Real Time Feature Detection and Threat Analysis with USRP SDR

Boris Shishkin, Danh Nguyen, Cem Sahin, Caitlin Miller
{bs44, dhn24, cs486}@drexel.edu, caitlin.miller@utexas.edu

November 2012

Contents

1 Introduction 4

2 Setting up UHD + GNURadio 4
  2.1 Install Drexel version of UHD 4
      firmware 5
      fpga 5
      host 5
      images 5
      Examples 5
  2.2 Install GNURadio 6
  2.3 Connect the USRP to the Host 6
  2.4 UHD Host Drivers and GNU Radio UHD Library Rebuild 7

3 Process for Custom FPGA Hardware Development 7
  3.1 Install and Run Xilinx on Ubuntu 7
  3.2 Hardware Build Process 7
     Creating a Custom Makefile 7
     Build the Hardware Design Using Make 8
  3.3 Firmware Build Process 9
  3.4 Adding Custom Verilog Files 9
     Inputs 10
     Outputs 10

4 Drexel FPGA Hardware Development for USRP N210 10
  4.1 Default USRP n210 Verilog Modules 11
     wb_1master (wb_1master.v) 11
     sysctrl (system_control.v) 11
     zpu_top0 (zpu_wb_top.v) 11
     bootram (bootram.v) 11
<table>
<thead>
<tr>
<th>Component</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_ram (ram_harvard2.v)</td>
<td>12</td>
</tr>
<tr>
<td>packet_router (packet_router.v)</td>
<td>12</td>
</tr>
<tr>
<td>shared_spi (simple_spi_core.v)</td>
<td>12</td>
</tr>
<tr>
<td>i2c (i2c_master_top.v)</td>
<td>12</td>
</tr>
<tr>
<td>gpio_atr (gpio_atr.v)</td>
<td>12</td>
</tr>
<tr>
<td>buff_pool_status (wb_readback_mux.v)</td>
<td>12</td>
</tr>
<tr>
<td>simple_gmac_wrapper (simple_gmac_wrapper.v)</td>
<td>12</td>
</tr>
<tr>
<td>settings_bus (settings_bus.v)</td>
<td>12</td>
</tr>
<tr>
<td>settings_bus_crossclock (settings_bus_crosslock.v)</td>
<td>13</td>
</tr>
<tr>
<td>user_settings (user_settings.v)</td>
<td>13</td>
</tr>
<tr>
<td>sfc (settings_fifo_ctrl.v)</td>
<td>13</td>
</tr>
<tr>
<td>sr_clear_sfc (setting_reg.v)</td>
<td>13</td>
</tr>
<tr>
<td>sr_clk, sr_ser, sr_adc, sr_phy, sr_bld, sr_led, sr_led_src (setting_reg.v)</td>
<td>13</td>
</tr>
<tr>
<td>pic (pic.v)</td>
<td>13</td>
</tr>
<tr>
<td>uart (quad_uart.v)</td>
<td>13</td>
</tr>
<tr>
<td>s3a_icap_wb (s3a_icap_wb.v)</td>
<td>13</td>
</tr>
<tr>
<td>flash_spi (spi_top.v)</td>
<td>13</td>
</tr>
<tr>
<td>rx_frontend (rx_frontend.v)</td>
<td>13</td>
</tr>
<tr>
<td>ddc_chain0, ddc_chain1 (ddc_chain.v)</td>
<td>14</td>
</tr>
<tr>
<td>vita_rx_chain0, vita_rx_chain1 (vita_rx_chain.v)</td>
<td>14</td>
</tr>
<tr>
<td>ext_fifo_i1 (ext_fifo.v)</td>
<td>14</td>
</tr>
<tr>
<td>vita_tx_chain (vita_tx_chain.v)</td>
<td>14</td>
</tr>
<tr>
<td>duc_chain (duc_chain.v)</td>
<td>14</td>
</tr>
<tr>
<td>tx_frontend (tx_frontend.v)</td>
<td>14</td>
</tr>
<tr>
<td>serdes (serdes.v)</td>
<td>14</td>
</tr>
<tr>
<td>time_64bit (time_64bit.v)</td>
<td>14</td>
</tr>
</tbody>
</table>

4.2 Drexel FPGA Hardware Modifications

5 Host Driver Modification: UHD and GNU Radio

5.1 Build and Install Custom Host Drivers

5.2 Details of Host Driver Changes

UHD Host Driver Changes

GNU Radio ‘gr-uhd’ Library Changes

6 ZPU Firmware

6.1 Packet Detected Interrupt Handler

7 Host-ZPU-Hardware Communication
8 GUI Development and Data Collection
  8.1 GUI Development .................................................. 26
  8.2 Data Collection ..................................................... 26
  8.3 How to Run Experiments ........................................... 28

9 Resources
  9.1 Books ................................................................. 28
  9.2 Links ................................................................. 28
  9.3 Questions? ............................................................. 29
1 Introduction

