

# High-level programming of the FPGA on NetFPGA

Michael Attig and Gordon Brebner

Xilinx Labs

2100 Logic Drive

San Jose, CA 95124

{mike.attig,gordon.brebner}@xilinx.com

## ABSTRACT

The NetFPGA platform enables users to build working prototypes of high-speed, hardware-accelerated networking systems. However, one roadblock is that a typical networking specialist with a software-side background will find the programming of the FPGA to be a challenge because of the need for hardware design and description skills. This paper introduces G, which is a high-level packet-centric language for describing packet processing specifications in an implementation-independent manner. This language can be compiled to give high-speed FPGA-based components. An extension has been produced that allows these components to be dropped easily into the data path of the standard NetFPGA framework. This allows a user to write and debug packet processing functions at a high-level in G, and then use these on the NetFPGA alongside other components designed in the traditional way.

## 1. INTRODUCTION

Diversity and flexibility are increasingly important characteristics of the Internet, both to encourage innovation in services provided and to harness innovation in physical infrastructure. This means that it is necessary to provide multiple packet processing solutions in different contexts. At the application level, there is a need to implement deep packet inspection for security or packetization of video data, for example. At the transmission level, there is a need to keep pace with rapidly-evolving technologies such as MPLS and carrier Ethernet. This is in addition to support for evolution in protocols like IP and TCP that mediate between application and transmission.

Recent research in Xilinx Labs has led to four main contributions which, taken together, offer a flexible *and high-performance* solution to the problems posed by allowing increasing diversity in packet processing:

- Introducing G, which is a domain-specific high-level language for describing packet processing. G focuses on packets and is protocol-agnostic, thus supporting diversity and experimentation. In addition, G is concerned with the ‘what’ (specifica-

tion) not the ‘how’ (implementation). In other words, G describes the problem, not the solution.

- Using the rich — but raw — concurrent processing capabilities of modern Field Programmable Gate Array (FPGA) devices to enable the creation of tailored virtual processing architectures that match the individual needs of particular packet processing solutions. These provide the required packet processing performance.
- Demonstrating a fast compiler that maps a G description to a matching virtual architecture, which is then mapped to an FPGA-based implementation. This facilitates experimentation both with options for the packet processing itself and with options for the characteristics of the implementation. It also removes existing barriers to ease of use of FPGAs by non-hardware experts.
- Generating modules with standard interfaces that are harmonious with the Click [8] modular router framework. This facilitates the assembly of complete network node capabilities by smoothly integrating varied FPGA-based components, and by interfacing processor-based components.

The technology that has been developed is scaleable. The initial target of the research was networking at 10Gb/s and above rates, with current experiments addressing 100Gb/s rates. However, going in the other direction, the G technology is also applicable to the current NetFPGA platform, for single 1Gb/s rate channels, or an aggregate 4Gb/s rate channel. In order to enable seamless integration with NetFPGA, wrappers have been developed that allow the generated modules to be dropped into the standard packet processing pipeline used in NetFPGA designs.

In this paper, we first provide a short overview of G and the compilation of G to FPGA-based modules. We then describe how these modules can be integrated and tested on the NetFPGA platform. This is illustrated by an example where G is used to describe VLAN header manipulation, and this is added into the reference router datapath.

## 2. OVERVIEW OF G

The full version of the G language is targeted at specifying requirements across network processing. The initial implementation has focused on a subset of G that is concerned with expressing rules for packet parsing and packet editing, the area which has been most neglected in terms of harnessing FPGA technology to its best effect. Other aspects of G beyond this subset are still the subject of continuing research.

Although independent of particular implementation targets, a G description is mapped onto an abstract ‘black box’ component that may be integrated with other components to form complete packet processing — or more general processing — systems. The setting for such components is that of the Click modular router framework [8] (an aspect not covered in this paper). In Click, an individual system component is called an element, and each element has a number of input and output ports. This enclosing Click context means that a G description is regarded as describing the internal behavior of an element, and also declaring the external input and output ports of that element.

A G element description consists of three parts. The first part is concerned with declaring the nature of the element’s external interactions, that is, its input and output ports. The second part is concerned with declaring packet formats, and the format of any other data structures used. Finally, the third part contains packet handling rules.

Input and output ports in G have types, two of which will be introduced here. The packet type is one on which packets arrive (input) or depart (output), and this corresponds to the standard kind of (untyped) Click port. The access type is one on which read and/or write data requests can be made (output) or handled (input). This is provided to allow interaction between elements, separately from the passing of packets.

