ENHANCING GNU RADIO FOR RUN-TIME ASSEMBLY OF FPGA-BASED ACCELERATORS

Richard Stroop and Peter Athanas
Virginia Tech, Blacksburg, Virginia, United States; {blvninri,athanas}@vt.edu

ABSTRACT

An enhanced GNU Radio flow is presented that seamlessly augments the standard GNU Radio framework with modules that reside in FPGAs, yet preserves the GNU Radio dynamics by providing full-custom radio hardware/software structures in seconds. By delegating portions of a GNU Radio flow graph to networked FPGAs, a larger class of software-defined radios can be implemented. Assembly of the signal processing structures within the FPGAs is accomplished using an enhanced flow where modules are customized, placed, and routed in a fraction of the time required by the vendor tools. With rapid FPGA assembly, a GNU Radio designer retains the ability to perform "what-if" experiments, which in turn greatly enhances productivity.

1. INTRODUCTION

Software defined radios (SDRs) have changed the paradigm of slowly designing custom radios, instead allowing designers to quickly iterate designs with a large range of functionality. With the help of environments like the open-source project, GNU Radio, a designer can prototype radios with greatly improved productivity. The inherent reconfigurability of CPUs makes them useful tools for realizing rapid development of radios, but their inability to process large amounts of data at once has always limited their use for high-throughput signal processing [1]. Even the most sophisticated processor can only achieve limited data rates and it becomes difficult to meet latency requirements as more processes are added.

Due to the software performance limitations in the GNU Radio framework, only radios with a certain level of complexity can be realized. For SDRs to become more prevalent in radio prototyping and development, accelerators are needed to address the high-throughput and computationally intensive portions of the radio. FPGAs are nicely suited for this needed acceleration; however, they do have properties that make them undesirable for rapid SDR development. Due to their long compile times, adding FPGAs into a flow designed for only software brings prototyping to a crawl. Furthermore, special hardware design skills are typically needed, and hardware design languages -- far removed from radio design environments -- are used for building the accelerator structures.

In this paper, enhancements to the GNU Radio development environment are presented that provide an easy way for adding hardware acceleration to a software radio. GNU Radio operates by piecing together software modules, either graphically or via a Python script, calling different functions on data that stream through a flow graph. In the enhanced flow, one or more modules in a design graph can be designated to reside in one or more FPGAs. The concept of module-based assembly is preserved in the enhanced flow, where hardware-centric modules are pre-compiled relocatable objects that are placed in a library, and can be retrieved, placed, and routed with other modules in a designated device when called upon by the GNU Radio run-time engine. Unlike other approaches, the FPGA compilation flow has been altered in favor of rapid assembly (place and route) at run time. The communication interfaces are pre-implemented for the FPGA so that radio hardware designers do not have to develop their own system. The data are abstracted to types that GNU Radio software blocks use and presented in a standard way to the designer, regardless of the interface used to receive the data. In this way only the basic signal processing blocks need to be designed in order to see them work within the framework of GNU Radio. The signal processing blocks that are used often can be added to a community-based library for easy reuse. The slow compile times are overcome with a modified back-end FPGA assembly suite. The project is titled `GReasy`, GNU Radio easy, since implementing signal processing with an FPGA accelerator is as easy and fast as creating a normal flow graph in GNU Radio. To the radio designer, the complexity of the underlying hardware is abstracted away, making it appear as if everything compiles and runs in software, allowing many iterations to be realized quickly. Radio design can continue at the speeds that GNU Radio designers are accustomed to but with the range of possible waveforms and general functionality extended.

1.1. Organization

This paper is organized as follows; Chapter 2 discusses the background to this work, with a focus on the hardware involved. Related works are mentioned to show the

necessity of these improvements. Chapter 3 explains the enhancements made to GNU Radio to add FPGA accelerators without some of the drawbacks of normal hardware additions. Chapter 4 describes how the hardware is built with a focus on quick designs rather than an optimal placement. Chapter 5 presents a demonstration of the basic functionality of this new model. Chapter 6 concludes the paper.

2. BACKGROUND

Despite being called *software*-defined radios, a real radio can never be completely built with only a processor. There must be an antenna and front-end capable of receiving/transmitting a signal and possibly doing data reduction and front-end filtering to transform that signal into a format usable by a software processor. A common way to handle this processing requirement is by adding a small amount of appropriate hardware back into the flow.

2.1. GNU Radio

GNU Radio is an extensive framework that enables many complex designs to be prototyped and built with only a General Purpose Processor (GPP) and off-the-shelf radio hardware. It is open source and commonly used due to its wide range of available radio blocks [2] and has a simple user interface. GNU Radio builds SDRs as flow graphs that can either be expressed in a Python script or graphically wired together with a tool called GNU Radio Companion (GRC) [3]. All of the signal processing is done in C++, a common language for software development, so that adding another custom block to GNU Radio is simple [4].

