EXPLORING ABSTRACT INTERFACES IN SYSTEM-ON-CHIP INTEGRATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Andrew Danowitz
July 2014

This dissertation is online at: http://purl.stanford.edu/tx933ws5747

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Stephen Richardson

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Modern mobile devices are marvels of computation. They can encode high-definition video, processing and compressing over 350MB/s of image data in real time. They have no trouble driving displays with as much resolution as a full laptop, and smartphone manufacturers boast of running games with "console quality" graphics. Mobile devices pack all of this computational power into a 1-2W hand-held package by integrating a number of specialized hardware accelerators (IP) along with conventional CPU and GPUs in a system-on-chip (SoC).

Unfortunately, creating these specialized systems is becoming increasingly expensive. Since hardware accelerators come from a number of different sources and design cycles, different accelerator blocks will often contain incompatible hardware interfaces. Therefore, a large portion of SoC design cost comes in the form of designers manually interfacing each accelerator into a system. This work includes everything from building custom logic to wire up a block, to developing the drivers and API needed to take advantage of the hardware.

My research focuses on generating these interfaces, including the physical hardware used to tie IP blocks into a system and the associated software collateral. Leveraging recent trends such as High Level Synthesis and other hardware "generator" methodologies, I propose an IP interface abstraction and parameterization designed to describe the interface of most current IP blocks. By encoding this knowledge at a higher-level of abstraction, I am able to construct and demonstrate a hardware generator that maps an interface protocol description into synthesizable register transfer language (RTL), and that can automatically create hardware bridges between different interconnect standards.

To ease the integration of the next generation of IP blocks—blocks that are automatically generated based off of user specification—I propose a set of interface primitives. When integrated into an IP generator, these primitives can automatically generate an interface that my interface system can tie to the rest of the system. I also demonstrate how the information stored in these types of primitives can be used to automatically generate a low-level software driver that manages access to the IP blocks.

Finally, I show how the simulation environment provided with an IP generator can be used to provide a domain appropriate application programming interface (API) to drive the software. Using an image signal processor generator as my platform, I demonstrate the construction of a map between the simulation software and hardware driver that enables a full one-button flow from algorithm development to applications running on specialized hardware within a working system.

Acknowledgments

I dedicate this thesis to my family and friends whose love and support helped me through graduate school. I would also like to thank my advisor, Mark Horowitz for his guidance, insight, and patience, and my mentors, Ofer Shacham and Stephen Richardson.

Contents

Abstract **iv**

Acknowledgments **vi**

1 Introduction **1**

1.1 Hitting a Power Wall: The Continued Case for Custom Design 3

1.2 Automating SoC Integration 6

2 Previous Work in Interface Generation **10**

3 Interface Abstraction **16**

3.1 Defining a Basic Interface 17

3.2 Creating an Interface Description 18

3.3 Handling Interface Complexity 22

3.4 Mapping Real Interfaces 23

3.5 Generating Interface Bridges 30

3.6 Validating the Abstraction 34

3.7 Extending the Generator 35

3.8 Summary 37

4 Advertising Native Interfaces **38**

4.1 Image Signal Processor Generator 40

4.2 Mapping HLS to RTL 42

4.2.1 Specifying Flow Control and Access 45

4.3 Building an HLS-to-RTL System 48
4.4 Interconnect Generator 51
4.5 Testing and Summary 54

5 Automating Software Generation **56**
5.1 Building Drivers 57
5.1.1 Driver Design Techniques 58
5.1.2 Generating the Driver 61
5.2 Generating the API 66
5.2.1 Mapping the API to the Driver 68
5.2.2 API Limitations and Future Work 69
5.2.3 API Summary 70

6 Conclusions **71**

Bibliography **74**

List of Tables

2.1 Signal and encoding differences between AXI3 and AXI4. 11

3.1 Parameters required to encode the basic IP interface. 19
3.2 Parameters required to encode the basic bus. 24
3.3 Parameter Mappings for the system buses. 27
3.4 Keywords used to map interface functionality to specific signal names. 28

4.1 Summary of my Genesis 2 interface primitives. 45
4.2 Summary of flow control details primitives pass to HLS. 49
4.3 Suffixes for constructing different control signal names. 50

5.1 Summary of IP interface information provided by each primitive. . . . 63

List of Figures

1.1 Die photo and block diagram of NVidia Tegra K1 processor [23]. . . . 2
1.2 Annotated die photo of Intel “Haswell" processor [27]. 3
1.3 Frequency scaling of processor designs over time [14]. 4
1.4 Power density of processor designs over time [14]. 5
1.5 Voltage versus feature time of historical processors [14]. 5
1.6 Energy efficiency for algorithms on different platforms [33]. 7

3.1 High level flow for the interface generator. 30
3.2 High level architecture of the prototype bridge generator. 32

4.1 Block diagram of an RTL IP architecture. 43
4.2 Code to instantiate and configure *buffered* primitive. 50
4.3 Code used to map an interface bus in an IP design to a *buffered* primitive. 51
4.4 Code used to generate Verilog module instantiation from primitives. . 51
4.5 Code used to map an interconnect standard to my primitives. 53
4.6 Flow diagram of how my primitives work with Genesis 2. 55

5.1 Block diagram of my driver generator system. 61
5.2 A C struct mapping interface resources to the `mmap` structure. 65

Chapter 1

Introduction

As seen in Figure 1.1, SoCs incorporate traditional processor cores as well as a myriad of custom hardware accelerators. These accelerators are designed to compute data intensive applications in real-time, like graphics, image processing, and wireless communications, and are optimized to do so in a very energy efficient manner. As a result, they are widely used in the mobile space where packaging and battery life requirements necessitate highly efficient computing solutions.

In recent years, the energy and performance benefits of integrating custom hardware on-die with the processor has led traditional desktop processor manufacturers to start moving towards SoC-like designs. Figure 1.2 shows the die of Intel's latest generation "Haswell" desktop processor. While the processor cores and cache take up more area here than in mobile SoCs, over a third of the die is dedicated to accelerators and peripheral controllers, including graphics, memory controllers, and display handlers.

The energy benefits of SoCs, however, come with a price. The complexity of getting all of these different hardware accelerators, or "IP blocks" to work together has caused the engineering costs of developing and verifying an SoC design to skyrocket, and by some estimates, the cost of developing the software to get an SoC system to function now dwarfs the cost of actual hardware development [21]. These factors have meant that the number of new custom chip starts is actually decreasing [20]. To help alleviate these factors, my doctoral work attempts to leverage recent trends in

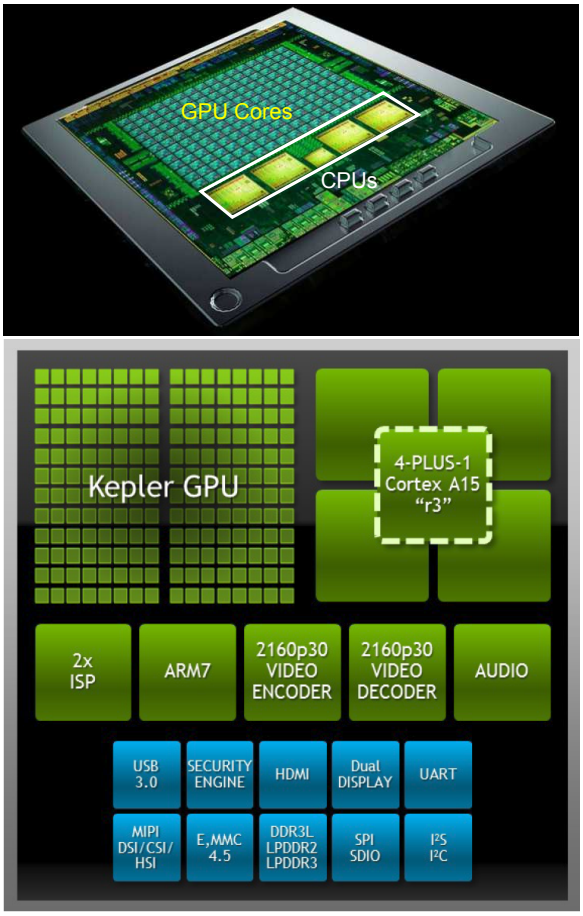

Figure 1.1: Die photo (top) and block diagram (bottom) of upcoming NVidia Tegra K1 processor. In addition to processor cores, the chip contains a substantial graphics fabric, video and image processing units, and a range of other peripherals [23].

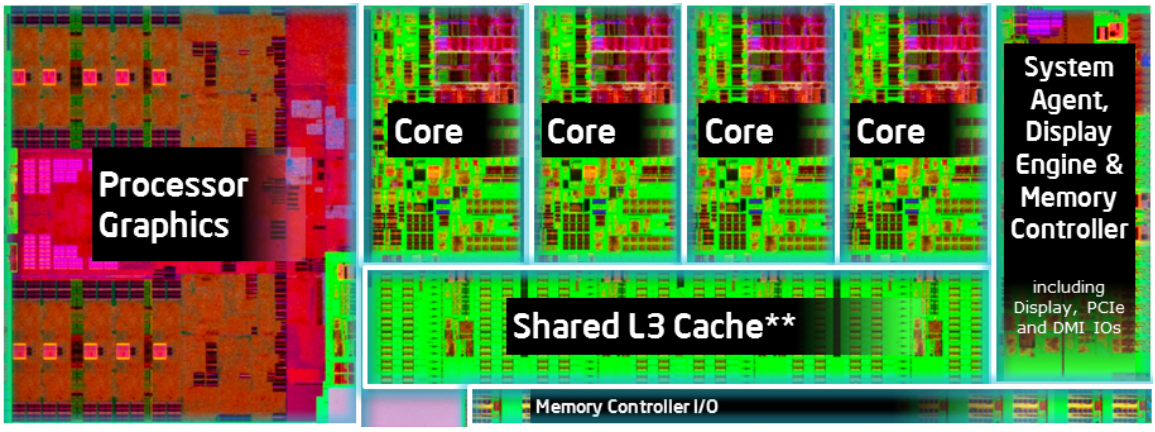

Figure 1.2: Annotated die photo of Intel "Haswell" (4th Generation Core Architecture). Less than half of the die area is dedicated to processor cores [27].

high-level synthesis (HLS), hardware generation, and domain specific programming languages (DSLs) to help automate the process of IP integration.

1.1 Hitting a Power Wall: The Continued Case for Custom Design

With the rising costs of chip design, one might expect that custom would be on its way out. The historical growth in the performance of general purpose processors made it seem like many of the applications that used to require custom accelerators could eventually be migrated into software. In practice, however, more and more portions of die area dedicated to custom accelerators and other types of specialized computation engines: since the release of Intel's Sandy Bridge and AMD's Llano architectures in 2011, many mainline desktop parts have started incorporating programmable graphics engines, among other accelerators directly on-die, providing a huge boost to the mathematical abilities and parallel computing resources of these parts. Also, modern mobile and desktop processing parts like the K1 and Haswell continue to dedicate large portions of die area to custom logic.

The reason for the continued success of custom is that most modern processor designs are power limited. As seen in Figure 1.3, the rate of general purpose processor

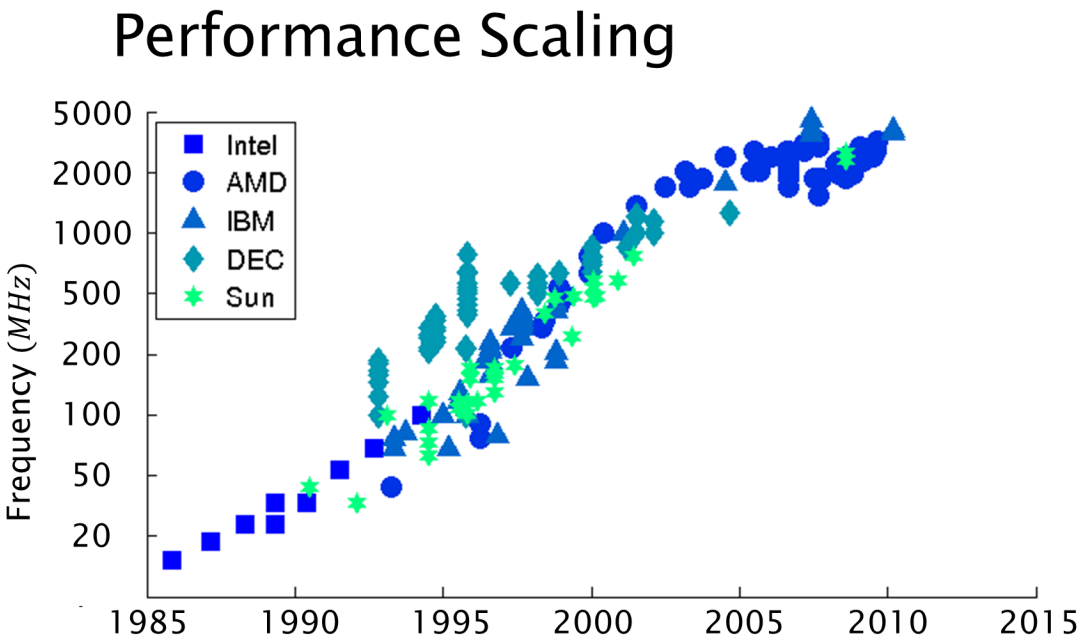

Figure 1.3: Frequency scaling of processor designs over time [14].

performance growth, as measured by operating frequency, has slowed considerably since 2005. The reason for this is that designers had been exploiting architectural techniques that increase performance at the cost of increasing power density [14]. This is shown in Figure 1.4. When processor power density reached roughly $1W/mm^2$ [14] in 2005, however, designers reached the limit of what could be efficiently air-cooled. From that point on, architects could no longer trade power for performance, which greatly slowed the rate of performance scaling.

To make matters worse, since roughly the 45nm generation, power and performance benefits from technology scaling have declined. According to Dennard's Constant Field Scaling [15], if all of the physical dimensions and the threshold voltage of a transistor are scaled down by a factor of α, the energy required to switch a transistor drops by a factor of α^3. Historically, this meant that as feature size has dropped by a factor of $\sqrt{2}$ with each technology node, designers were able to double the number of

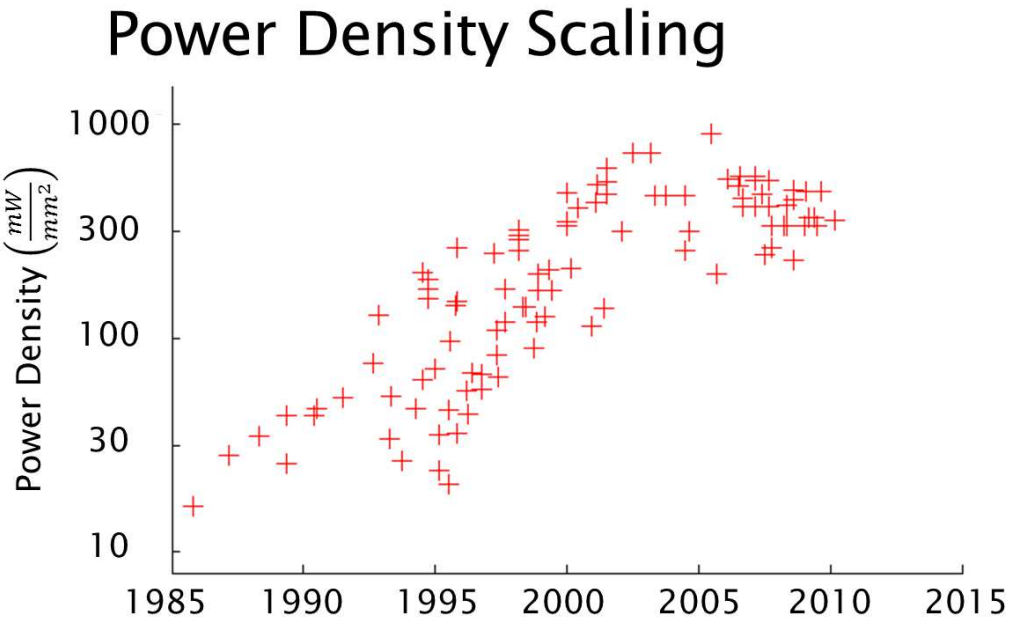

Figure 1.4: Power density of processor designs over time [14].