G has no built-in packet formats, since it is deliberately protocol-agnostic to give maximum flexibility. Packet formats and other data formats may either be declared directly in a G description or be incorporated from standard libraries through included header files. A format consists of an ordered sequence of typed fields. Types are scalar or non-scalar. The scalar types were selected to be those deemed of particular relevance to the needs of the packet processing domain. They are bit vector, boolean (true or false), character (ISO 8859-1), and natural number (in range 0 to  $2^{32} - 1$ ). For example, negative integers and real numbers were not included. Non-scalar type fields may be of another declared format, or be a list of sub-fields, or be a set of alternative fields. G itself does not explicitly differentiate between uses of packet fields (e.g., header, trailer, payload) — this is expressed through the actual packet handling rules.

The substantive portion of a G description provides the rules to be applied for packet handling. These refer as appropriate to the preceding declarations of ports and formats. In particular, the handler heading for the set of rules includes the name of the packet input port and the name of the input packet format. Prior to the rules themselves, local variables can be declared. These have a lifetime corresponding solely to a particular packet being handled. Thus, if several packets are being handled simultaneously, each has its own unique set of the local variables. In general, concurrency is maximized, here across multiple packets.

Packet handling rules implement the requirements of protocols for handling packets. When writing rules, the G user is liberated from having to think about when rules are applied, or how rules are applied. In particular, the G user does not have to be concerned with where fields occur within packets, or how packet data arrives and departs at input and output ports. Application of different rules is intended to be done as independently as possible.

In terms of the rules themselves, there are two categories: packet rules and external state rules. Packet rules are used to change the content or format of the packet, or to approve it for forwarding. There are four types of packet rules in the G subset being considered:

- **Set:** change packet fields
- **Insert:** insert one or more additional fields into the packet
- **Remove:** remove one or more fields from the packet
- **Forward:** indicate that the packet should be output after handling

For a simple forwarding function, combinations of set and forward rules are sufficient. For encapsulation or decapsulation, for example, insert or remove rules respectively are used also. External state rules are used to perform reading and writing of arbitrary data format values on external access ports. These rules can optionally specify additional read or write parameters (e.g., memory addresses, lookup keys, or function arguments, depending on the type of external element).

Guards provide for the conditional application of rules, and so form the basis for packet parsing and simple packet classification. The Boolean expression within a guard condition is always fully evaluated and tested for truth before the guarded rule will be applied. There can be a nest of disjoint guarded rules, in which each sequential guard condition is always fully evaluated and tested for falsity before a subsequent rule in the nest will be considered. In other words, the first rule in the nest with a true guard condition, or a final unguarded rule if none of the preceding rules have a true guard condition, is the one that is applied.

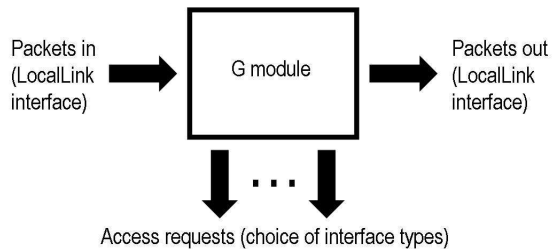


Figure 1: Generated module

### 3. COMPILATION OF G TO FPGA

#### 3.1 Underlying principles

The compilation of G towards an FPGA implementation is founded upon three important principles: (i) an expected low level of dependencies between rules in G descriptions; (ii) the creation of *virtual architectures* to implement the packet handling rules; and (iii) the generation of modules with standard interfaces that can be plugged together to create complete systems.

The first principle is derived from observation of the nature of packet parsing and editing for protocols. There is considerable independence in how packet fields are treated with respect to other packet fields. This gives packet processing a distinctive flavor when compared with normal data processing or digital signal processing. The fruit of the independence is to provide greater opportunity for concurrent operations, with consequent benefits for throughput and latency. In turn, the huge amount of potential for concurrent processing in an FPGA provides the physical support for this. The G language itself is designed to encourage the expression of independence in descriptions, something attractive to its user but also avoiding the need for too much ‘automatic discovery of parallelism’ magic in the compiler.

The second principle reveals an analogy between virtual networks and virtual architecture. Just as it is beneficial to implement problem-specific networks on programmable physical infrastructure, so it is beneficial to implement problem-specific microarchitectures on the programmable physical infrastructure of an FPGA. The G compiler builds a processing microarchitecture that matches the needs of the particular G description. This contrasts with the usual need to contort the problem to fit a fixed microarchitecture, whether expressing it sequentially in C for a normal processor, or warping it to fit the properties of a specialized network processor.

The third principle is that each G description is compiled to generate a module targeted at an FPGA, and that this module has standard interfaces. This is so that it can be integrated with other modules, either also generated from G or from other sources, to build a complete FPGA-based system.