Each block runs a signal-processing task in a dedicated thread that passes data to other threads through shared memory buffers. The GNU Radio scheduler handles the threads and data with little overhead, but requires the blocks to be written in their format and only for GPPs. Currently the only supported off-loading of the signal in the flow graph is for transmission and reception of analog signals with a radio front-end. No other processing hardware can be added easily but GNU Radio does allow for rapid software-only prototypes.

One common hardware device for SDRs is a Universal Software Radio Peripheral (USRP). There are different iterations designed by Ettus Research [5] based on what a user needs. All of them are relatively inexpensive, as far as hardware goes, so they are common in research and among SDR hobbyists. With the use of replaceable daughter cards the hardware can be quickly changed to handle different frequencies for transmitting and receiving. They are well supported in the popular GNU Radio community and can be connected either by USB or Gigabit Ethernet for more bandwidth. When used as receivers, the USRPs convert analog signals to digital signals and decimate them to appropriate sample rates before passing the signal to a host computer for additional signal processing. When used as transmitters, the USRPs perform the inverse of the functions described for reception.

The USRP is an example of a required hardware accelerator to provide an RF front-end. It contains a motherboard, multiple ADCs and DACs, and a million gate FPGA [6]. High-speed general-purpose operations are done on the embedded FPGA such as decimation, interpolation, and digital conversions [7]. By moving some processing to hardware, GNU Radio has already enabled many real-time radio prototypes [8].

2.2. FPGAs

Although there are many powerful options in the field of signal processing, FPGAs provide a reconfigurable means of handling intensive processing tasks. FPGAs consist of a large set of hierarchically connected logic blocks that are re-programmable to any implementation that can be created with a hardware description language. Where GPPs do everything in sequence, FPGAs can implement large parallel operations and deep computational pipelines, which make them indispensable in the SDR domain [9].

A typical FPGA compilation flow involves transforming a hardware description into a binary image, or *bitstream*, that is used to program a device. This is a multistage process that has many variations, but all of which take a large amount of time. The *Tools for Open-source Reconfigurable Computing* (TORC) are useful for user manipulation of EDIF and XDL files as well as bitstream packets [10]. TORC provides a powerful API for exploring alternative ways of processing FPGA designs. By itself, it is not a method of FPGA assembly, yet with intricate knowledge of the Xilinx flow, custom tools can be created with different objectives in mind. The Virginia Tech `qFlow/tFlow` project uses TORC as a foundation to create an enhanced Xilinx back-end bitstream generation process with the primary objective of rapid compilation [11].

2.3. Related Work

GNU Radio has seen many iterations and enhancements throughout its development. Due to the large community, there are constantly software-processing blocks that are being produced and new methods of implementing radios faster are being shared. Developments more concerned with this paper are those that have attempted to add hardware to a largely software based design environment.

Many projects have been developed to make use of FPGAs for software-defined radio acceleration. These projects include the Kansas University Agile Radio [12], the Japanese National Institute of Information and

Communications Technology SDR Platform [13], and the Berkeley Cognitive Radio Platform [14] to name a few. These implementations all required using special boards and software for communication rather than augmenting already available systems. They were also limited by the slow compile times of FPGAs, although their gains in performance were seen as worth this cost.

FPGAs have not been added to stock GNU Radio yet as "hardware is strictly not part of GNU Radio" [2]. There is an FPGA on the USRP devices that can be modified [15] for custom applications though. With the recent increase in FPGA size on the newer devices these customizations have become more popular. Unfortunately the changes made are permanently left on the board and run before every flow graph receiving data from the USRP. All flexibility is lost and their operation is not presented to the user in the flow graph.

3. GNU RADIO ENHANCEMENTS

Changes were made to the GNU Radio software distribution and additional capabilities were integrated into it to facilitate the building of attached hardware. The normal GNU Radio functionality is maintained, yet slight changes to the system have been made to accommodate distributed hardware and more flexible communications, and are outlined in this section.

3.1. GNU Radio FPGA Extension Class

The GNU Radio module library is organized into a set of well-defined classes. There are separate classes for filters, operators, input/output, converters, and more. The entire structure is set up to allow easy addition of new processing blocks simply by adding another class. The classes inherit all of the necessary block information from GNU Radio and thus never create new dependencies within the rest of the system. This same model was used to develop a hardware class that would fit alongside all of the other blocks.