With increasing number of wireless packet standards and usage of different frequencies everyday, it is getting more important to be able to detect the traffic that is present in the air. Wireless service providers are paying closer attention to their traffic distribution to ensure enough resources are being allocated to the correct area. In this project, we present a quick and reliable preamble based packet detection. Our project utilizes the USRP N210 boards’ capability of streaming and processing at high speeds. We generated and plugged into the hardware design a custom fpga block to cross-correlate the received packet preambles with the predefined correlation values in order to detect predefined / known wireless standards (currently our system is capable of detecting WiFi and WiMax packet standards). We also added a cyclostationaty feature detector in order to detect non-predefined / unknown wireless standards. This type of energy detector is designed to both detect the rising edge (i.e. the beginning of the transmission) and the falling edge (i.e. the end of the transmission). We also modified the ZPU firmware for running specific tasks once a packet preamble is detected. In this project, we also deliver a full setup for capturing GPS locked timestamps for when valid packets are received. We make use of the metadata of each USRP packet to provide the means of transmitting the timestamps back to the host. This timestamps structure was also modified to provide additional information regarding the type of the timestamp. A timestamp could be generated due to a trigger from preamble based cross-correlator packet detector along with the falling and the rising edge of the energy detector. This was achieved by making changes to the host drivers. Our system is then complemented with a Python graphical user interface (GUI), where the user is allowed to configure the packet detection method on the USRP board on-the-fly, and a MATLAB data collection and analysis system, where timestamps are being stored and can be saved for future tasks.

2 Setting up UHD + GNURadio

This guide assumes that you will be installing the latest master or release branch of GNU-Radio on an Ubuntu Linux machine, although there are also Fedora, Mac, and Windows instructions available online. It also assumes that GNURadio is being used with the USRP n210 by Ettus Research, with the UHD drivers provided by Ettus. Instructions for other USRP models should be similar in setting up GNURadio, but information on FPGA development and USRP architecture will generally not apply.

* NOTE: Drexel has modified significantly the UHD host drivers and GNU Radio gr-uhd library. To install these custom code libraries, follow these instructions:

2.1 Install Drexel version of UHD

The UHD binaries must be installed BEFORE GNURadio for the two to be integrated properly. Run the following commands from the terminal to check out the Drexel-modified version of UHD from the master branch. Note that this is different from the original release from Ettus Research. Some of the driver functionalities are changed, such as received times-
tamp interpretation. These new driver APIs will not work with Ettus-provided examples and tests. Thus, when building the uhd library from source, make sure to disable the build of examples and test by passing options ‘-DENABLE_EXAMPLES=OFF -DENABLE_TESTS=OFF’ to cmake.

```sh
git clone git@wireless.ece.drexel.edu:~/atl/uhd
cd uhd/host
mkdir build
cd build
cmake .. -DENABLE_EXAMPLES=OFF -DENABLE_TESTS=OFF
make
sudo make install
```

Git will create a directory called “uhd”. Within this directory are four sub-directories.

**firmware**

The “firmware” folder contains firmware files for the on board processor. The n210 uses the ZPU, and so only the “zpu” folder is important to us. The source code in this directory does not need to be compiled unless you are creating custom firmware or FPGA images for the USRP. This will be discussed in the section on FPGA development.

**fpga**

The “fpga” folder contains all of the verilog source files for the FPGA. These files can be edited to add new functionalities to the FPGA. Again, these will only need to be built if creating custom images, which will be discussed later.

**host**

The “host” folder contains Python scripts for various USRP utilities, including burning firmware and FPGA images to the board.

**images**

The “images” folder should contain the firmware and FPGA images (.bin) files you would like to burn to the USRP. The lastest image files should be named ‘usrp_n210_fw.bin’ for ZPU firmware and ‘usrp_n210_r4_fpga.bin’ for FPGA hardware.

**Examples**

The “Examples” folder contains test scripts and demonstration scripts for testing out GNU Radio and UHD operations. The file ‘uhd_fft_with_WiMax20.py’ contains the most up-to-date GUI for adjusting correlation coefficients and detection types.

To run the application, run ‘python <script.py>’ on the command line.
2.2 Install GNURadio

Drexel Wireless System Lab (DWSL) modified the gr-uhd library in order to receive the process packet detection timestamps. This library provides source and sink blocks in GNU Radio environment.

The complete GNU Radio package must be built and installed at least once to function correctly. To build, get the custom GNU Radio library from DWSL and follow these instructions:

```bash
git clone git@wireless.ece.drexel.edu:~/atl/gnuradio
cd gnuradio
mkdir build
cd build
cmake ../
made
sudo make install
```

2.3 Connect the USRP to the Host

Connect to the USRP via ethernet:

Address: 192.168.10.1

Netmask: 255.255.255.0

To make sure the host can communicate with the USRP:

```bash
ping 192.168.10.2
```

Verify communications and UHD driver functionality by using UHD host commands:

```bash
uhd_find_devices --args="addr=192.168.10.2"
```

or

```bash
uhd_usrp_probe --args="addr=192.168.10.2"
```

Download the latest FPGA images from Ettus (see above in the description of the “images” folder. Find uhd/host/utils/usrp_n2xx_net_burner_gui.py and run the Python script. This is the easier version of the net burner with a user interface - the one without the GUI will also work if run from the terminal. Use the following:

Firmware Image: usrp_n210_fw.bin
FPGA Image: usrp_n210_r4_fpga.bin
This should write the FPGA image, the firmware image, and then ask you if you want to restart the FPGA. Click “Yes” and wait for the FPGA to reboot.

At this point you should be able to run all of the various GNU Radio blocks.

2.4 UHD Host Drivers and GNU Radio UHD Library Rebuild

At various points in development, in order to receive packet detection timestamps passed from UHD hardware to host, the host drivers for UHD devices and GNU UHD library (gr-uhd) must be rebuilt and re-installed on the host computer. For more information, refer to section 5 below.