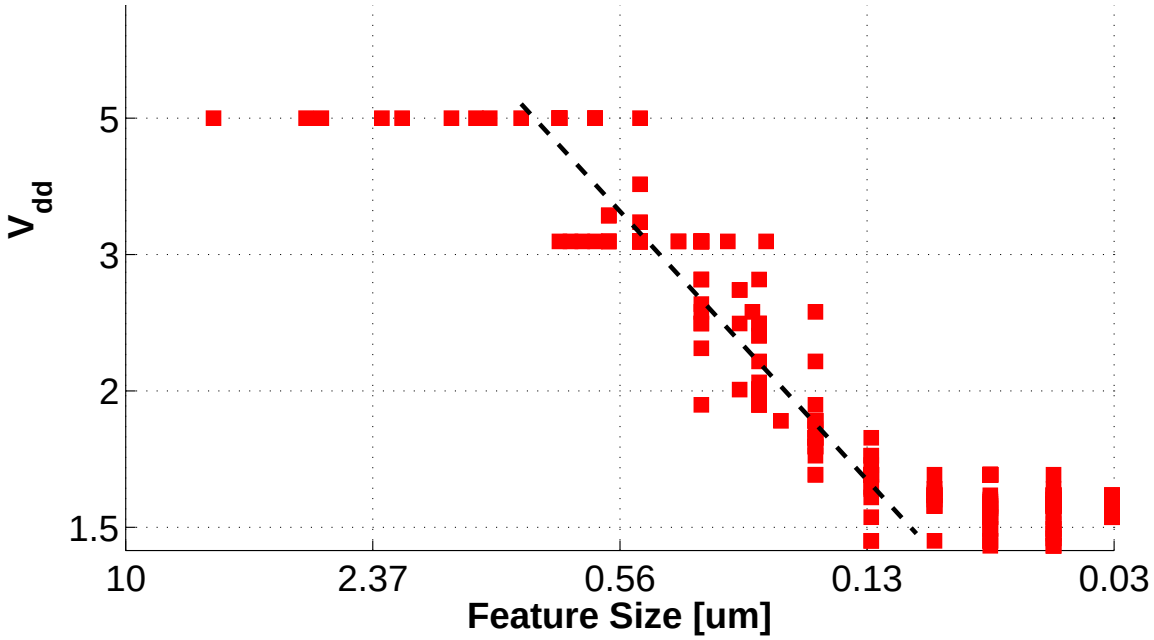

Figure 1.5: Voltage versus feature size. Voltage scaling, which began at roughly the half micron node, has largely leveled off since the 45nm generation [14]. The trend-line is provided to show the sharp cutoff in scaling.

transistors, increase the processor's operating frequency, and still maintain a constant power density. Unfortunately, for performance to improve as operating voltage, V_{dd}, scaled, the threshold voltage, V_{th}, needs to scale as well. Due to leakage power concerns, however, threshold voltages are no longer scaling at the same rate as the rest of the transistor, as illustrated in Figure 1.5, and thus V_{dd} scaling has dramatically slowed as well.

Since performance is proportional to operations per second and power is proportional to the product of energy per operation and performance ($P \propto \frac{ops}{s} \times \frac{E}{op}$), if power is fixed, the only way to increase performance is through decreasing energy per operation. This can be accomplished by tailoring hardware to specifically match the needs of the underlying algorithms. Custom accelerators are designed to do this.

By giving up the generality found in general purpose processors and optimizing data paths for a certain class of algorithms, custom accelerators can achieve up 1000 times lower energy than general purpose processors [22], as shown in Figure 1.6. As a result, the SoC design methodology and customization are likely to play an important role in chip design for the foreseeable future.

1.2 Automating SoC Integration

In this thesis, I attempt to reduce some of the hardware and software related design costs through automation. My work does this by using the hardware generator design methodology—rather than building a single hardware instance, we build software-like constructors to generate customized hardware instances—to automate the integration of IP blocks into an SoC design.

Much work has already been done to simplify the process of wiring an IP block into a system on chip, and this work is overviewed in Chapter 2. IP blocks generally adhere to one of many industry-standard interfaces that were designed to aid the problem of integration. Unfortunately, the wide range of interface standards, and backwards compatibility issues between interface revisions mean that an SoC integrator will likely end up using IP blocks with incompatible interface standards. To address this problem, researchers have explored various ways to encode the IP communication

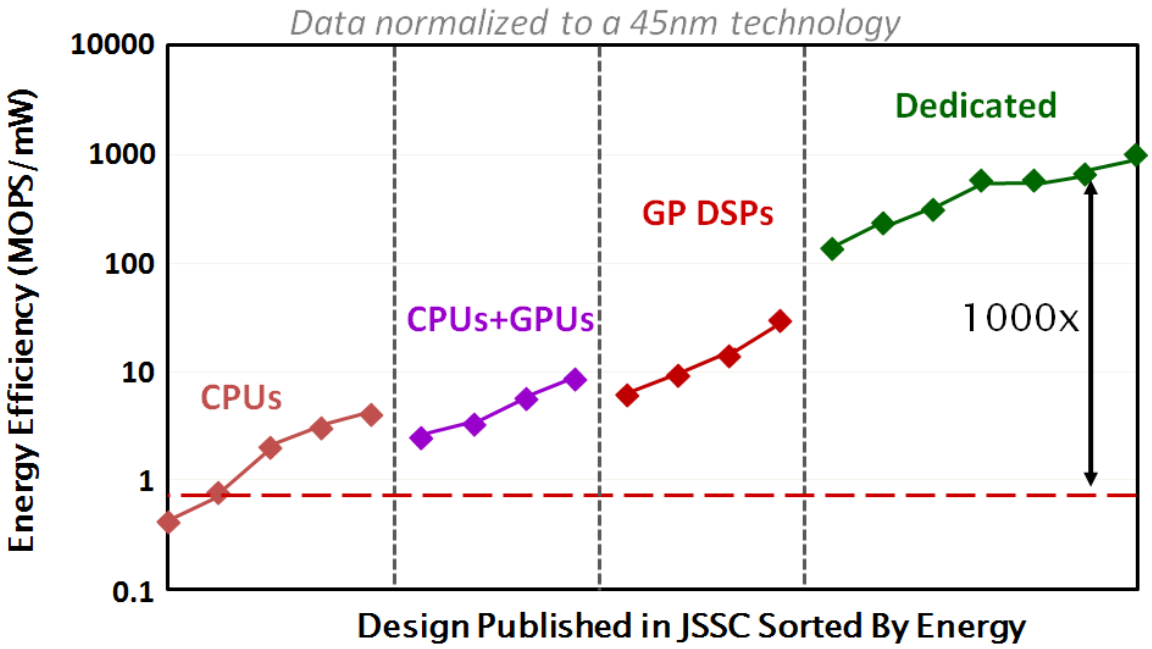

Figure 1.6: Energy efficiency for algorithms implemented on different platforms [33]. Each mark on the X-Axis represents a unique design published in JSSC.

protocol at a higher level, and automatically generate bridge logic. This chapter will review these methods, and highlight some improvements that a generator-based approach can address.

Chapter 3 attempts to address the problems of connecting IP with fixed interface standards to each other and to modern high-level designs by introducing an abstraction for IP style interfaces. I identify four key characteristics that must be present in any IP interface and then propose a parameterization of this interface space that is flexible enough to account for the differences between most IP buses. Using my abstraction, I demonstrate a method for automatically generating hardware to bridge between any two supported interfaces. The conversion method I use is based on three steps that handle signal resynchronization, physical resources provisioning, and control signal conversion, and is implemented in the Genesis 2 [43] generator language. I verify my generator's functionality by producing and simulating bridges for a number of popular interconnect standards.

While my interface abstraction works well for existing IP blocks, future SoC designs are likely to use high-level design methodologies to help ease the process of IP design and to automate hardware integration. Since designers may still rely on RTL-based design methodologies for certain specialized accelerators that don't map well to their high-level design tools, in Chapter 4, I demonstrate a system for mapping high-level interface elements to RTL-based accelerator blocks. The work discussed here allows any IP designs that must be specified through RTL to still take advantage of the system-integration benefits of high-level synthesis.

Next, to push the capabilities of IP integration into the software domain, Chapters 4 and 5 discuss my work to introduce transaction level model (TLM) based interfaces into Genesis 2, and to adapt the interface information found in these interfaces to automatically generate a C driver for generated IP blocks. I demonstrate and verify the functionality of my interface primitives and driver generator in the context of an image signal processor (ISP) IP generator (ISPGen) [10].

With a mechanism for automatically generating a software driver, Chapter 5 then focuses on how we can automatically generate software APIs for the IP so that domain experts without hardware knowledge can take advantage of the hardware. This

chapter demonstrates a proof-of-concept methodology for automatically generating hardware APIs. All IP generators will come with some sort of simulation collateral that allow a system designer to test various parameter options before settling on a final fixed IP. Since the same set of generator parameters and design constraints govern how the hardware and the software simulator are created, for a given set of IP parameters, it is generally possible to create a mapping between the relevant simulation interface and the required driver commands to complete the same computation on the hardware. By integrating this technique into the ISP generator, we are able to construct a system that allows for domain experts with little knowledge of computer hardware to experiment with novel image processing algorithms in real-time.

Chapter 2

Previous Work in Interface Generation

SoC design methodology is widely used in chip design. It allows users to assemble entire systems out of pre-built, pre-verified IP blocks. These IP blocks can be sourced from both internal and external vendors, so, for example, an SoC might feature a processor design from ARM, and graphics from Imagination. It is the job of the SoC designer to integrate these devices together to form a fully functional system.

To help manage the complexity of integrating IP blocks from different vendors, most IP blocks adhere to one of a variety of interface standards. These standards define the signals, timing, and handshaking protocols used by the IP blocks to communicate. Major system designers like IBM [26], Intel [28], and ARM [2], and a number of consortia like Accelera [36] and Hypertransport [24] all maintain sets of incompatible interconnect standards. Therefore, it is likely that not all of the blocks a system designer plans to use advertise the same interface standard.

To make matters worse, standard groups often maintain more than one standard for IP interfaces. The widely used ARM AMBA standard, for example, is actually a family of 10 buses and bus variants[1]. Each of these buses is designed with a different use case in mind. For example, AXI is used as an interface for generic high-performance peripherals. AXI-Stream, on the other hand, is specifically designed for

[1] APB, ASB, AHB, AHB-Lite, ATB, AXI, AXI-Stream, AXI-Lite, ACE, ACE-Lite

blocks with streaming interfaces, and APB is used for low performance functions, like control interfaces. These buses are all incompatible, and custom bridge logic is needed to convert between the ARM interfaces. Therefore, even if a particular group's standard dominates a market, like ARM buses in the mobile space, designers may still face the problem of trying to integrate multiple interface standards.

Finally, even if blocks conform to the same nominal standard, there is no guarantee that they can actually communicate with each other. As technology progresses and designers find new ways to optimize their interconnects, interface standards are revised. Often these revisions are not backwards compatible with the version they replace. For example, in version 3 of the AXI protocol, information sent along the *write* bus can be reordered independently of the corresponding data sent on the *address* bus. To keep track of which data is associated with each address, the address and data values of a single transaction are assigned a transaction ID. In AXI version 4, however, address and data can no longer be independently reordered, and the IP blocks are designed to assume that any address and data pairs they receive, regardless of timing, correspond to the same bus transaction. Other discrepancies between the two standards are shown in Table 2.1. Because of these discrepancies, AXI3 peripherals are not directly compatible with AXI4, and require a custom hardware bridge to convert between protocols.

Table 2.1: List of signal and encoding differences between AXI3 and AXI4. Note that "x" can refer to either "R" or "W" (e.g. ARLOCK)

Signal	Change
AxLEN	Incrementing burst extended to support up to 256 transfers.
AxLOCK	AXI4 removes support for locked transaction. This simply becomes a directive to the interconnect arbiter.
AxCACHE	Adds new order requirements for certain transaction types, updated definitions of bit meanings.
WID	Only exists in AXI3. AXI4 eliminates ability to reorder write data relative to write access.

All of these factors highlight major issues with the use of IP standards to integrate systems on chip: there is no single standard interface that will allow any IP block to

seamlessly integrate into any system, nor can there be. It is impossible to know what will be needed in the future, so we need to plan to deal with changing bus interface descriptions.

Automation seems like a solution to this evolving interface problem. Rather than forcing system integrators to build custom bridges between different interfaces, it would be much more convenient to have a hardware constructor that can take in the protocol used by each component, and automatically synthesize the logic needed to tie everything together. As a result, there has been a substantial amount of work on describing and synthesizing protocols going as far back as the mid 1980s [34, 9]. Early work focused on synthesizing interfaces from event graphs and event sequences. In 1997, Rowson et al. proposed that, for design purposes, interface communication could be treated separately from the low-level interconnect implementation [42]. Much like the OSI 7-layer network abstraction [41] which provides an abstract framework for communicating data over heterogeneous networking equipment, this abstract separation has helped shape modern work on system integration.

Inspired, in part, by Rowson, much work has already been completed in creating IP independent interconnect networks [31, 13, 46]. Several works propose that the bulk of communication and data routing on chip be completed by a purpose-built high-performance network on chip (NOC). Since the interconnect is designed independently of the IP blocks, these systems need interfaces between the IP blocks, and have been a target for automatic interface generation. Products from companies like Sonics [45], Arteris [4] and various research projects [7, 40] all offer the ability to automatically generate a custom, high-performance interconnect system. These NOC generators leave options like data-widths, network switching characteristics, and other characteristics as optimization parameters so that SoC integrators can tune the performance, energy, and area to meet the requirements of their final system.

While NOC generators have been successfully implemented in the interconnect space, a number of factors have limited the success of attempts to automatically bridge the interconnect to the IP interfaces. One major hurdle towards automatic IP integration is that the SoC integrator often has no control over what the interfaces to each IP block looks like. While many blocks may adhere to popular standards like

AMBA, individual blocks may have unique protocols or cutting edge features that the system integrator has not faced before. In order to accommodate these blocks, much of the previous work in automating IP-to-interconnect connections has focused on building extremely flexible generators.

Finite state machines (FSMs), for example, are widely used to specify formal definitions of interface behavior protocol and for synthesizing protocol conversion hardware. FSMs formally encode the functionality of an interface protocol by modeling all of the possible transactions and transitions that may occur on an interface. Recent work by Avnit et al. has demonstrated that FSM-based models are sufficient to represent all of the functionality found in many modern IP bus standards [5]. This same work also demonstrates how FSMs-based interface models can be used to formally prove whether two interfaces are compatible. Avnit's work also demonstrates an algorithm for synthesizing protocol converters from two protocol models. The formalism introduced by such approaches greatly aids the process of design verification by providing increased confidence in the correctness of the IP-to-system interface and by providing simulatable models for each connection.

In Avnit's work, the user enumerates the number of distinct states that the bus can operate in and divide channels up into categories of data, input control, and output control. For each state transition, the user mathematically specifies the "guard" conditions for transitioning to various states. These guards either involve checking for the presence or absence of a desired value on an input control signal, or checking the value of special user-defined bound counters. For each state, the user specifies whether a value should be read or written from the data channel, and specifies which values should be asserted on the output control channels. All of these specifications exist as mathematical equations. To convert between two protocols, the user specifies a mapping between data and control channels for two protocols, and Avnit's system mathematically determines a state machine description that can convert between the two buses.

For complicated buses, building and verifying one of these FSM protocol descriptions requires a significant design effort. To make matters worse, there's no clear mechanism for reusing portions of FSMs to describe other interfaces. For example,

while AXI and APB both use a similar "valid-ready" handshake mechanism, the FSMs are not partitioned in such a way to allow the handshake mechanism from one definition to be reused in another. Therefore, much of the effort of specifying common interface mechanisms may need to be repeated for each interface modeled.

To make finite state machine models more approachable for designers, we would like to have a high-level way of specifying them. Ideally, we could develop an abstraction for IP interfaces that can encapsulate complex interface protocols in a succinct description. We could then build a generator to either map the description into a mathematical FSM description or create bus converters from the high-level descriptions directly.

Attempting to address this complexity issue are interface specific languages or grammars that can be used to define custom interfaces. Most of these languages directly map grammar elements to hardware implementation [30][19]. These grammars tend to limit the communication protocols to what can be defined by composing a set of fixed hardware stages or block, limiting the impact of such tools on design cost.

Other groups working on automating IP interconnect have avoided the complexity problem by limiting the number of protocols supported. For example, companies like Sonics have developed bridges capable of connecting interconnects produced by their NOC generator to some AMBA and OCP buses. They have not published the mechanisms that they use to complete this conversion, however, and it is not clear how much the interface produced by their NOC changes with different implementations. Also, if an IP interface is not explicitly supported, it is up to the system-integrator to manually create a bridge capable of tying the block to the NOC interconnect.

If bus details may vary, perhaps it is better to define interfaces at a higher level using high-level synthesis (HLS) design methodologies. One example of an HLS approach is the use of transaction level modeling (TLM) to generate IP-to-interconnect RTL, as exemplified by the works of Cho et al. and Lee et al. [12, 32]. Transaction level modeling allows designers to focus on high-level communications between the controller and various resources on an IP block. From a designer's perspective TLM interface can be as simple as issuing "read" and "write" commands. Lower level details, such as flow control mechanisms, and whether data is transfered via memory

mapped I/O (MMIO) or through a direct memory access engine (DMA), are obscured from the user and automatically implemented by the HLS software. By freeing system designers from specifying the interface implementation details, TLMs provide an important tool for tackling the issues of system integration design cost. Also, since the low-level RTL is algorithmically generated from the model by an HLS software package, it potentially reduces the amount of human error in the interface RTL.