Prior work was done to develop an auxiliary FPGA (*afpga*) class for GNU Radio [16]. This class contained an *afpga_source* and *afpga_sink* that mimicked the behavior of a USRP2 source and sink using raw Ethernet frames. The only difference was that an *afpga_sink* sent a packet to a connected USRP2 that redirected the stream of data to the FPGA. This paradigm was changed because it did not allow for the user to have control over how hardware was added to the FPGA. The benefits of the new flow can be seen in Figure 1. The legacy blocks evolved into the new *afpga_in* and *afpga_out* blocks of the current system. Instead of presenting data as disappearing into a sink, the *afpga_in* block presents data as entering an FPGA. The new block enables GNU Radio data from the computer to be sent to the FPGA as well as from the USRP2, and even from other FPGAs. The *afpga_out* block represents the output of a

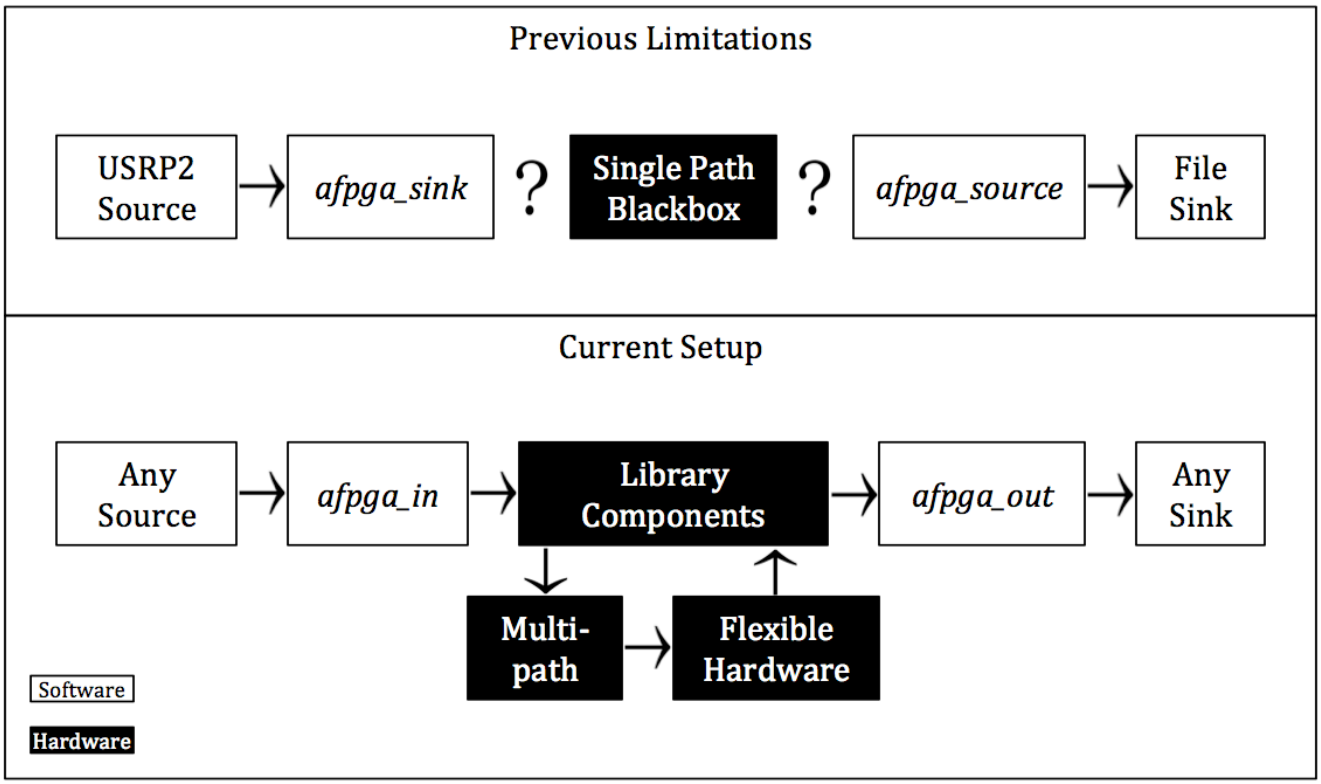

Figure 1. Previous vs. Current *afpga* Model

controllable system rather than an unknown source. The standard GNU Radio complex I/Q data format is included as an output type so that further processing can occur on the host if desired. In this way hardware and software blocks can go into the same flow graph and communicate with one another. The only limitation is that hardware blocks can only be placed within *afpga_in* and *afpga_out* blocks, and regular GNU Radio blocks must be placed outside of them. If the functionality of a software block is desired on the hardware, currently it must be written in HDL and added to the *afpga* library.

The real shift in how hardware is represented in GNU Radio comes with the addition of signal processing *afpga* blocks. As with the GNU Radio paradigm, arbitrary signal processing chains are composed by connecting various *afpga* blocks. By preserving this assembly paradigm, the designer is free to build hardware, and have it interact with other hardware or software. Since multiple paths are supported, modules can communicate to one or more other blocks that may be connected to it. This scheme lends itself to improved prototyping because it now has hardware blocks that are free to operate the same way GNU Radio already treats software blocks. Presenting the hardware to radio designers in a familiar and flexible fashion is only the first step to enhancing this SDR environment.

