Adventures in RFNoC™: Lessons Learned From Developing a Real-Time Spectrum Sensing Block

Rylee G. Mattingly RMATTINGLY@OU.EDU
School of Electrical and Computer Engineering, Advanced Radar Research Center, University of Oklahoma

Justin G. Metcalf JMETCALF@OU.EDU
School of Electrical and Computer Engineering, Advanced Radar Research Center, University of Oklahoma

Abstract

RF Network-on-Chip (RFNoC™) is an open source framework from Ettus Research that allows for convenient development access to the field-programmable gate array (FPGA) within Ettus brand USRP devices. By utilizing the floor space available on the FPGA of select radio models, digital signal processing DSP can be done in hardware before the data is ever streamed to the host computer. Cross device heterogeneous processing can increase the speed of computationally intensive algorithms by helping to parallelize operations on the FPGA prior to generalized processing on the host computer. This paper and the accompanying presentation discuss a design flow that was derived from lessons learned while developing with RFNoC for use on the Ettus X310 radio. An implementation of a real-time spectrum sensing block is shown as an example of a successful use of the proposed process.

1. Introduction

The RFNoC framework lowers the barrier of entry to develop FPGA based digital signal processing (DSP) blocks that can be used with UHD and GNURadio (Ettus Research, 2020b). Deploying algorithms to the FPGA allows for computational savings on the general purpose CPU of the host PC. Additionally, utilizing the FPGA for processing can reduce the latency of the critical path by removing the need to send data to the host computer or reduce the amount of data that needs to be streamed to and processed by the host computer.

RFNoC provides several tools to try to reduce the overhead required to build a custom FPGA block. Design complexity is managing by utilizing abstraction and limiting the number of files a user must interact with to succesfully implement an RFNoC block. This work describes the lessons learned from the implementation of a fast spectrum sensing (FSS) block by walking through the design flow from custom IP module to functioning RFNoC block controlled through GNURadio. All development took place on a desktop machine with an Intel® Xeon® processor and 256 gigabytes of ram running Ubuntu 20.04 and using UHD 4.0 with no out-of-branch patches applied. Of course, the more complex a block is and the more the structure deviates from the default configuration, the more management must take place. Numerous topics will be covered in an order that mimics the actual order that a user may take to build a block.

FSS operates by taking a sample of the spectrum and then uses an a priori estimate of the noise environment to classify each bin, where each bin is a consecutive sample from a fast Fourier transform (FFT), of a spectral frame as either high-power or low-power (Kirk et al., 2018). Low-power bins are considered unoccupied and available for use. There are many ways to process or optimize utilization from the resultant low-power "meso-bands", or groups of consecutive low-power bins, identified by the FSS algorithm (Martone et al., 2018). This work focuses on a greedy approach and implements an architecture that returns the widest low-power meso-band (Mattingly, 2021).

Due to the time-frequency agility of modern communication networks, any spectrum sensing algorithm must be able to sense a new primary user of the band in sub millisecond timescales in order to minimize interference. Implementing the FSS algorithm on the FPGA in the radio helps to reduce latency and reduce the response time to new emitters entering and leaving the environment. A brief description of the block architecture will be shown and the latency metrics of that block will be described.

2. Selecting an Interface

Understanding the data flow between blocks is critical and this starts by understanding how blocks are connected.

Proceedings of GNU Radio Conference 2021,

Data ports of blocks within the RFNoC framework are connected using the Compressed Hierarchical Datagram for RFNoC (CHDR) bus. The CHDR bus is a lightweight implementation of the AXI bus interface introduced by Xilinx to standardize the connection between Xilinx-provided IP blocks. The RFNoC implementation of this interface utilizes only tdata, tvalid, tready, and tlast signal lines for each input and output (Ettus Research, 2020b). This means that a framer/de-framer is needed to divide the data packets into their respective header, metadata, and data elements. Fortunately, RFNoC provides two framer/de-framer interfaces, AXI-Stream Payload Context and AXI-Stream Data, as part of their development tools.