While techniques like this show great promise for systems that are fully generated, there is still the problem of connecting to existing IP blocks with fixed interfaces. TLM-based solutions like Cho et al.'s rely on fixed libraries of protocol definitions for this compatibility. This means that somewhere in the process, a designer must still model every existing interface that they would like to use. Therefore, like FSM-based techniques, TLM-based interface synthesis techniques could also benefit from the creation of a high-level interface abstraction that's capable of succinctly capturing the functionality of existing interface blocks.

For the first major contribution of my doctoral work, I introduce and validate such an abstraction. In Chapter 3 I identify and define a constrained interface design space directly applicable to IP interfaces. Within this space, I propose a parameterization of interface features that is rich enough to capture the physical designs I have found in a simple description.

Chapter 3

Interface Abstraction

My ability to generate a simple high-level description of interfaces rests on one basic idea: all IP interface standards are very similar. Most IP interfaces serve the same function: to move data to/from a location in the hardware from/to the processor/memory space. This function requires 3 major pieces of information: the data being moved, the address it is being moved to or from, and the operation that should be performed on this data. In addition to these pieces of information that are communicated, these IP interfaces need to specify policies for how to control the flow of this information in the network. This chapter uses these concepts to allow a small number of parameters to specify a large number of current IP interfaces, and provides a way to semantically link signals from different interface standards.

In Section 3.1, I show that completing different types of transactions operations requires each interface standard to transmit a common set of information. Since different standards may include a dizzying array of special operation types—streaming operations, atomic accesses, cache coherent accesses, etc.—Section 3.1 only focuses on the information each interface must encode to perform basic reads and write, leaving a discussion of more advanced interface functionality for later in the chapter. Section 3.2 uses this simplified interface model to outline my strategy for creating a simple way to specify these buses at a higher level. Later, in Section 3.3, I discuss how I extended my interface definition to include more advanced functionalities. Finally, I close out the chapter with a discussion of how I used my interface description to

construct a hardware bridge generator, capable of automatically creating interface-to-interface bridge RTL.

3.1 Defining a Basic Interface

When analyzed at a high-level, all of the standards I have encountered are concerned with executing some flavor of read and write operations. To complete these operations, every IP interface must encode a core set of information. First, IP interfaces must have a **data** field for the read and written operations. Second, since SoC interconnect networks are generally designed to connect one or more masters to a set of hardware peripherals, and since many IP blocks have a number of interface resources that the master may want to access, I can also assume that the majority of IP interface standards will have a concept of an **address**, that can be used to route read and write requests to the proper desination. Finally, if interface standards can handle multiple types of operations, both a read and write, for example, it must also have a mechanism for specifying the fundamental **operation** type. Regardless of physical layer implementation, all three of these sets of information must be communicated between the sending and receiving blocks in order for the blocks to correctly process each transaction.

To complete these transactions, each interface must also define some aspects of their protocol for sending information over the physical layer. I categorize this type of information as **flow control**. To effectively communicate more than one piece of information, the interface's flow control must define what makes a distinct transaction, or message. This includes defining the information that is contained in a single transaction—generally some combination of data, address, and operation and control signal.

From a flow-control perspective, the interface also needs to define when different signals on the interface are part of a valid transaction, and mechanisms for the sender and receiver to negotiate when valid information can be sent—in other words, IP interfaces must have a definition for synchronization and handshake.

Taken together, these four categories of functionality form the bare-minimum functionality of what's found in any IP interface: **data**, **address**, **operation**, and **flow control**.

3.2 Creating an Interface Description

While all basic buses are similar at a high-level, they vary greatly in terms of their implementation details and mechanisms. These implementation details must be accurately captured for each standard in order to accurately represent and reconstruct the bus.

I have developed a set of parameters for each category of *data*, *address*, *operation*, and *flow control* capable of capturing these mechanisms. My parameters are shown in Table 3.1. Since my aim is to reduce the amount of effort required to specify an interface definition, I tried to keep my set of parameters as small as possible.

For the basic interface discussed in Section 3.1, the parameterization was fairly simple. I started with the assumption that every interface is going to have a dedicated data bus for each supported operation type. Therefore, the first thing I need to know about the bus is what operation types are supported, read and/or write. This is encoded in my *op_enable* parameter. I also need to know how the data is encoded at the physical level. In my current set of parameters, I capture information about the size of the data bus (*data_size*), the size of a data word (*data_word*), and the endianness of any information sent over the bus (*endian*).

The parameterization for the address space was also fairly straightforward, as I have only come across a handful of mechanisms for specifying address. While some IP interfaces may multiplex addressing information with other buses, such as data, the IP blocks and interface standards I have worked with in my research either maintain an explicit address bus, or are point-to-point links where the address is implicit. Therefore, my parameterization assumes that if the bus is addressable (*address_enable*), an address bus exists.

One of the most common variations that I have seen in the address space is the inclusion of a *chip_select* signal. One bit of this signal is routed to each IP

Table 3.1: Parameters required to encode the basic IP interface.

Parameter	Description
Data	
Interface scope	*Defines the directionality and basic op types*
op_enable	Determines if interface is read, write, or both
Data characteristics	*Describes the format of data to be passed*
endian	Big or little
data_size	Size of data bus
word_size	Size of a data word
Address	
Address scope	*Defines how addressing is accomplished*
address_enable	Specifies whether bus is addressable
address_size	Specifies the width of the address bus
slave_select_enable	Specifies a onehot IP enable bus
slave_select_map	Maps address range to slave-select signals
shared_rw_channels	Is address shared by read and data
Flow Control and Timing	
Handshake	*Handshake used for data transfer*
flow_control	High level handshake protocol
flow_map	Map between valid ops and encoding
reply_path	Does slave send replies?
shared_ready_valid	Does the reply valid signal also act as slave ready
reply_map	Encodings of valid and error responses
master_stallable	Can the slave insert idle cycles?
slave_stallable	Can the master delay a response from a slave?
max_ops_outstanding	Number of ops that may be in-flight at once
Synchronization	*Relative timing between components of a message*
address_data_sync	Is address to data timing fixed or variable

Continued on next page

Table 3.1 – continued from previous page

Parameter	Description
sync_cycles	If fixed, how many cycles are they separated by
write_sync_address	List of control signals sync'd with the address bus
write_sync_data	List of control signals sync'd with the data bus
trans_id	Specifies if transactions have IDs
reorder	Specifies if transactions can be reordered
Operation	
Operation	*Defines how read/write are specified*
operation_enabled	Is there an "op" field?
read_write_type	Is rd/wr specified by a bit or bus?
read_write_encoding	Encodings for read vs. write ops

block and indicates whether a transaction is relevant to each IP. The other common variant is whether a single address bus is shared by read and write channels (*shared_rw_channels*). My parameterization supports both of these variations, and offers a *chip_select_map* parameter to allow the system to convert from a raw address on one side of the converter to the appropriate chip select bit on the other. Additionally, there is a category of address characterization that describes the size of the address. The endianness of the address is assumed to be inherited from the data characteristics.

As previously discussed, I divided the functionality of flow control into two distinct categories. The first category, the *handshake*, encodes the mechanism used by a master to signal valid data on a bus and the mechanism used by the slave to communicate that it is sampling the data. For the various system bus standards I analyzed, the handshake method was generally limited to simple *ready-valid* or *ready-operation_type* mechanisms, where the valid and op signals are combined into one bus. Since handshake protocols are generally designed to be used as a unit, my parameters encode the handshake by protocol name (e.g. *ready-valid*), and my system currently allows for both *ready-valid* and *ready-operation* mechanisms to be specified.

The number of handshake mechanisms may be expanded, however, to accommodate different mechanisms such as credit-based flow control.

It is also important to note that the naming convention of some of these handshake protocols implies the timing of the protocol itself. *Ready-valid* for example implies that the slave will issue a ready a signal whenever it can receive data, so that valid data can be consumed as soon as it is available. Conversely, in a *valid-ready* handshake, the slave will only issue ready after the valid signal has been asserted. While the signals in the interface are the same in either case, this distinction in protocol can cause a system to lock up if both master and slave are waiting for the other to advertise a possible transaction.

In parameterizing the synchronization aspects of flow control, my system makes the assumption that all control signals in an interface are synchronized with either the address or data buses. With this assumption, the parameters only need to encode the relative timing between address and data and which control signals are synchronized with which bus to fully capture how each interface is synchronized. My system uses the *sync_per_channel* map parameter to indicate for both read and write operations which control signals are associated with data and which are associated with the address.

My system uses another two parameters to encode the timing between address and data signals. In some cases, there is no fixed timing relationship between address and data, but instead each bus has its own set of handshake signals to handle synchronization. In many cases, however, there is a fixed timing relationship between the two. Therefore, my system includes one parameter to encode whether the timing between address and data is constant or variable. A second parameter specifies the number of cycles address arrives before data in a fixed timing system. This parameter is only used if there is a fixed timing relationship between address and data signals, and users are allowed to set this value to a negative number if data arrives first. So long as my assumption about control signal synchronization holds, these parameters should be sufficient to encode any synchronization found in an IP interface.

Finally, for each basic IP interface, my parameterization assumes that there is a mechanism for transmitting whether an operation is a read or write. This distinction

can be determined automatically if read and write independent address and handshake buses—e.g. a transaction on the read channel is a read operation. For other protocols, however, it is possible for users to specify a signal in the interface that indicates the type of each operation, and so I offer them a parameter to map encodings on the selected signal as either reads or writes. Unless an IP interface distributes whether an operation is a read or a write across multiple signals, these two parameters are enough to convey basic operation types.

3.3 Handling Interface Complexity

While I was able to develop a relatively simple set of parameters to describe the operations of a simple bus, most buses found on IP interfaces are far more complex. Rather than sending a fixed-size word on every transaction, many modern buses are designed to send transactions of varying sizes, or even to allow users to mask out certain bytes of the data word. In fact, many common IP interface standards from ARM [2] and IBM [25] simply do not work for basic transactions if the variable size signals are not implemented.

Also, as interface standards evolve and are optimized for different use cases, the types of operations that they are capable of completing tends to grow. Interfaces specialized for high-bandwidth applications, for example, may incorporate streaming reads and writes. Different interfaces also incorporate features like atomic data operations, or support for cache coherent operations. These types of features often involve sending additional control information with each interface transaction.

To support the addition of high-level features like these, my bus definition must be expanded with new sets of parameters. As part of my research, I expanded my bus definition to cover common features, such as variable data size transactions, and streaming or "burst" transmission modes. I prioritized these functionalities over other features since several common interface standards, including ARM's AXI, require both variable transaction size signals and burst-mode signals for even basic basic read and write operations to work correctly.

In the bus standards I have analyzed, the mechanisms used to communicate data

size vary substantially. In AXI and AHB-Lite, masters communicate size over a dedicated size bus. Masters in OPB, however, uses predefined one-hot signals to indicate transaction width (e.g. there is a bit to indicate a half-word transaction, and another to indicate a whole word). Also, while in certain systems only the master reports transaction size, in others, including OPB, the slave reports its own width to the master. This allows for the master to determine the maximim transaction size accepted by each IP block at run-time, but adds to the complexity of the bus specification.

These differences can all be abstracted into a set of three parameters: one that encodes which elements (masters and slaves) report size; how the size is encoded, either as a value on a bus or through a set of one-hot signals; and a parameter that maps values on the size signals into the numerical word sizes. If the first parameter indicates that neither master nor slave reports size, the other two size reporting parameters are simply ignored. By giving the system designer the flexibility of specifying a map for how size is encoded, these three parameters allow a wide range of sizing mechanisms to be specified.

The full range of parameters for each functionality is enumerated in Table 3.2. Combined, these parameters form my IP interface specification.

3.4 Mapping Real Interfaces

To ensure that my interface definition was expressive enough to capture the functionality of real IP interfaces, I map several standards from ARM and IBM into my definition. Table 3.3 shows the resulting parameterization for each of the buses. With the exception of some advanced features like cacheability that were intentionally left unimplemented, the parameters were able to represent all of the buses' protocol and physical specifications in the model. This was not surprising, however, since the design of my parameterization space was informed by the variations found in many bus standards, including the ones I mapped.

During the course of my development, whenever I ran into a required bus feature that did not exist in my current description, I either refactored existing parameters or

Table 3.2: Parameters required to encode the basic bus.

Parameter	Description
Data	
Interface scope	*Defines the directionality and basic op types*
op_enable	Determines if bus is read, write, or both
Data characteristics	*Describes the format of data to be passed*
endian	Big or little
data_size	Size of data bus
word_size	Size of a data word
dynamic_sizing	*How is transaction size reported (opt)*
size_reply	Adds size buses from slave
size_encoding	Is it a "bus" or "onehot"
sizes	What sizes are supported "word," "halfword," etc.
mask_enable	Enables mask bus
mask_granularity	Number of data bits a match bit applies to
Address	
Address scope	*Defines how addressing is accomplished*
address_enable	Specifies whether bus is addressable
address_size	Specifies the width of the address bus
slave_select_enable	Specifies a onehot IP enable bus
slave_select_map	Maps address range to slave-select signals
shared_rw_channels	Is address shared by read and data
Flow Control and Timing	
Handshake	*Handshake used for data transfer*
flow_control	High level handshake protocol
flow_map	Map between valid ops and encoding
reply_path	Does slave send replies?
shared_ready_valid	Does the reply valid signal also act as slave ready

Continued on next page

Table 3.2 – continued from previous page

Parameter	Description
reply_map	Encodings of valid and error responses
master_stallable	Can the slave insert idle cycles?
slave_stallable	Can the master delay a response from a slave?
max_ops_outstanding	Number of ops that may be in-flight at once
Synchronization	*Relative timing between components of a message*
address_data_sync	Is address to data timing fixed or variable
sync_cycles	If fixed, how many cycles are they separated by
write_sync_address	List of control signals sync'd with the address bus
write_sync_data	List of control signals sync'd with the data bus
trans_id	Specifies if transactions have IDs
reorder	Specifies if transactions can be reordered
Operation	
Operation	*Defines how read/write are specified*
operation_enabled	Is there an "op" field?
read_write_type	Is rd/wr specified by a bit or bus?
read_write_encoding	Encodings for read vs. write ops
Burst mode	*Defines burst mode mechanisms*
burst_enabled	Does bus support burst?
burst_only	Are all transactions "bursts"?
early_term	Can a master terminate a burst?
wrap_enable,	Does the bus support address wrapping bursts?
inc_enable,	Does the bus support incrementing address bursts?
fixed_enable	Does the bus support fixed address bursts?
length_provided	Does burst send number of transactions in the burst?
length_map	Map between burst length and signal encodings
last_provided	Does the burst raise a flag on the last transmission?
first_provided	Does the burst raise a flag on the first transmission?

Continued on next page

Table 3.2 – continued from previous page

Parameter	Description
master_updates_addr	Does the master update the burst address each cycle?

added new ones to implement the required features. For example, an early version of my parameters only allowed bus masters to report the data size of a given transaction, and assumed that the data value being passed on the data size bus would be the number of bytes being transmitted. While these assumptions held for AMBA buses, the OPB implementation did not fit. OPB requires both masters and slaves to report their sizes on a per transaction basis, use a one-hot mechanism for advertising size, and encodes larger message sizes in terms of number of data words, rather than number of bytes.

To accommodate the OPB bus, I expanded the set of parameters dealing with advertising transaction size. Since my old definition had no concept of a slave offering a size and always assumed that size information traveled over a single bus, I added two new parameters to specify whether slaves replied and how these messages are physically transmitted. To accommodate sizes defined in terms of word-length, I merely expanded the scope of the data-size encoding map parameter to allow users to define size encodings in terms of number of words as well as number of bytes. Since my definition captures word size in a separate parameter, it is trivial for any system using my definition to convert between words and bytes. Note that while adding support for OPB required me to add some new parameters, by splitting OPB's size reply behavior into several orthogonal components and by incorporating these parameters into the existing size reply parameter subset, I was able to expand my interface definition in a way that could potentially allow me to support size reply mechanisms that differ from any of the buses I have already seen.