3 Process for Custom FPGA Hardware Development

We developed a custom FPGA hardware based on the reference designed provided by Ettus Research. In this hardware we added custom packet detector, energy detector, timestamp-collection blocks, and host communication auxiliary functions. If this hardware design is changed in any way, a new FPGA image must be generated and programmed into the USRP N210 FPGA board. This section covers our hardware build process.

3.1 Install and Run Xilinx on Ubuntu

Install the licensed version of Xilinx the FPGA on the USRP n210 is larger than the USRP n200 FPGA, and so it is not compatible with the free WebPack license.

After installing Xilinx, the following must be run in the terminal each time you use ISE (on Ubuntu):

```
source opt/Xilinx/14.1/ISE_DS/settings32.sh
ise
```

3.2 Hardware Build Process

Creating a Custom Makefile

Locate uhd/fpga/usrp2/top/N2x0/Makefile.N210R4 and open it in a text editor. There are four parameters at the beginning of the makefile that can be customized. The following is an example that creates the images in uhd/fpga/usrp2/top/N2x0/build-custom, inserting a custom DSP module into the receive chain.

Multiple custom Verilog source files need to be added in addition to the `custom_dsp_rx.v` file. These files specify the top-level modules of our custom packet detector and are declared in
the `custom/Makefile.srcs`. Thus, this file must be included in the list of custom Makefiles as well.

```
TOP_MODULE = u2plus
BUILD_DIR = $(abspath build$(ISE)-custom)

# Path to custom dsp file
CUSTOM_SRCS = /home/lmco/uhd/fpga/usrp2/custom/custom_dsp_rx.v
CUSTOM_DEFS = RX_DSP0_MODULE=custom_dsp_rx

# Include other makefiles
...
include ../../custom/Makefile.srcs
```

### Build the Hardware Design Using Make

Run the following from the terminal under `top/N2x0/` to build the custom project:

```
# Clean the build folder
make clean
# Make new build
mkdir build-N210R4
# Copy the necessary pre-generated netlists into build folder
./copy_cores
# Build
make N210R4
```

Note the 3rd step from the build process above. The `copy_cores` script was created to copy the necessary pre-generated netlist cores from `custom/` folder into the `build/` folder. Without this step, Xilinx synthesis tool will report errors on missing cores.

**Note:** the `copy_cores` script uses absolute system path for copying the cores. It will fail if the path defined in the script is non-existent. Before running, make sure to open the script, update your specific system path to the correct ones. To check if the script runs successfully, go into `build-N210R4` folder and check whether there exist `.ngc` and `.mif` files. See section 4.2 below for more information.

Make will kick off the build process using Xilinx ISE toolchain. The process takes about 30 minutes, and we may encounter the following warning regarding an unmet timing constraint:

```
Process "Generate Programming File" completed successfully
INFO:TclTasksC:1850 - process run : Generate Programming File is done.
touch /home/dhn24/research/gnu-radio/uhd/fpga/usrp2/top/N2x0/build-N210R4/u2plus.bin
python /home/dhn24/research/gnu-radio/uhd/fpga/usrp2/top/python/check_timing.py
/home/dhn24/research/gnu-radio/uhd/fpga/usrp2/top/N2x0/build-N210R4/u2plus.twr
```
This warning can be safely ignored. It will not affect the hardware operation.
Make will create two important files. First of all, it will create a .bin file that will be used as the custom FPGA image in the net burner. This file must be renamed to the name of the default image file (in our case for the n210, the file MUST be named usrp_n210_r4_fpga.bin).

3.3 Firmware Build Process

We also must compile the firmware ourselves instead of using the default firmware images. Otherwise, the firmware and FPGA images may not be compatible due to different versions of Xilinx and other dependency issues, which may brick the FPGA. Go to the firmware directory (`uhd/firmware/zpu`) and run the following to compile custom firmware:

```
mkdir build-custom
cd build-custom
cmake ../
make
```

Make generates a firmware image named `usrp2_txrx_uhd.bin` in the `build-custom/usrp2` folder. Again, rename the file to the proper name (`usrp_n210_fw.bin`) before use.

After compiling BOTH of these .bin files (hardware and firmware), they can be burned to the FPGA using the net burner tool as before. If the FPGA becomes bricked at any point in this process (loses connection with the host and the net burner tool), simply hold the reset button on the board while power-cycling the USRP to reboot the FPGA in safe mode, and then load new images using the net burner tool.

More information on creating custom makefiles and building custom images can be found in the README text documents in the “fpga” and “firmware” directories.

3.4 Adding Custom Verilog Files

To enhance the functionalities of USRP N210 FPGA hardware, custom DSP cores are added to the design. This section describes the process used in hardware code development. The
custom_dsp_rx.v file is a verilog file already included with the uhd directory that can be modified to insert custom DSP modules. The following signals are inputs and outputs to this module that you can utilize in your own custom DSP code without further modification to the original code.