Since both of these interfaces use the AXI-Stream interface to some extent, a brief overview should be provided before continuing. The AXI-Stream bus is a subset of a larger set of AXI bus protocols (Xilinx, 2017). Table 1 describes the signals that are used in AXI.

Signal Name	Signal Description
tdata	The payload word for the data stream.
tlast	Asserted on the last payload word of the packet
tvalid	Asserted when the value on tdata is valid.
tready	Asserted when by the recipient to signal readiness.
tuser	Describes the word type for the current word on the bus.

Table 1. A description of the signal used in a simple AXI-Stream interface.

Notably, this interface provides a valid signal that is fed into the receiving block and a ready signal that comes from the receiving block to the sender. This allows for the blocks to agree on a successful transmission before the transaction is complete. This also makes the logic for advancing the stream incredibly simple. If the valid signal and ready signals are both asserted on a clock cycle then the stream can be advanced. The user signal is not strictly required for the AXI-Implementation but is used in the AXI-Stream Payload Context interface. This signal provides information about the currently presented word on the data line, which is useful if different types of data encodings are used in a single data stream.

The AXI-Data interface is the simpler of the two available interfaces. The interface provides a standard AXI-Stream interface for the payload data of the packet but removes user-facing complexity by presenting the header data as separate signals that are valid throughout the receipt of the data packet. Table 2 provides a list of signals that are provided to the block in addition to the previously described AXI-Stream signals that are used for the data stream.

Signal Name	Signal Description
ttimestamp	Timestamp associated with the packet.
tlength	Length of the packet in bytes.
teov	Signals the end of a vector.
teob	Signals the end of a burst of associated packets.

Table 2. The relevant signals for the AXI-Stream Data user interface.

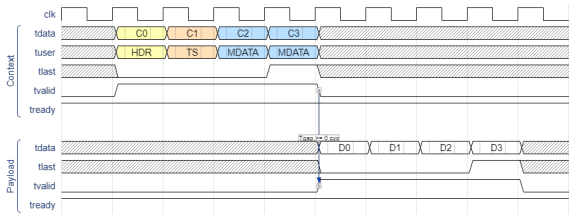

Figure 1. The timing diagram for a the AXI-Stream Payload Context interface. It shows a 4 word data packet with timestamp (Ettus Research, 2020b).

Although the AXI-Data interface does provide an easier interface to use, the lock timing of the signal fields is not constant. There is a setting that defines when the signal is valid and how long it remains valid. The valid time of these signals and the timing necessary to successfully output updated header information can be ambiguous.

The other interface option is AXI-Stream Payload Context. This interface provides an AXI interface for data, like the previous option, but it also provides the header, timestamp, and metadata information in an additional AXI-stream called the context stream. Because both of these streams are being put onto a single bus by the framer/de-framer logic it is important to serve the context data before the payload data. Both of the streams use the signals as they are described in Table 1 to provide the AXI-Streams, except that the user signal is only provided in the context stream as the data stream only has one encoding type. The timing diagram shown in Figure 1 shows how the context and payload streams are linked together and the structure that must be maintained when the packet is forwarded or a new packet is created. Although this interface requires more consideration for two separate data streams that are linked in time, it allows a consistent model to be applied to all data and allows metadata to be used if desired. It is for those reasons that the AXI-Stream Payload Context interface was chosen for this work.

The framer/de-framers are inside of the NoC shell where data goes in from the CHDR bus and is output in the according to the selected protocol as discussed above before interacting with user logic. Similarly, the output interface signals go back through the NoC shell on their way out

of the block. The NoC shell is mostly a black box to the user, except for a first in first out (FIFO) buffer on each of the ports. This buffer allows the user block to stop accepting data briefly, using the tready signal, without dropping packets or packet words. A backend interface is also implemented in the NoC shell as a 512-bit status interface and a separate control interface that is used by the larger framework and is not of use to the user (Ettus Research, 2020b).