In addition to providing parameters capable of specifying the basic architecture of each IP bus, I also added a separate set of parameters that map interface functionality to the physical wire names found in each interface standard. This list of parameters is shown in Table 3.4, and can be filled out for each interface instance to generate

Table 3.3: Parameter Mappings for the system buses. These parameters are defined in Table 3.2

Parameter	APB	AHB-Lite	AXI	OPB	IXF
Data					
op_enable	Both	Both	Both	Both	Both
dynamic_sizing	False	True	True	True	True
size_reply	False	False	False	True	True
size_encoding	NA	bus	bus	onehot	bus
sizes	NA	(byte, halfword, word, double, quad)			
mask_enable	True	True	True	False	True
mask_granularity	8	8	8	NA	8
endian	little	little	either	big	little
Address					
slave_select_enable	True	True	False	False	False
shared_rw_channels	True	True	False	True	True
Flow Control and Timing					
flow_control	rdy-val	rdy-op	rdy-val	rdy-val	rdy-valid
shared_ready_repvalid	True	True	False	False	False
slave_stallable	False	False	True	False	True
max_ops_outstanding	1	1	NA	1	NA
address_data_sync	Fixed	Fixed	Variable	Fixed	Fixed
sync_cycles	0	1	NA	0	0
trans_id	False	False	True	False	False
reorder	null	null	rd, wr, rdwr	null	null
Operation					
read_write_type	bit	bit	NA	bit	bit
burst_enabled	False	True	True	True	True
burst_only	NA	False	True	False	False
early_term	NA	True	False	True	True

Continued on next page

Table 3.3 – continued from previous page

Parameter	APB	AHB-Lite	AXI	OPB	IXF
wrap_enable	NA	True	True	False	True
inc_enable	NA	True	True	True	True
fixed_enable	NA	False	True	False	True
length_provided	NA	True	True	False	True
last_provided	NA	False	True	False	True
first_provided	NA	True	False	False	True
master_updates_addr	NA	True	False	True	True
lock	False	False	True	True	True

Table 3.4: A sampling of the keywords used to map interface functionality to interface specific signal names. Note that if the interface has fully independent read and write channels, many of the keywords below must be duplicated to distinguish the read channel signals from the write channel signals.

Keyword	Description
Data	
rddata	Read data bus
wrdata	Write data bus
rdid	Read transaction ID
wrdid	Write transaction ID
id	Transaction ID for buses with shared address
size	Size of data in transaction
mask	Mask for the data bus
Address	
rdaddr	Read address bus
wraddr	Write address bus
Continued on next page	

Table3.4 – continued from previous page

Keyword	Description
addr	Address bus for buses with shared address
rdslvselect	Slave select for read
wrslvselect	Slave select for write
slvselect	Slave select for buses with shared address
Flow Control and Timing	
rdvalid	There is a valid read transaction
wrvalid	There is a valid write transaction
valid	There is a valid transaction on a shared bus
rdrdy	Slave is ready for a read transaction
wrrdy	Slave is ready for a write transaction
rdy	Slave on a shared bus is ready for transaction
transtyp	Bus conveying whether a valid transaction is occurring
repvalid	Slave is transmitting a valid reply.
reprdy	Master is ready for slave reply
Operation	
rdwr	Transaction is read/write
rep	Holds reply to transaction
bsttyp	Type of burst, fixed, wrap, or increment
bstlgnth	Number of transactions in a burst
bstfst	Flag/bus specifying the first transaction of a new burst
bstlst	Flag/bus specifying the last transaction in a burst
hwxfer	Specifies a half word data size
wxfer	Specifies a full word data size
dwxfer	Specifies a double word data size
hwack	Specifies slave width is a half-word
wack	Specifies slave width is a word
dwack	Specifies slave width is a double word

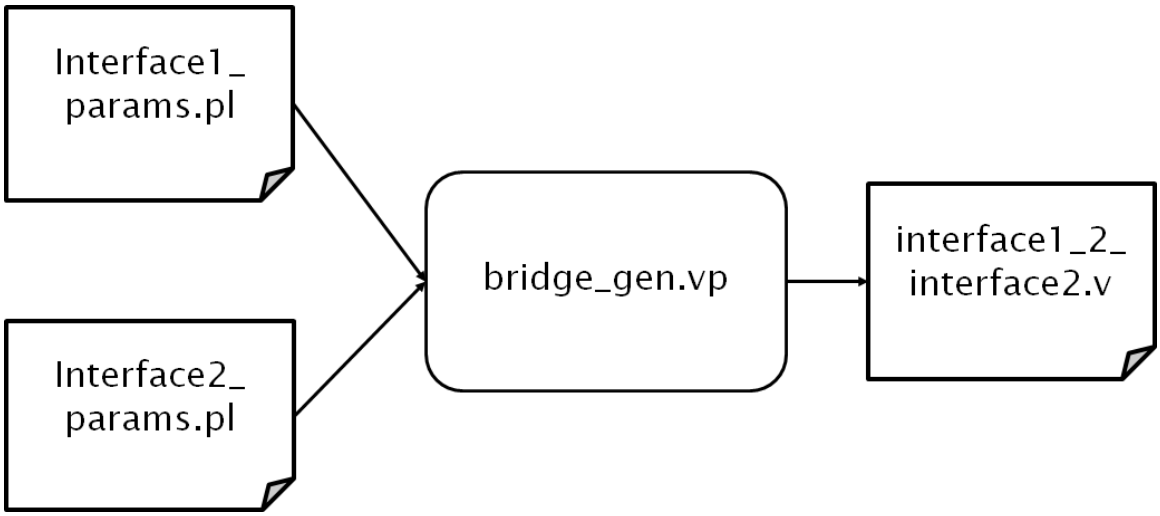

Figure 3.1: High level flow for the interface generator. It takes in two interface descriptions—one master, one slave—and produces RTL for a bridge capable of converting between them.

a pin-compatible interface converter. If not specified, my definition will still faithfully capture the interface functionality, but the RTL wire names for any interface generated by my description may be different from those found in the standard.

3.5 Generating Interface Bridges

Even though each bus feature from my sample of ARM and IBM buses could map to my definition there was no guarantee that the description was complete enough to fully reconstruct the full interface protocols. To test my IP abstraction's ability to encode and interface with existing IP, I used the parameterized bus description to build an IP-interface-to-IP-interface converter generator.

The flow of my converter generator is shown in Figure 3.1. The idea behind this converter is that it would take in two of the descriptions of IP interfaces and, using only the knowledge encoded in the description, would generate RTL capable of translating from one protocol to the other. Such a generator would indicate that the parameterization is sufficient to fully describe the physical signals and high-level protocol advertised by each IP.

I implemented the converter system with a design tool called Genesis2 [43]. High level synthesis languages like Bluespec [35] and Chisel [6] could also have been adapted to build this converter; however, generators are ready made for converting a list of architectural parameters and implementation mechanisms like those found in my bus definition into efficient, domain-specific hardware.

As exemplified by Ofer Shacham's Genesis 2 [43] tool, generators enable the creation of domain-specific hardware generators. With generators, domain experts codify all of the design decisions that they would make in developing a hardware instance into a set of high-level architectural parameters. They then use a tool like Genesis 2 to create a hardware template capable of directly parsing these architectural parameters and creating RTL for a fixed hardware instance. Domain experts are able to place limits on the values that users can select for each parameter to help ensure that the generated hardware instances are efficient. Researchers have already used these tools to create a floating point mathematical unit generator capable of generating highly-efficient hardware implementations across a range of area, energy, and performance targets [18].

Genesis 2 hardware generator templates are composed Perl interleaved with the designer's RTL of choice. The template developer uses Perl to describe how the design should be elaborated, e.g. how many of which instances to create and which algorithmic RTL implementation to include, while all of the underlying hardware for each elaboration choice is specified in RTL. During the elaboration, or generation phase of compilation, Genesis 2 parses out the Perl code to construct a final, fully specified RTL module. The tool elaborates the design hierarchically, meaning that Perl elaboration code can be written to take into account the module's position in a design and adjust its parameters based off of values set for its child and parent blocks. A full description of the Genesis 2 language and design principles can be found in the doctoral thesis of Ofer Shacham [44].

Data structures containing all of the parameters from my interface definition were used to encode each bus description consumed by my interface generator. The parameters in the data structure are identical to the parameters enumerated in Table 3.2. For my generator, I created data structures that define AMBA's APB, AHB-Lite,

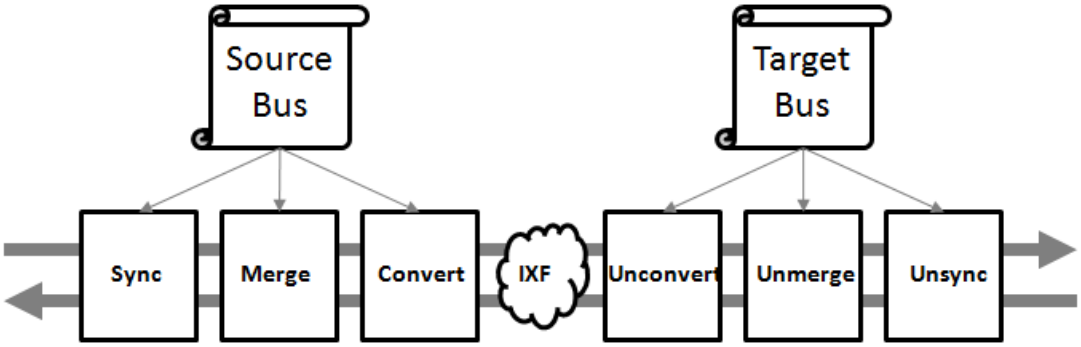

Figure 3.2: High level architecture of the prototype bridge generator. The source and target interface objects are used to specify the functionality of all six blocks.

and AXI standards, and IBM's OPB.

The architecture of the interface-to-interface translator is shown in Figure 3.2. At a high level, the translator operates by converting both input bus definitions into a common interchange format (IXF), and then connecting the buses through this intermediary interface. The detailed interface definition for the IXF block is available in Table 3.3. The interface generator architecture is conceptually similar to the Universal Bridge proposed by Cho et al. [12], except instead of using a microcontroller to handle all aspects of protocol and encoding conversion, I break the bus conversion into three distinct steps and generate custom logic for all control.

My generator separates the master-to-IXF conversion into three architectural steps: *sync*, *merge*, and *convert*. Internally, the stages communicate in a latency insensitive manner, using the master's handshake format to determine when the next stage can accept new data. The convert block handles the actual handshake conversion, and the IXF communicates using a ready-valid protocol.

The sync stage is responsible for converting the input interface's signal synchronization into the synchronization used by the IXF format—all parts of a transaction are synchronized to the same cycle. The sync stage accomplishes this task through the use of a set of FIFOs for each input signal. Based on the interface description, the synchronization stage is configured to determine the basic operation type taking place, generally a read or a write, and determine which signals are necessary to complete a transaction of this type. Once all of the signals required for a transaction are present, and the slave side of the interface indicates that it is ready for data, the sync

stage sends the transaction along.

Since different protocols use different combinations of handshakes and synchronization to indicate when data is ready, the sync stage has a built in controller that interprets the handshake protocol and manages the FIFOs. For example, in the AHB-Lite protocol where address information is sent the cycle before data, the controller will capture the address signals when it receives a valid transaction, and, if the operation is a write, will capture the associated write signals on the next cycle. The controller is also designed to optimize for latency, and, when possible, will bypass buffers and retransmit data on the same cycle it is received.

The *merge* block takes buses that, like AXI, have separate read and write addresses, and merges their transactions onto a single shared address bus. The merge block arbitrates between requests on the two input channels to serialize the bus's operations. By default, the arbiter uses a round-robin scheme; since this is a generator, however, it is a simple matter to implement other priority schemes.

On the return path, the merge unit keeps track of the outstanding bus transaction types. When a response comes back through IXF, the merge unit uses this record to route the response to the appropriate interface channel (read or write). The number of transactions that the merge unit keeps track of is determined by the *max_ ops_ outstd* parameter.

Finally, the *convert* stage of the generator implements the logic necessary to convert control, handshake, and other signals from the way they are specified in the input bus into the format expected by IXF. For signals that exist in IXF but not in the master interface—data mask, for example—this stage maps them to a logical default value—data mask is hard-coded to all 1's. This also handles all of the handshake conversion work. While some of this is a simple combinational mapping of different signal types, other conversions involve limited synchronization. For example, if the master protocol has *shared_ rdy_ repvalid* enabled, where ready and reply valid are represented by the same signal, the convert block tracks outstanding operations and valid responses to ensure that each op replies valid at the appropriate time.

The IFX-to-slave conversion operates much the same way. The *unconvert* block reformats the information transmitted by IXF into the types of information specified

in the output bus. Most of its functionality is analogous to what's found in the convert block. There is also additional buffering to hold a slave's reply if the slave cannot be stalled directly.

For buses like AXI, *unmerge* separates operations on the single shared read-write channel onto dedicated read and dedicated write channels. While conceptually this should just be a demultiplexer, since the IXF bus only has a single ready signal, there is no way for the unmerge unit to advertise which type of operation the AXI bus is ready for. The unmerge unit solves this with an input buffer that allows it to accept and store a valid transaction of either read or write if the slave is not yet ready for that type of operation.

Finally, *unsync* converts from the synchronization format offered by IXF into the format required by the output bus. The unsync unit uses FIFOs to capture valid transactions from IXF and release various signals at the protocol-determined timing interval.

3.6 Validating the Abstraction

I tested my bridge generator by feeding different combinations of the model objects for APB, AHB-Lite, and AXI into the bridge generator. I then tested each of the resulting RTL-level protocol converters. To test the converters, I obtained RTL for peripheral memory blocks that advertised compatibility with one of the four mapped standard system interfaces [37, 39, 38][1], and issued read and write operations across the converter. For AXI, I was able to obtain ARM-provided SystemVerilog assertions designed to test the protocol [3]. While the validation suite is not of production quality, the bridge generators performed correctly in simulation, indicating that my proposed parameterized interface specification can map both the physical signals and high-level protocols of IP interfaces.

In practice, my bridge generator has some performance limitations. The design choice to first convert to a fixed intermediate standard, IXF, before converting to

[1] Minor alterations were made to integrate these modules into my SystemVerilog-based test environment.

the final interface format leads to inefficiencies and extra logic in cases where the master and slave buses have characteristics that are different from IXF. An AXI-to-AXI bridge, for example, loses about half of AXI's theoretical peak performance due to the need to merge reads and writes onto IXF's single communication channel. A more efficient implementation is likely possible if I directly converted from input to output interface formats.

As a more promising alternative, however, I could create a generator that converts my bus descriptions into finite state machine representations like those proposed by Avnit et al. and discussed in Chapter 2. This would allow bridges generated from my definition all of the formalism advantages of FSMs and would allow me to leverage the synthesis work already completed for FSM structures. My bridge generator was only constructed to help me test the completeness of my interface abstraction, however, and the current limited architecture accomplishes that task, so I leave new and improved implementations to future work.

3.7 Extending the Generator

As we mentioned, buses evolve over time, so it is criticial that the generator can evolve as well. This raises two questions: how hard is it to modify the bridge generator to support the expanded definition, and how hard is it to maintain backwards compatibility with older interfaces? Since a major motivation for this research is to ease the integration of existing blocks into an SoC: it is essential for the generator to connect older IP blocks to newer interface standards.

In some cases, maintaining backwards compatibility can be relatively straightforward. If a user comes across a bus that has a new implementation mechanism for a feature that is already handled by the generator—a new handshake protocol for example—they can go through the generator code and modify any areas that handle the affected parameters to support the new definition. As part of this task, they would teach the generator how the hardware specified by the new mechanism translates into the older mechanisms, allowing the generator to map old and new interfaces together.

When a user adds a new high-level feature to the interface definition, however, backwards compatibility becomes trickier to achieve. The problem is that not all new features have a close analog to features found in existing buses. For example, when mapping IBM's OPB bus into the parameter set, I was confronted with the fact that OPB peripherals can request rearbitration through the *sln_ retry* signal if they cannot complete the request in time. The AMBA buses I had already mapped, however, do not support peripheral-initiated re-arbitration. If the OPB bus is used as a master, there is no major backwards compatibility issue: AMBA peripherals are incapable of requesting rearbitration. By tying the OPB master's rearbitrate signal to ground, I am able to ensure backwards compatibility with AMBA peripherals.

If an AMBA bus were updated in a future revision to support a peripheral rearbitration request, compatibility would also not be an issue. In this case I could simply map OPB's implementation of rearbitrate to the new AMBA bus's, and the system would work properly.

For the case where a current AMBA standard with no concept of rearbitration is the master, however, there is no simple mapping that will suffice. If I allow the AMBA master to ignore the re-arbitration request, the interconnect may stall waiting on a response that will never come. On the other hand, if I map the re-arbitrate to an AMBA bus error, which seems like the only mechanism AMBA has for handling a peripheral that is unable to complete a request, and if the master is not programmed to know about errors caused by re-arbitration, it may simply give up on a request, rather than try again later. This could affect overall system performance, and make use of certain OPB peripherals unreliable. Fortunately, the AMBA master's driver software could be modified to handle the re-arbitration error in an appropriate way. Therefore, I chose to map the re-arbitrate signal to an AMBA error response.