With GRC, a completely visual representation of the software and hardware can be built. By enabling a visual component, prototypes can be presented for the designers in a clearer way. All of the current *afpga* blocks have GRC representations allowing them to be managed visually. There are also scripts in place to automatically add newly registered *afpga* blocks to the GRC repository for dragging and dropping into designs. It should be noted that if a designer does not want to use GRC, they could continue using the GNU Radio Python scripts without losing any functionality of this work.

3.2. Core Code Changes

The changes to *afpga_in* and *afpga_out* for GRC are purely visual. These modules are still sources and sinks within the core of GNU Radio. Since the FPGA processing blocks are not executed on the host machine, the data stream does not necessarily need to be handled by the host computer. In order to make this work, the GNU Radio scheduler is ignored for these blocks. When two blocks are connected from a GNU Radio script, the function `gr_flowgraph::connect(source, destination)` is run. This does a check to make sure the modules have the same type and are not already connected. Once this is done the modules are added to the scheduler to be run later. Since no software data is being sent through the hardware blocks, they are not scheduled. The process for ignoring the FPGA blocks is currently rudimentary: if both the source and the sink contain *afpga* in the name, then they are not scheduled. Also if a USRP2 block is connected to an *afpga* block, neither are scheduled. The USRP2 was modified to send directly to an FPGA, which clears an overload of traffic to the host machine. This means that the block on the host is no longer doing any processing, which is why it too is no longer scheduled. If an FPGA block connects to another FPGA block, since GNU Radio is no longer recognizing this connection, it is logged in 'fpga_connections.txt'. These connections will be used to build a netlist and eventually a bitstream for every FPGA in the flow graph.

GNU Radio does not run the unscheduled hardware blocks, but it does call their constructor functions. Each block's constructor opens a file called 'edif.dat' and appends one line of information to it (Figure 2). This line is a condensed description of the ports available to the Verilog module (Figure 3). The file is used by the program `EdifWriter` to build a completely Xilinx compatible EDIF file. `EdifWriter` is part of the `qFlow` package that will be presented soon [11]. The information that is written by each block contains a unique name for that block as well as all of the port information. The ports that are buses are described as ARRAYS with a certain DIRECTION, NAME, and SIZE. The ports that are only one bit are described as PORTS with just a DIRECTION and NAME. This means that most data lines are described as ARRAYS and the clock/reset lines are usually PORTS.

```
Cell;zb_radio;ZB3;Array;output;out;33;Array;
    input;in;33;Port;input;rst;Port;input;clk;
```

Figure 2. Sample Module edif.dat Line

```
module zb_radio (
    output reg [32:0] out,
    input [32:0] in,
    input rst,
    input clk
);
```

Figure 3. Verilog of Sample Module

Within the framework of GNU Radio, all of the blocks are created, connected, and then run. The creation process yields an 'edif.dat' file, and the module connections are recorded in the file 'fpga_connections.txt'. After all of this is done, the flow graph is started. This normally calls the `start()` function on all of the blocks that have been scheduled, but code was inserted just before this step to assemble and program the necessary FPGAs first. The code

was placed into a separate C++ file called 'edif_connector.h' so that modifications could be made without disrupting more of the GNU Radio core.

The `connect_edif()` function starts by cleaning up 'fpga_connections.txt'. This means clearing duplicates and moving valid information to a file called 'connections.txt'. Since 'fpga_connections.txt' is appended to, it has to be deleted after each run or the next run will contain all of the information from both runs. Using this connection information, the function builds a netlist for each FPGA. The different FPGAs are identified by different base MAC addresses associated with the *afpga_in* and *afpga_out* blocks.

After all of the block's CELL information is added to the file, the connection information is added. The connections are denoted as either NET or LOOP. A NET is a one-bit connection that declares a NAME for itself and then points to the bits that it is connecting. The bits are identified by pointing to a CELL with a certain INSTANCE and then to the desired PORT. If the single bit being connected is part of an ARRAY of wires, then the INDEX is also given. Otherwise a negative one INDEX tells the code that the PORT is only one bit wide and should be treated as such. If more than one bit should be connected, a LOOP is used instead. A LOOP also starts by declaring a NAME but includes a SIZE to make sure that everything connecting to it has the same width. After that, all of the wires are identified by pointing to a CELL with a certain INSTANCE and then to the desired ARRAY.

Every CELL is automatically given a NET that connects the clock and a NET that connects the reset to a global clock and global reset respectively. If a NET or LOOP has the same name as another NET or LOOP on a different line, they will be automatically concatenated by `EdifWriter`. This allows for multiple lines containing the same clock and reset information, but can be used to connect a complicated wire set in the future. A complete 'edif.dat' file showing the modules and connections used for the implementation of this paper is shown in Figure 4.