3. The FSS Architecture

FSS must scan through the data stream as it arrives and classify each frequency bin as being high power (i.e occupied by a signal) or low power (i.e. noise only) based on the threshold. The payload AXI-Stream that feeds data into the block provides a single bin value per clock cycle. Therefore a simple comparison can be carried out on each element to determine if it is low power or not.

If a bin is the first low-power signal to be encountered, then a 'bucket', with start and size fields, should be initialized with a start point of the current index and a size of one. For each consecutive low power sample after a bucket has been initialized the size of the existing bucket should be incremented. Once a high-power bin is encountered the bucket that was being used should be frozen.

This bucket initialization could lead to a very large number of buckets being instantiated, which is unnecessary since the only bucket of interest is the largest one. This means that only two buckets should ever be needed to retrieve the desired data. Instead of initializing a new bucket when a new section of the available spectrum is encountered then the smallest of the two buckets should be selected to be overwritten. This is a simple comparison between the size parameter of the buckets. Selecting the bucket with the smallest size to be the new write space preserves the data of the currently longest sample set.

Once the largest bucket is found, it needs to be sent out of the block. There are a few options available. The first option is to add the start point and size to the metadata at the front of the packet. This was quickly ruled out as it would require the entire packet to be cached until after a decision had been made. Ideally, the block would allow the data to stream through such that there was no interruption to the stream. That means that the block will need a secondary port to move the data along.

Secondly, the data could be sent across the control infrastructure to the next block. This would allow the FSS block to utilize a single stream endpoint. Unfortunately, this would prevent the data from being streamed directly to the host PC without an intermediary block to convert the control data to a standard data stream. Although the block data is intended to be consumed by another block in hardware without traveling to the PC, for this work it is necessary to offload the data to the computer for verification and analysis. This meant that the block would need to employ a secondary data output stream to enable the full flexibility that is needed. This comes with the additional overhead of a secondary stream endpoint to dynamically connect the secondary data output to other blocks of interest. The two buckets need to have each of their fields maintained as each frame of data is processed. The size of these registers should be determined by the length of the packet, as each of the registers should be able to store this value. The block designed for this work is fed by a hardware FFT block with an FFT length of 1024 setting the packet size. 1024 was selected as the FFT length because 1024 is approaching the largest power of two of samples that will comfortably fit in a jumbo Ethernet frame. The default transporter block puts each CHDR packet in its own Ethernet frames and cannot split large CHDR packets between multiple Ethernet frames. This means that our bucket sizes are stored in 11-bit registers so that they could, if necessary, store 1024 as the possible maximum size of available spectrum in a frame. The start point fields are stored in 10-bit registers so that that the index packet word index from 0 to 1023 can be stored.

The packet words that are passed into the block are not tagged with their number in the packet. That means that an 11 bit counter should also be kept. This packet counter allows the block to keep track of the word index in the packet. This will be used to generate the index value that will be stored in the start field of a bucket when a low power bin is detected.

A final set of buckets is needed to allow for seamless operation. A send bucket is maintained and on the last sample of a packet, the data associated with the largest bucket is transferred to this set of registers. This allows all of the fields associated with the FSS processing state machine to be reset on this last sample. Of course, it is important to consider the effect that the last sample has on the data. This means that if the last bin is below the threshold then the largest bucket comparison should be made with this change considered. Similarly, if the value belongs in the bucket that is being selected, then the value of the size of the send bucket should be incremented with the transfer. With the variables and structures laid out, the FSS state machine can be presented. Figure 2 shows the FSS state machine with the transition conditions and transition actions listed. Figure 3 shows the state machine diagram for sending a packet on the secondary output.