**Inputs**

- clock - control signal, DSP clock
- reset - control signal, active high synchronous reset
- clear - control signal, active high on packet control initialization
- enable - control signal, active high when streaming enable
- set_stb - user setting, controlled through user setting register API
- set_addr[7:0] - user setting, controlled through user setting register API
- set_data[31:0] - user setting, controlled through user setting register API
- frontend_i[23:0] - full rate real input directly from the RX frontend
- frontend_q[23:0] - full rate imaginary input directly from the RX frontend
- ddc_out_sample[31:0] - strobed samples I16,Q16 from the RX DDC chain
- ddc_out_strobe - high on valid sample

**Outputs**

- ddc_in_i - full rate real output directly to the DDC chain
- ddc_in_q - full rate imaginary output directly to the DDC chain
- ddc_out_enable - enables DDC module
- bb_sample[31:0] - strobed baseband samples I16,Q16 from this module
- bb_strobe - high on valid sample

4 Drexel FPGA Hardware Development for USRP N210

This section describes our customization of the USRP N210 FPGA hardware in order to enhance its packet detection capabilities.
4.1 Default USRP n210 Verilog Modules

The top-level verilog module for the USRP n210 is u2plus, found in the u2plus.v file. This file mainly instantiates the u2plus_core module, found in u2plus_core.v

Within the USRP2 core module are instantiations of the important modules for the various functions of the USRP. The following are descriptions of these modules, as well as some of the modules instantiated within them that may be important for custom DSP development on the FPGA.

**wb_1master (wb_1master.v)**

The Wishbone Single Master Interconnect is the wishbone connection bus to the ZPU. It segments the address space to provide access to the Main RAM, Packet Router, SPI, I2C, Ethernet MAC, Settings Bus, PIC, UART, ICAP, SPI Flash, and Boot RAM. It is mostly used to set parameters, not for relaying high-speed data.

The Wishbone Bus is a hardware computer that lets different cores within a chip communicate with each other.

Important types of signals found in this module include:

- **cyc** - indicates that a valid bus cycle is in progress
- **stb** - indicates a valid data transfer cycle
- **we** - indicates whether the current local bus cycle is a READ or WRITE cycle
- **ack** - indicates the termination of a normal bus cycle by a slave device

**sysctrl (system_control.v)**

System Reset Controller controls system-wide reset. When the system boots, this block is reset, then everything else is put in reset. Then this module takes the RAM Loader is taken out of reset, and finally takes the processor and wishbone out of reset to reset the main system.

**zpu_top0 (zpu_wb_top.v)**

This is the main, top-level instantiation of the ZPU processor. It is used to interface peripherals but does not process raw data. This module interfaces the ZPU to the wishbone connection bus.

**bootram (bootram.v)**

Instantiates the Boot RAM.
sys_ram (ram_harvard2.v)
Instantiates the Main RAM.

packet_router (packet_router.v)
The packet router controls the sending of packets across Ethernet, SerDes, and Wishbone. We want our matched filter data structure to be sent here as an input to the packet_router module in order to communicate custom packets (such as those containing timestamps) to the host via Ethernet.

shared_spi (simple_spi_core.v)
This is the Serial Peripheral Interface bus to the ZPU. It is used for data transmission including to the ADC and DAC.

- skl - serial clock
- mosi - master output, slave input
- miso - master input, slave output
- sen - slave select

i2c (i2c_master_top.v)
This is wishbone compliant I2C top-level master controller. It only interacts with the daughterboards.

gpio_atr (gpio_atr.v)
This module allows the GPIO ports connected by the motherboard to be controlled via the host.

buff_pool_status (wb_readback_mux.v)
This module appears to be a mux allowing data sources to be connected to the wishbone bus.

simple_gemac_wrapper (simple_gemac_wrapper.v)
This is the wrapper interface to the Ethernet. This handles most of the buffering for transmitting and receiving via Ethernet.

settings_bus (settings_bus.v)
The settings bus allows the ZPU to change the values stored in the various settings registers.
settings_bus_crossclock (settings_bus_crosslock.v)

Crosslock for the settings bus.

user_settings (user_settings.v)

The host can configure certain settings. The host code should be configured to input new coefficients and threshold values for a new communication protocol. This should in the future be sent to the matched filter module, instead of hardwiring these values.

sfc (settingsfifo_ctrl.v)

Settings and readback bus controlled by a FIFO.

sr_clear_sfc (setting_reg.v)

Settings changed by ZPU.

sr_clk, sr_ser, sr_adc, sr_phy, sr_bld, sr_led, sr_led_src (setting_reg.v)

All of these modules are settings registers that can be changed by the ZPU via the settings_bus module.

pic (pic.v)

PIC is an interrupt controller.

uart (quad_uart.v)

UART output for the ZPU.

s3a_icap wb (s3a_icap wb.v)

ICAP for Xilinx s3a FPGAs.

flash_spi (spi_top.v)

SPI top module.

rx_frontend (rx_frontend.v)

The first filtering stage for incoming I/Q samples. They are transformed to higher-precision samples and sent to the DDC chain for continued signal processing.
ddc_chain0, ddc_chain1 (ddc_chain.v)
This is the receive chain, which performs the signal processing operations on signals received by the USRP. Within this file are additional modules, including a CORDIC filter, CIC filter, and half-band decimators, which the I/Q signals pass through in that order. After the filters, the I/Q streams are rounded.

After the signal processing, a custom module is instantiated in the DDC chain. This module can be found in the dsp_rx_glue file. In the section on creating custom verilog modules, creating a custom makefile was explained. In this makefile, variables such as RX_DSP0_MODULE are defined by the names of the custom modules. If such a module is defined, the signals are passed into the custom verilog module. If not, we pass through generic assign statements and the final I/Q streams are ready to be passed in a packet to the host.

