Transmit Dongle for Communications Lab

B-16
Abhin Shah - 140070013
abhinshah02@gmail.com
Karan Chadha - 140070014
karanchadhaiitb@gmail.com
Kalpesh Krishna - 140070017
kalpeshk2011@gmail.com
Faculty Mentor: Prof. Shalabh Gupta
RA: Shubham Dhage

08 April 2017

Abstract

Arbitrary function generators(AFG) are used to generate baseband analog signals. Besides being costly equipment, the AFG cannot be interfaced directly with our PC/GNU-Radio. As of now, we need to first synthesize the signals, transfer them to the USB stick. This stick is then connected to the AFG which serves as an input to a Modulator board for transmission. To achieve this end goal, we generally do not need the complicated functionality offered by an AFG. In our project, we wish to build a low-cost portable transmit dongle, which could be used in the communications lab at teaching institutes. Waveforms generated from the laptop by the user are given as input to the transmit dongle. The dongle would then give a RF modulated output which can be transmitted using an antenna.

Contents

1 Introduction

1.1 Project Objective

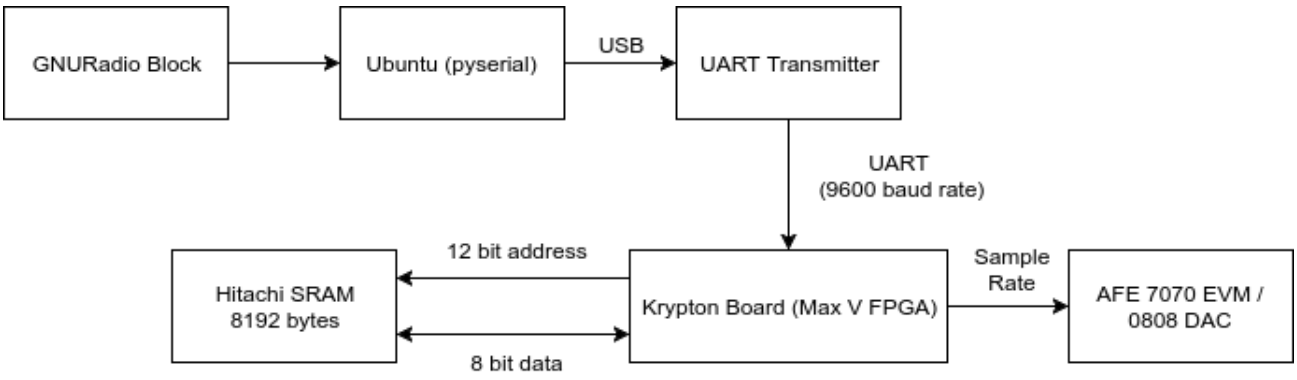
Our project goal was to achieve the following - A low cost, portable transmit dongle compatible with GNURadio which can be used for our communication lab[EE340] at the very least. We hoped to produce and transmit signals up to 100 KHz.

We divided the solution to this problem into four larger components -

1. **Open Source Software** - Generation of digital samples using Python/GNU-Radio. This software is meant to be compatible with any PC / OS.
2. **Communication** - Communication between PC/GNU-Radio and FPGA via a suitable communication protocol like UART.
3. **Buffering and DAC** - The microcontroller/FPGA stores a copy of these samples (buffer) in an SRAM and finally outputs them onto a DAC, which converts them to an analog signal.
4. **Modulator** - This component takes the analog signal and a local oscillator frequency as the input and outputs/transmits a modulated wave.

In our project, both the points 3. and 4. are achieved by the AFE7070 Evaluation Module (by Texas Instruments).

1.2 Block Diagram



As shown in the above diagram, at a higher level the project flow is as follows:

- Firstly, desired waveform can be generated in GNURadio and dumped into a custom made GNURadio block. This block internally sends data to the USB port using pyserial.
- On the USB port is a USB to UART converter, which has an FPGA connected on the other side for transferring the digital data samples.
- The FPGA reads the data received via UART and stores the cycle of the periodic input in the SRAM. When the transmission of the data is com-

plete, the FPGA outputs the equally spaced data on data pins at the correct specified sample frequency.

- The output of the FPGA is given to a DAC which converts the digital data into analog signal and in the case of AFE 7070 it also modulates the data with the local oscillator frequency given as input to the module and outputs/transmits the modulated signal. In the case of 0808 DAC circuit, it outputs the analog signal after passing through a series of low pass filters for smoothening of the output.

1.3 Motivation

Arbitrary Function Generators (Arbitrary Waveform Generators) are generally complicated and expensive devices which offer a large variety of useful functions. Unlike function generators AFGs can generate any arbitrarily defined waveshape as their output. The waveform is usually defined as a series of "waypoints" (specific voltage targets occurring at specific times along the waveform) and the AFG can either jump to those levels or use any of several methods to interpolate between those levels.
Today, a number of excellent AFGs are available in the market designed by Textronics, Keysight and other companies. Most of the modern AFGs have proprietary software which can interface with the AFG hardware to generate arbitrary waveforms. There are some portable AFGs as well, such as the Hantek DDS3X25. A few AFGs have an in-built Windows operating system as well!
However, these devices suffer from the following problems -

1. **Proprietary** - The software for these products is proprietary and cannot be dissected easily by an end user. This makes it difficult to interface this with open source software like GNURadio. Although the software for generation of the digital samples can be open source, but these products accept data only in specific formats, conversion of samples to which is proprietary.
2. **Expensive and Overkill** - The complicated hardware is very costly. Essentially, we are using the AFGs since we have them available in the lab for other purposes. Buying an AFG just for this purpose is an overkill for EE-340. It would be much more convenient to provide input to a hardware directly using our PC.

Our project has the following advantages over the above:

1. **Open Source** - The software is completely open source and can be used by anyone to generate signals as long as he/she has the attached board.
2. **Low Cost Kit** - The board will have a FPGA and a DAC/Modulator. Since it is indigenous, it will be a low cost device and easy to interface.
3. **EE - 340** - Using this kit will eliminate the use of an AFG for many of the EE-340 lab experiments. It will be much easier now, since one can directly use GNURadio to do the trick.