4. Generating Verilog

The development journey begins with the rfnocmodtool, a command-line tool that is used to create RFNoC modules

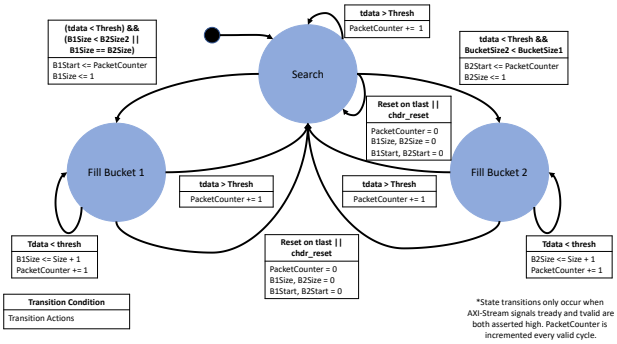

Figure 2. The basic FSS state machine shown with transition conditions and transition actions.

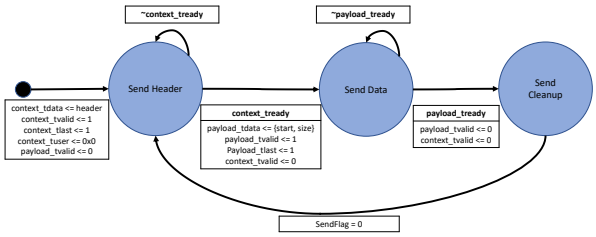

Figure 3. The state machine used for transmitting the a packet of data out of an AXI-Stream port on an RFNoC block.

and blocks. Modules contain RFNoC blocks, this hierarchy allows users to group similar blocks together and help manage files when many blocks are in development (Ettus Research, 2020a). A module can be created using the following command:

```
$ rfnocmodtool newmod
```

The user will be prompted to enter information about the module that will be used to create a directory with the required folder structure. Navigating inside of the file directory that was created from the newmod command, a user can generate the files necessary to start work on the block.

```
$ rfnocmodtool add
```

The add command above prompts the user for information about the block, of which only the name is a required field, the defaults for the remaining options are sufficient for a successful start to the process.

There are seven main files that the user needs to manipulate to make a fully flexible RFNoC block. Table 3 lists the files of interest and their location relative to the top level of the module. These files will be used throughout the development flow and their purpose will be discussed as they are used.

The Verilog files rfnoc_block_blockName.v and rfnoc_shell_blockName.v are both generated automatically. rfnoc_block_blockName.v is the file that describes the RFNoC block under development and this

File Name	Location
rfnoc_block_blockName.v	/rfnoc/fpga
noc_shell_blockName.v	/rfnoc/fpga
blockName_x310_rfnoc_image_core.yml	/rfnoc/icores
blockName_block_ctl_impl.cpp	/lib
blockName.yml	/rfnoc/blocks
module_blockName.block.yml	/grc

Table 3. The relevant files for the RFNoC block development cycle.

is where the HDL for the user's design will go. The rfnoc_shell_blockName.v is instantiated inside the main design and is responsible for the framer/de-framer of the selected interface as discussed in the previous section. The RFNoC shell also implements a backend interface that helps control RFNoC that is completely invisible to the user and should not be modified.

Before jumping into the Verilog files the user should first configure the block and generate new HDL files to meet their needs. Block configuration settings are the responsibility of the blockName.yml file. YAML Ain't Markup Language (YAML) is a human readable format that is used in several places to define parameters and describe blocks and is denoted with the .yml or .yaml file extension (YAML, 2009).

There are four main sections to the blockName.yml file: block information, clocks, control ports, and data ports. Block information should not be changed as it is mostly informative. The clocks section controls the clock frequencies of the various busses, there are two clocks that can be chosen from on the X310: 200 MHz and 184.32 MHz. It is critical to ensure that the CE clock is in the list and that the desired clock speed is selected. Setting the clock is as simple as

```
clocks:
  - name: ce
    freq:  "[200 MHz]"
```

This work did not utilize the control port as a master port and therefore the default settings of this section were always used.