For more information, see the section concerning custom FPGA images.

vita_rx_chain0, vita_rx_chain1 (vita_rx_chain.v)
Prepares packets from the receive chain to be sent to the packet router and on through the Ethernet connection.

ext_fifo_i1 (ext_fifo.v)
This FIFO is a buffer for signals to be transmitted before they are sent to the DAC.

vita_tx_chain (vita_tx_chain.v)
Takes data from packets to be transmitted.

duc_chain (duc_chain.v)
This is the signal processing chain for the transmit side.

tax_frontend (tx_frontend.v)
Similar to the receive side - this module performs initial I/Q balancing and DC offset correction.

serdes (serdes.v)
Serializer/deserializer serializes data for passing through a serial link. It communicates the data over the MIMO cable. In other words, it communicates data from one USRP to another when using multiple USRPs connected via MIMO.

time_64bit (time_64bit.v)
All of the samples are timestamped.
4.2 Drexel FPGA Hardware Modifications

We modified the UHD hardware project in order to accommodate our custom packet detector hardware implementation. The packet detector was implemented in Xilinx System Generator and generated to Verilog. The output of the process included several Verilog files describing top level modules of our IP core, together with multiple Xilinx post-compiled netlist files, ending in “.ngc” and “.mif”. These .ngc & .mif files are critical to the hardware build process (synthesis, map, place & route, and generate programming file). These files must be copied into the 'build-N210R4/' folder before the hardware build process to ensure no errors. Otherwise, the Xilinx synthesis tool will report missing cores during build.

Custom Packet Detector Hardware Implementation

The custom packet detection hardware for this project was generated using Xilinx System Generator within Simulink. Source files can be found under custom build directory within the uhd hardware implementation. Figure 1 shows the general structure of the implemented design, including the two detection cores and the memory interface structure.

![Custom FPGA DSP Core](image)

Figure 1: System Generator DSP core

The basis of the implemented packet detection core was a correlation core extracted from Rice University WARP reference design 15. Within the Simulink design under the top level receiver block the ”Long Corr HP” block performs streaming cross correlations on complex inputs. It was extracted from the WARP reference design and modified to provide inputs...
for custom cross correlation coefficients.

Structurally the core is a phase correlator with 1 bit I and 1 bit Q resolution. However cross correlation coefficients are 3 bit signed values which effectively provide confidence weights to the phase correlation. The correlator is 64 samples long and results in a 2.56 microsecond correlation in time using a fixed 25 MSPS sample rate on the USRP N210. Currently other sampling rates are not supported for this core. A logical block diagram of the correlator is shown in Figure 2.

![Cross Correlator Diagram](image)

Figure 2: System Generator Cross Correlation Core

Modifications to the core consisted of replacing ROM modules in the core with block RAM’s and modifying the internal structure of the correlation blocks to accept the new memory structures. The resulting package required a custom bus structure that will be explained in the following section.

A second detection core was added to the design to allow for detection of energy variation of the input stream without knowledge of the input signal characteristics. The core consists of two full rate moving accumulators which produce a sum of 32 consecutive sample magnitudes squared. The two summers are separated by a 32 sample delay. The resultant system can provide feedback for changes in power of the incoming signal, for both rising signal energy and falling signal energy. Moreover a rising and falling signal thresholds are provided to force a trigger at a predetermined difference in values for the two summers. As an example a value of 10 for the rising energy detector indicates that a trigger will be passed to the host when the non delayed summer exceeds the delayed summer by a factor of 10. Since the
summers contain sums of magnitudes squared a simple conversion of $10 \log_{10}$ will produce an equivalent dB gain required to trigger the system. A logical block diagram of the energy differentiator is shown in Figure 3.

**Energy Differentiator**

![Energy Differentiator Diagram]

**Figure 3: System Generator Energy Differentiator Core**

<table>
<thead>
<tr>
<th>Gateway Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gateway In</td>
<td>I Raw Data In</td>
<td>Fix 16.14</td>
<td>NA</td>
</tr>
<tr>
<td>Gateway In1</td>
<td>Q Raw Data In</td>
<td>Fix 16.14</td>
<td>NA</td>
</tr>
<tr>
<td>Gateway In2</td>
<td>address_user_registers</td>
<td>8Bit Unsigned</td>
<td>NA</td>
</tr>
<tr>
<td>Gateway In3</td>
<td>Data User Registers</td>
<td>32Bit Unsigned</td>
<td>NA</td>
</tr>
<tr>
<td>Gateway Out</td>
<td>Signal Detection Trigger</td>
<td>1bit Boolean (Active High)</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 4: Input and Output Ports for the Packet Detector**

**User Register Bus Interface and IO** At the top level the user registers are interfaced using the Wishbone bus which can be controlled by the user. The host–Wishbone interaction will be discussed in later sections. Once the new register values are received by the Wishbone controller, it pushes its changes to the respective output. In our case, we used "Slave #7", which interfaces to the user registers. The register addresses are defined in Fig. 5.

Fig. 6 shows the details of the register 18 structure. Note that this specific register controls multiple functionalities. By changing the top 3 MSBs, the packet detection type can be
controlled. This register also controls the Energy Detector thresholds.

Similarly register address 1 is reserved for the cross correlation threshold where allowed values can range from 0 to $2^{32} - 1$. Reasonable values for core functionality should be selected between 10,000 and 25,000 depending on applied coefficients and tolerated rate of false alarms.

