Abstract:
The goal of this project was to implement the computation of frame synchronization in an FPGA to reduce the processing burden on the CPU of an SDR (Software Defined Radio) platform, and to exploit the parallelism inherent in a hardware implementation. Thorough consideration of all possible options, Vivado High-Level Synthesis was used to compile C++ code directly into HDL code, and GNU Radio was used to describe the movement of signals on the top level design and to place RFNoC blocks in, which are designed utilizing USRP Hardware Driver (UHD) code.

1. Introduction
A prominent application of Software-Defined Radios (SDR) is for dynamic spectrum sensing. This is the process of sensing vacant “blocks” in the RF (Radio Frequency) spectrum, in the time and frequency domains, of varying sizes. A radio node then accesses one or more of these blocks to transmit a message. The message will be sent along with a preamble, which is a unique short signal that is prepended to this message. All other radio nodes in a network of nodes will scan the entire RF operating spectrum, typically around 100MHz, looking for the preamble that relates to them.

However, the signal processing computations required for the preamble search takes a significant amount of time. Due to this constraint, there is an opportunity to improve the performance of the current implementation of signal processing in SDRs by utilizing components which are already contained in the NI/Ettus X310, but are not currently being used to their maximum efficiency. The Kintex-7 FPGA (Field-Programmable Gate Array), which is included inside of the X310, can be programmed to handle the process of detecting and extracting an embedded message from an analog front-end that receives wireless signals. This process can bottleneck the performance/throughput of the CPU (Central Processing Unit) when using a software solution rather than hardware solution. Also, since the X310 already provides compatibility with GNU Radio, we were able to seamlessly interface RFNoC (RF Network on Chip) into our system, which enables the integration of our FPGA into the USRP to process the incoming data stream. Accelerating the detection of an embedded message allows the software to execute other DSP (Digital Signal Processing) operations.
2. Project implementation
For this entire project the development platform was a Dell R730 server which is hardwired into the USRP (Ettus Research X-310). The USRP uses, at the highest level, the hardware drivers which are referred to as UHD. These drivers are programmed, in our case, via GNU Radio. The radio drivers, UHD, interface with the user defined Verilog files. Shown above is a picture of the Server and Radio that were used in this project.

The overall objective in terms of development was to write C++ code and use high level synthesis (HLS) to turn that code into Verilog code. This Verilog code, which is essentially a function, is then paired with a NoC Block. This NoC Block is a Verilog file that calls the function from the HLS Verilog file and integrates this with the higher levels of RFNoC. This is the process that ties the C++ code to the FPGA. The image in Figure 1, credit to Ettus Research, shows in a pictorial form the process that does this. The diagram shows the separate layers that are in play. The whole lower 9/10ths is the NoC Block file. Inside this file, there are different layers that are all wrapped together. In the bottom we can see the user IP, which is our HLS-generated Verilog file.

Figure 1: Turning a C++ code onto FPGA

The data that is inputted to the USRP via an antenna is passed to various radio peripherals. This manipulates the data to be in the form of I/Q samples. For this project, the data is oversampled 16 times, which are referred to as phase classes. These 16 phase classes are essentially representing the same input value. For the Matched Filter, having the 16 phase classes does not affect the logic. However, for the Correlator, the phase classes must be separated into 16 data flows, which are each correlated separately and have their own correlation output power. Using GNU radio to check the input signal to the USRP, the input is a complex I/Q signal which contains a high level of noise.

The primary step in development was the creation of MATLAB simulations of the Correlator and Matched filter. This process was a key component in developing the approach that was used for both blocks. These simulations also allowed the results of our design to be tested against data that can be easily manipulated. In both cases, the function of both blocks is the same as the MATLAB simulations, but modified to work in a way that is better suited for hardware development.

The first type of simulation that was considered is called a “C simulation”. This is the process of testing the high-level code, the C++ code. This is a high-level test that compiles the C code and runs it. There are outputs in the below sections that detail the results given. It is important to note that these simulations do not simulate hardware, and thus do not reflect final implementation, but they do test our core logic. In order to simulate the hardware, a different type of simulation called a Cosim, or co-simulation, was used. This simulation tests the C code in conjunction with Register Transfer Language. This is similar to running “testbenches” in Xilinx. The output of these tests was a “waveform.” This simulation, if successful, ensures that the design can be implemented on the FPGA. It is also a vital step because the final test is flashing the FPGA with the desired block.