As this example indicates, backwards compatibility for new types of transactions can sometimes be maintained by carefully choosing a default mapping of new features to older buses and modifying the software driver controlling the interface masters. Unfortunately, the framework presented in this chapter is only concerned with mapping the low-level signals and protocols between multiple interface standards and has no mechanism for changing the software used to power the devices.

If there was an automated mechanism that handled hardware integration and driver generation, however, it could make achieving backwards compatibility between old and new buses much simpler for the system integrator. In Chapter 5, I discuss the creation of a software driver generator that uses information about the low-level IP interface to build a custom driver. While such a driver generator could be extended to incorporate interconnect information to handle these sorts of interface mismatch problems, implementing and testing that feature is left for future work.

3.8 Summary

Integrating IP blocks into a system is much harder than it could be. While standardized buses were supposed to address this issue, the evolution and proliferation of different standards means every SoC design is likely to incorporate IP blocks that use multiple different interface protocols. While much work has been completed to try to automate the IP integration process, current techniques could benefit from a simple way of specifying existing IP interfaces at a higher level of abstraction.

To address this issue, I created a flexible, IP-specific interface abstraction. My system is designed to be extensible so as SoC interconnects evolve over time, the generator and interface description can be expanded to connect older IP blocks to these new interfaces. To test my interface abstraction, I constructed a hardware generator capable of creating bridge logic between any two IP buses that can be described in my description. I then used this system successfully to encode and translate among a number of different IP interface standards, including AXI, AHB-Lite, APB, indicating that my description has sufficient flexibility to represent commercial IP block standards.

Chapter 4

Advertising Native Interfaces

While the interface definition proposed in Chapter 3 provides a mechanism for integrating existing IP blocks that already advertise fixed bus standards, this mechanism is not always the most efficient way to link IP blocks into designs. In fact, translating between interfaces in a system can add excess logic and hurt overall bandwidth.

Any time we have to bridge an IP block that offers one interface to a system interconnect that offers another, we are essentially instantiating two interface translations in our design. First, the IP designer had to translate from the communication expected by the internals of the IP block into the communication protocol specified by the advertised bus standard. Second, my system, or any other bridge mechanism, converts from the advertised bus standard into the interconnect standard. Depending on the protocol and synchronization changes required by each of the steps, the double conversion process may introduce throughput bottlenecks into the system-to-IP communication. The double conversion also puts us into a position where the logic in the IP interface-to-interconnect conversion step may be primarily designed to undo some of the translations that occur in the IP-to-IP interface conversion step, adding unnecessary logic and complexity into the design.

In older design methodologies, the convenience for system designers of having a single standard bus per IP block meant that these inefficiencies were often worth the cost. With high-level synthesis design methodologies, the interface information can be communicated at a higher-level, allowing IPs to be automatically integrated

without advertising a fixed physical standard.

High-level synthesis tools can encode information about the different data structures that the IP would like to receive and pass directly, and automatically synthesize the necessary IP interface hardware. High-level synthesis languages enable this by requiring both IP designers and system integrators to deal with information flow at a high-level. In Bluespec, for example, designers can declare interface classes and define access methods for each piece of information they wish to pass to an IP block. This sort of interface synthesis is also known as transaction level modeling (TLM), since it allows IP and system designers to only focus on the higher-level, transactional data-flow between modules without needing to worry about implementation details.

When the designer uses TLM, the synthesis program can use information provided by the interface class to determine the required timing and flow-control mechanisms for communicating data between the IP blocks and the rest of the system. The synthesis program then uses this information, and its knowledge about how all of the different peripherals in the system are related and interconnected to automatically generate low-level interconnect hardware. While the generated hardware may constitute a custom interconnect network, some researchers have proposed mechanisms for using high-level synthesis to map these IP interfaces into existing system bus interface standards [12].

There are still many design cases that cannot benefit from TLM, however. First, if a system designer has a large amount of legacy hardware collateral and RTL they would like to use, the designer may not have the resources to rewrite it in a TLM friendly manner. In this case, the designer needs a way to map their legacy modules into a TLM-like flow.

Even if a designer is not tied to legacy RTL, an HLS-based flow may not be the best for all of the components the designer would like to integrate into the design. For example, for certain high-performance, high-efficiency accelerators, an IP designer may find that his or her high-level synthesis tool does not encode all of the knowledge required to synthesize an efficient IP block, and decide to implement the IP in RTL instead. There are many benefits to a system integrator, however, to relying on an HLS tool to design the system interconnect; not the least of these being the ability of

an HLS tool to pull in required bandwidth information from each of the blocks and provision the interconnect network accordingly. Therefore, it would be ideal if these RTL-based blocks could be included in the HLS interconnect design flow.

Any mechanism designed to map an RTL block to a TLM-like interface must provide two things. First, it must encode the same set of flow control information that a high-level interface language would normally extract from a block, such as flow control and timing information. This is required so that the high-level synthesis tool knows what sort of interconnect hardware it must generate in order to properly communicate with the block. Second, the mapping mechanism must provide predefined RTL hardware access points that the high-level synthesis environment can map the generated interconnect into. In other words, it would need to have a model for the sorts of hardware that it will be connected into.

In this chapter, I discuss a system I have built for mapping high-level synthesis interfaces into standard RTL. In Section 4.1, I briefly discuss the image signal processor generator system that provided my main inspiration and test case for pursuing this work. In Section 4.2, I discuss the types of RTL hardware I expect to find in the interface of an IP block, which I need to map into to successfully integrate HLS-style system integration with existing designs. I then discuss the types of flow control and timing information I need to know for each interface element in order to automatically generate the low-level interface hardware in Section 4.2.1. Finally, I discuss my implementation of this high-level interface system in the Genesis 2 design language.

4.1 Image Signal Processor Generator

To make this discussion more concrete, this chapter will use the image signal processor generator (ISPGen) created by Brunhaver [10] as the IP generator example to demonstrate the issues that need to be addressed and the mechanisms used to address them. The key concept behind ISPGen is that almost all image processing tasks can be put into the form of a *stencil-based* computation which can be calculated with high energy efficiency. The ISPGen creates efficient compute engines for the specific stencil program it is given.

Stencil based computation describes any types of operation where a set of map and map-reduce mathematical operations are applied to a 2-D array, or *stencil* of matrix operands. This operation is then iterated over every matrix element, with the result of each map-reduce operation stored in a separate output matrix. This class of operations includes basic 1-D and 2-D linear convolutions, and weighted-averaging techniques among other mathematical operations. These operations are highly parallel and have extreme locality by nature, feature large operating sets and low precision operations, and they can be cascaded together into extremely efficient hardware implementations.

Stencil operations have a broad application to the domain of image processing. Operations such as applying a Gaussian blur or a sharpness filter to an image can be modeled as stencil operations, as can most steps in a basic photographic pipeline. The basic image pipeline in any camera requires tens of different stencil computations to complete. To create IP blocks capable of handling these operations, the image signal processor generator allows users to specify their desired algorithm in an assembly-like programming language called Data Path Description Assembly, or *DPDA*. A special DPDA compiler analyzes the user's algorithm, creates custom stencil hardware to perform each of the stencil operations, and creates a custom IP block that cascades stencil operations together.

Depending on the algorithm specified, the generated IP block will feature a wide range of interface resources. For each stencil operation, there will generally be a set of registers designed to hold the map coefficients for the map-reduce operation. Depending on the operation, the IP block may also feature items like interface memories, for example to hold look-up values to be used during the computation. When specifying the DPDA, the user may also request that certain statistics be exported from the IP block. For example, a user can specify that a histogram of pixel brightness be created.

Each of these interface resources may have flow control restrictions on when they can be updated and read. This is a critical issue for control registers in particular, which should generally not be changed partway through processing an image. These constraints then need to be converted into access-control hardware on the IP interface.

In addition to hardware timing issues, these blocks require drivers to interact with the IP interface from software.

ISPGen is in part being developed to offer hardware acceleration for experimental image processing algorithms on the next generation Frankencamera [1]. For this platform, we want to enable computational photography and computer vision researchers with no background in hardware design or system architecture to write an algorithm in software, and have it automatically implemented on the Frankencamera's FPGA so that the algorithm can process a live image stream in real time. If the final system requires researchers to write IP specific drivers, however, then researchers will still need to know about the underlying hardware mechanisms in order to use the system. Manual driver development would also slow down the rate at which new algorithms could be prototyped and integrated into a test system.

The wide range of interfaces and interface requirements found in the ISP generator made it an ideal target for my research into interface synthesis and driver software generation. As configurable data engines, each ISPGen instance advertises a wide variety of interface elements ranging from flow-control heavy programmable registers, to high-bandwidth ports designed to stream in image data as quickly as possible. The number of interface elements found in these blocks can also vary widely, ranging from a few registers used as filter taps for a basic filtering IP to hundreds of registers and memory structures used to control an entire photographic pipeline. Finally, the fact that this generator may be used by programmers with limited hardware knowledge means that any solutions I implement to generate interface hardware and software must be completely automated and transparent to the user; e.g. there's no way for a solution to "cheat" and require manual user intervention for synthesis.

4.2 Mapping HLS to RTL

In order to use RTL-based modules in the context of a high-level synthesis flow, we need a set of well defined interface hardware for the high-level tool to map into. In Chapter 3, I addressed this problem by limiting my work to cover RTL modules that

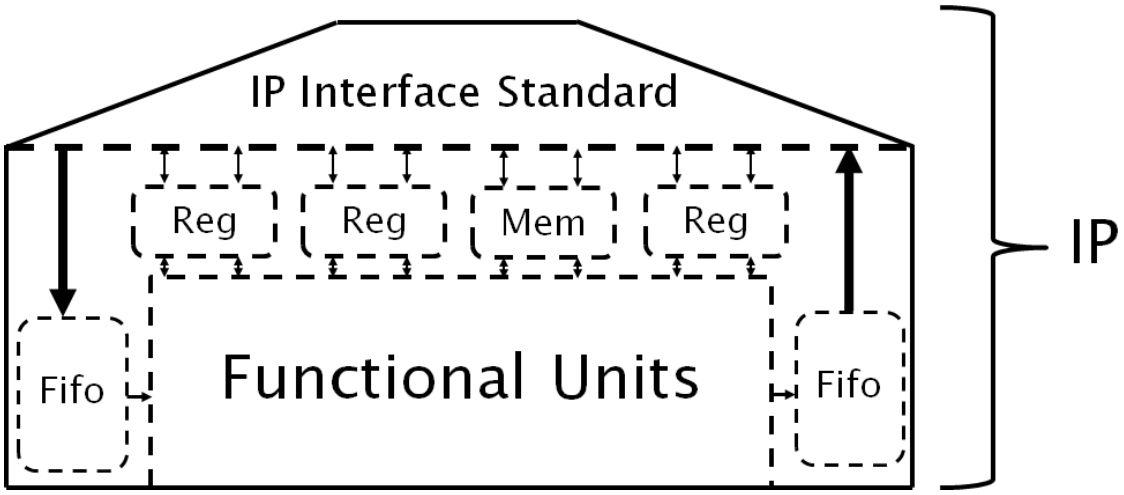

Figure 4.1: Block diagram of an RTL IP architecture, illustrating the functional units, the intrinsic interface, and standard bus interface. In this chapter, I attempt to eliminate the bus interface segment and bring the high-level interface flow to the level of the intrinsic interface.

advertise an instance of an IP interface. This ensured that there were always well-defined address, data, and flow-control mechanisms. As already discussed, however, the reliance on fixed interface standards can lead to interconnect inefficiencies and bandwidth limitations if there are conflicts between how the IP interface and system interconnect expect to communicate.

Rather than rely on a fixed interface standard, the goal of this part of the work was to bring the high-level synthesis tool directly to the internal interface of the IP block. This meant defining what an internal, or *intrinsic* IP interface looks like. For the purposes of my research, I view the internal IP interface as a number of data storage elements that exist at the periphery of the hardware's functional units. This setup is shown in Figure 4.1.

At the lowest level, an IP block may feature resources that correspond to simply bits and buses. Since these resources require no additional flow or access control, all a high-level synthesis tool needs to do to map to these is to create a single set of wires, making this the simplest type of resource that a high-level synthesis flow can be mapped into. I refer to this class of intrinsic IP interface resources as a *message*.

As a special case of message, the IP blocks that I have encountered often feature a

number of one-bit signals used for control and synchronization. These signals include things like clock, reset, and idle. Control signals may either be active high or low (or trigger on posedge or negedge) as there is no single convention that all designers follow. I capture these signals with the control port or *cport* primitive. The only difference between a cport and a message is that a cport is assumed to be associated with additional encoding information: e.g. is it *active_low* or *active_high*. This information is required for extending the high-level synthesis tool to the IP block since, at generation time, the HLS tool will need to know how IP control signals map to the same signals coming from the system scope.

The next object, or interface primitive, is the *buffered* type, which represents simple buffering memory elements like registers, queues, and FIFOs. Much of the hardware I have run into uses control registers to hold configuration values at the interface, and uses FIFO-like elements to quickly stream in data, therefore, this is arguably one of the most common types of hardware that HLS needs to be mapped into. Fortunately, these elements all have very similar physical interfaces: they likely have a physical port for writing in a new word, a port for reading out the current word, and an enable (or push) signal that controls when they can be written. The resource may also have a full, busy, or similar signal indicating that they can temporarily not be written. Therefore, the HLS interface generator must be able to map all of these signals in order to properly interface with buffered type interface elements.

Finally, on the intrinsic interfaces I have encountered, I have also run into a number of *addressable* memory elements, such as register banks, SRAMs, and lookup tables. Basic addressable memory structures also tend to share similar physical interfaces as well. All of these elements contain physical address buses for accessing a specific data location, buses for taking in write data and driving out read data, and either a bus for specifying the operation type (read or write), or specific ports for a given access type. The HLS tools must be taught how to map into this interface as well in order to support interface generation for RTL blocks. As a side-note, some memories that may exist on an interface are multi-ported to handle multiple memory requests, and each port generally has its own address, data and flow controller, meaning that it can be encoded by modeling the multi-ported block as multiple single-ported memories.

Therefore, for now, I have not worried about explicitly mapping HLS tools into multi-ported blocks.

To bring an HLS type interface generation flow into the IP intrinsic interface, the HLS hardware must be able to connect the low-level interfaces that are likely to be advertised by the IP block. These four distinct interface types cover most of the mechanisms that will be found on an intrinsic interface: message, cport, buffered, and addressable. This list of primitives, the type of hardware each primitive is capable of mapping to, and their low-level physical interface are summarized in Table 4.1.

Table 4.1: Summary of primitives used to map TLM style interfaces into Genesis 2 generator designs.

Type	**Sample Hardware**	**Interface Signals**
Message	signal bus	data
Cport	1-bit control signals	data
Buffered	register, FIFO, queue	rd_data, wr_data, enable, clock, reset, full/busy
Addressable	memory, look-up table	address, rd_data, wr_data, enable, clock, reset, full/busy

4.2.1 Specifying Flow Control and Access

Simply giving the RTL's hardware connection points to the high-level synthesis tool is not enough to automatically generate an interface between the IP block and the rest of the system. For example, if an IP designer is creating a hardware accelerator to apply a uniform Gaussian blur across an entire image, the designer may configure the block to read the filter weighting coefficients from control registers. Once integrated into a system, the interconnect will need to write these control registers in order to set up each new Gaussian blur. If these values are changed in the middle of processing a frame, however, the final image may exhibit tearing from where the old values were replaced by the new. Likewise, if an IP block advertises a set of statistics registers that store data about the current run, the IP designer may wish to prevent the processor from writing to it. Given the way I have defined the intrinsic interface for the IP

block, in each of these cases, it would be up to the high-level interface generator to implement the low-level hardware or software mechanisms necessary to prevent these kinds of illegal accesses from happening.

In a conventional HLS flow, creating these flow restrictions is possible since the tool has created the hardware it is interfacing, and is "aware" of the required access patterns. In our case, however, we must first develop a method for communicating the flow control information to the high-level synthesis tool.

For a mechanism to effectively communicate RTL flow control requirements to a high-level synthesis tool, it must allow flow control information to be specified on a per IP resource, or per interface primitive basis. Different sets of control registers and interface hardware may have different requirements, and the RTL-to-HLS mapping mechanism must be flexible enough to support this.

Beyond that, to create an RTL-to-HLS mapping mechanism, it is necessary to define what sorts of flow-control options an IP interface can request. For this work, I used my experience with ISPGen to try to develop a comprehensive set of access control information to share with the HLS tool.

The first flow control parameter that the user can set is the direction of the primitive. This can be set to "input," "output," or "both" depending on whether the interface resource is an input, output, or a bi-directional element from the perspective of the IP block. During generation, the primitives use this distinction to determine which input/output (I/O) signals to generate. For example, if a message primitive is set to "input," the generator will create a unidirectional signal that only allows data to be transmitted from the interconnect to the message resource. Since the processor cannot read this resource, a return data path is omitted. Since these primitives are used to generate the connection with the interconnect network, it would also be possible to implement hardware checks that send an error response to the bus if an illegal read or write is attempted. We leave this as an exercise for future work, however. Regardless of the specified direction, the interface generator can still provide a full two-way interface to support hardware testing.