It is necessary to know that these `edif.dat' files contain all of the same information as a Xilinx EDIF file but are presented in an easily readable, manipulatable, and compact format. `EdifWriter` parses this information and uses TORC to build the more complex netlist. Although these files are automatically generated and run, they can be modified or written from scratch by any user who wishes to describe a netlist in a simpler format before running the Xilinx or `qFlow` tools, which require an EDIF.

3.3. Fast Bitstream Creation

Once a connections list is built, one or more bitstreams must be generated in order to actually program any of the FPGAs. The remainder of the code mostly calls external scripts to

```
Cell;zb_radio;ZB3;Array;output;out;33;Array;
    input;in;33;Port;input;rst;Port;input;clk;
Net;rst;blacktop;BT0;rst;-1;zb_radio;ZB3; rst;-
    1;
Net;clk;blacktop;BT0;clk;-1;zb_radio;ZB3;
    clk;-1;
Cell;blacktop;BT0;Array;input;in0;33;
    Array;input;in1;33;Array;output;out0;33;
    Array;output;out1;33;Port;input;rst;
    Port;input;clk;
Loop;BT_in_00;33;blacktop;BT0;in1;
    zb_radio;ZB3;in;
Loop;afpga_zb_radio_ZB_3_wire_0;33;
    zb_radio;ZB3;out;blacktop;BT0;out0;
```

Figure 4. Sample edif.dat with Connections

accomplish this goal. Scripts were used for two reasons. The first is that the tools being used are still in active development as a separate project. It would be impossible to incorporate them into the core of GNU Radio. The second reason is that these scripts are easily modifiable and can be run outside of the GNU Radio framework to interface with the tools. This allows any user or program to take an 'edif.dat' file and run through the process of putting a bitstream on an FPGA. Also if GNU Radio fails, the process can be picked back up from the scripts without running everything again.

The first script is called 'edif', which takes one input: the base MAC address of the FPGA it is building. This calls `EdifWriter` on the respective 'edif.dat'. Once a true EDIF is created, a checksum is generated with `crc32` [17] to represent the contents of the FPGA. This checksum is stored on the FPGA so that it can be requested later to determine if anything has changed. If the FPGA has never been programmed, then there is no checksum on it and the next script will be called. If it has been programmed before with the exact same netlist, in the case where GNU Radio is run twice with only software changes occurring, then the rest of the scripts are unnecessary and not run for this FPGA.

The second script is called 'qflow', which actually builds a bitstream using a rapid modular based assembler called `qFlow`. `qFlow` implements a custom placer that quickly decides where to place pre-synthesized modules on any FPGA. It then makes use of Xilinx's router to wire everything up. This routing is still optimized and thus takes around two minutes to complete with the shortest paths.

The same script can be modified to call `tFlow`, or Xilinx tools based on what the designer needs. `tFlow` has a little bit more information about the organization of the Virtex 5 family, so it is able to do bit-wise manipulation to place already routed versions of the modules on the FPGA. It then does a small amount of routing at the bit level to

connect the modules that were placed. This process is fast for building a working bitstream. It requires that all of the pieces be compiled to the bitstream level in advance, but all of the blocks in the current library have been registered with `tFlow` to make this possible. The downside to `tFlow` is that it requires an intimate knowledge of the FPGA family that is being worked on.

GNU Radio is a system that has a quick turn around time, this means that the hardware needs to compile as quickly as the software so that prototyping is not hindered. The commercially available Xilinx tools offer the most optimized, and often only, way of compiling a hardware design for their FPGAs. The price for this optimization is a long build time, which on large systems can take over a day to complete [18]. By relaxing the placement optimization, `qFlow` is able to build a basic radio design in around two minutes. By removing the routing optimization and utilizing bitstream manipulation, `tFlow` is able to build a basic radio in around twenty seconds. The script could also run the original Xilinx tools commonly used today to produce an optimal bitstream at the cost of a long run time. No matter which process is run, the final output of the 'qflow' script is a bitstream for an FPGA.

The final script called 'program', places the bitstream that was just built onto the appropriate FPGA. Currently this is accomplished using a tool called `Impact` provided by Xilinx from the command line.

Once every script has been run, the code moves on to another FPGA if one exists. All of the scripts are run again for each FPGA. The final step is sending control data to the FPGAs to tell them where they should direct data. This control data, along with the checksum data from earlier, are sent using raw Ethernet packets. Most commonly the data are sent back to the host machine but can be directed to another FPGA or USRP2 or any other system that is listening on the network. When all of the operations in the `connect_edif()` function are completed successfully, it returns a true value allowing the rest of the program to start the software blocks. If the program detects errors it will stop the flow graph and throw a runtime error. The function can also determine that there are no appropriate FPGA blocks to run and allow GNU Radio to run normally with only software blocks.