The process of creating the image to flash can take upwards of 1 hour. It involves selecting the RFNoC blocks and NoC blocks to be loaded into the image, and thus onto the server. A picture that shows the selection of these blocks is shown in Figure 2. After the blocks are selected the image is generated, which can take around 45 minutes or more. This is where any errors will be discovered as the process involves checking all relevant files for
syntax or linking errors. After the FPGA image has been created it is then loaded onto the FPGA using UHD commands. The process of loading the image can take around 10 minutes, so in total it takes around 1 hour to generate and load an image onto the FPGA. Before live data testing can commence, a couple more steps are necessary after we reboot the radio. The XML files that control the behavior of the NoC block need to be edited, since these blocks link the IP loaded onto the FPGA with GNU Radio along with the radio drivers, allowing them to be referenced by GNU radio. Finally, the blocks can be used in testing, using GNU Radio.

The data that we send into the GNU Radio block, which contains the developed FPGA implementation, is generated using a Python script. All the streams in GNU Radio use complex 16-bit integers. It is important to note that this has the same functionality as a complex short in C++. To create this complex short in Python, 16-bit integers need to be encoded one at a time, and when the file is read, it is read 32 bits at a time in GNU radio so that two groups of 16 bits are read and fed into the complex int16 data stream for the developed GNU Radio block. Then this GNU Radio block, which contains the team’s FPGA implementation, outputs its relevant data in the same format and that data is collected using a GNU Radio default block called “File Sync”.

For all the simulations and tests, the same set of input data was used (Figure 3). It consists of the full preamble followed by some high noise, followed by a lower amplitude preamble. This is an unrealistic scenario using realistic data. This is only to verify the developed code with high certainty. For the input data to the correlator, the output of the matched filter was used. This can be seen in the Matched filter section below.

3. Correlator

The algorithm used for correlator is premised on splitting the incoming data into 16 phase classes - data streams - where a phase class is comprised of samples which have the same sampling period. This is done because the signal
is 16 times over-sampled. For example, the first sample coming in is phase class 0 sample 1 and the 17th sample is phase class 0 sample 2. Each time additional data is inputted, it is shifted into its phase class from the right, and this simulates the shifting involved in correlation. After the data is shifted in, the sum of the phase classes is taken, and this is the correlation output. The correlator (Figure 4) was implemented in Matlab and was used as a baseline for future Vivado HLS simulations/implementations.

**Level-2 (Correlator)**

Figure 4: Correlator architecture

Figure 5 shows the final MATLAB simulation output of the Correlator. It is important to note that this is the correlation power for a single phase class. The correlation power slightly varies for each different phase class. An additional image displays the results for a different phase class.

Figure 5: Sample output of a Correlator

Once the correlation algorithm was verified through MATLAB testing, the next step in the team’s development was to build this algorithm in Vivado HLS using C++ code. In order to build the correlator, one function was made with a state machine that would perform the shifting and calculating of the correlation power. A correlator object was also created which contained helper functions and fields that are essential to the correlator. Once the development of this C++ code was complete, it was tested using Vivado HLS C-simulations. The results of the C-Simulation are shown in Figure 6. However, the most useful information that was gathered from these C-Simulations was that the data-types that would be used on the FPGA could be used in the C++ implementation. This allowed the data to be lined up correctly for the AXI stream interface using fixed point arithmetic.

Hardware definitions were added to the code and were synthesized in Vivado HLS. These hardware definitions included techniques such as unrolling and/or pipelining for loops, and arrays were partitioned to be registers. Then, a clock period of 5ns was chosen and a synthesis was run. Synthesis in Vivado HLS provides the ability to convert C++ code into Verilog, VHDL, and System C descriptions. In the case of this project, Verilog files would be going onto the Kintex-7 FPGA, however these files required testing first. To test the hardware description, Cosimulations and RTL simulations were run, which synthesizes the testbench to call and test the C++ implementation and
generate a waveform. For the Cosimulations, a detector was implemented, to look for the position of the spike in correlation power in the stream of data.

