Mapping a Domain Specific Language to a Platform FPGA

Chidamber Kulkarni
Xilinx Inc
San Jose, Ca
Chidamber.Kulkarni@xilinx.com

Gordon Brebner
Xilinx Inc
San Jose, Ca
Gordon.Brebner@xilinx.com

Graham Schelle
University of Colorado
Boulder, Co
schelleg@cs.colorado.edu

ABSTRACT
A domain specific language (DSL) enables designers to rapidly specify and implement systems for a particular domain, yielding designs that are easy to understand, reason about, re-use and maintain. However, there is usually a significant overhead in the required infrastructure to map such a DSL on to a programmable logic device. In this paper, we present a mapping of an existing DSL for the networking domain on to a platform FPGA by embedding the DSL into an existing language infrastructure. In particular, we will show that, using few basic concepts, we are able to achieve a successful mapping of the DSL on to a platform FPGA and create a re-usable structure that also makes it easy to extend the DSL. Finally we will present some results of mapping the DSL on to a platform FPGA and comment on the resulting overhead.

Categories and Subject Descriptors
C.1.3 [**Other Architecture Styles**]: Adaptable architecture

General Terms
Design, Experimentation, Languages.

Keywords
Domain Specific Language, Platform FPGA, Network Processing

1. INTRODUCTION
A domain specific language (DSL) is a programming language tailored for a particular application domain. An effective DSL enables development of a complete program or design for a domain quickly and effectively. A fundamental requirement for an effective DSL is capturing precisely the semantics of the application domain. Common examples of DSLs include Matlab for signal processing, HTML for document markup and OpenGL for 3D graphics. Potentially, there are many advantages to using DSLs, the most fundamental being that programs are generally easier to write, reason about and modify compared to using general purpose languages (such as Verilog and C). Typically, DSLs will be at a higher abstraction level than general-purpose languages and used by domain experts.

DAC 2004, June 7–11, 2004, San Diego, CA, USA.

We have so far discussed the advantages of using DSLs in application design. However, the single most inhibiting factor against using DSLs is the significant initial cost related to the infrastructure required to support a DSL. For example, transforming programs in DSLs such as Matlab onto a hardware description language such as Verilog requires significant effort and tool support. There have been numerous research projects related to such efforts, for example [1,2]. In this paper, we present a case study of mapping a DSL for the networking domain - Click [3] - to a platform FPGA. The goal of this work is twofold: first, to understand and quantify the initial cost of design with a DSL; and second, to evaluate the overhead in performance of using a DSL.

The search for application-specific solutions with ever decreasing time-to-market is pushing system designers away from the risky time-consuming ASIC design process towards programmable platform solutions, such as platform FPGAs. An example of platform FPGA is the Xilinx Virtex-II Pro family of programmable logic devices that include hard IP cores such as the PowerPC microprocessor and RocketIO serial transceivers [4]. The current state-of-the-art design flow for FPGAs in the networking domain requires entering the design with a hardware description language (HDL) and then following synthesis, place, route, and bit stream generation steps. One of the goals of this work is to present a higher abstraction level for the networking domain by mapping Click on to a platform FPGA, thus enabling easier access to platform FPGAs for domain experts.

Network processors are an alternative implementation platform for networking applications. However, it is now apparent that most of these network processors have been developed without paying much attention to the underlying programming model. Hence, programming network processors has become a great challenge and an active research topic [5]. Platform FPGAs, on the other hand, have been used in real-life networking solutions for a long time. Thus it is natural to provide a higher abstraction for mapping networking applications on to platform FPGAs for domain experts.

We argue based on a similar argument as Hudak's for software development [6], namely that, although the start-up cost for using a DSL as compared to say a HDL is higher, the aggregate development cost of a particular design using DSL should yield significant savings in total design cost.

Two fundamental assumptions are crucial to success in implementing the chosen DSL on platform FPGAs. First, we do not intend to build the infrastructure for mapping the DSL to platform FPGAs from scratch. Rather, we will reuse the existing

infrastructure of a popular HDL, here Verilog, to embed domain aspects into. Second, we will focus on implementing semantic issues crucial to the particular domain and if required, build domain specific tools at this level. This approach promises reuse of syntax, semantics and domain specific tools, and has a larger chance of enabling the savings stated earlier.