The low cost receive dongle(10$ RTL-SDR) is available, but a similar low-cost hardware is not available for the transmitter. Expensive alternatives (USRPs) cost around USD 1500, and therefore can't be used on a mass-scale. This motivates us to make a low-cost RF transmit-Dongle.

1.3.1 Old Implementation

This project was pursued last year in EDL (see the report here) and achieved some success. More specifically,

1. **Communication** - They succeeded in taking samples from a custom GNU-Radio block and transferring them via USB to a Tiva microcontroller. The Tiva code was written using the Energia interface.
2. **AFG Waves** - They obtained standard waveforms such as sine, square, triangle and sawtooth at certain frequencies less than 30 KHz.

The limitations in their version of the project were -

1. **Wrong Sampling** - While transmitting from GNU-Radio 8 samples were obtained per wave when sampling rate is 10 times the frequency. Ideally 10 samples per period should have been obtained. Besides this, the product only worked at certain discrete frequencies.
2. **Slow Speed** - Since USB 2.0 was used, they could not achieve frequencies higher than 30 KHz.
3. **Irregular Waveform** - There were spikes in the waveform and it is difficult to filter out high frequency glitches.
4. **TIVA** - Since Energia was used, the full functionality of TIVA was not used (such as Port C and JTAG). Also, a number of TIVA boards were lost in the process of building this.

After their project, the WEL Lab RAs worked on this project. They were able to fix some of the problems, especially # 2 and # 4. They wrote a new non-Energia TIVA code which could transfer data via Ethernet at higher speeds. However, they could not achieve perfect sampling or a regular waveform.

2 Project Design

The first part of this section describes the FPGA subsystem. The second part of this section describes the AFE-DAC subsystem.

2.1 FPGA / GNURadio Subsystems

The FPGA system is a digital circuit design that's been written in VHDL. We have open-sourced the code and you can have a look at the Github. `https://github.com/Abhin02/EDL/tree/master/uart_receiver`. This part of the documentation explains the major components of the digital circuit design along with the the motivation.

2.1.1 Motivation for FPGA

A major subsystem in this project is a "sample-player", a system which can accept a vector of sample values from a Python script running on a PC and output them on a port at a constant rate. A standard approach to solve this problem would be using timers on a microcontroller which supports some form of communication with the PC (generally Ethernet, UART or USB). We chose the avoid this approach for the following reasons -

1. **Accuracy** - We initially started working on the Tiva-C based microcontrollers (prior to EVAL-1). What we realized was the difficulty in reaching high sampling rates. While the microcontrollers performed well at lower frequencies (close to 50 kSamp/sec), we were not getting accurately spaced samples at higher frequencies (closer to 500 kSamp/sec). Additionally, software delays via the
   Typical GNURadio applications do require sampling rates upto 1-2 MSamp/sec and an FPGA seemed to be a better choice. With the FPGA system and a 0808 DAC circuit, we have been able to achieve *extremely* accurate signals upto 3 MSamp/sec. The FPGA is expected to work upto 6-7 MSamp/sec with a more powerful DAC. Exact calculations have been given later. This could *potentially* be optimized further, as will be described in the *Future Work* section.
2. **Simplicity & Flexibility** - Microcontrollers are generally complicated systems used to provided solutions to a large number of problems. As a result, it's documentation is often large and difficult to fully understand in a limited time frame. Since we've extensively worked on VHDL code in the past, digital design seemed like a more viable option to get a system up and running quickly.
   An FPGA would also provide us with more flexibility, since we would be designing a system **from scratch**, giving us a chance to optimize a lot more.

On the downside, FPGA systems often need a lot more planning, coding and are sometimes difficult to debug. A lot of VHDL code is often not synthesizable,

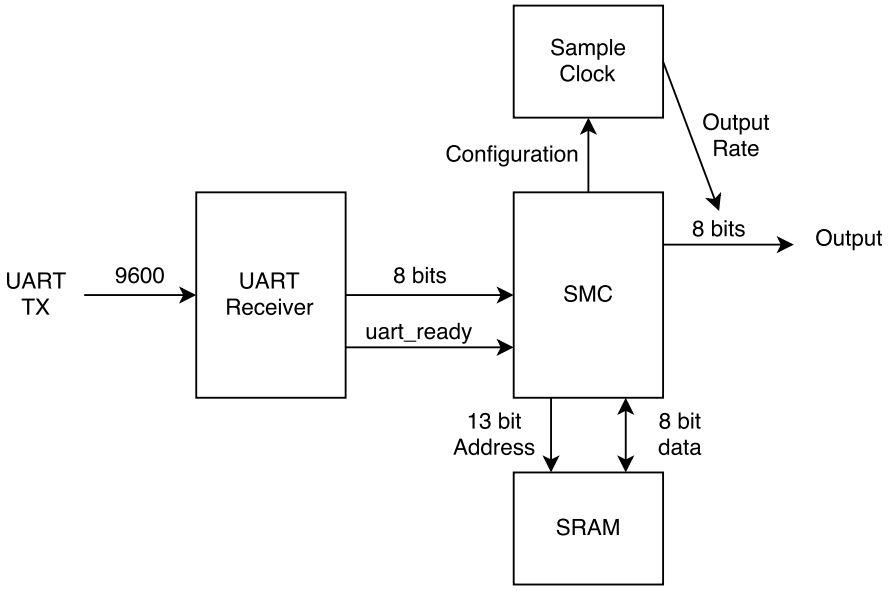
leading to stricter coding guidelines. Fortunately, two prior projects in VHDL had taught us basic tricks of the trade.

2.1.2 FPGA Specifications

In our current implementation, we've used the WEL Lab's `https://www.ee.iitb.ac.in/~wel_iitb/resources_development_boards.html`. Strictly speaking, this is a CPLD, and not an FPGA but we will stick to the loose FPGA notation since the code is general enough to work on most CPLDs / FPGAs. The Kryton Board uses a Altera Max V CPLD chip, which works on an internal clock of 50MHz.
Additionally, the Krypton Board has other developmental features such as on-board switches, LEDs, and ports that have been used in our final implementation.
The FPGA is programmed via urJTAG, using FT2232 as a USB-JTAG bridge.
We are using 13 bit addresses of a Hitachi SRAM to store our samples, giving the system a capability to store upto 8192 samples.
Finally, our frontend has been implemented in GNURadio and uses `pyserial` internally.