Figure 1 shows a high-level schematic of the generated module. For this research, Xilinx FPGA devices have been the target technology, so the module interfacing was chosen for compatibility with standard modules produced by Xilinx. Packet input and output ports in the G description are mapped to module interfaces that use the LocalLink standard [12] for signaling packets word-wise between modules. Access ports are mapped to interfaces that can follow several different standards for accessing modules that support read/write requests.

#### 3.2 Compilation process

The generated module is not described directly in terms of the underlying FPGA resources: programmable logic gates, interconnect, and other embedded units. It is described in an intermediate form, expressed in a hardware description language, VHDL in the case of this compiler. The intent is to generate VHDL of a quality comparable with that achieved by hand design, in a much shorter time and a much more maintainable manner. The VHDL description is then processed by the standard design tools provided for FPGAs, which have had many thousands of person-years devoted to achieving high quality of results. In particular, these tools seek to harness all of the features of the specific target FPGA. Thus, the G compiler itself is decoupled from specific FPGA device detail, and works in partnership with an FPGA vendor’s specific processing tools.

Since a G description is abstracted from any implementation detail, additional input is supplied to the compiler. This input concerns the nature of the interfaces to the module. It includes structural details, such as word width and minimum/maximum packet sizes, and temporal details of required throughput. At present, the compiler reports on achieved latency and achieved FPGA resource count; in the future, these will also be supplied as targets to the the compiler. The main steps carried out by the compiler are:

1. **Parsing** of the G description, and of the additional implementation-specific information.
2. **Analysis** to determine dependencies through multiple uses of packet fields or local variables, or through multiple uses of external access ports.
3. **Partitioning** of rules into dependency-respecting clusters to form an optimal parallel processing microarchitecture.
4. **Scheduling** of rule applications within each cluster to respect arrival and departure times of packet data via word-wise interfaces, and the word-wise nature of access interfaces.
5. **Generating** a VHDL description of a module that implements the virtual packet-handling architecture, plus other inputs for the standard tools.

## 4. G DROP-IN MODULES

Generated G modules are a natural fit within the NetFPGA framework. Streaming packet interfaces enable G modules to be instantiated at various locations in the standard packet processing pipeline of the NetFPGA. However, to fully enmesh G modules into the NetFPGA environment, some infrastructure modifications were necessary. The goal was to make G modules indistinguishable from other NetFPGA library components. This allows the current NetFPGA implementation approach to remain unaltered, while enabling the inherent ability of the high-level simulation approach that G components bring. Essentially, development of G modules becomes an extension of the methodology for contributing NetFPGA modules [3].

A G component is packaged to appear as any other NetFPGA module, such as the *input arbiter* or *output port lookup* modules used in the reference router project. Figure 2 shows a wrapped G module. The G module is wrapped amongst various infrastructure pieces. An input-side protocol bridge converts from the NetFPGA bus signaling protocol to Xilinx’s LocalLink signaling protocol [12]. G modules can operate as pipeline stages, as they support cut-through processing mode. The streaming LocalLink interfaces enable this desirable feature. Once the G module begins to emit the packet, it is sent through the output-side protocol bridge that converts back to the NetFPGA signaling protocol from LocalLink.

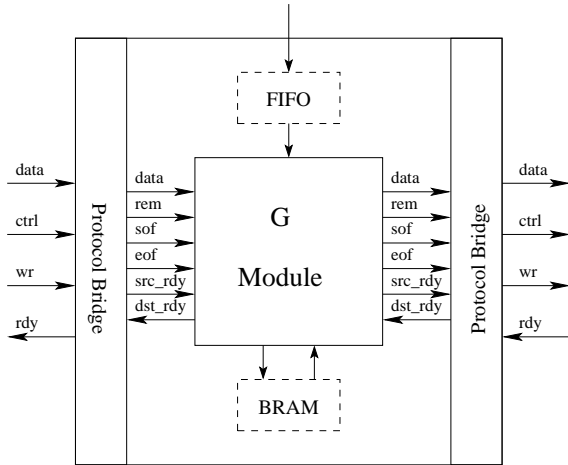


Figure 2: G Drop-in Module

The structure shown in Figure 2 is actually a top-level verilog wrapper. This wrapper instantiates the protocol bridges, the compiler-generated VHDL G module hierarchy, and the access components attached to the G module, such as registers, FIFOs, or BRAMs. This wrapper is auto-generated by a Perl script that scans the G module for interface signals.