4. HARDWARE

To facilitate the easy integration of hardware with GNU Radio, certain steps are taken on the hardware end to manage communication. By organizing the hardware in a certain way, the time it takes to assemble is significantly reduced. The most beneficial modification to the hardware

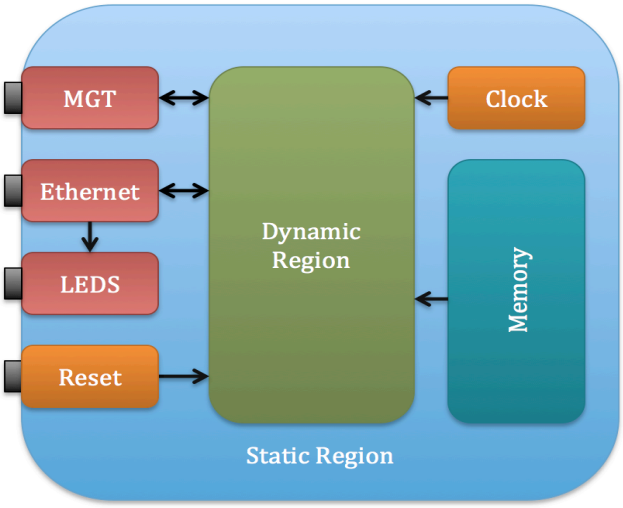

Figure 5. Static with I/O buffers, Ethernet and Dynamic Region

is the segregation of static and dynamic regions on the FPGA (Figure 5). There are a few core modules that are necessary for every design to communicate with GNU Radio. These make up the static region, which never changes and thus does not have to be rebuilt every time. This saves a large amount of time and is similar to the concept of partial reconfiguration where only a section of the FPGA is reprogrammed. The difference in this system is that a whole bitstream is still built and programmed, but it is done faster with a custom assembler that integrates the static and dynamic regions.

Black boxes are used in the Xilinx tool ISE to provide the ability to separate modules from the static region and dynamic design. The static region is hard coded and used for communication interface modules. The top of the dynamic region is a black box that is named *blacktop* since as a black box it appears to be the top of all of the modules being placed on the FPGA. All of the signal processing modules are also expressed as EDIF black boxes so that *blacktop* only has to see the connections between them. The dynamic region is what 'edif.dat' represents and covers effectively everything that is not the static region. The outer layer of the dynamic region is represented by the CELL called 'blacktop' with INSTANCE name BT0. Each path is represented by an `in` and `out` ARRAY. The `clock` is provided by a PLL in the static region, and the `reset` is tied through the static region to a physical button on the FPGA.

To make this flow more desirable, there is a library of hardware components in development currently. The process is constructed so that the community can easily contribute to the hardware library development. A set of standards is being used to ensure that all of the blocks can work properly together on the targeted hardware.

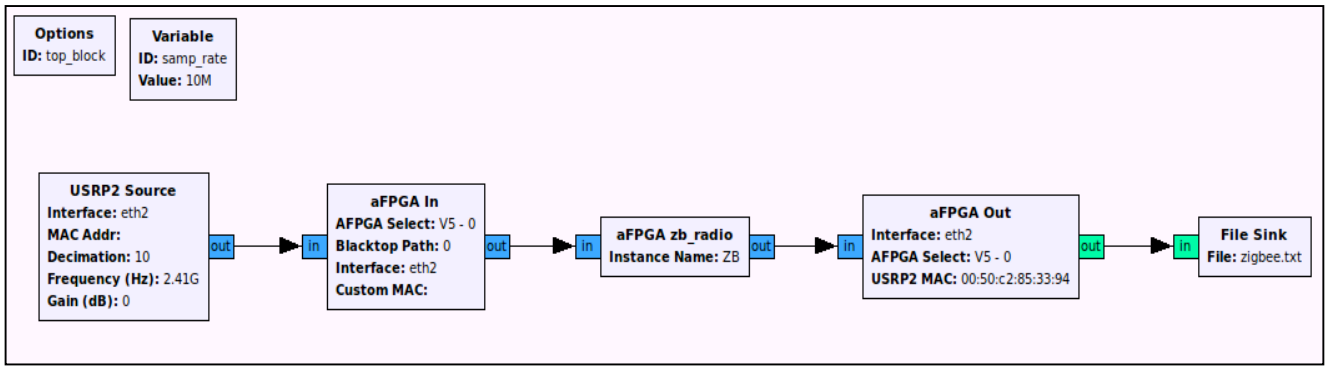

Figure 6. ZigBee Flow Graph in GRC

5. IMPLEMENTATION

The enhancements to GNU Radio allow FPGA blocks to be "dragged and dropped" into flow graphs and still keep the ability to perform 'what-if' experiments. The rapid iteration of designs with hardware makes the prototyping of more complex radios feasible. But in order to build the hardware a library has to exist.