The remainder of the paper is organized as follows: Section 2 introduces Click, the chosen DSL for networking. Section 3 discusses the main aspects, specifically semantics and tool flow, of mapping Click on to platform FPGAs. Section 4 presents the results and related discussion. Section 5 presents a brief overview of related work. We end the paper with the main conclusions and directions for future work

2. Introduction to CLICK

Click is a domain specific language for describing networking applications [7]. It is tailored for domain experts, and is based on a set of simple principles. The fundamental unit of computation in Click is an element. An element represents a basic networking task such as classification, header verification, route table lookup, and decrement time to live. Each element has input and output ports. Communication between elements takes place via the ports. A typical Click user specifies the connectivity graph between different elements corresponding to a particular application, and this graph then represents the packet (and control) flow between elements. There are two types of communication between ports in Click: push and pull. Push represents a communication initiated by a source element and pull is initiated by a sink element. A typical Click design has chains of both push and pull elements. A queue element, for example, has a push input port and a pull output port, and thus is ideal for de-coupling push and pull chains.

Click was originally implemented on Linux using C++ classes, and the corresponding C++ and Linux infrastructure. Thus, a main strength of Click is modular, re-usable and open source elements. However, even though Click is a DSL for the networking domain, it cannot fruitfully utilize the inherent concurrency in the domain for implementation due to its strong ties with sequential language infrastructure. There have been attempts at mapping Click on to network processors. However, these tend to be restricted to mapping on to a single specific device, thus difficult to reuse across different network processors. In this work, we have developed methods and tools to enable a mapping of Click on to platform FPGAs. This in turn enables a wide variety of implementations that enable designers to utilize the inherent concurrency in the domain. This is possible both due to mapping Click on to a generic HDL, as well as the generic nature of the FPGA fabric. In the next section, we provide further details of how we embed Click into Verilog.

3. CLIFF – Click for FPGAs

Cliff is an embedding of Click in Verilog targeted to platform FPGAs. As motivated earlier, the goal of embedding Click in Verilog is to leverage the existing Verilog infrastructure to ease the path to implementation. At a meta level this has two advantages: first, it enables domain experts to achieve hardware realizations of their applications without delving into many HDL details; and second, it provides HDL proficient designers a faster starting point for further optimization of the design.

As in Click, elements are also central to a Cliff design. The interfaces to an element were chosen after careful design space consideration, with respect to different possible implementations of Cliff elements, specifically in pipelined, pool, or hybrid manners. The three main interfaces for an element in Cliff are illustrated in Figure 1. These are inter-element, data and memory interfaces respectively. These three aspects form the semantic backbone of Cliff.

The inter-element interface is used to implement communication between elements. The data interface is used for passing data between elements (for example, a packet header) and is optional. The memory interface provides access to a memory implemented using on-chip memory (e.g. BRAM) and where necessary, off-chip memory (e.g. DRAM). We will now discuss further each of the above three semantic aspects.

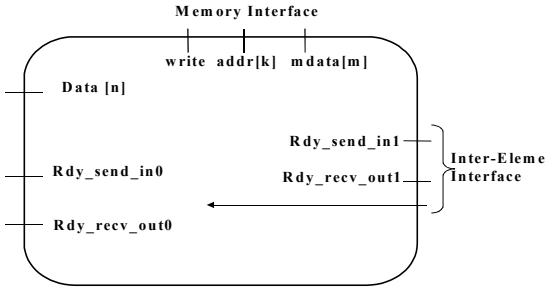

Figure 1 Element Interfaces in Cliff

3.1 Implementation of Elements

Each Click element is implemented using a finite state machine (FSM) model that is reused for all elements. Although data flow semantics seem a natural fit, considering implementation efficiency (of synthesis) a FSM model was chosen. The FSM states are extendable by the user for writing either new elements or extending existing functionality. The base FSM model contains the following three states, of which two are pre-defined:

- RDY_RECV_STATE – in this pre-defined state, the element is ready to receive packet information from an upstream element. Accordingly, it sets appropriate signals (for example, Rdy_send_in0 is set).
- USER_STATE – in this state, the relevant logic related to a particular Click element is implemented using either one or more states. An example extension here is that one state could always be reserved as a memory access state. This helps in providing a clean and transparent memory access interface.
- RDY_SEND_STATE – in this pre-defined state, the element has performed the required packet manipulation and is ready to send the packet information to a downstream element.

The above FSM model for each element is a simple, re-usable, and extendable model that is able to implement a large number of Click elements. For the two pre-defined states, we use two signals to indicate ability to receive data and acknowledgement of receipt of data. Thus, using the two pre-defined states we are able to provide an elegant implementation of the push and pull semantics

between elements. In fact, the push-pull interface in Cliff is quite similar to the C++ inheritance found in Click in the class Element. In addition, one should view the two pre-defined states as communication primitives for each element. We discuss this a little further in the next sub-section.