2.1.3 FPGA Workflow

On a higher level, the digital circuit is a "sample-player", that can play samples at rates upto 5 MSPS. The following `TopLevel` block diagram shows the different modules and their links.



1. **UART** - The system starts on GNURadio, where we've designed our custom block to send samples via UART via an external USB-to-UART converter. The block uses `pyserial` internally, and more about this block in the *Implementation* section. This is running at a configurable baud rate, which is 9600 currently.

2. `UARTReceiver` - This is a VHDL module built by us, designed to receive UART signals. It is currently tuned to accept baud rate 9600, but can be modified with edits in VHDL. This module raises a flag `uart_ready` whenever the next byte of data is produced on a port.

3. **SRAM** - This is an external chip which is interfaced to HEADER2 of the Krypton Board. It has a unidirectional address bus of 13 bits (a total of 15 bits, but we are using the lower 13 bits) and a bidirectional 8 bit data bus. This stores all the samples.

4. `SMC` - This module is the SRAM controller. It is responsible to accept UART samples, feed in sample rate to registers, read/write from/to the SRAM and finally output the samples to the external port at the sample rate.

5. **Sample Clock** - This is a part of the `SMC` in the final implementation. It is responsible to generate a clock having frequency identical to the sample rate. As a result, this clock is configurable by the `SMC`, and the sample rate is sent out to the `SMC` using UART.

The next sub-sections will go over each of the design strategy used for each of the mentioned components. It will not have specific implementation details since that would be covered in the *Implementation* section.

2.1.4 GNURadio - UART

End Goal - *To build a GNURadio block that can send samples to the FPGA board over UART. This block should be convinient enough to interface with other GNURadio signal processing blocks and should have a mode to support I-Q channel interleaving as well.*
Design - Clearly, this block has to be designed as a *Sink* block. Since it can potentially send both I and Q channels interleaved, it needs to have two input ports. The parameters needed would be,

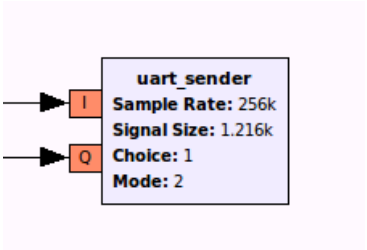
1. `sample_rate` - This needs to be communicated to the FPGA (who will finally output the samples at this rate). This is sent in the *Configuration Mode* of the system.

2. `signal_size` - The total number of samples in the signal. The FPGA board will continuously play these samples in a loop. The GNURadio block takes only the first `signal_size` samples and ignores everything else.

3. `mode` - For `mode` = 1, samples are output at a rate `sample_rate` * 2 and I and Q channels are interleaved. For `mode` = 2, the Q channel signal is ignored and I is played at `sample_rate`.

4. `choose` - This is interfaced with QT GUI Chooser and is used to select the step of operation of the GNURadio block. For `choose` = 1, the GNURadio block is in an idle state. For `choose` = 2, the GNURadio block outputs the `samp_rate` and `signal_size` to the FPGA module. Finally for `choose` = 3, the samples are sent.

Note - The block must be re-started for a change in configuration of samp_rate or signal_size, since the block sends this data ONLY ONCE.



The I and Q channels must receive periodic signals (or aperiodic signals with less than 8192 samples). The length of one period is specified in the `signal_size` parameter.
Pre-Processing - Finally, it is essential to convert the float values to the ASCII range, 0-255. This is done via a simple scaling of the maxima and the minima among the input samples.

2.1.5 System Modes

The FPGA system operates in three modes which are described below. Note that this is not the same as the GNURadio `mode` parameter. Users have to correctly change `choose` and the FPGA modes to successfully run the system in its current implementation. More details in **Implementation**.

1. *Configuration Mode* - The FPGA system is expecting configuration details from the **UART** interface. More specifically, the system is expecting the values of `signal_size` and `sample_rate`. These are stored in registers inside the `SMC` and **Sample Rate** blocks. (This is triggered by turning ON switch S4 in the Krypton board and switching OFF switch S1.)
2. *Sample Write Mode* - This is the default mode of the digital circuit, and it waits for incoming samples (signal data). This data is directly dumped in the SRAM.
3. *Output Mode* - In this mode, samples are output to a port at the configured `sample_rate`. No writing of samples / configuration details can take place in this mode. (This is triggered by switching ON switch S1 in the Krypton board and switching OFF S4.)

2.1.6 UARTReceiver

End Goal - *The final module must receive and interpret UART signals coming on an FPGA pin. The system should raise a flag whenever new data is ready on its output port.*
Design - Several steps are needed to correctly design this system, which has been explained below. These steps were derived from WEL Lab's `http://wel.ee.iitb.ac.in/teaching_labs/Microprocessor/Microcontroller%20Notes/8051/Serial.pdf` on UART by Prof. DK Sharma.

1. *UART Clock* - The FPGA originally runs at a clock frequency of 50MHz. However, typical UART baud rates are a lot lesser (we've used 9600). It's very important to exactly produce this clock rate, since slight offsets will accumalate over time and lead to significant clocking errors. Since all baud rates are not a divisor of 50MHz, the clock's pulsewidth is kept adaptive to ensure an insignificant accumalated error.

2. *Clock Synchronization* - For maximum accuracy, it's preferred to sample the UART pulses at the middle of the pulses. As a result, whenever a new start bit is received, the sampling clock needs to be shifted by half the pulsewidth to sample exactly at the midpoints of the data pulses.

3. *Debouncing* - Similar to keyboards, we observed several spurious pulses (perhaps noise). This would find itself inside the FSM, but wasn't an actual input sequence. Hence the FSM needs to be robust enough to counter noise. We've used the technique *Keyboard Debouncing*, where a delay is added after the initial start bit to confirm whether the system actually saw the start bit. This delay must be siginificantly lesser than the UART pulse period for minimal errors.