The final section defines the configuration of the data ports. The framer/de-framer interface needs to be selected as either axis_chdr, axis_pyld_ctxt, or axis_data and a clock needs to be assigned and multiple named input and output ports can be defined using the following parameters. Defining the clocks is critical as it can lead to very hard to diagnose drops in block performance and throughput.

```
data:
  fpga_iface: axis_pyld_ctxt
  clk_domain: ce
  inputs:
    input1:
      item_width: 32
      nipc: 1
      info_fifo_depth: 32
      context_fifo_depth: 32
      payload_fifo_depth: 32
      format: sc16
      mdata_sig: ~
```

Listing 1. Data interface settings that should be considered for each block.

Buffer sizes are mostly up to the user to determine. These FIFOs sit at the port edges and help to prevent overflows if the user block very briefly stops accepting input. The same settings apply to the output ports that have identical parameters under an outputs tag instead of an inputs tag.

Once the block YAML is configured to the users desired state, new Verilog files can be generated. RFNoC provides a script called rfnoc_create_verilog.py that generates the skeleton verilog files for the configuration defined in blockName.yml. The script is a part of UHD and exits in the UHD install directory under /host/utils/rfnoc_backtool directory. When the user is in the same directory as the script it can be executed from the command line using the following,

```
$ ./rfnoc_create_verilog.py -c
path/to/block/yaml
-d path/to/Verilog
```

Now that the two Verilog files have been updated, the user can insert their IP module and connect it to the defined interface.

5. Adding Registers and Ports

Now that the two Verilog files have been updated, the user can begin to decide how many user registers they want. Register logic is generated inside of the rfnoc_block_blockName.yml file and a single register is defined by default, providing a template for the addition of more registers. Each register has an address parameter and a reset parameter, with the default address set to zero and a default value of 0. These two parameters should be copied and used for each new register, incrementing the address by four as the register words are passed in as 32-bit integers.

First, the user should instantiate a 32-bit Verilog register for each new RFNoC controlled register that is needed. The next part of the user logic is an *always* block synchronized with the control port clock. Inside this *always* block there are three *if* statements that control register functions. The first of these *if* blocks checks the reset line, here the user should add a simple register assignment resetting the Verilog register to the default value parameter

The other two *if* statements check for a register read or a register write. Both of these contain *case* statements using the address to multiplex between the registers. A copy of the default register logic with the appropriate changes to the parameters will successfully set up the hardware of the registers. The Verilog code in Listing 2 provides this user logic as it can sometimes be absent from generated files in UHD version 4.0.

```
// Address for user register
localparam REG_USER1_ADDR     = 0;
// Defult value
localparam REG_USER1_DEFAULT = 0;

reg [31:0] Reg_One_Value
        = REG_USER_DEFAULT;

always @(posedge ctrlport_clk) begin
  if (ctrlport_rst) begin
    Reg_One_Value = REG_USER_DEFAULT;
  end else begin
    // Default assignment
    m_ctrlport_resp_ack <= 0;

    // Read user register
    if (m_ctrlport_req_rd) begin
      case (m_ctrlport_req_addr)
        REG_USER_ADDR: begin
            m_ctrlport_resp_ack  <= 1;
            m_ctrlport_resp_data
                <= Reg_One_Value;
        end
      endcase
    end

    // Write user register
    if (m_ctrlport_req_wr) begin
      case (m_ctrlport_req_addr)
        REG_USER_ADDR: begin
          m_ctrlport_resp_ack <= 1;
          Reg_One_Value
            <= m_ctrlport_req_data[31:0];
        end
      endcase
    end
  end
end
```

Listing 2. The default register logic described above. Provided as

it is not always correctly generated in version 4.0.

After the (well-tested and simulated!) user module is inserted into the user logic section of the verilog file and the hardware definitions for the registers is complete, the work in the verilog files is done.