Addresses 2-17 are used for cross correlation coefficient values. Signed 3 bit coefficients are loaded into the registers as follows: 4 adjacent coefficients are grouped and assigned to the 24 top bits of the register in the order they appear in the series (i.e.: MSB of coefficient 1, I is assigned to MSB of address 2, MSB of coefficient 2, I is assigned to (MSB -3) of address 2. In this way I coefficients occupy registers 2 through 9. Q coefficients occupy register 10 through 17.

A series of MATLAB scripts is available for generating correlation coefficients from time domain or frequency domain samples. For reference, the Matlab script located in
'uhd/Documentation/CrossCorr_Coefficients/Long_symbol_generation.m' provides a simple example for register population.

Address 0 is reserved and must be written in order to bring the user register bus into a known state after every transaction. As an example, to perform a complete write operation to any register, two write operations must be performed:

write to address=X, Data=Y
write to address=0. Data=Y

Incorporating Custom IP Core Into Design

In order to incorporate our custom packet detector IP core into the design, we modified the following Verilog modules to accommodate the changes:

1. custom/custom_dsp_rx.v
   - This file was used as a template to instantiate our custom packet detector IP core into the design. A top-level output port packet_valid was added to route the packet detection signal from our IP core to higher level modules.

2. sdr_lib/dsp_rx_glue.v
   - This file was modified to route internal packet_valid from the custom_dsp_rx core to higher level modules.

3. sdr_lib/ddc_chain.v
   - We commented out the work by Caitlin Miller regarding “timestamp calculation”, as we now use the vita_timer to provide timings of the packet detection signal in higher level modules. Thus, timing calculation is no longer necessary here. See section 4.2 below for more information. Note that the vita_timer automatically sinks to the GPS PPS reference if the GPS module is present for UHD builds 3.4 and later versions.
     - Internal packet_valid signal from dsp_rx_glue was routed to higher level modules.

4. top/N2x0/u2plus_core.v
   - At this module level, packet_valid signal from ddc_chain is routed and re-distributed internally to the ZPU interrupt vector and vita_rx_chain for timing calculations of packet detection time. See section 4.2 for details on how timings are calculated and sent back to host.
     - In addition, the debug vector was modified to reflect only current status of “detection timestamps FIFO” in the vita_rx_chain. The debug vector is all ‘1’ if the FIFO is full, and all ‘0’ otherwise. This change was made for debugging purposes.

5. top/N2x0/u2plus.v
   - This top-level module was only modified to output packet_valid signal from the down converter DSP chain to an output port. This signal is then mapped to an
external LED pin in the system ucf file for visual indication. Note that the ETH_LED signal was commented out to make place for the packet_valid signal usage.

**Hardware Debugging Framework using Xilinx Chipscope and Core-Inserter**

Our hardware development process was tested using Xilinx Chipscope, Core-Inserter, and Analyzer. Chipscope is a hardware debugging IP that enables real-time monitoring of signals through the JTAG header. A Xilinx platform USB cable was used to transmit JTAG information back to the Analyzer software running on host. This framework allowed us to perform in-fabric hardware debugging at run-time.

We used Xilinx Core-Inserter to insert a Chipscope core into a post-synthesis, netlisted design. We must synthesize the design first before inserting chipscope this way. The detailed process is as follows:

1. Run the build process for custom uhd fpga build using instruction from section 3.2 above.

2. Open the generated Xilinx “ISE” project file in top/build-N210R4/u2plus.xise in ISE Project Navigator. Add a new Chipscope Definition Core (a “.cdc” file). This file should be saved in the custom/ folder. A sample core called “chipscope.cdc” is provided in the same folder.
   
   - This core allows us to specify Chipscope ILA and ICON core connections, as well as the desired nets we are interested in monitoring at run-time.

3. After modifying the ISE project, we need to incorporate the Chipscope core into the Make build process. Modify custom/Makefile.srcs to add the Chipscope custom source file.

4. Restart the build process from the top/N2x0/ folder. Chipscope cores should be generated and included in the design this time.

5. Start Xilinx Chipscope Analyzer on host, set up trigger conditions and start monitoring hardware signals.

**Getting Detection Timestamps Back to Host**

Each time the packet detection signal is triggered, we record the timestamp of the event using the vita_timer and store it in a FIFO of length 16 for transmission back to host. We incorporated the timestamp transmission seamlessly into the receiver chain by modifying the vita_rx_chain module.

The default UHD VITA RX chain is shown in Figure 7. The VITA RX Control block adds timing information for each sample output of the DSP RX Core. Most of these timing information are discarded, and only the timestamp of the first sample in each VITA frame is recorded in the “timestamp” field of the VITA frame. Currently this timestamp information is ignored during host-side processing.
To get our packet detection timestamp information back to host, we make use of the “timestamp” filed of the VITA frames. Each VITA frame “timestamp” is modified to store the timestamp of the last packet detection time. On the host side, the uhd drivers are modified to constantly watch these timestamp values, and only record new timestamps if there are a change in the VITA “timestamp” field. The host driver modifications are documented below.

In addition, to keep up with a granularity of the packet detection timestamps (which can happen every 10-20 sample time), a size-16 FIFO is added to buffer the timestamp values and stream them out sequentially following each VITA frame.