Using a library of pre-built and pre-registered hardware blocks, a radio can be stitched together for an FPGA just like software designers are used to doing. When this library has grown into the size that GNU Radio software blocks are at now, then any designer can pick up the system and prototype applicable radios. The currently available blocks are limited to the ones built for demonstrating proofs of concept. More are being constructed at Virginia Tech to increase the basic available block library.

A demonstration of the enhanced GNU Radio is run within a virtual machine running on a MacBook Pro with a 2.3 GHz Core i7 processor, 8 MB shared level 3 cache, 8 GB of 1333 MHz DDR3 SDRAM, and a 251 GB Apple SSD. The software is not limited to one specialized computer configuration -- any computer that can run GNU Radio can also run `GReasy`. The MacBook is used here since it offers portability and easy access to the attached hardware. The FPGA used is the Virtex 5 on the XUPV5-LX110T development board, but any FPGA in the supported families with an Ethernet adapter can be used.

A ZigBee demonstration was done because it is a complex standard that can benefit from hardware acceleration. The ability to add custom hardware blocks to `GReasy` lets the designer view what is being placed on the FPGA and control how it is connected. In Figure 6 the USRP2 connects to the *afpga_in* block, which connects to the ZigBee demodulator, which then connects to the *afpga_out* block, which pipes the decoded and demodulated stream to a file. The USRP2 is set to run at a center frequency of 2.41 GHz with a decimation of 10 and unity gain. The USRP2's constructor sends these settings over Ethernet and the host computer is set up as the receiver for data. `GReasy` needs a USRP2 MAC address in order to redirect network traffic from the USRP2 to an FPGA instead. The hardware setup can be seen in Figure 7.

The *afpga_in* block is set to run *FPGA-0 on eth2*, which means that it will program the first board available using the second Ethernet interface on the computer. The 'Blacktop Path' is set to 0 so that `qFlow/tFlow` can connect blocks to the first path in 'blacktop', which produces an output stream to the Ethernet. The only option for the ZigBee demodulator (*afpga_zb_radio*) is to set the instance name. This name is simply a way of identifying which blocks are which in the final EDIF created by GNU Radio; it is not necessary to set these names for proper operation. The *afpga_out* block is also running on FPGA-0 across the second Ethernet interface. Its output is green in GRC since it is decoding the data as an integer containing four characters. In GRC, the software types have to match, so the file sink is green as well for this example application. GNU Radio handles the conversion of the data to the proper type before exposing it to the software blocks.

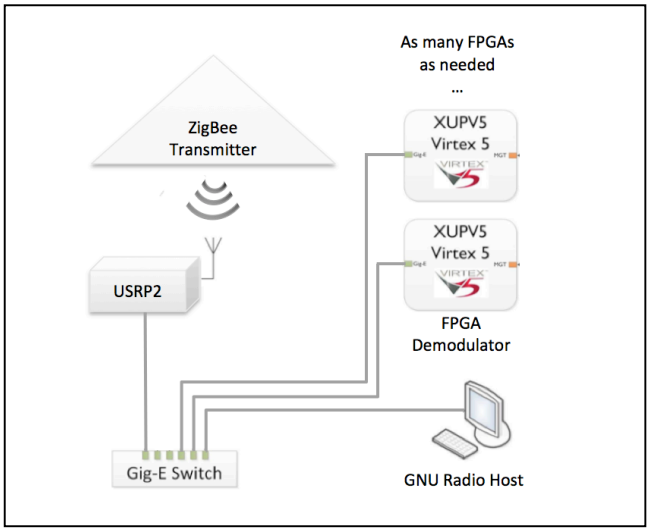

Figure 7. ZigBee Hardware Setup

The compile times for FPGA hardware in this prototyped radio have satisfied the "instant gratification" experience of GNU Radio. The addition of hardware does not slow down the design process and only enhances the number of radios that can be successfully implemented.

Table 1 shows a comparison of the time it takes to build the ZigBee radio using the Xilinx tool ISE against the custom tools `qFlow` and `tFlow`. For the custom tools, Synthesis is only required during the first run of a design that changes the internal logic of a hardware block (which should not happen often). Unlike the current tools, this also only forces a synthesis of one block instead of re-synthesizing the entire design. ISE still runs Synthesis and Map even when no changes are made to the design logic. As seen in Table 1, for the average design iteration no synthesis is performed when using `qFlow` and `tFlow`. The custom tools are only concerned with placing and routing the specified modules, as long as room remains on the FPGA. ISE tries to optimize the entire design, so the more resources used the longer assembly takes. It can also be seen that `tFlow` does not have any Bit Generate time. This is because `tFlow` works by stitching together pre-built bitstreams of individual modules. The placing and routing is already happening at the bit level, so when it is complete the final product is a working bitstream. Both of the custom tools used have performed at more than reasonable speeds and this comparison is meant to show where the accelerations occur in relation to the tools designed for the FPGA.