Next, the software interface must be informed of the changes to the port configuration and the additional user registers. blockName_block_ctl_impl.cpp is where the the API goes to generate the control signals necessary to configure the block when the block constructor is called by the user script or GNURadio. First, the user needs to create new uint32_t constants for the registers that were just created in the Verilog. A call to the register_property function should be made inside of the private _register_props function with a closure to provide the functional information. After the property is registered, a custom property type needs to be instantiated at the bottom of the file, this is the property that is passed into the register_property. The constant declaration, property call, and property type should be inferred from the default single register instantiated in the file by default.

Similar to the process to create a register property, a block edge property is required for each of the ports that are defined in the YAML files discussed in the previous section. Just like with the register, constants need to be defined for each of the ports, each property must be registered and a property type must be instantiated. Unlike the register, the constants should only be incremented by 1 and the input and output edges are indexed separately so there should be an output zero and an input zero. A further difference from the register is an additional property resolver that helps the software to understand what data types should be sent in and out of the block. This should match the value that was used in the port definition of the block YAML, but there is not a check to make sure that the values are consistent. How to instantiate these new ports should be infered from the existing definitions in the blockName_block_ctl_impl.cpp file.

6. GNURadio Integration

If the user is planning on using the block with GNURadio then there is one more step before moving on to the final image synthesis tasks. The graphical representation of the block that is used for drag and drop connection of the block needs to know what the port configuration is and what each of the register callbacks and default values should be. All of this configuration is done in the module_blockName.block.yml file, utilizing three main sections.

The first section is the template section, this provides the import argument and the constructor parameters so that GNURadio can correctly setup the auto generated script. Of these parameters only the callbacks need to be manipulated. These callbacks connect through to the previous properties that were instantiated to manage the control read and writes to the registers in the blocks. One call back is needed for each of the registers that are used and it is important that the first argument in the callback is the same as the one provided in the last property object in the implementation file.

The second section, parameters, is all of the fields within the GNURadio block. Register fields for the user to input values into the GUI block are created here. It is critical that the ID of this value is the same as the one given to the callback in the previous section of this file. The label and default value can be anything the user wants to have visible in the GUI. It is a good idea to have the block defaults in this file be the same as the default parameters previously set but, again, this is not a requirement.

The last section of this file is used to create the actual ports on the GUI block. Each port has three parameters: domain, label and dtype. For the purpose of this work the domain will always be rfnoc and the dtype will be sc16, the label can be whatever the user wants to display. The port assignment between these definitions and the properties defined in the previous file are based on the port number assigned previously, meaning that the first import port reference in the .block.yml file corresponds to the zeroth port in the property definition.

An example of each of the necessary fields are provided below for illustration.

```
callbacks:
- set_int_property('user_reg',
                   $(user_reg))
...
parameters:
- id: user_reg
  label: User Register Name
  dtype: int
  default: 0
...
inputs:
- domain: rfnoc
  label: Input Port Name
  dtype: 'sc16'
outputs:
- domain: rfnoc
  label: Output Port Name
  dtype: 'sc16'
```

7. Image Synthesis and Loading

With all of the initial block setup and design complete the user can define the blocks that are needed in the FPGA image and how they are connected through the various crossbars. Settings for the layout of the blocks are handled in the blockName_rfnoc_image_core.yml file. This file is structured into four section: streaming endpoint definitions, block definitions, block connections, and clock assignment.

First the streaming endpoints need to be defined, each have three block parameters: ctrl, data and buff_size. The ctrl and data parameters determine whether or not traffic from that plane flows through the endpoint. buff_size determines the size of the buffer that is used to prevent dropped packets if the recipient of the stream halts operation momentarily. The parameters are formatted as follows

```
stream_endpoints:
  ep0: #Label can be anything
    ctrl: False
    data: True
    buff_size: 32768
```

Block definitions are next with each having a different set of parameters. The only common parameter is the block_desc, this is the name of the blockName.yml file and should be formatted as seen below. The parameters field can also be used to set HDL parameters of the block adding additional flexibility.

```
noc_blocks:
  blockName0:  #Label can be anything
    block_desc: 'blockName.yml'
    parameters:
```

Since the block and module were created by the modtool then the software knows the directory structure and where to find these files.