4. *FSM Rate* - It is advisable to use the maximum available frequency to run this FSM. In any case, the frequency should be much higher than the baud rate. This gives the FSM enough flexibility to carry out debouncing, generate the UART Clock and synchronize quite accurately.

2.1.7 SRAM

No design steps are needed here, just ensuring that the SRAM chip works correctly! The guidelines to interface a Hitachi SRAM with the Krypton board are outlined on the WEL's course project for EE-214 (Sampling Analog Signals).

2.1.8 SMC

End Goal - *To build a module that accepts configuration details and samples from the* `UARTReceiver`*, correctly store them in the SRAM or local registers, and read them from the SRAM at a rate dictated by the* ***Sample Clock*** *module.*
Design - The following guidelines should be followed while interfacing a system with an SRAM. Detailed timing diagrams can be found in the EE-214 course project resources.

1. *Writing to SRAM* - In a nutshell, the FSM needs to ensure that the correct address is on the address port before the process starts. $\overline{CS}$ and $\overline{WR}$ need to be lowered to begin the process. A wait of 7 clock cycles of a 50MHz clock is recommended by the EE-214 resources.

2. *Reading from SRAM* - Similar to writing, here $\overline{CS}$ and $\overline{OE}$ are lowered in the beginning. Again, a wait of 7 clock cycles of a 50MHz clock is recommended.

3. *FSM Rate* - This rate needs to be atleast 8-10 times faster than the baud rate. This is to ensure that samples coming from the `UARTReceiver` are

written into the SRAM before the arrival of the next sample, to prevent unnecessary delays. Again, the FPGA system clock is recommended.

4. *Communication with **Sample Clock*** - The FSM should be designed to expect ticks / clock edges from the **Sample Clock** module when the system is in the *Output* mode. This is designed using a *Wait* state in the FSM. The FSM design for reading from the SRAM will decide the *SRAM Read Offset* value (explained in the next section).

5. *Address Registers* - Two different registers need to be kept for reading / writing from the **SRAM**. The read address register is expected to loop across `signal_size` locations in the SRAM, starting from 0. The value of `signal_size` is sent in the *Configuration* mode and needs to be stored in an internal register.
   The write register on the other hand needs to store samples in continuous memory locations starting from 0. It should not be incremented when the system is in the *Configuration* mode.

2.1.9 Sample Clock

(*Note - In the final implementation, this has been merged inside the `SMC`. The block has been separated here for a clearer understanding of the system design.*)
End Goal - *To build a module which accepts a sample-rate value from the `UARTReceiver` and outputs a clock running at that sample rate.*
Design - This acts like a support module for the `SMC` and is critical to obtain correct output signals. A few design strategies -

1. *FSM Rate* - Once again, this needs to be higher than the intended sampling frequency. This can be designed using a counter than increments on every pulse of the system clock. Frequencies which are not divisors of the system clock will accumulate some error in frequencies. The error is very less at lower frequencies. In our application, small errors will not matter.

2. *Configuration* - The sample rates must not be hard-coded in the VHDL code (unlike the UART system). These sample rates are redirected to the **Sample Clock** module when the system is in the *Configuration* mode and must be stored in internal registers.

3. *SRAM Read Offset* - Since the `SMC` is bound to take some time to read the samples from the **SRAM** (atleast 7 system clock cycles), the sample rates sent in the *Configuration* step must be slightly more than the intended sample rate. Exact calculations have been mentioned in the implementation section.

2.2 DAC Subsystems

Principally two DAC subsystems are used to demonstrate the results:

1. **0808 DAC with filters**:
    - The DAC0808 is an 8-bit monolithic digital-to-analog converter (DAC). The WEL RA's designed a DAC circuit board using this DAC.
    - We used this circuit to test the Tiva Code initially. Later we used this for debug purposes to check if the output was as expected. It consists of a current to voltage converter along with some filters for smoothening of the DAC output.
    - The output of the DAC circuit could be connected to an external board like the IQ modulator for modulation and transmission of the signal.
2. **AFE7070 EVM**:
    - The AFE7070EVM is a circuit board that allows designers to evaluate the performance of Texas Instruments' AFE7070 transmitter, a dual 14-bit 65-MSPS digital-to-analog converter with an integrated programmable fourth-order baseband filter and analog quadrature modulator.
    - The EVM provides a flexible environment to test the AFE7070 under a variety of clock, data input and RF output conditions. For ease of use the AFE7071EVM includes the CDCM7005 clock generator/jitter cleaner for clocking the AFE7071.
    - It was chosen because it serves the purpose of being both a DAC and a modulator. Also, it provides us with functionalities such as amplitude and phase offset correction which are very important such a transmission.
    - There is a software support for the evaluation module with a full featured GUI for easy testing and prototyping. The offset correction, selection of clock modes, output conditions can all be done using this propriety Texas instruments software.
    - The Evaluation Module as described above contains the chip FT245RL which does USB to SPI conversion and uses it to write to registers of the AFE7070 IC. In the final transmit-dongle, SPI is to be implemented on the FPGA side so that the offset can be directly controlled using GNURadio.