The following Verilog files are modified for packet detection timestamp transmission to host:

1. **vita_rx_framer.v:**
   - Modify the VITA state machine to record timestamps based on packet detection signal, add the timestamps into a FIFO, and sequentially stream them out in each VITA frame.
   - Note that the time information streamed out here is not the default timestamps provided by the VITA timer. We clear out the MSB (most significant bits) 4-bit of the 64-bit timestamp field (currently holds only zeros), and put in our detection type information (4-bit). See Figure 8 below for the modified timestamp structure.

2. **vita_rx_chain.v:**
   - Modify debug vector to route debug information to higher-level modules.
3. **vita_rx_tb.v:**
   - Create and modify testbench for the *vita_rx_framer* module.

![Figure 8: Packet Structure for Timestamps Sent to the Host](image)

Fig. 8 displays the structure of the timestamps being sent to the host as packet detections are happening. As it can be seen, it is a 64 bit structure, where 60 bits are reserved for timing information. The 4 most significant bits are reserved for flagging the timestamp for specific type of detection. Bit 60 is reserved and is forced zero (0).

## 5 Host Driver Modification: UHD and GNU Radio

The UHD host drivers (stored in `uhd/host/lib`) and GNU Radio UHD library package (stored in `gnuradio/gr-uhd`) are modified to accommodate timestamp transmission from UHD hardware. The UHD drivers now simply watch the “timestamp” field of each VITA frame, detect a change in the timestamp, record them as new packet detection time in a C++ queue data structure, and pass this information back to higher-level processing in GNU Radio.

### 5.1 Build and Install Custom Host Drivers

To build and install the modified UHD host drivers, go into `uhd/host` folder:

```
mkdir build
cd build
cmake ../ -DENABLED_EXAMPLES=OFF -DENABLE_TESTS=OFF
make
sudo make install
```

GNU Radio `gr-uhd` package is also modified and must be rebuilt and re-installed. This build is automatically included if you build and install the complete customized GNU Radio library as instructed in section 2.2.

If you make modifications to only the `gr-uhd` library, then only this library needs to be rebuilt to reflect the changes. To do this, go into the `gnuradio` folder:
After this, we should be able to run all the applications with the new libraries and host drivers.

5.2 Details of Host Driver Changes

UHD Host Driver Changes

The timestamp and detection type information is passed to uhd host drivers from FPGA hardware via the Ethernet interface. Each incoming UDP packet includes metadata, which contains a 64-bit timestamp detailing the last detection time (LSB 60-bit) and the actual detection type (correlation, energy hi-low, etc.) (MSB 4-bit). The host drivers are modified to catch this information, extract the detection type and store in a separate field called detection_type of rx_metadata_t, and then correct the timestamp value to 64-bit. Lastly, just before adding the timestamp second and fractional fields into the C++ queues to pass to GNU Radio processing, the uhd drivers re-attaches detection_type information (4-bit) into the MSB 4-bit of the 64-bit second field. The GNU Radio gr-uhd library is responsible for extracting this detection information out of the second field before processing detection timestamps out.

All code changes happened in the uhd/host/lib and uhd/host/include folders. Search for “DWSL” to see code changes.

1. include/uhd/types/metadata.hpp:
   - Add a detection_type field of type char to the rx_metadata_t structure.

2. super_recv_packet_handler.hpp:
   - Adding C++ queue data structures for catching second and fractional timestamps passed from the FPGA hardware.
   - This change in API results in multiple escalated changes necessary in the function signature at other upper level files (search for “DWSL” in uhd/host to see all files that get changed. Except for the ones noted here, most other files were changed in only minor details to accommodate new driver API signatures.
   - Timestamps catching and re-package happens in recv_one_packet() function.
GNU Radio ‘gr-uhd’ Library Changes

All changes to GNU Radio ‘gr-uhd’ happened in the `gr-uhd/lib/gr_uhd_usrp_source.cc` file. Search for “DWSL” to see details.

- In `work()` function, enable tag information to always get process per receive buffer (4096 samples). Process metadata tags by passing in 2 C++ queues: one for `timestamp_second` and one for `timestamp_fraction` when making the UHD API driver calls.

- Upon returning from the driver calls, extract second and fractional timestamp information from the 2 queues, post-process the `timestamp_second` field to extract `detection_type` (4 MSB bits), then print them.

6 ZPU Firmware

The ZPU firmware was updated to handle the interrupt flag related to packet detector. As explained in the previous sections, each time a packet is detected, an interrupt is triggered. This interrupt is then caught by the ZPU and the handler is called. It should be noted that USRP interrupts are handled differently than any other interrupt on a microcontroller. USRP interrupts can be interpreted as a polite reminder to the ZPU that something has happened and that it probably should be taken care of. Therefore, they can’t be used for time-sensitive applications. However, for debugging purposes this method works great, and, in fact, it is still integrated into our design.

6.1 Packet Detected Interrupt Handler

As explained in the previous section, the cross-correlator packet detector block in our hardware design outputs a flag signal. This signal goes high every time the cross-correlator output is above the set threshold. This packet detected is then passed onto the interrupt vector defined in the `u2plus_core.v`. It was noticed that many of the available bits in this vector were still available for user usage for additional programming. The 15th bit of the interrupt vector was changed to be the `packet0_valid` flag. This can be seen by searching the file for ”irq”.