The buffered and addressable primitive types also have the ability to take in a designer specified "blocking signal." When active, this signal blocks the interconnect

from writing to the data structures. To use this feature, the user provides a cport type object to be used as the blocking signal. The register and memory primitive use the cport's active high/low parameter to determine when writes should be blocked.

This signal is distinct from any sort of "full" signal that might be found on a queue or FIFO, and is used to tell the interconnect generator if there are conditions relating to the IP's state that prevent the resource from being accessed. The reason for this distinction is largely one of optimization. If a resource is only temporarily blocked because of a self-correcting issue—e.g. the input FIFO is full because the IP is not running fast enough internally to clear it—it may make sense to leave any requests to write to the FIFO on the interface so that the value can be written as soon as the FIFO frees up. In fact, most fixed interface standards explicitly support handling these sorts of temporary stoppages through some sort of internal "ready-valid" flow control mechanisms.

On the other hand, if an IP resource is going to be frozen for an extended period of time—for example, if a user tries to reprogram a control register that cannot be overwritten while the IP is processing an image—it may make more sense to send the master a "resource busy" bus error to the bus master and discard the transaction so as to not lock up the interface resources.

Finally, the *buffered* and *addressable* primitive types may also need to pass some performance information to the HLS system in order to ensure that it properly provisions them with network resources.

One important consideration when integrating an IP block into a system is whether it can be fed fast enough to make full use of the block's computational resources. For especially data hungry units, the interconnect generator may need to provision extra data links to a particular unit, or ensure that an IP block can be directly written by a DMA engine to ensure a consistent high-bandwidth transfer of data. My system communicates these needs by providing two pieces of information for each primitive: whether the interface resource is "streaming," and, if so, what its required streaming bandwidth is. Using this information, the generator can appropriately generate the interconnect for these units.

As a further optimization, the RTL-to-HLS interface generator may also need to

know about shadow buffers. Shadow buffers are groups of registers that sit on the path to the control register. They can be written at any time, and their output is multiplexed together so the user can select which buffer gets written to the actual control register. Shadow buffers allow the user to load successive configurations to the IP block in advance and quickly switch between them, helping to mask any interconnect congestion or latency issues that might be present. This resource exists solely as an optimization parameter. Therefore, in an ideal world, the high-level interface synthesis tool would be able to analyze latency issues and implement shadow buffers automatically if it would help system performance. Since this is an optimization parameter, however, and since a tool may not always implement shadow buffers automatically or generate the correct number, I leave this as a parameter that a system designer can optionally set.

The full list of flow control and optimization options is shown in Table 4.2. While the flow control parameters implemented here are limited to basic functionality, they could easily be extended to encode higher level protocol information about how the IP block expects to interact with the rest of the system. For example, if an IP block has a list of "illegal" or "unimplemented" values that should never be written to a control register, the buffered and addressable primitives could be extended to store this information. During generation time, the high-level interface synthesis tool could then use this protocol information to issue errors if an illegal value is ever sent. So long as such changes do not affect the primitive's interface to the IP or interconnect, generator designers are free to tweak and customize the design primitives. If the modifications do change the interface, however, it is up to the designer to modify the interconnect generator portion of the design to handle the new functionality.

4.3 Building an HLS-to-RTL System

Based off of the hardware mappings discussed in Section 4.2, and the information about necessary control flow information in Section 4.2.1, I built a system capable of advertising high-level interfaces on RTL.

Table 4.2: Summary flow control and optimization information passed to the high-level interface synthesis tool. Note that "streaming," and "bandwidth" options are only available for *buffered* and *addressable* types, while the "shadow buffers" option is only available for *buffered* types.

Feature	**Allowed Values**	**Description**
Direction	"input", "output", "both"	Specifies whether the interface element is an input, output, or both to the IP block
Blocking	cport	Uses the provided cport to block access to the element.
Streaming	True, False	Specifies whether element should be streamed to
Bandwidth	integer	Specifies required bandwidth for full performance (MB/s)
Shadow Buffers	integer	Specifies numbers of shadow buffers

Since my target application, ISPGen was already constructed in the Genesis 2 design language, I created my system in Genesis 2 as well. In Genesis 2, I made a software object for each of the hardware interface primitives summarized in Table 4.1. Each of these software objects not only contains information about the low-level hardware interface that it must map to, but also have methods for setting all of the flow-control and optimization options summarized in Table 4.2.

To map these primitives into an actual RTL design, the IP designer instantiates one primitive of the appropriate type for each IP resource they would like to advertise. On instantiation, the user provides each of these objects with basic information about the IP resource it represents, including the name, and width. Using built-in methods, the user can also set the more advanced flow-control information used by each primitive to generate the appropriate interconnect hardware. These steps are illustrated in Figure 4.2.

On instantiation, the object internally creates a unique set of Verilog signal names for the data, address, and control signals of the IP interface resource it represents. To connect the interface hardware to each primitive, the IP designer uses "assign" statements to attach their internal Verilog signals to the signals advertised by the interface object. The designer can get the interface object's basic Verilog signal name using the object's built-in `m2v` function call. The signal name returned by this function

```
//;# For each piece of the interface, create a primitive
//;# First define some of the primitive's basic parameters
//; my $direction = 'both';
//; my $width = 5;
//;
//;# Then instantiate the object for the primitive.
//; my $msg = new buffered('ifc_resource_name', $direction,
//;      $width);
//; $msg->set_acc_name('isp_instance_1');
//; $msg->set_blocking($idle);
//; $msg->set_streaming(False);
//;
//;# Create a Genesis 2 parameter to hold all of the
//;# primitives, so that they will be available at other
//;# levels of the design hierarchy.
//; my $ifc = parameter(Name => 'interface', Val =>[$msg],
//;       Doc => 'Array of objects to define interface');
```

Figure 4.2: Code used to map an interface resource in an IP design to a *buffered* primitive.

call corresponds to the base Verilog name that my HLS-to-RTL system will use to construct the names for each signal advertised by the primitive. The IP designer can "construct" the other Verilog signals for each resource by appending the signal name suffixes listed in Table 4.3. This mapping process is illustrated in Figure 4.3.

Table 4.3: Summary of suffixes that must be appended to an object's `m2v` provided signal name to "construct" the other signal names advertised by the primitive.

Signal	Suffix
rd_data	_rd
wr_data	_wr
address	_addr
enable	_en
full	_full

While the process of manually mapping the object's Verilog signals to the IP's internal signals can be tedious, my interface objects also contain macros for quickly

```
// Assign the interface element signals to the primitive
//; my $base name = $msg->m2v();

//Assign the signals that originate outside the IP
assign elem1_data_in = `$base_name`_rd;
assign elem1_write_enable = `$base_name`_en;

//Assign the IP signals to the output
assign `$base_name`_wr = elem1_data_out;
```

Figure 4.3: Code used to map an interface bus in an IP design to a *buffered* primitive.

```
module myIP( `msg1->genAcc()`,
             `msg2->genAcc()`,
             ....
           );
```

Figure 4.4: Code used to generate the Verilog module instantiation for an IP block that relies on my primitives for interface synthesis.

defining all of the object's input and output Verilog signals in the Verilog module's header. To do this, the user simply invokes each primitive's `genAcc` method. A code sample illustrating this step is provided in Figure 4.4 When instantiating an instance of each of these modules, the user can pull out the list of messages from the Genesis module object and use each message's `gen_hw` method to automatically create an I/O list for the instantiated instance that is compatible with the signal names created by the interface-to-interconnect generator discussed in Section 4.4.

Once the user has mapped all the interface signals into my primitives, he or she then feeds all of their primitives into another Genesis 2 object I developed called "system_connector". This object, discussed in greater detail in Section 4.4 forms the basis for implementing my HLS to interface primitive objects mapping.

4.4 Interconnect Generator

With Genesis 2 objects exporting fixed physical interfaces and high-level control information for each interface primitive, the next step in realizing an HLS-to-IP intrinsic interface system was mapping a high-level language to the objects I created. Since my

objects were already in Genesis 2, and since the ISPGen, my test platform, was also in the Genesis 2 design flow, I chose to use Genesis 2 as the high level language for my implementation. Existing HLS tools like Bluespec and SystemC could be mapped to my objects as well, however, this is left for future work.

My Genesis 2 HLS mapping solution is based around a Genesis2 object I created called the *system_ connector*. This object gathers together all of the primitive objects in an IP block, and generates the RTL required to map the IP's interface primitives into a given interface standard. The object also governs the generation of hardware required to enforce the flow control constraints on each interface primitive.

For the sake of limiting implementation complexity, the *system_ connector* object has been created to map the hardware primitives into any number of pre-defined interface standards. While ideally my object would make use of the interface abstraction described in Chapter 3 to map the primitives into a wide variety of interconnect networks, my HLS-to-RTL interface system was actually completed before I developed my interface abstraction.

The use of my *system_ connector* primitive is illustrated in Figure 4.5. The object is designed to be used at the level of the design hierarchy where the IP block it is connecting is instantiated. To instantiate the interface connection hardware created by the *system_ connector* object, the user invokes the connector's `gen_hw` function call.

Internally, my *system_ connector* object is designed to use Genesis 2 generators to convert between the primitive objects and the interconnect network. This is done to make my tool more easily extensible to more interconnect standards. The template is responsible for handling address decoding for the IP resources, and connecting the IP buses to the read and write data stream. The template must also convert the flow-control and control mechanisms used by the system-interconnect into the format required by the IP primitives. This last stage may require special hardware to handle synchronization and other handshaking issues.

When generating, the *system_ connector* object assigns each interface resource an address space starting from a user defined offset address offset. In general, each *cport* and *buffered* type is granted one 32 bit address word. If the register has shadow

```
//;# Create a new system connector
//; my $sys conn = new system connector('IPIF');

//;# Add in the primitive objects pulled from the IP instance
//; $sys conn->add msg($ip1 msg1);
//; $sys conn->add msg{$ip1 msg2);

//Instantiate the conversion hardware
`$sys conn->gen acc()`

//;# Create geneis module instance and pull out interface objects
//; my $ipInst = generate('myIP', 'myIP1');
//; my $ipInstIfc = ipInst->get param('interface');

//;# Build the I/O list
//; my $ipInstIo = [];
//; foreach my $msg (@{ipInstIfc}){
//;     push(@{$ipInstIo}, $msg->gen hw());
//; }

//Instantiate the IP and connect it to the interconnect
`$ipInst->instantiate()`(
    `join(',', @{$ipInstIo});
);
```

Figure 4.5: Code used to automatically map an interconnect standard (in this case, Xilinx's IPIF) to my primitives.

buffers enabled, however, it is granted one address per shadow buffer. *Addressable* memory types are also granted an address space proportional to their capacity.

Once the address spaces have been assigned, the template creates a combinational decoder that maps and translates the handshake to each protocol. For a *buffered* type, this decoder activates the primitive's "enable" signal whenever the bus issues a write to that address. For *addressable* type, the decoder will send the write operation to the primitive whenever a valid write is registered to that primitive's address space. For *cports* and *messages*, the decoder output is used to mask the data signal to these elements unless they are the target of a transaction. The mapper also computes the correct local address for indexing elements within the addressable primitive based off of the global interconnect address.

For reads, my template multiplexes the response signals from the primitives, and uses the address as a select signal. The mapper also automatically acknowledges all operation requests on the cycle after it receives each transaction.

Finally, the template calls the built-in `generate` function on each interface primitive, which triggers the primitives to generate all of the hardware required to implement the user defined flow control and optimization requirements. The primitives enforce the block signal by inverting it and "anding" it with the enable signal. If busy is high, the enable passed into the IP interface resource will be low, preventing new values from being written. Finally, my buffered primitives automatically generate the hardware required to implement any shadow buffers specified by the IP designer. My primitives instantiate the appropriate number of registers and a multiplexer to select among them.

The overall flow of my interconnect generator is shown in Figure 4.6.

4.5 Testing and Summary

I integrated this system into the ISP generator to automatically integrate ISPGen IPs into the Xilinx Zynq development platform Using my system, we were able to automatically integrate and prototype accelerators for FAST, Canny, Harris, Stereo, Lucas Kanada Optical Flow, SLIC super pixel segmentation, and camera pipeline

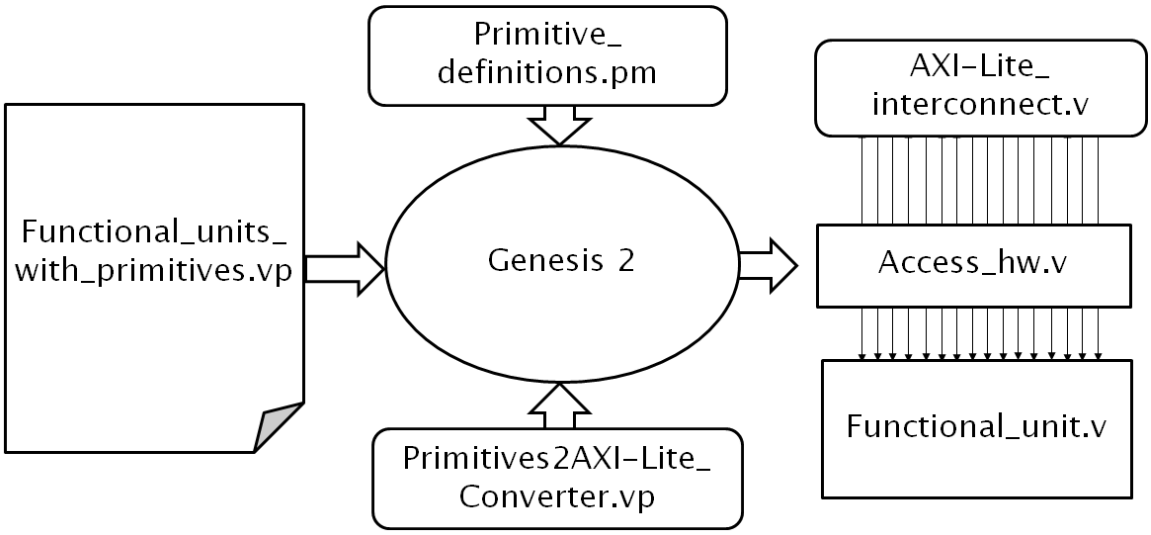

Figure 4.6: Flow diagram of how my primitives work with Genesis 2.

algorithms, demonstrating the functionality of my primitives.

While bringing high-level synthesis integration capabilities to RTL-like designs like ISPGen proved useful for our goals, it is not necessarily groundbreaking. The real benefit of my primitives is that they export a substantial amount of information about the physical IP interface in a predictable, parseable data-structure. As I will discuss in Chapter 5, I can leverage the standardized interface information contained in the primitives and *system_ connector* to automatically generate custom low-level software drivers for IP blocks produced by ISPGen.

Chapter 5

Automating Software Generation

Up until now, my contributions have mostly focused on wiring an IP block into a larger SoC system. This, however, only solves part of the interface problem. For the hardware to be used in the system, the processor still needs software collateral, including low-level C drivers to make the hardware accessible to the software, and a high-level API to make it accessible to application developers.

How this software is created has major ramifications for the nascent IP generator design methodology. One of the proposed benefits of generator framework, and one that features a prominent role in the goals of ISPGen, is the generator's ability to enable rapid prototyping and design refinement at low non-recurring engineering costs. This would enable field-testing a wide variety of designs on reconfigurable fabrics like FPGAs. If a new driver and software stack needs to be manually written for each instance that the generator creates, however, system designers will still be severely limited in their abilities to iterate through and test multiple designs, and a sizable portion of the benefits from using generators will be lost. If the software collateral could be automatically generated along with the hardware, however, then the generator design methodology could truly enable rapid prototyping.

This chapter discusses the driver and API generator I created for ISPGen. My generator leverages the IP and interconnect information encoded in the primitives I proposed in Chapter 4 and combines it with an Operating System specific template in order to create a full Linux driver. I then leverage this same information along

with other algorithm specific collateral produced by the Darkroom DSL-to-DPDA hardware synthesis flow to create the high-level API.

5.1 Building Drivers

To function, a device driver must contain information about the hardware it is driving—including the IP's advertised interface, and knowledge about the system interconnect—to be able to efficiently send data to the IP. For my generator, I always assume that the IP is mapped via some type of memory-mapped input/output (MMIO) system—e.g. from the processor's perspective, the various device resources can be accessed just like standard memory addresses.

From the IP, the driver needs to know the number and types of interface elements, the flow control requirements, and how each of these elements map to the device's address range. To make things easier for the programmer and driver designer, it is also helpful to know the mapping of the architectural name for each resource to each of the interface elements. From the interconnect, the driver needs information like the base address of each block, whether there are DMA engines available, and how to use all of these resources.