3.2 Inter-Element Communication

Inter-element communication in Cliff is implemented using a simple protocol that is based on the two pre-defined states introduced earlier. A three-way handshake is used, wherein the downstream element signals the upstream element that it is ready to accept a packet. On receiving this signal, the upstream element passes the required information to the downstream element and the downstream element acknowledges the receipt of this information to the upstream element. At this point, the upstream element is ready to receive another packet's information. Each of the two states relies on two signals (as shown in Figure 1) to provide a reliable control flow between elements. Using this mechanism, we can implement a classic pipelined architecture with different packet flows interacting with each other. The reason three-way handshaking is needed on the interface is due to the unknown latencies from upstream and downstream elements. Anything simpler would require knowledge of the element graph, compromising the reusability aspect of Cliff elements.

However, to exploit more concurrency, one can also connect the elements in a more parallel fashion, for example using a pipeline of element pools. Here, based on domain specific knowledge, one can decide to connect a set of elements in parallel and then pipeline remaining elements. Adding more inter-element connections to this model only requires addition of a pre-defined state with two signals for every additional communication.

3.3 Memory Interface and Organization

The memory interface for each element consists of two different interfaces. First, the data interface that enables passing data (by value) between elements. The second memory interface consists of three signals. They are write, which signals the memory whether the data is to be written or not; address, which provides the particular relative address of/in a packet; and data, the actual data that is read or written. As with inter-element communication, these two simple interfaces enable one to implement a range of memory organizations. Note that in practice, there is no limitation preventing multiple memory interfaces (analogous to multiple ports) for an element.

For example, in one of our benchmark designs of an IP router, a data interface consisting of a data pointer to a payload data in BRAM (block RAM) and complete 344-bit header data for Ethernet/IP (including optional ARP bits) was used. This resulted in a 344-bit wide data path that was mapped on to a Xilinx Virtex-II Pro part. It is notable that, excepting "FromDevice" and "ToDevice", no other elements used their (second) memory interface and so these were removed by the later synthesis step.

We have implemented two different memory controllers that provide encapsulation of BRAM memories and present them as a single large many-ported memory. The number of ports is derived from the number of stages of a pipeline of Click elements. We implemented the memory controllers using the FSM model presented earlier, in the same manner as for Click elements. A fundamental advantage of this sort of generic memory interface is that one can easily plug and play with different memory organizations just by simple modification of the generic Verilog infrastructure.

3.4 Cliff Design Flow

The Cliff design flow is shown in figure 4.

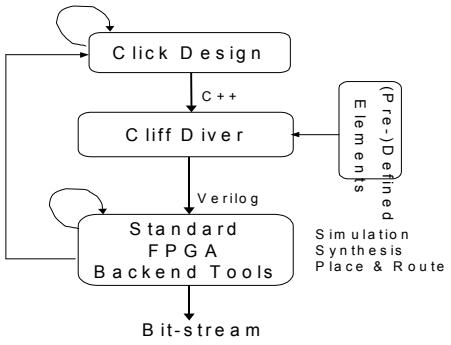

Figure 2 Cliff design flow

Applications are specified in Click. A C++ based Click parser called "CliffDiver" transforms Click descriptions into top-level Verilog design based on the library of Cliff elements. This Verilog design is then entered into standard FPGA tools, such as Xilinx ISE, where one can perform simulation, synthesis, and place and route of the particular design to generate the final bit-stream. In our experiments we used the Xilinx ISE 5.2 as the standard backend tool. The two self-loops indicate the ability to perform local optimizations at each of those steps.

4. Results and Analysis

Currently, we have implemented around 20 Click elements, including some architectural elements such as two different memory controllers. In particular, we have implemented "FromDevice" and "ToDevice" elements with Gigabit Ethernet interfaces to demonstrate the feasibility of designing high performance networking applications. Based on this, we have generated three different Click designs mapped to a Xilinx Virtex-II Pro device. The three designs that we have generated are a basic IP router, a network address translator (NAT), and a differentiated services (DiffServ) router. The results of implementing these three designs onto a platform FPGA are given in table 1.

Table 1 Implemented Cliff Benchmarks

Benchmark	Area (# Slices)	Performance	
		Freq (MHz)	Throughput (Gb/s)
IP Router	4016	125.31	2
NAT	3248	125.64	2
DiffServ	9114	85.10	1.6

The slice count presented here does not include four non-Click Gigabit Ethernet media access controllers that occupy 4580 slices together. Note that, for Xilinx Virtex-II Pro devices, each slice has two four-input LUTs, carry logic, arithmetic logic gates, wide function multiplexers and two storage elements. We observe that both the IP router and NAT have similar performance and both meet their desired throughput of 2 Gb/s (duplex) at a consistent clock rate of 125 MHz. However, for the DiffServ router, we were able to achieve a clock rate of only 85 MHz, resulting in throughput of 1.6 Gb/s. However, our goal in these first

experiments has been to obtain synthesizable code, and it is indeed the case that our implementations have scope for further optimization. The overhead of using the Cliff semantics in the above implementations, as compared to using flat HDL, are eight signals (wires) and five registers minimum for every element. The upper bound depends on the width of the data interface and how the synthesis steps imply the data interface. In practice, we found that the overhead is less than 2% of the total area of the designs we have looked at. This is an attractive trade-off.