3 Project Implementation

3.1 AFE7070 EVM Evaluation Module

The AFE7070 EVM consists of the AFE7070 integrated circuit along with CDCM7005 clock synchronizer and the required peripherals. The AFE7070 integrated circuit is a dual 14-bit 65-MSPS digital-to-analog converter with an integrated quadrature modulator. The CDCM7005 is a high-performance, low phase noise and low skew clock synchronizer that synchronizes a VCXO (voltage controlled crystal oscillator) or VCO (voltage controlled oscillator) frequency to one of the two reference clocks. The CDCM7005 clock synchronizer provides the necessary clocks for the AFE7070. A simplified block diagram of the EVM in its default configuration is below:

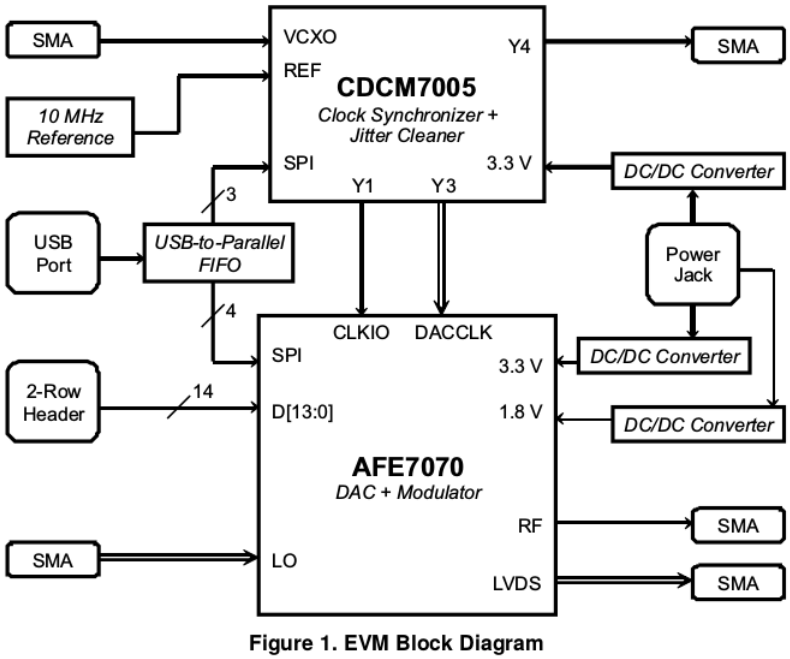


Figure 1. EVM Block Diagram

Figure 1: Block Diagram

The key features of the AFE7070 EVM are:-

1. **Sampling Rate** - The maximum sampling are of the AFE7070 EVM is 65 MSPS.
2. **Parallel Input Data** - The AFE7071 can input either interleaved 14-bit complex I and Q 1.8-V to 3.3-V CMOS input or just 14-bit phase 1.8-V to 3.3-V CMOS input.
3. **Digital signal-processing features** - The AFE7070 includes additional digital signal-processing features such as a numerically controlled oscillator for frequency generation/translation, and a quadrature modulator correction circuit, providing LO and sideband suppression capability.

4. **Signal Bandwidth** - The AFE7070 provides 20 MHz of RF signal bandwidth with an RF output frequency range of 100 MHz to 2.7 GHz.

5. **Clocking modes** - The AFE7070 has four clocking modes: Dual Input Clock, Dual Output Clock, Single Differential DDR Clock, and Single Differential SDR Clock.

The configurations of the AFE7070 EVM used are :-

1. **Bypassing the CDCM7005** :- The CDCM7005 can lock to one of two reference clock inputs (PRI_REF and SEC_REF). The clock signals on the EVM can be generated with the CDCM7005 or supplied externally. By default, the CDCM7005 is configured to use an on-board 10-MHz reference(SEC_REF) and external VCXO input signal to generate the AFE7070's DACCLK and CLKIO signals. The on-board 10-MHz reference limits the usage of the AFE7070 EVM. To overcome this, we are providing a clock signal ((PRI_REF) to the AFE7070EVM from the FPGA to generate the DACCLK. We are also providing the 14-bit data from the FPGA and hence there is no need to latch in the incoming data to the clock. Thus there is no real requirement of the CLKIO. This allows us to bypass the CDCM7005 chip.

2. **The Single Differential DDR Clock** :- In the first two clocking modes the user needs to provide a differential DAC clock at DACCLK and a second single-ended CMOS-level clock at CLKIO as seen in the above block diagram. The single differential SDR clock mode is only used for transferring 14-bit phase data. Hence we decided to go forward with the Single Differential DDR Clock.
   In this mode the user needs to provide just a differential clock to DACCLK at the internal output sample clock frequency. The rising and falling edges of DACCLK are used to latch I and Q data, respectively. The internal output sample clock is derived from DACCLK. The timing diagram of this mode can be seen in Figure 2:

3. **Synchronization**:- The AFE7070 has a synchonization input pin, SYNC_SLEEP, that is sampled by the same clock mode as the input data to reset/initialize internal signal processing blocks. In our mode as the internal processing blocks process I and Q in parallel, the user can provide the sync signal during either the I or Q input times (or both).

4. **Quadrature Modulator Correction (QMC) Block**:-The quadrature modulator correction (QMC) block provides a means for changing the phase balance of the complex signal to compensate for I and Q imbalance present in an analog quadrature modulator. The QMC block contains three programmable parameters. Two of the gain registers control the I and Q path gains and are 11-bit values with a range of 0 to approximately 2.0. The phase register controls the phase imbalance between I and Q and is a 10-bit value with a range of –1/8 to approximately +1/8.
   LO feedthrough can be minimized by adjusting the DAC offset feature. Two offset registers can be used to control the I and Q path offsets and are 13-bit

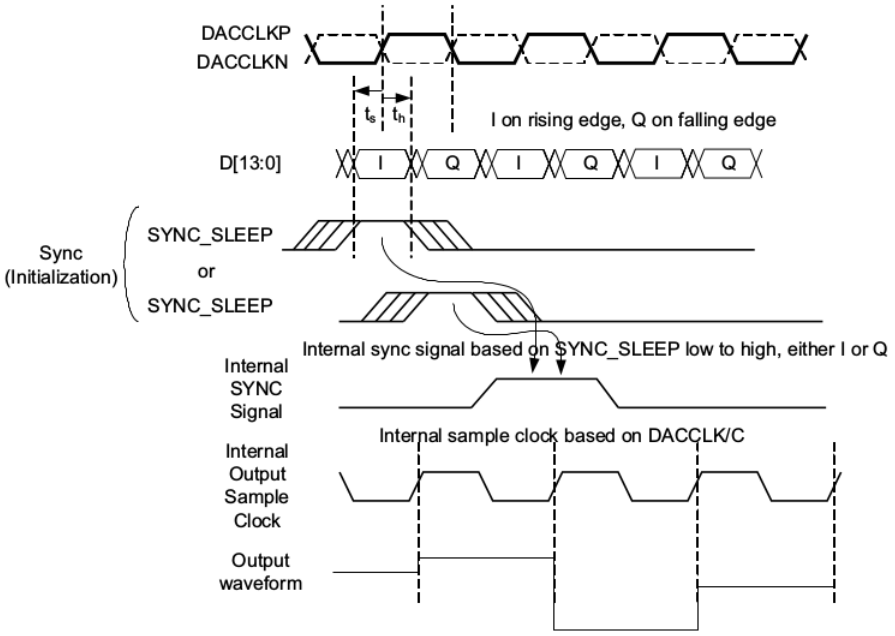


Figure 2: Single Clock Mode Timing Diagram

values with a range of –4096 to 4095. The QMC Gain/Phase Block diagram is as below:-