In addition to information about the hardware, the driver also needs to contain mechanisms on how to interact with both the operating system kernel and the user to advertise the hardware's functionality. Operating systems like Linux often have a set of software methods that all drivers are required to implement. For example, in Linux character and block drivers, which allow devices to be advertised to the user as a file handle, the driver must implement `open` and `close` methods that define actions the system should take when the device's "file" is accessed. While every Linux driver must implement these methods, other operating systems may have different hardware access models

Additionally, the operating system also places some restrictions on *how* driver functionality can be advertised to users. In Linux, for example, all driver functionality must be advertised through a handful of standard function calls—`read`, `write`, `iocontrol`, `mmap`, and a few others.

Within these constraints, it is then up to the designer or tool that is creating the driver to determine how best to advertise the device's capabilities to the programmer, and to implement the driver's functionality.

5.1.1 Driver Design Techniques

While there are some conventions on how basic drivers interact with the user space—in Linux, simple character drivers use the Linux write and read methods for moving operands to and from the devices—for specialized IPs like those produced by the ISP Generator, that can have hundreds of individually settable IP interface resources, and where there might be a need for some software processing in the kernel space, there may be no single "correct" way to advertise the IP resources to user space software. This means that most driver generator systems will still need some level of user interaction to specify how the generated driver should interact with the system.

From an implementation perspective, there are many existing conventions and techniques that help to make driver creation simpler. One of these is to encode the high-level functionality of a class of devices directly into the communication protocol. One example of this is the USB mass storage device class, which defines a set of protocols for how all USB storage devices, such as flash drives should interact with the system. This standard is specifically written to enable and simplify high-level storage tasks like file transfer, and file system management. As a result, once a USB mass storage driver is written for an operating system, most USB storage devices will automatically be compatible, negating the need for per device drivers.

The concept of one driver being used to power a class of devices has implications for individual hardware instances created by generators. In ISPGen, while all of the generated hardware instances have unique interfaces and functionality, they also share a number of characteristics. Functionally, they all use the same mechanisms for moving those images in and out of the device, they share the same high-level control mechanisms for controlling image processing and per stage flow control, and all have distinct configuration and operation states. Therefore, the kernel optimizations and high-level I/O protocol for accessing each device, and the general structure of the

driver could be shared.

The challenge with creating such a general driver is that the control interface of a generated IP is likely to vary from instance to instance. Therefore, the generic driver would need some mechanism for knowing about the IP's specific configuration interface. This interface information could be communicated dynamically by the device as part of a generic ISPGen communication protocol, but this would require added complexity on the part of the IP to store and transmit configuration information. A "driver generator" approach could help to rectify this problem by allowing a user to create a general driver *template*, and then populating the template with the per-instance implementation details.

People working in the reconfigurable computing field have already started to build such driver generators. The Xilinx Vivado design suite [17], for example, addresses this problem by automatically generating a C header file that maps the architectural name of each IP interface resource to its physical address on the bus. The person or software in charge of creating the full driver can then plug this generated C collateral into a driver template to handle the I/O communications with hardware. My implementation of a driver generator expands on the work of previous driver generators by automatically incorporating advanced flow-control mechanisms. By leveraging the information encoded in my interface objects from Chapter 4, my driver generator provides functionality like automatically managing shadow buffers, deciding which interface elements should be accessed via DMA, and ensuring that interface resources are only written at legal times. While incremental, these advances help to ensure that the resulting driver is both high-performance and easy to interface with.

While the use of a generic driver or driver template to create a driver for a class of IP blocks helps amortize the required driver development effort across many devices, someone still has to write the driver. To address this fact, a few researchers have built varying types of driver generators over the years. One approach, exemplified by the work of Bombieri et al. builds drivers directly from test benches [8]. Bombieri uses software to convert the test bench's functionality into a finite state machine. Either the IP or system designer then manually annotates the sub graphs of the finite state machine into tasks that they would like to see in the driver. The designer is also

asked to provide the MMIO addresses for each of the IP block's interface elements. Bombieri's driver generator takes the annotated state machine and mapping table and produces C code for a basic driver.

Since most IP blocks come with test benches for use in verification, Bombieri's method has a low barrier to entry. Also, by separating out the graph annotation and IP resource memory mappings, this technique opens up the possibility for an IP designer to carefully construct and annotate a test bench and driver graph for his block. System integrators would then only need to provide MMIO mappings for the IP block in their system to get a fully functional driver.

For generated systems, however, this approach has a few drawbacks. Mainly, this method requires manual user intervention to construct a driver for each generated IP block. The ISPGen for example is capable of creating hardware for anything from a simple Gaussian blur filter to a complete photographic pipeline capable of processing raw images from a camera sensor, and the details of the interface for each of these blocks varies substantially both in terms of number and types of interface resources that need to be programmed to compute a task. While ISPGen could produce a unique test bench for each generated instance, each time a system integrator wished to specify a new IP block they would have to manually annotate the test bench finite state machine and edit the MMIO mappings to create a driver. This would require the system integrator to know about the IP block's functionality, and the added manual design effort would likely limit the ability to rapidly prototype these designs. Bombieri's annotation technique could potentially be expanded to automatically handle the sorts of small interface variations found between ISPGen's generated instances, but that is beyond the scope of this work.

Therefore, for my driver generator, I still rely on a pre-written driver template to provide me with most of the driver implementation details. While the template I use here was specifically built and tested for the ISP generator architecture, many of the features of the drivers I generate are applicable to a range of fixed hardware accelerators. Therefore, it is my hope that as part of future work, some of the mechanisms of the ISPGen driver template can be generalized into a more generic driver generator.

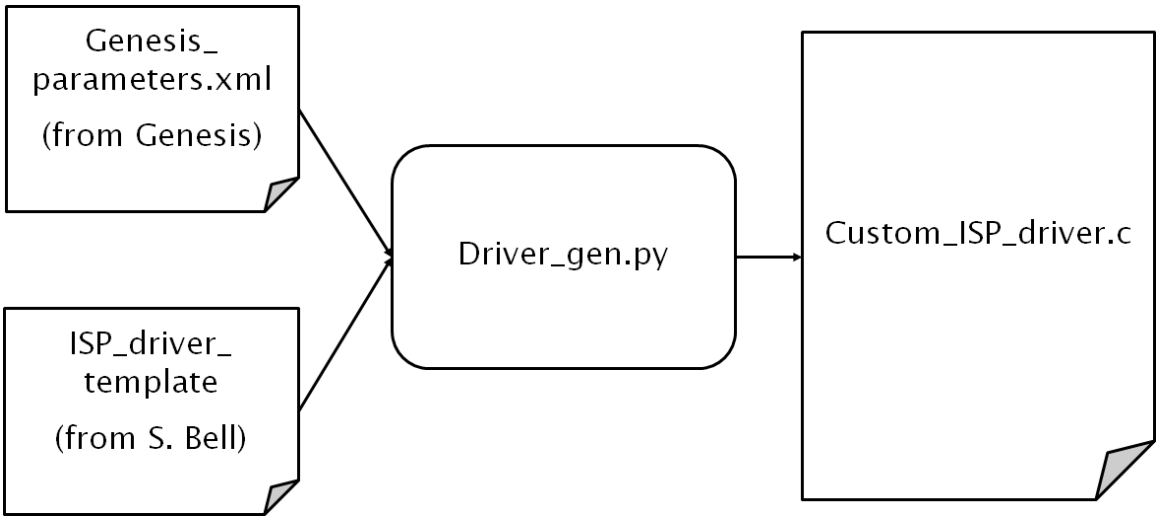

Figure 5.1: Block diagram of my driver generator system. It combines design information from Genesis 2 with a driver template to create an IP specific custom driver.

5.1.2 Generating the Driver

As discussed in Section 5.1.2, driver software has two main jobs: handling I/O, and interfacing with the operating system and user. For the I/O portion, my driver generator is able to get most of the information it needs from the interface primitives introduced in Chapter 4. To manage the device, however, and to optimize the driver performance through kernel functions, additional knowledge about the low-level device functionality and architecture is required. In my generator flow, illustrated in Figure 5.1, I rely on a pre-built driver template to provide these higher-level functions.

The driver generator template is raw C code that implements kernel driver functions. To create a complete driver, I insert generated code into specific places in the template to complete the driver. My template advertises the control interface to the programmer through the Linux `mmap` command. For each of the resources found on the control interface of the ISPGen, there is a corresponding set of addresses in the memory pointer returned by invoking mmap. To change a configuration value, the user simply writes a value to the corresponding mmap pointer, and the driver ensures that the value is passed to the IP block the next time the IP block is idle. Rather than transferring these values directly to the IP block, the driver captures any value

sent to the mmap command internally, and sends the value to the IP only when the user requests that a frame be processed.

Linux convention would normally dictate that transferring an image to the IP for processing should be handled by the user passing a pointer to the image to the driver's `write` method. For performance reasons related to our Zynq-based system, our template designer, Steven Bell, decided to compel users to specially request a pre-allocated kernel-space memory buffer for storing their images and transfer the image to there. The template uses the Linux `ioctl` driver interface to provide a pointer to a free buffer to the user. There is also an ioctl command to "deallocate" each buffer, which essentially just tells the driver that the buffer is no longer in use. There are an additional set of ioctl commands for activating the IP and getting results.

This implementation choice made the process of DMA'ing images from user space to the IP device simpler, as otherwise the driver would need to build large scatter-gather tables to DMA the image from user-space memory to the hardware. The changes made to the advanced ISP driver made the driver more complex to interact with, however. While the `read` and `write` driver commands are defined as part of the standard Linux driver model, `ioctl` commands are driver specific. Therefore, users need to know specific details about the ISP generator driver in order to work with it. Also, depending on how common this sort of buffer pre-allocation is, optimizations like these may limit the re-usability of the driver template. Ultimately, the person in charge of creating the driver template must determine if these trade-offs are worth it.

Internally, the driver template uses a queue structure to manage multiple frame requests at a time, associating each requested frame with the `mmap` control information specified for it.

To generate the finished driver, my system is responsible for pulling in information about the IP's communication interface and integrating this information into the template code. My generator reads information about the IP interface from the outputs of the Genesis II hardware generator language. Whenever Genesis II is used to generate an IP instance it produces an XML file listing all of the different configuration parameters used to generate that instance. This XML file includes all of

the information about the IP interface encoded by my interface primitives and interconnect generator organized into a fixed, predictable data structure. The full list of interface data that my primitives encode is shown Table 5.1. The interface data from each primitive is aggregated into a single Genesis 2 parameter, `DRIVER_DATA`, by the sys_connector object, discussed in Chapter 4.

Table 5.1: Summary of IP interface information provided by each primitive. For the ISPGen implementation, `accelerator` identifies the kernel, and `local_name` identifies the specific interface resource.

Key	Description
`local_name`	Interface signal name as specified within the IP RTL
`accelerator`	Name of the IP accelerator
`arch_name`	Architectural name of interface resource (accelerator name + local name)
`start_addr`	Base address for interface element
`end_addr`	Last address for interface element
`addr_space`	Total address space covered by element
`groups`	Number of shadow buffers
`data_width`	Width of data bus
`enable_index`	Chip enable index for the primitive
`block_sig`	The signal that gates access to this primitive
`direction`	Whether the primitive is an input, output, or both from the IP's perspective
`streaming`	Whether the primitive should be accessed via DMA
`bandwidth`	Bandwidth required by the IP for full performance

Using this information, my generator sets up the template with the MMIO addresses for each interface resource. I also use this information to automatically handle access control to the IP resources. The information stored in the generator allows me to enforce directionality of data to and from IP blocks. For example, I can separate out read-only elements from write and read/write elements, and write the driver so that commands cannot be written to these blocks except in debug modes. My generator also integrates knowledge of the interface primitives' "block" signals into the driver. This guarantees that interface resources are only written at legal times. For any IP resources that are set as "stream" enabled, my driver generator will automatically set up the driver to DMA data to these blocks.

The information allows the generator to build the driver to automatically manage the shadow buffers on control registers. As discussed in Chapter 4, shadow buffers are groups of registers that exist between the interconnect and a configuration register on the IP interface. These extra registers are used to buffer future configuration values for the IP block so that the IP can be quickly configured for the next run. My generator knows which addresses correspond to which IP resources, and which of these correspond to shadow buffers. Using this information, whenever there are more than one requests pending in the work queue, the generated driver can automatically pre-load the shadow registers with the program values used by subsequent frames. As soon as a control sequence is transferred from the shadow buffers to the IP, the driver can reuse the shadow buffer for the next queued set of control values. When combined with driver-side optimizations, like only rewriting configuration values when values change between frames, or using a dedicated way of the shadow buffers to "memoize" popular or default control settings (left for future work), such optimizations can decrease interconnect traffic and reduce the time between successive IP runs, and all can be automatically implemented by my generator.

Finally, my generator uses knowledge about shadow buffers to simplify the control interface presented to the driver user. Thanks to my generator, when a user memory maps the IP interface, they only see one instance of each control register. All of the complexity of keeping track of which shadow buffer to write and which set of shadow buffers corresponds to which frames is handled automatically by the driver.

The information in the generated interface also allows my generator to present a higher level interface to the driver user. The Linux memory map function advertises the control interface as one contiguous buffer of memory space. It does not, however, encode or communicate any knowledge about which index corresponds with which IP interface element. As shown in Figure 5.2, I use the information stored in the Genesis file to create a C struct that maps between the hierarchical IP interface names and their corresponding memory map indices. This is provided to the user as a C header file.

By design, my generated C struct only includes the IP interface elements that can be written by the user—a separate struct (not pictured) is provided for reading

```
/* Generated map */
struct name2mem{
  unsigned int isp filter tap1,
  unsigned int isp filter tap2,
  unsigned int isp approx table[5],
  unsigned int* isp ctrl fifo,
  unsigned int isp ctrl fifo len
};
```

Figure 5.2: A C struct mapping architectural names of interface resources to their index in the `mmap` buffer. Both the driver's `mmap` interface and C structure are designed to be aware of the directions of the primitives, the type of hardware the primitives represent, and whether they are streaming.

values from elements—and the struct elements are typed depending on the type of interface primitive they represent. "Addressable" blocks that are not marked as requiring streaming access are advertised as a C array. Interface primitives that the hardware designer has set as "streaming" show up as pointers, so the user can point to the memory location where the data is located. The driver then uses this pointer to set up a DMA between the data and the IP streaming interface element.

To program the IP block, programmers simply populate this C struct (from Figure 5.2) with the desired parameters, and copy its contents directly into the memory map buffer provided by the driver.

Combined, these features offer full access to the ISP generator hardware. Programs can easily transfer images stored in memory to the accelerator, and have them processed in real time.

While the specialized template we constructed was only tested with John Brunhaver's ISPGen, it is to a broader class of streaming hardware devices: functionalities like programming control registers and transferring large data sets between the device and OS are not exclusive to the domain of image processing. In fact, it is likely that after building driver generators and templates for a number of different types of IP generators, we can identify commonalities and design patterns among the types of driver functions that are implemented in different classes of devices. This would allow us to create a very general driver template that could be used to create efficient drivers for a wide swath of IP generators. In addition to commonly used features,

like handling DMAs to various IP interface elements and registering interrupts, such a template could even be built to implement more advanced features like system profiling, where the driver generator automatically builds performance counters into the driver.

5.2 Generating the API

Even with automated driver constructions, application programmers still need to know some low-level details about the Linux driver model, and the generated IP to use the hardware. For applications like the ISP generator, where the target user is algorithm developers in the domain of computational photography, many of our users may lack this kind of knowledge. Therefore, we need to create a high-level API to allow our users to take advantage of our generated driver and hardware.

One of the big challenges with automatically creating an API is that APIs are generally written to reflect how the IP is going to be used. This is tied to the functionality of the IP block. Up until now, the generators discussed here all rely on the abstractions specified and implied by the SoC methodology. I used general architectural models for IP control interfaces and well known communication models to integrate and implement basic communications with IP blocks. With heterogeneous IP blocks, however, no single use abstraction exists; there is not enough information to create an API. Fortunately, in the case of the ISP Generator, we have another mechanism for determining the IP's high-level functionality: the Darkroom domain specific language [29].

Domain specific languages (DSLs) have recently gained popularity as a way for programmers to create efficient code in a specific application domain. These languages specially tailor their programming models and capabilities to fit the constructs and types of computations generally used for writing high-level algorithms in a given application domain. The back-ends of these languages then leverage knowledge about the domain and the DSLs tailored programming model to create highly-efficient, optimized implementations for heterogeneous hardware platforms [11].