Now that the Verilog files were verified to be correct, the implementation on the Kintex-7 could be tested. To do that, an X310 image was built, which included the correlator block. Once this image was built and flashed onto the SDR, the correlator block was placed in GNU Radio and run using test input data. When correctly implemented, the output of the data from the FPGA on the SDR can be observed in Figure 7.

4. Matched filter
A matched filter is a system that takes in a noisy signal and filters it to enhance the signal to noise ratio where the signal is in a known form. This process is done by simply convolving a noise-free version of the signal to be detected, which is the preamble in our case, with the entire input. Consequently, all data being input to the radio is being convolved with the preamble signal.

The team needed to develop convolution logic from scratch. For this project, it was more efficient to use linearity
properties of convolution to take each sample as it comes into the system, convolve it with the entire preamble and then add it back to the other samples. This implementation creates a system that can manipulate data as it comes in, instead of waiting for data to come in before performing necessary calculations. The results of the system logic that was developed can be seen in Figure 9.

![Graph](image)

Figure 9: Match filter simulation results

5. Experimental validation
The first experiment ensured that the team could develop in Vivado HLS, generate hardware descriptions, and run testbenches on those hardware descriptions to properly implement the desired functionality on the FPGA using GNU Radio while evaluating the results through the usage of graphs and logs.

The second experiment was used to prove: (1) We can use the skills learned from completing the first experiment to create a different block for filtering and correlation, and (2) The blocks created were viable and were able to be used effectively. Schematics for the second experiment is shown in Figure 10. The first step in validating our test was to look at the data coming out of the matched filter. It is expected that the data coming out will be filtered and have a high SNR and look almost exactly like a non-noisy signal. This result is verified in Figure 11, which compares a noisy input to the clean output that comes from the FPGA. Looking at this plot we can then claim that this part of the test was a success and that our Matched Filtering block is working.

![Diagram](image)

Figure 10: System diagram

The second half of our verification of the experiment was to look at the output of the correlator after the above data has been passed into it. We should see two peaks that are 16 samples apart and the value of the peak should be significantly higher than the base power of the output. The exact values will not match up due to the way we are interpreting the binary representation of the data. Figure 12 shows the output of the correlator that is implemented on the FPGA. We can clearly see that the data matches the expected value well. This means that the correlator is deemed as working in conjunction with the matched filter. This means that the entire system is working as expected.

For the final test, the process was to look at the relative speeds of the various systems used. The FPGA implementation worked within our 5 nanoseconds clock period target. The Matched Filter takes 133 clock cycles
to get through the pipeline. The correlators take 14 clock cycles to get through their pipeline. So for one sample to get through the pipeline of the matched filter and correlator it would take \((133+14)\times 5\) nanoseconds = 735 nanoseconds to get an output from the sample in. This doesn’t take into account any handshaking that the AXI stream or GNU Radio does because we do not have control over that. Table 1 shows the average times for software simulations of the correlator and matched filter.

![Input Data and Filtering Sequence](image1.png)

![Matched Filter Output From FPGA](image2.png)

**Figure 11: Matched filter results**

![Output of USRP](image3.png)

**Figure 12: Correlator results**

**Table 1: Time taken by simulations**

<table>
<thead>
<tr>
<th></th>
<th>Correlator Csim</th>
<th>Matched Filter Csim</th>
<th>Correlator Matlab</th>
<th>Matched Filter Matlab</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Time(ms) Per Sample</td>
<td>0.0127156</td>
<td>0.146062</td>
<td>0.0433198</td>
<td>0.0173541</td>
</tr>
</tbody>
</table>

6. Conclusions

Successful completion of this project would benefit various aspects of different industries and fields of study. A prominent area of impact would be in military applications, due to the fact that the military is tasked with facilitating communications that require high levels of security, reliability, and most importantly, speed. A radio communications system that revolves around the integration of both software and hardware is inherently more secure based on the fact that it is much more complex because it utilizes several different integration components that make it more difficult to infiltrate. By implementing hardware acceleration in military radio communications, response time could be improved dramatically, and since many military operations consist of high danger scenarios, it is vital that they are able to quickly react and communicate problems that come their way. Another meaningful effect our project could have is on the upcoming evolution of mobile communications, 5G.