Table 1. Average ZigBee Build Times in Seconds

	Synthesis	Map/Place	Route	Bit Generate	Total
ISE	198	89	46	33	366
qFlow	0	35	46	39	120
tFlow	0	10	5	0	15

6. CONCLUSION

From the perspective of a radio designer, library-based assembly is more natural than low-level hardware description languages and hides the complexity of FPGAs. In the flow presented here, chains of computation were specified for FPGA implementation within the GNU Radio framework just as if they were original radio blocks in the flow. Once one designer has written standard radios and filters in hardware, they can be passed around in the form of blocks, which anyone can drop into their design without the tedious step of developing for FPGAs. The tools can place multiple hardware accelerator blocks on one FPGA as long as there is still available room. The nature of the tools means that more area resources are required, so the FPGA will fill up faster, but this is seen as an acceptable trade off for many designs that see a large decrease in build time. Implementing multiple accelerator paths on one FPGA will also begin to cause performance issues in communication over the Ethernet line if large amounts of data are sent back to GNU Radio at once. As with any heterogeneous system, it will perform best in a situation with minimal communication between hardware; but the gigabit speeds have been more than adequate for transferring signal data between the USRP2 and GNU Radio in the past.

The enhanced GNU Radio flow was demonstrated using a USRP2 and a Virtex 5 FPGA, all networked to a host computer with gigabit Ethernet. One clear benefit of this flow was that the FPGAs could be added or taken away just like any other module, and were not a forced part of the design. Any number of FPGAs could be added, and all of the communication and interconnects would be handled implicitly. The radio designer could pick the composition of a radio using available hardware components, and chose where they go in the flow.

These enhancements to GNU Radio showed how an FPGA system could be built in near real-time for an SDR environment. GNU Radio was used as a test bench for `qFlow` and `tFlow` because of its open nature and current lack of FPGA integration. As more hardware accelerators are added to GNU Radio a completely heterogeneous system could be built even in the prototype phase.

7. REFERENCES

[1] T. Ulversoy, *Software defined radio: Challenges and opportunities*, Communications Surveys Tutorials, IEEE, 12(4):531 –550, quarter 2010.

[2] E. Blossom, *GNU Radio*, http://gnuradio.org, 2012.

[3] J. Blum, *GNU Radio Companion*, http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion, 2012.

[4] A. Alonso, *GNU Radio Tutorials*, http://gnuradio.org/redmine/projects/gnuradio/wiki/Tutorials, 2012.

[5] M. Ettus, *Ettus research*, http://www.ettus.com/, 2012.

[6] E. Blossom, *Exploring GNU Radio*, http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html#{}fpga, 2004.

[7] J. Corgan, *USRP Intro*, http://gnuradio.org/redmine/projects/gnuradio/wiki/UsrpFAQIntro, 2012.

[8] The Comprehensive GNU Radio Archive Network, *Available projects*, https: //www.cgran.org/wiki/Projects, 2012.

[9] C. Dick, *A case for using FPGAs in SDR PHY*, EE Times Design, http://www.eetimes.com/design/communications-design/4142853/A-case-for-using-FPGAs-in-SDR-PHY, 2002.

[10] N. Steiner, *Tools for open reconfigurable computing*, http://torc-isi.sourceforge.net/index.php, October 2011.

[11] T. Frangieh, *Design Assembly Techniques for FPGA Back-End Acceleration*, PhD thesis, Virginia Polytechnic Institute and State University, In Progress.

[12] G.J. Minden, J.B. Evans, L. Searl, D. DePardo, V.R. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A.M. Wyglinski, and A. Agah. *KUAR: A flexible software-defined radio development platform.* New Frontiers in Dynamic Spectrum

Access Networks, 2007. DySPAN 2007. 2nd IEEE International Symposium, pages 428 –439, 2007.

[13] H. Harada. *Software defined radio prototype toward cognitive radio communication systems.* New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. First IEEE International Symposium, pages 539 –547, 2005.

[14] S.M. Mishra, D. Cabric, C. Chang, D. Willkomm, B. van Schewick, S. Wolisz, and B.W. Brodersen. *A real time cognitive radio testbed for physical and link layer experiments.* New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. First IEEE International Symposium, pages 562 –567, 2005.

[15] J. Corgan, *USRP FPGA Verilog*, http://gnuradio.org/redmine/projects/gnuradio/wiki/UsrpFAQFpgaVerilog, 2011.

[16] C. Irick, *Enhancing GNU Radio for hardware accelerated radio design*. Master's thesis, Virginia Polytechnic Institute and State University, 2010.

[17] Ubuntu Manuals, *crc32*, http://manpages.ubuntu.com/manpages/lucid/man1/crc32.1.html, 2005.

[18] K. Pereira, *Characterization of FPGA-based high performance computers*, Master's thesis, Virginia Polytechnic Institute and State University, 2011.