## 5. EXAMPLE

The following section highlights the methodology in using a G module in the NetFPGA framework. For illustrative purposes, a simple VLAN extraction component will serve as the G component. The functionality of VLAN extraction is to remove VLAN tags attached to packet data that are no longer necessary to aid routing.

### 5.1 Step 1: Setup environment

The standard directory structure associated with the NetFPGA environment should be utilized. Users work out of the *projects* directory. Files that are developed for using G should be placed in a new directory, *gsrc*. This includes all *g*, *xml*, *fv*, testbench, and synthesis files. The expected structure is shown in Figure 3.

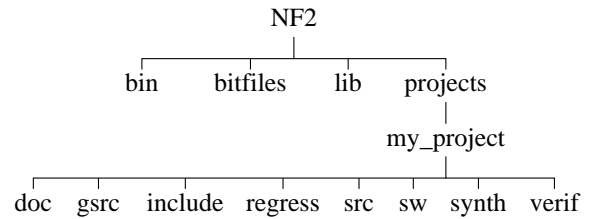


Figure 3: Environment directory structure

### 5.2 Step 2: Compose G

The first development step in utilizing G is to compose the G to describe the desired functionality. The entire G for the VLAN extraction example is shown below. A companion file is created to describe the characteristics of the interface ports, such as the width.

```

element vlan_extract{

    input packetin : packet;
    output packetout : packet;

    #define VLAN_TYPE      0x8100
    #define NON_VLAN_TYPE 0xabcd

    format MyPacket = (
        IOQHeader : (
            dst_port_one_hot : 8,
            resv             : 8,
            word_length     : 16,
            src_port_binary : 16,
            byte_length     : 16
        ),
        type : 16,
        VLAN_header : 32,
        : *
    );

```

```

handle MyPacket on packetin {

    [type == VLAN_TYPE]{
        remove VLAN_header;
        set type = NON_VLAN_TYPE;
        set IOQHeader.word_length =
            IOQHeader.word_length - 1;
        set IOQHeader.byte_length =
            IOQHeader.byte_length - 8;
    }

    forward on packetout;
}
}

```

The packet format for this example defines the NetFPGA control header (*IOQHeader*), the *type*, and the *VLAN\_header*. The remainder of the packet is indicated using the *\** field to mean that the *VLAN\_header* is the last field of the packet this G module is interested in. The packet handler for this example is relatively simple. The *type* field is checked. If it is a VLAN type, the *VLAN\_header* is removed, and the control header is updated to reflect that there is one less word in the packet. Finally, the packet is forwarded on the output packet interface.

### 5.3 Step 3: High-level G simulation

The functionality of the written G description must be verified. The G development environment comes with two tools to aid with high-level G description verification. The first is *gdebug*. This tool steps through a G description, allowing the user to select from a list of available actions to invoke. (Recall that G descriptions are declarative rather than imperative, so multiple actions could potentially be executed at any given debug step.) The second tool is *gsim*. This tool reads *packet instances* corresponding to input and produces the resulting output. The *gsim* tool is the main simulation vehicle for G descriptions.

Packet instances are created through use of the *gfv* language (G format value). This language is used to specify the values that packet fields can take. An example of this is shown below.

```

formatvalue ioq = (
    0x20 : 8,
    0 : 8,
    4 : 16,
    1 : 16,
    0x20 : 16
);

```

```

formatvalue test1 = (
    : ioq,

```

```

    [0x8100 | 0xaaaa] : 16,
    0xfeedcafe : 32,
    0x000011112222 : 48,
    0x1011121314151617 : 64,
    0x18191a1b1c1d1e1f : 64
);

```

This format value description will create two packet instances to input. One instance will fill the *type* field with *0x8100* and the other instance with *0xAAAA*. The *gsim* tool reads these packet instances and simulates the expected output. The processed output for the first packet instance is shown below. Note the inclusion of field names matched to bit patterns in the output. This aids quick verification. Simulation with *gsim* is a fast process, enabling quick functional verification cycles.

```

formatvalue test1_1 = (
: (
    0x20 : bit[8] /* dst_port_one_hot */,
    0x00 : bit[8] /* resv */,
    0x0003 : bit[16] /* word_length */,
    0x0001 : bit[16] /* src_port_binary */,
    0x0018 : bit[16] /* byte_length */
) /* IOQHeader */ ,
0xabcd : bit[16] /* type */,
0x000011112222 : bit[48],
0x1011121314151617 : bit[64],
0x18191a1b1c1d1e1f : bit[64]
);

```

### 5.4 Step 4: Embed in NetFPGA pipeline

This G module fits between the input arbiter and the output port lookup modules in the reference router pipeline, as shown in Figure 4. G modules are currently limited to support single input and output packet streaming interfaces, so G modules can only be placed between the input arbiter and the output queues in the reference router pipeline.