Next, connections must be defined for each of the blocks. Decisions about the routing through an endpoint or through the static crossbar only are made at this point. Two different connection examples are given below. The first provides a connection from the ep0 as the source of the data to the input of duc0 (Digital Up Converter) block, then it is routed from the duc0 output to the input of the radio0 block. This is an example of a static connection between two block using only the static crossbar between the DUC and radio, this forms the transmit chain to the radio hardware.

The second example shows a block connected with the streaming endpoint on each end. It is important to note that each endpoint should only be used for a single pair of signals. If the user's block has an odd number of ports or multiple pairs of ports, multiple streaming endpoints are necessary.

```
connections:
- {srcblk: ep0,   srcport: out0,
dstblk: duc0,    dstport: in_0}
- {srcblk: suc0, srcport: out_0,
dstblk: radio0, dstport: in_0}
```

The final piece of this file defines the clock connections to the block. Clocking is a critical piece for ensuring that the operation of all of the blocks goes smoothly. For this work all of the clocks were defined using the compute engine (CE) clock except for radio blocks which are connected to the radio clock. An example of both types of clocking connections are shown as,

```
clk_domains:
- { srcblk: _device_, srcport: radio,
    dstblk: radio0, dstport: radio }
- { srcblk: _device_, srcport: ce,
    dstblk: ddc0,    dstport: ce    }
```

Finally, after all of these files have been updated to set up the configuration of the block, the module can be built and the FPGA image synthesized. To start the build and synthesis process the user needs to create and navigate, within the terminal, to the module build directory. Once in the build directory, the make files for the install need to be created using and the c files built :

```
$ cmake ..
-DUHD_FPGA_DIR=./path/to/uhd/fpga/dir
```

```
$ make
```

After this process completes the make files can be used to install the module into GNURadio with the command,

```
$ sudo make install
```

At this point the block should be visible inside of GNURadio at the bottom of the module tree. The make files are now ready for synthesis to the FPGA. Synthesis requires that the Xilinx Vivado is already installed with version 2019.1 being the tested version. The Vivado build is initiated by executing the following in the build directory.

```
$ make blockName_rfnoc_image_core.yml
```

The build process can take a significant amount of time, throughout this work 45 minutes was found to be very common but times of up to four or five hours were also encountered depending on the content of the user logic. Once the build is complete a .bit file is generated in the uhd/rfnoc/-top/usrp3/x300/build directory. The .bit file is used to program the FPGA using the uhd image loader tool. The image loader was the most successful when using the following argument configurations:

```
$ uhd_image_loader
    --args="addr=Radio.IP.Address"
    --fpga_path ./path/to/.bit
```

8. Performance

It is important to set a uniform way to measure performance such that performance across different implementations can be assessed and compared. Creating a metric for the designs in this work is even more challenging, as much of the design exists as a module within a black box. Therefore, the metric must be defined for only the logic that is managed by the user block itself.

Two metrics are developed for the user blocks in this work. The first describes the streaming efficiency of the block and is the number of clock cycles between the receipt of the final word in the packet and the transmission of the final word in the packet. This data latency metric helps classify the efficiency of data management for blocks that require any data retention or data stream halts. When a result is produced based on the data in a packet, it is useful to know how many cycles after the last word of a packet is received does the resultant data product get transmitted. This is the measure of the second metric that this work will refer to as product latency.

The first metric is easy to determine. The design goal of this block was to implement the algorithm without interrupting the AXI-Stream flow through the block. This was accomplished as the input ports and output AXI ports are wired directly together. This means that any stoppage of the stream comes from the block connected to the output of the FSS block. So the last sample is registered into the output on the clock cycle after it is registered onto the wire running through the block. This is the absolute minimum data latency possible for an RFNoC block other than not being present in the stream chain at all.