Once the hardware was configured to modify the interrupt vector, the ZPU was programmed to interpret this interrupt request. The following files were changed to provide this capability:

`memory_map.c` The memory mapping of the new interrupt flag is done in this file. Location 15 was defined to be `IRQ_PACKET_DET`, this definition was then used for the right amount of shift to extract the interrupt flag.

`pic.c` The number of bits being read from the interrupt vector has been changed to 16. The original setting was 8. Also `PIC_PACKET_DET` was added to the enabled interrupts. See the line that changes the edge_enable of the `pic_regs` data structure.
net_common.h  A new unsigned integer was defined for internal processing.

net_common.c  This is the main file where all the Ethernet frames are being compiled. Our plan was to utilize the unused 1 flags in the IPv4 header. The priority bit was toggled to represent the packet detection. IPH_VHLTOS_SET macro is used to add the packet detected signal into the Priority flag of the IPv4 header. This can then be captured by any packet analyzer on the host side. For our testing purposes, we used a Java program that we wrote that utilized the jpcap library (http://netresearch.ics.uci.edu/kfujii/Jpcap/doc/index.html) to analyze the received Ethernet frames on the host side. This code is also provided with the rest of our testing scripts.

txrx_uhd.c  This is the main file that’s run once the ZPU boots up on the USRP N210 board. The main while loop was changed to toggle the integer defined in net_common.h. This is then used for internal calculations.

7  Host-ZPU-Hardware Communication

One of the most vital parts of this project was to be able to change the cross-correlation values that were being stored in registers on the USRP board. The UHD comes with a variety of commands that’s available to the users. As explained in the previous sections, all the cross-correlation values were stored in "user registers" on the USRP board. These are registers that are connected to the wishbone bus and can interact with the ZPU. The basic idea here is to instruct the ZPU to change the register values. In order to do this, we used the UHD calls to open our USRP device. Once the handle was received, the set_user_registers() function was used. The function takes two arguments: (1) address of the register and (2) the value that’s going to be written. Once this was tested and confirmed to work, these functions were integrated into the main GUI script. Refer to the next section for more details regarding the graphical user interface usage.

NOTE: Due to the way our registers were defined, there has to be two register writes that needs to happen each time. First write is to set the actual register value and the second one to move the address pointer back to 0 while still keeping the same register value. Unless this is done, you will notice the register value to be fluctuating. See the main Python script for example usage.

8  GUI Development and Data Collection

Our group also developed a graphical user interface and a MATLAB based data collection system. These two components complement each other and, therefore, will be explained in the same section. Figure 9 shows the integration of the components.

---

1These flags are defined in the IPv4 protocol; however, now that every router used these days is doing "best effort" service, no body uses the quality of service flags anymore.
8.1 GUI Development

For the GUI development, the gnuradio-companion (GRC) was used. The standard uhd_fft.grc design that came with the example designs was modified to include additional dropdown boxes, textboxes, and buttons. The design was modified to incorporate the functionality needed for changing the cross-correlation register values. Energy detection and packet detector threshold values are also changeable using the GUI along with activation and de-activation of the different types of packet detectors. Figure 10 shows the final state of the GUI.

The GUI seen in Figure 10 was created with GRC to place all the graphical components. Using GRC, the accompanying Python script was generated. All other modifications were made on the Python script. It should be kept in mind that the current Python script shouldn’t be modified in the GRC any more, since this will overwrite all the manual changes.

8.2 Data Collection

The GUI script explained in the previous section outputs its timestamp values to the stdout. Fig 11 provides a sample screenshot of the outputs. The format of the outputted messages / timestamps can also be seen in the same figure.

The data collection portion was handled by writing MATLAB scripts. As shown in Fig. 9, a pipe is used to redirect all the Python stdout printouts into MATLAB. The pipe is then read in MATLAB as a regular file. We first open the file and read lines off of it one by one. The read lines are then processed to extract the timestamps. There are two different versions of the MATLAB script:

stdout_read.m  This is the main file that will handle reading values and storing the timestamps. This file will also plot a scrolling figure to mark the timestamps visually.

stdout_read_without_plot.m  This file is similar to the previous one with one exception: There is no plot output. The advantage of using this file is that it is fast enough to catch up
Figure 10: GUI Used for Testing Packet Detection.

Figure 11: Sample Timestamp Output shown on Terminal.

with the speed of the USRP receptions. *Unless a visual demonstration is needed, it is highly recommended that this file is used.*
MATLAB scripts also handle creation and deletion of the fifo pipe. Every time the script is executed, it deletes the pipe and creates it again. This is to ensure that no leftover data is being transferred from one experiment to the other.

8.3 How to Run Experiments

All that needs to be done to collect data using our setup is the following:

- Run the MATLAB script that will collect the timestamps
- Run the Python script that will initiate the packet transmission and to collect the power data. Use the following code to run the script:

```
python -u uhd_fft_with_WiMax20.py >& CONNECTOR_PIPE
```

The -u flag is to run Python in an unbuffered fashion—i.e. We want the GUI to push all the output to the `stdout` as soon as something is ready to be printed. Once the experiment is done, the items above should be stopped in the opposite order.

- Stop the Python script
- Stop the MATLAB script

After these steps, the power data will be saved in ”values.dat” which is going to be in the same folder. The timestamps can also be found in the vector ”triggers” saved in MATLAB.

9 Resources

9.1 Books


9.2 Links


28
9.3 Questions?

Feel free to contact any of us using the email addresses above.