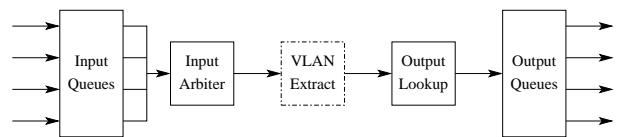


Figure 4: VLAN extraction G module included in reference router pipeline.

To include the G module, the *user\_data\_path* verilog top-level is modified to instantiate the G wrapper and wire it into the data path. The *user\_data\_path* file found in *NF2/lib/verilog/user\_data\_path/reference-user\_data\_path/src/* is copied into and modified within the *src* directory of the project. Recall that the G wrapper instantiates protocol bridges, the G module itself, and any access memories that the G module uses.

## 5.5 Step 5: Simulate system

The simulation environment for NetFPGA is at the HDL level. The G compiler produces synthesizable and simulation-ready VHDL, so the simulation scripts distributed with a standard NetFPGA distribution were modified to incorporate simulation of G components. These scripts are run in the same manner as any other NetFPGA project. Test data and regression tests are also created in a similar way.

## 5.6 Step 6: Implement

The build scripts have also been modified to accommodate the synthesis of G components. These scripts are run as if a standard NetFPGA project is being constructed.

## 6. RELATED WORK

Various domain-specific packet processing languages have been proposed. PacLang [4] is an imperative, concurrent, linearly typed language designed for expressing packet processing applications. FPL [2] is a functional protocol processing language for performing complex packet classification and analysis. PPL [7] is another functional programming language for packet processing, oriented towards TCP/IP as well as deep packet inspection. SNORT [10] is a well-known intrusion detection system based on complex rule sets that specify the processing of packet headers and the matching of patterns in packet payloads.

There has been considerable research and development on compiling C to FPGA (so-called ‘C-to-gates’). In general, C subsets and/or C annotated with pragmas must be used. A focus has been on loop unrolling as a means of introducing concurrency, particularly for digital signal processing applications. Examples include Catapult C [9], Handel-C [1], SpecC [5], Streams-C (now Impulse C) [6], and Synfora C [11]. To date, there has been little evidence of this style of technology being of benefit for high-speed packet processing.

## 7. CONCLUSIONS AND FUTURE WORK

This paper has presented the G language and the efficient compilation of G to FPGA-based modules. In particular, it has presented a new framework which allows G modules to be easily incorporated into NetFPGA-based system designs. Taken together, these advances will ensure greater ease of use for networking experts who are not familiar with traditional FPGA design approaches.

The paper focused on just a subset of the full G language. Subsequent publications will explore other aspects including, for example: describing relationships between packets and hence functions like segmentation and reassembly; structuring of packet handlers and hence

modularity between protocols; and support for more flexible packet formats. Ultimately, G is intended to cover the full spectrum of packet processing.

Future work will include seeking better integration between the standard software Click environment and an FPGA Click environment that has been developed around G. The linkage to the NetFPGA platform now offers the hardware contribution to developing an ecosystem around Click and G. It provides a convenient basis for interfacing software Click elements with FPGA-based elements described in G, thus enabling experiments where time-critical data plane functions benefit from the speed of the FPGA.

## 8. REFERENCES

- [1] Celoxica. Handel-C. [www.celoxica.com](http://www.celoxica.com).
- [2] D. Comer. *Network Systems Design Using Network Processors, Agere version*. Prentice Hall, 2004.
- [3] G. A. Covington, G. Gibb, J. Naous, J. Lockwood, and N. McKeown. Methodology to contribute netfpga modules. In *International Conference on Microelectronic Systems Education (submitted to)*, 2009.
- [4] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In *Proc. ESOP 2004*, pages 204–218, Barcelona, Spain, Mar. 2004. Springer-Verlag LNCS 2986.
- [5] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer, 2000.
- [6] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 49–56, Napa, CA, Apr. 2000.
- [7] IP Fabrics. Packet Processing Language (PPL). [www.ipfabrics.com](http://www.ipfabrics.com).
- [8] 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, August 2000.
- [9] Mentor. Catapult C. [www.mentor.com](http://www.mentor.com).
- [10] M. Rosesch. SNORT — lightweight intrusion detection for networks. In *Proc. LISA 1999*, pages 229–238, Seattle, WA, Nov. 1999.
- [11] Synfora. Synfora C. [www.synfora.com](http://www.synfora.com).
- [12] Xilinx. FPGA and CPLD solutions. [www.xilinx.com](http://www.xilinx.com).