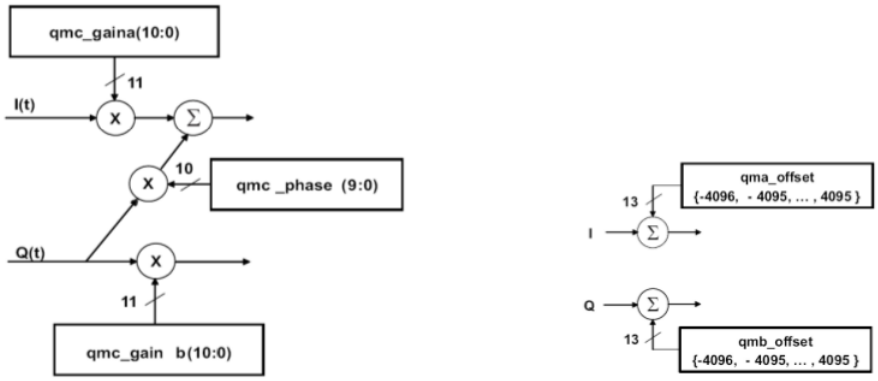


Figure 3: QMC Gain/Phase and offset Block Diagrams

The various settings of the AFE7070 EVM can be controlled using a proprietary Texas instruments software. It provides options for the following settings of the AFE7070 IC:- Power setting, SYNC Settings, FIFO Settings, Clock Settings, Mixer/NCO Settings, Digital Input Settings, Misc. Digital Signals, QMC Settings, Analog Output Settings. Along with these, it also provides CDCM7005 controls and general GUI controls.
For the prototype the basic settings of the AFE7070 required are:-

1. **Local Oscillator** - A local oscillator (LO) signal must be provided via the SMA connector J10. This signal's amplitude must be between –5 dBm and 5 dBm with a frequency between 100 MHz and 2.7 GHz.

2. **RF Out** - The AFE7070's RF output (pin RFOUT) is ac-coupled by a

100 pF capacitor to the SMA connector J3. This output can be connected directly to a 50-ohm spectrum analyzer or to an antenna to transmit the RF output.

3. **Clock Input** - A differential clock is to be provided at the DACLKP/N.
4. **Power Supply** - The EVM provides multiple options for powering the AFE7070. By default, users can power the board with a 6-Vdc adapter. Switched-mode dc/dc (buck) converters step down this voltage to the necessary 3.3-V and 1.8-V rail voltages. There is also a provision to power the board with a 5-Vdc supply.
5. **Software setup** - For the first time setup we need to install the USB drivers. For regular use we first do reset USB ports to get the default settings. Then we need to put the CDCM7005 in the buffer mode in the CDCM7005 tab. Then we put the clock in Single Differential DDR Clock. Either IQ or Phase data is allowed, and the FIFO is disabled automatically. Then we disable the AFE7070's NCO settings and press the send all button which reflects the changes.

The operations performed by the AFE can be seen via this block diagram:-

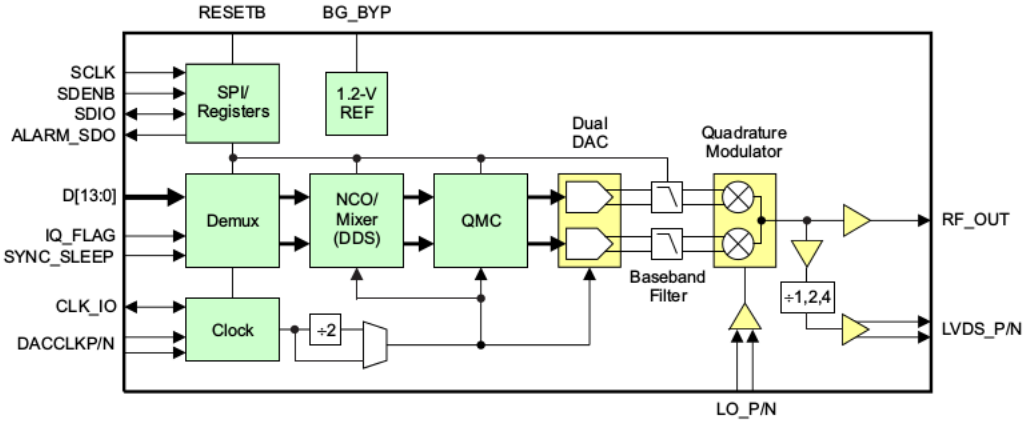


Figure 4: AFE7070 Integrated chip block diagram

3.2 Operation & Implementation - FPGA

This is a more detailed description of the final implementation of the system. It uses the guidelines mentioned in the previous section to build each component. This section begins with a detailed instruction set to run the system.

3.2.1 Installation Instructions

You must have installed GNURadio. You will need the Quartus tool-chain if you wish to modify the internal VHDL code as well. You can read about the Quartus

installation instructions by contacting the WEL team. You can view the instructions for installing the GNURadio block on the Github repository. Additionally, you will also need to install `pyserial`.

3.2.2 Running the System

Make sure you've read the Krypton Board `https://www.ee.iitb.ac.in/~wel_iitb/resources_development_boards.html` and used GNURadio before trying to run this system. You can find more on the WEL website.
We have provided a sample `https://github.com/Abhin02/EDL/blob/master/BPSK_UART.grc` which generates a BPSK signal and sends it to our GNURadio block on the Github repository.