Table 2 Memory Design Exploration

Memory organization	Area (# Slices)	Performance		
		Frequency (MHz)	Throughput (Gb/s)	Latency (Cycles)
Mem Org 1	4016	125.10	2	28-31
Mem Org 2	3460	126.64	2	39-46

In addition to the above three designs, we have also carried out a small design exploration of the memory organization space. We implemented two different memory configurations for the IP router design. The first memory organization is as described in Section 3, with a 344-bit wide data path. The second memory organization uses four single-ported BRAMs connected to all of the Click elements. The second organization does not use the data interface. The goal of this experiment was to illustrate that, as stated earlier, we can indeed plug and play with different memory organizations and quickly generate accurate implementations and related design data. From Table 2, we observe that the second memory organization has a lower area requirement and is also able to achieve a slightly higher clock rate at similar throughputs. However the advantage of first memory organization is smaller latency compared to the second one. Thus, one can conclude that for latency-constrained designs, the first memory organization should be used.

5. Related Work

Related work on mapping domain specific languages to embedded platforms can be found in the context of digital signal processing (DSP), graphics applications, and network processing. Two approaches based on mapping Matlab, a popular DSL for specifying and implementing DSP algorithms, to FPGAs are well known. Haldar et al [1] use a classic compilation of Matlab programs to a HDL and Hwang et al [2] use a library based mapping of Matlab programs to HDL.

In the context of network processing, there is no widely used DSL. However, in the academic community, Click has been used for research. Shah et al present a mapping of Click to the Intel IXP1200 network processor using Intel micro-engine C and assembly language [5]. Teja uses a graphical interface with underlying TejaC language for entering the design, and performs code generation for either the Intel IXP1200 or Broadcom network processors [8]. There has been no prior work in this domain on mapping a DSL on to a generic HDL, and then on to a platform FPGA. For exploring implementation alternatives for network applications, a critical first step is enabling the mapping of a DSL, such as Click, on to a platform FPGA – enabling design space exploration of a wide variety of architectural alternatives.

6. Conclusions and Future Work

As the complexity of applications increases, designing for ASIC becomes increasingly challenging and expensive, resulting in more system designers moving towards programmable platforms. A key element in the success of programmable platforms will be the ability to map applications easily and efficiently. The underlying programming model thus becomes a key enabler, and domain specific languages are fundamental to a top-down design process. In this paper, we have presented one such approach: mapping a networking DSL to a platform FPGA. The design time using the mapping of such a DSL to an FPGA is significantly lower.

Going further, we plan to incorporate corresponding ANSI C implementations of each of the Click elements, thus providing users with the ability of partition their implementation between hardware and software mapped to a platform FPGA with one or more embedded processors. In addition, we plan to raise the abstraction of some of the architectural elements – such as those related to memory organization – incorporated into Cliff, thus resulting in a DSL with an ability to refine implementations more smoothly. Domain specific optimizations based on compile-time analysis of DSL remains a topic of active research.

7. Acknowledgements

We thank Phil James-Roxby and Eric Keller for their useful and stimulating contributions both to discussions of this work and to its practical implementation.

8. REFERENCES

[1] M. Haldar, A. Nayak, A. Choudhary, P. Banerjee, "A System for Synthesizing Optimized FPGA Hardware from MATLAB," Proc. Int. Conf. on Computer Aided Design, Nov. 4-8, 2001, San Jose, CA.

[2] J. Hwang, B. Milne, N. Shirazi and J. Bloomer, "*System Level Tools for DSP in FPGAs*," in Proc. FPL2001, (Ed.s), 2001.

[3] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," ACM Transactions on Computer Systems, 18(3):{263-297}, Aug 2000.

[4] Virtex™-II Pro Platform FPGA Handbook (v1.0), Xilinx, January 2002.

[5] N. Shah, W. Plishker, and K. Keutzer. "NP-Click: A programming model for the Intel IXP 1200," In 2nd Workshop on Network Processors (NP2) along with HPCA-9, Feb. 2003.

[6] P. Hudak, "Building Domain-Specific Embedded Languages," ACM Computing Surveys, (4es): 196 (1996).

[7] E. Kohler, "The Click Modular Router," Doctoral Dissertation, Dept. of EECS, MIT, Feb 2001.

[8] Teja Technologies, Teja C: A C based programming language for multiprocessor architectures, White Paper, Oct 2003.