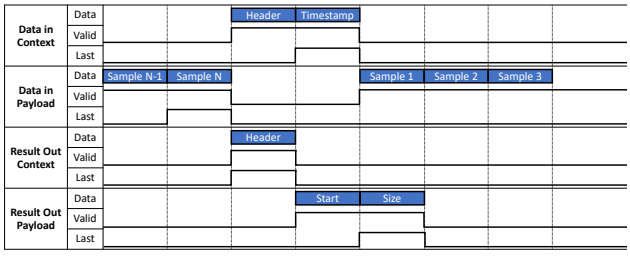

Figure 4. The state machine used for transmitting the a packet of data out of an AXI-Stream port on an RFNoC block.

To understand the result product latency, a timing diagram showing the end of the packet must be examined. Figure 4 shows a timing diagram at the end of the packet where the send flag is triggered by the tlast signal.

Here, the send flag is asserted by the tlast signal during that same cycle that the copy of one FSS bucket into the send bucket is carried out. On the clock after the tlast, the send bucket is loaded, the FSS buckets are clear and a valid marked header is on the context bus. Two clocks after tlast, assuming the consumer of the result is ready, the header is removed from the context bus and the data is on the payload bus. This means that the data is consumed by the output block on the rising edge of the third clock after tlast is asserted. This means that the product latency is three clock cycles. On the Ettus X310, the FPGA operates at 200 MHz, giving a total time of 15 nanoseconds between the receipt of the last data word and the output consuming the result.

9. Conclusion

This paper presented a development flow for implementing RFNoC blocks that was derived from the lessons learned from the design of a Fast Spectrum Sensing (FSS) block. The data interface of the framework was described before identifying critical files necessary for creating custom blocks. Additionally, an example block architecture for FSS was presented and shown to operate at full streaming rates.

New performance metrics were defined to determine performance from only the user defined logic. These new definitions include a data latency metric to define streaming performance and product latency to determine he time required to generate a result. The given FSS architecture adds no additional latency to the data stream, other than that introduced by the presence of the block. The block achieves a product latency of three clock cycles or 15 nanoseconds when running the X310 at its full clock rate of 200 MHz.

Acknowledgements

This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) under grant HR0011-20-1-0007. The views expressed in this article are those of the authors and do not reflect official policy or position of DARPA, or the U.S. Government, No official endorsement by DARPA should be inferred. Approved for public release; distribution is unlimited.

References

Ettus Research. *Getting Started with RFNoC Development*, October 2020a. URL https://kb.ettus.com/index.php?title=Getting_Started_with_RFNoC_Development&redirect=no.

Ettus Research. *RF Network-On-Chip (RFNoC™) Specification*, 2020b. Rev. 1.0.

Kirk, Benjamin H., Narayanan, Ram M., Gallagher, Kyle A., Martone, Anthony F., and Sherbondy, Kelly D. Avoidance of time-varying radio frequency interference with software-defined cognitive radar. *IEEE Transactions on Aerospace and Electronic Systems*, pp. 1090–1107, November 2018. doi: 10.1109.

Martone, Anthony F., Ranney, Kenneth I., Sherbondy, Kelly, Gallagher, Kyle A., and Blunt, Shannon D. Spectrum allocation for noncooperative radar coexistence. *IEEE Transactions on Aerospace and Electronic Systems*, 54(1):90–105, feb 2018.

Mattingly, Rylee G. Implementation and analysis of adaptive spectrum sensing. Master's thesis, University of Oklahoma, 2021.

Xilinx. *AXI Reference Guide*, 2017.

YAML. YAML Ain't Markup Language (YAML) Version 1.2, 2009. URL https://yaml.org/spec/1.2/spec.html.