1. **GNURadio Schematic** - Make sure that the block is correctly installed. Make sure of the following things,
   - The same `sample_rate` must be used for both the signal generation and in the `uart_sender` block. Of course, if you are using interpolation / decimation blocks, make sure you mention the latest `sample_rate` in the `uart_sender` block.
   - The `signal_size` sent must cover all the samples that need to be looped by the system. Make sure this does not cross 8191. The block will only take `signal_size` samples from the beginning.
   - Make sure `choice` is set to 1 when you start the system.
2. **Connections** - Before executing the GNURadio schematic, connect the USB-to-UART circuit. Connect TX to pin 5 on HEADER0 and GND to a ground pin on the Krypton Board. Turn on the FPGA.
3. **Configuration** - Turn on S4 (on Krypton Board) to turn on the *Configuration Mode*. Execute the GNURadio script. In the QT GUI, change the *FPGA Mode* to "Send Samples". You should see an LED glow on the USB-to-UART module.
4. **Sending Data** - Turn off S4 to switch to the *Sample Write Mode*. In the same QT GUI window, change the *FPGA Mode* to "Send Samples". You will see the USB-to-UART LED glow for sometime. Do not move to the next step unless the LED stops glowing.
5. **Displaying Data** - Turn on S1. If you see the Krypton LEDs glow, the system worked successfully and you can see digital samples on HEADER1, from pin 2(LSB), 4, 6, 8, 10, 12, 14, 16(MSB).
6. **Debugging** - In case the LEDs do not glow, turn ON switch S2 (reset). Turn OFF switch S2. This clears the registers. Next, power OFF and power ON the system to clear the SRAM. Restart your GNURadio block.

   The next few sections discuss the implementation of this system.

3.2.3 GNURadio - UART

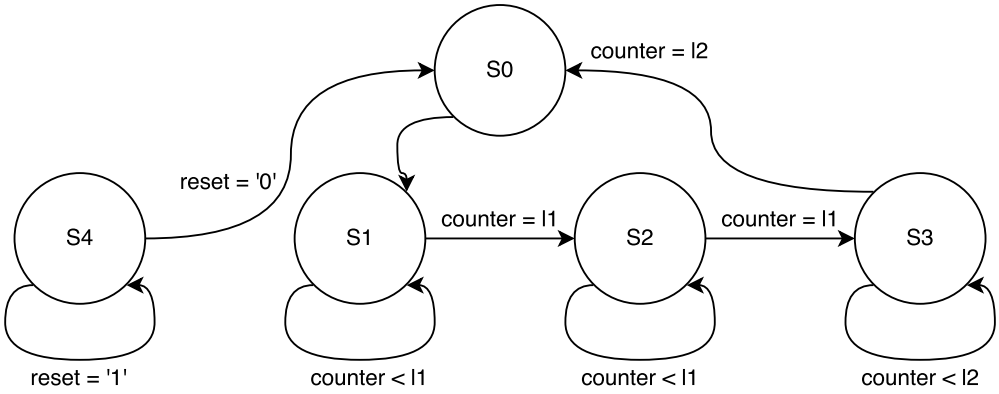
This has been implemented in `https://github.com/Abhin02/EDL/blob/master/gr-uart_sender/python/uart_sender.py`.

1. **Out of Tree Modules** - The standard practices (`https://wiki.gnuradio.org/index.php/OutOfTreeModules`) for *Sink* Out-Of-Tree Modules were used to set up the custom block. Callbacks were setup to ensure smooth interfacing with the GNURadio GUI widgets. This was done since `choice` needed to be chosen by a QT GUI Chooser.
2. **USB-to-UART Interfacing** - This was done using `pyserial`. Currently, the specific IC's device ID has been hardcoded in the GNURadio block's code. Hence, this will only work for USB-to-UART bridges having the same manafacturer ID / device ID. This ID is used to uniquely identify the device and set up a UART serial communication.
3. **Input Cleanup** - The first `signal_size` samples have been extracted and scaled between 0 and 255.
4. **Configuration** - The two configuration values are sent in a Little Endian fashion. This is done when the value of `choice` is 2 and rate has not been sent before.
5. **Sending Data** - Data is sent at a baud rate of 9600 without a parity bit. All samples are continuously sent when `choice` is set to 3 and data has not been sent earlier.

3.2.4 UARTReceiver

The UART clock has been implemented as a "ticker" in `https://github.com/Abhin02/EDL/blob/master/uart_receiver/UARTTicker.vhd` and the actual UART receiver module uses this clock in `https://github.com/Abhin02/EDL/blob/master/uart_receiver/UARTReceiver.vhd`.

1. **Ticker** - The `UARTTicker` has an internal counter and a couple of limit registers. The values of the limit registers have been pre-calculated and hard-coded in the VHDL code, configuring it for 9600 baud rate. Guidelines to recalculate this for different baud rates are given in the code.
   The FSM is designed to continuously increment a counter (unless it's RESET), until it hits a limit. On hitting the limit, a pulse ("tick") is output for one cycle and the counter is reset. Since 9600 does not exactly divide the clock frequency of 50MHz, two counter limits have been used alternately to ensure that the net timing error is zero in three clock cycles.
   Finally, to ensure correct clock synchronization, an additional midway limit has been provided causing a separate port to "tick" at the middle of the UART clock cycle.



(a) S0 / S4 - Reset state, stays in this state as long as `reset` is high.