Darkroom is a DSL based off of the Terra programming language [16]. It is

specifically designed to represent image processing pipelines and recently has been co-developed with ISPGen. Both projects are designed to represent the same classes of image processing algorithms, and the Darkroom DSL has been fitted with a DPDA-compiler back-end. Since DPDA is the language used to specify hardware synthesis in the ISPGen language, this means that Darkroom allows developers to specify their algorithms at a high-level, and automatically create custom hardware through the ISP generator.

Additionally, the use of Darkroom as a front-end for synthesizing hardware provides all of the high-level information necessary to automatically create an API. Since Darkroom and Terra are first and foremost software simulation languages, Darkroom programs are designed to be linked into high-level application code by producing linkable C function calls for the algorithm. Since software developers can use the Darkroom C simulation code to test their algorithms, it stands to reason that this software interface is both high level enough and fully featured enough to act as an API.

Also, the structure of the C API calls advertised by Darkroom map very directly to tap values used to configure the hardware. Regardless of the algorithm, the Darkroom C API always takes in two arguments, a pointer to the image to be processed, and a C struct that contains fields and configuration values for all of the configuration registers and memories on the IP block. Not coincidentally, this structure is very similar to the driver C structure I generate, illustrated in Figure 5.2, as the Darkroom–ISPGen flow ensures that all of the configuration values that must be set to process an image in Darkroom are represented in the IP block with one or more dedicated interface primitive.

In order to create an API, we must map the Darkroom function calls to the IP's driver interface. Darkroom by itself, however, does not contain sufficient information to map its high-level software interface to the driver: due to some quirks in the DPDA specification and ISPGen's implementation, the hardware interface requested by the DPDA can sometimes slightly differ from the interface advertised by the generated hardware. Since the driver's interface depends heavily on the IP's physical interface, this meant that my API generator had to synthesize information from both the

Darkroom and ISPGen tools to match the interfaces together.

5.2.1 Mapping the API to the Driver

To run Darkroom code in software, the first thing a user does is specify the filter coefficients and other image pipeline settings. Darkroom does this by providing users with a C struct with named entries for each different filter parameter. In the next step, the user makes a function call to run the pipeline, and the function returns the processed image.

In hardware, the Darkroom function call actually encompasses a number of functional processing steps. Mapping the kernel configuration to the driver is a fairly simple process. All of the members of the Darkroom-produced pipeline configuration struct generally map in a one-to-one fashion to interface resources on the IP block. Occasionally some hardware is duplicated to allow for parallel processing. In this case, the taps in hardware are duplicated, and given uniquified names, making the mapping between the Darkroom struct element and the IP interface element one-to-many.

All of the names of resources advertised on the generated hardware interface share a common base name with but are distinct from the name of the corresponding Darkroom struct item they are derived from. Mapping the API struct elements to driver memory mapped I/O values is as simple as doing a text match on the two sets of names. To get the Darkroom struct element names into my generator, I parse an XML file already generated by the Darkroom-to-DPDA flow. I get the names of the hardware taps from the interface primitives discussed in Chapter 4.

While the Darkroom API allows a user to pass a pointer to user-space memory containing the image to be processed, the driver expects the image to reside in a driver allocated buffer. To handle this, the API software uses the driver's "allocate buffer" `ioctl` command to get a buffer and copies the image into kernel space.

When the user calls the API, the API opens the driver, and copies the values from the Darkroom defined C struct, into the `mmap` control interface. It allocates a kernel space buffer from the driver, copies the image over, and uses the driver's "process image" *ioctl* command to start computation. The API then calls "read image" and

blocks until the processed image returns. All of this code is automatically generated for any IP block generated by the ISP generator. While this implementation is heavily defined by the inner-workings of Darkroom, the ISP generator, and the driver generator, these concepts can be applied to a general range of DSL-enabled hardware generators.

5.2.2 API Limitations and Future Work

The software interfaces provided by these DSLs, however, are not always the optimal choice for a hardware API. This often stems from the fact that coding techniques that make sense in software do not always make sense for driving high performance hardware. This is a problem that we have run into with Darkroom and the ISP generator, as Darkroom was built with a software implementation in mind.

A major issue is that there are different scheduling constraints between software and IP. Since processor-to-IP communication can take many cycles and often occurs over potentially congested shared-links, hardware often includes optimizations to try to mask communication latencies. Hardware optimizations like shadow buffers, for example, allow the programmer to queue up IP programs in advance, so the IP can start processing the next frame immediately after finishing the current one. These optimizations rely on the user queuing driver calls in advance, and work best with a non-blocking API, where users can queue new frames at any time. In software, however, these communication delays do not play as large a role, so the Darkroom C only offers single-frame, blocking calls to the algorithm. Once again, the Darkroom API limits the performance of the IP block.

One option to address this limitation is to simply model the API off of the DSL simulation interface, rather than copy it directly. In the case of the ISP generator, this would involve adding a function call to allocate kernel buffers, and creating non-blocking variants of the pipeline function calls. Of course, the downside of this method is that the generated API may no longer be fully compatible with the test code. Even if the API generator mapped the original DSL simulation function calls to driver commands, and just offered the hardware optimized calls as an expanded

feature set, any existing code would still need to be retooled. Since in most cases the desired changes are possible and not harmful in the software version (and might even help for parallel execution engines), the right solution when possible is to update the interface specification so that programmers and system testers only need to worry about one set of function calls. This will hopefully be completed in a future version of Darkroom.

5.2.3 API Summary

Looking forward, the ability to automatically generate unified software/hardware APIs also hints at a solution to one of the major hurdles for casual application developers looking to incorporate hardware acceleration into their programs: compatibility. Today's major mobile development platforms all are designed to run the same software across a variety of hardware platforms. This is especially notable in Android, where different handset manufacturers source a wide range of SoCs for their phones, and is to a lesser extent a problem on iOS where new generations of phones bring new hardware capabilities. If a developer wants to use hardware acceleration in a program, he or she must first detect whether the hardware is present in the system and, if not, provide a software implementation to perform the computation. Using consistent APIs between hardware and software implementations, however, can eliminate this concern. As part of API generation, the system can also build a wrapper around the DSL simulation C and the API. When a program makes a call to the wrapper, the wrapper can automatically detect whether the IP block is present, and if not, route the function call through software for processing. Therefore, regardless of the hardware platform the application developer is working on, they can safely use the API software and get the benefits of hardware acceleration wherever it is available.

Chapter 6

Conclusions

SoC design offers many benefits to chip designers. By allowing designers to integrate many pre-verified custom accelerators into a single chip, this methodology helps system designers achieve the energy and performance benefits of custom design across a wide range of application domains. As transistors get smaller, SoC provides a powerful framework for combining functionalities that used to span multiple chips onto a single die, further improving energy and performance while decreasing manufacturing costs for the target systems. These strengths have allowed SoCs to dominate the mobile compute space, and have allowed it to make some inroads into the high-performance desktop market.

As designers move to incorporate ever more functionality onto a single die, however, they are increasingly running into the limits of our abilities to design these heterogeneous systems. Every new accelerator needs to be physically connected into the system, and these connections need to be verified, and communication protocols need to be checked to ensure that the accelerator can properly communicate. Once the hardware is attached, software connections, a driver and an API, need to be built for the accelerator to be used in the system.

To allow designers to keep pushing the bounds of system performance with new and more powerful SoCs, we need to devise new ways to integrate these systems. One potential solution to this dilemma is automation. Ideally, IP modules would be simply "plug-and-play": the module would advertise its interface and how it expects to

be communicated with, and a design tool would automatically generate the logic required to integrate it with the system interconnect. This same information could then be combined with high-level information about the IP block to automatically generate low-level hardware drivers and a high-level software API. If possible, this could drastically cut the amount of design effort required to integrate each new IP block into a system. While many researchers have tried to address pieces of this problem, with this thesis, I have tried to propose a set of solutions that addresses everything from automatically wiring the IP blocks into the system to software generation.

Recognizing that one hurdle to automating hardware integration of various existing IP blocks using current HLS mechanisms was the difficulty of specifying IP interface protocols, my first contribution was to propose an IP bus interface abstraction and interface definition capable of succinctly capturing interface protocols. Using the observation that, for a given high-level bus functionality, all interface buses need to encode similar sets of information, I was able to distill a compact set of parameters that are capable of encoding the different mechanisms a bus is likely to use to transmit this information. I then used my parameters to demonstrate a prototype interface generator capable of creating synthesizable bridge RTL between different interface descriptions encoded in my parameters. While my definition currently does not support all of the advanced high-level operations found in high-performance buses like OCP–cache coherency, atomic operations, multi-threading, etc.–it does provide a model for how designers can continue to use existing IP blocks as they transition towards HLS design and integration methodologies. This work also teases the potential that if an automated system were used for both IP hardware integration and driver generation, we can help ensure that IPs built for old interface standards can continue to perform on interfaces with newer, incompatible features.

With a means of describing existing IP interfaces at a high-level, which would potentially allow them to be integrated with high-level design approaches to system integration, I then moved on to the problem of how future RTL-based IP blocks and generators can be built to take advantage of HLS-based interface synthesis techniques. While I believe HLS will become a predominant design methodology for creating IP blocks, there are still likely to be specialized accelerators that are not handled

optimally by a user's set of HLS tools and that must be written in RTL. For system integration, however, these blocks should still be able to benefit from HLS-based interface synthesis techniques. Therefore, I developed a set of Genesis 2 interface primitives that can be used to map the interface of these RTL blocks into HLS flows, and immediately put these primitives to use in automatically integrating the hardware of IP blocks generated by the ISPGen tool into an FPGA-on-SoC framework.

While integrating the IP hardware into a larger system certainly helps to address the problem of SoC design complexity, the system still needs a software driver for the IP block to be used in software. For the IP block to be accessible to the average application developer, it also needs an API that obscures the low-level tasks of interacting with the device driver with high-level function calls for using the hardware.

Using the IP information encoded in my RTL-to-HLS interface mechanisms in conjunction with a driver template tailored for use with ISPGen, I constructed a system capable of automatically creating a C software driver for generated IP blocks. While the driver template may need to be tweaked to work with each distinct IP generator a system integrator would like to use, the techniques used to automate the driver creation process are general, and will hopefully act as a basis for creating a powerful, largely application independent driver generator framework.

Finally, I tackled the issue of creating a high-level API for generated IP blocks. My work in this area was for IP generator systems that already use a domain specific language to specify the specialized hardware generated. I created a system that mapped the API of the Darkroom DSL to the driver calls required to process an image in the hardware generated from the Darkroom description.

Combined with the hardware created by ISPGen, this work allows for a true "one-button flow" that takes a Domain expert from testing a new way to process images in software to prototyping a real-time hardware implementation in the field. Such a setup could potentially enable boom in the creation of IP blocks and the use of programmable logic in general purpose computing, as it would allow domain experts with no hardware knowledge to experiment with custom hardware design.

Bibliography

[1] Andrew Adams, Eino-Ville Talvala, Sung Hee Park, David E. Jacobs, Boris Ajdin, Natasha Gelfand, Jennifer Dolson, Daniel Vaquero, Jongmin Baek, Marius Tico, Hendrik P. A. Lensch, Wojciech Matusik, Kari Pulli, Mark Horowitz, and Marc Levoy. The Frankencamera: An experimental platform for computational photography. In *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, pages 29:1–29:12, New York, NY, USA, 2010. ACM.

[2] ARM. *AMBA*. http://www.arm.com/products/solutions/ AMBAHomePage.html.

[3] ARM. *AMBA 4 AXI4, AXI4-Lite and AXI4-Stream System Verilog Assertions (SVAs)*. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022d/index.html.

[4] Arteris. *Arteris, The Network-On-Chip Company*. www.arteris.com.

[5] K. Avnit and A. Sowmya. A formal approach to design space exploration of protocol converters. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 129–134, 2009.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Krste Asanovic, and John Wawrzynek. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Design Automation Conference (DAC)*, 2012.

[7] Daniel Ulf Becker. *Efficient microarchitecture for network-on-chip routers*. 2012.

[8] Nicola Bombieri, Franco Fummi, Graziano Pravadelli, and Sara Vinco. Correct-by-construction generation of device drivers based on RTL testbenches. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1500–1505. European Design and Automation Association, 2009.

[9] Gaetano Borriello and Randy H Katz. Synthesis and optimization of interface transducer logic. In *Proceedings of the International Conference on Conputer Aided Design*, pages 481–494, 1987.

[10] John Brunhaver. *Design and Optimization of a Stencil Engine*. PhD thesis, Stanford University, 2014.

[11] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM.

[12] Hansu Cho, Samar Abdi, and Daniel Gajski. Interface synthesis for heterogeneous multi-core systems from transaction level models. In *ACM SIGPLAN Notices*, volume 42, pages 140–142. ACM, 2007.

[13] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689, 2001.

[14] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: recording microprocessor history. *Communications of the ACM*, 55(4):55–63, April 2012.

[15] R.H. Dennard, F.H. Gaensslen, H.N. Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Proceedings of the IEEE (reprinted from IEEE Journal Of Solid-State Circuits, 1974)*, 87(4):668–678, 1999.

[16] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A multi-stage language for high-performance computing. *SIGPLAN Not.*, 48(6):105–116, June 2013.

[17] Tom Feist. Vivado design suite. *Xilinx, White Paper Version*, 1, 2012.

[18] Sameh Galal, Ofer Shacham, JS Brunhaver, Jing Pu, Artem Vassiliev, and Mark Horowitz. FPU generator for design space exploration. In *Computer Arithmetic (ARITH), 2013 21st IEEE Symposium on*, pages 25–34. IEEE, 2013.

[19] A Grasset, F. Rousseau, and AA Jerraya. Automatic generation of component wrappers by composition of hardware library elements starting from communication service specification. In *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, pages 47–53, June 2005.

[20] Douglas Grose. Keynote: From contract to collaboration delivering a new approach to foundry. DAC '10: Design Automation Conference, June 2010.

[21] 2011 Technology Working Group. Design. In *International Technology Roadmap for Semiconductors*. 2011 edition edition, 2011.

[22] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA '10: Proc. 37th Annual International Symposium on Computer Architecture*. ACM, 2010.

[23] Jen-Hsun Huang. *nVidia CES 2013*. Las Vegas Convention Center, January 2013. http://www.nvidia.com/object/ces2013.html.

[24] HyperTransport. *HyperTransport*. http://www.hypertransport.org/.

[25] IBM. *OPB*. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/9A7AFA74DAD200D087256AB30005F0C8.

[26] IBM. *CoreConnect Bus Architecture - IBM Microelectronics*. https://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture, March 2006.

[27] Intel. *4th Generation Intel®Core™Processor Die Shot*. http://download.intel.com/newsroom/kits/core/4thgen/gallery/images.

[28] Intel. *Intel®Quickpath Interconnect Maximizes Multi-Core Performance*. http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html.

[29] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. In *Proceedings of SIGGRAPH 2014*, Vancouver, Canada, August 2014. ACM.

[30] Erica Jones and Jonathan Sprinkle. autoVHDL: A domain-specific modeling language for the auto-generation of VHDL core wrappers. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 71–76, New York, NY, USA, 2011. ACM.

[31] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112, 2002.

[32] Dongwook Lee, Hyungman Park, and Andreas Gerstlauer. Synthesis of optimized hardware transactors from abstract communication specifications. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 403–412. ACM, 2012.

[33] Dejan Marković and Robert W Brodersen. *DSP Architecture Design Essentials*. Springer, 2012.

[34] J Nestor and DE Thomas. Behavioral synthesis with interfaces. In *Proc. IEEE International Conference on Computer Aided Design*, pages 112–115, 1986.

[35] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69 – 70, june 2004.

[36] OCP. *OCP-IP: Get the Specifications.* http://ocpip.org/get_the_specifications.php.

[37] OpenCores. *Generic AHB Slave Stub.* http://opencores.org/project,ahb_slave.

[38] OpenCores. *Generic APB Slave Stub.* http://opencores.org/project,apb_slave.

[39] OpenCores. *Generic AXI Slave Stub.* http://opencores.org/project,axi_slave.

[40] Michael K. Papamichael. *CONNECT: Re-Examining Conventional Wisdom for Designing NoCs in the Context of FPGAs.*

[41] ITUTX Recommendation. 200 (1994)| iso/iec 7498-1: 1994. *Information technology–Open Systems Interconnection–Basic Reference Model: The basic model.*

[42] James A Rowson and Alberto Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the 34th annual Design Automation Conference*, pages 178–183. ACM, 1997.

[43] Ofer Shacham. *Creating Chip Generators Using Genesis2.* Stanford, http://genesis2.stanford.edu/.

[44] Ofer Shacham. *Chip multiprocessor generator automatic generation of custom and heterogeneous compute platforms /.* 2011.

[45] Sonics. *Sonics On-Chip Communication Network for Advanced SoCs.* www.sonicsinc.com.

[46] D. Wingard. MicroNetwork-based integration for SOCs. In *Design Automation Conference, 2001. Proceedings*, pages 673–677, 2001.