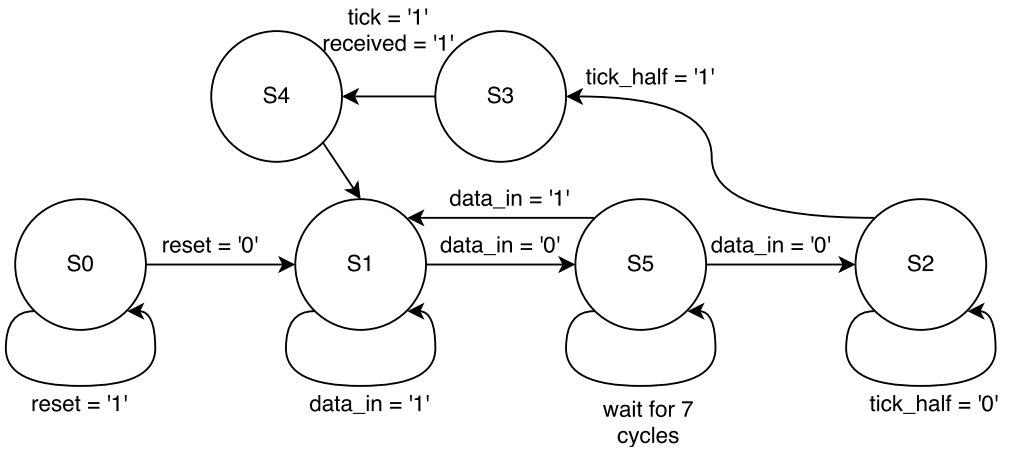
(b) S1 - First wait cycle, here `l1` is $\frac{50*10^6}{9600} - 1$.

(c) S2 - Second wait cycle, identical to first.

(d) S3 - Third wait cycle, here `l2` is $\frac{50*10^6}{9600}$.

The ticker outputs a pulse whenever there is a transition from S1 to S2, S2 to S3 and S3 to S0. It additionally outputs a pulse whenever half the limits are reached. This has been split into three states to exactly divide 50MHz over three baud-rate cycles.

2. **Receiver** - The UART Receiver circuit has an internal shift register that's to store the last 10 bits of the bit stream. The bits start shifting into the register whenever the start bit is seen. Once the stop bit is encountered, the middle 8 bits are read off the shift register and sent to the output port. A flag `data_ready` is also raised.
To take care of *de-bouncing* a delay of 7 clock cycles has been used whenever the start bit is encountered. To take care of clock synchronization, the receiver expects ticks from the **Ticker** component. Finally, data can be sent continuously without intervals. The FSM is running at 50MHz.



(a) S0 - Reset state, stays in this state whenever `reset` is ON.

(b) S1 - In this state, the system is expecting a UART signal (LOW start bit).

(c) S5 - This state takes care of debouncing. In case `data_in` is not 0, it returns to the wait state S1.

(d) S2 - This state helps in clock synchronization.

(e) S3 - This state samples data at the middle of the UART pulses.

(f) S4 - In this state, `data_ready` is raised.

3.2.5 SMC & Sample Clock

Both these modules were combined into one system `https://github.com/Abhin02/EDL/blob/master/uart_receiver/SMC.vhd`. Here are the key steps in the implementation -

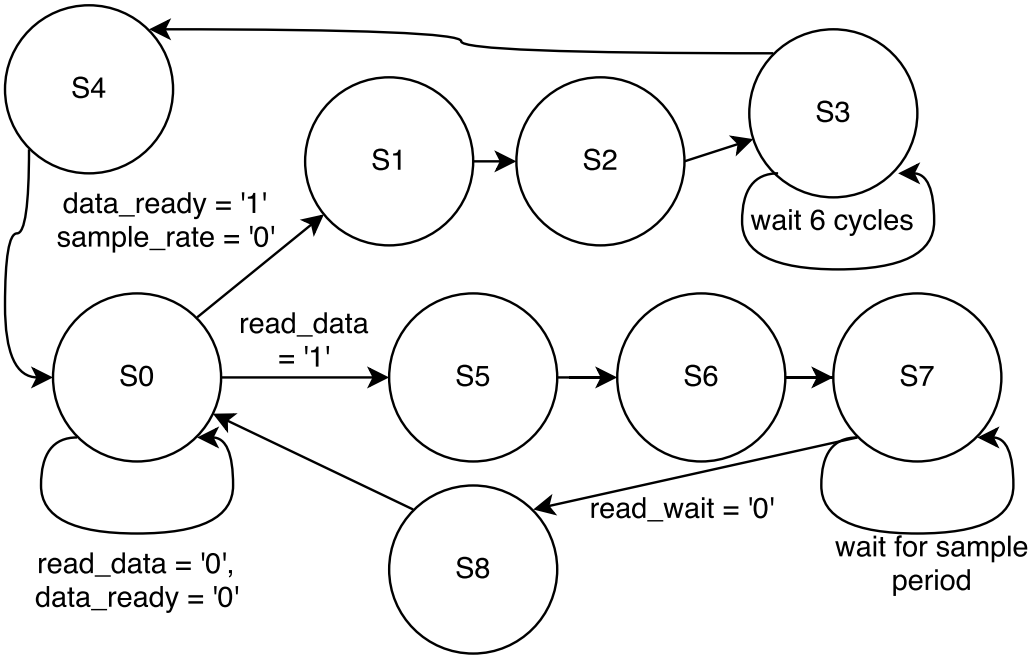
1. *Configuration* - The FSM remains in state S0 during configuration. There is a variable `limit_control` which loops from 0 to 5. The value of this denotes the register which is being written into. The first four registers are 4 bytes for the `sample_rate` (which is actually stored as the number of clock cycles needed to complete one sample period minus the *SRAM Read Offset*) and the upper two bytes are for the `signal_size`.

2. *Writing to SRAM* - The standard sequence of signals have been used to write to SRAM. An additional wait for 6 clock cycles has been added to ensure writing is complete. An address register is incremented by 1 each time a write in the SRAM takes place.

3. *Reading from SRAM* - An internal counter is maintained that counts upto the value stored in configuration. Since this is done in S7, an additional delay occurs due to the traversal from S5, S6, S8. As a result, a slightly lower value of sample period has been used. This can be computed using,

   $$x = \frac{50}{f_s} - 4$$

   Here, x is sent as the `sample_rate` value and f_s is the sampling frequency. Once samples are read, they are sent to an output port. A `debug` port has also been added which interfaces with the LEDs in the current implementation. Once the read address has cycled through the `signal_size` addresses of the SRAM, it comes back to 0 and repeats the process.
   Here is our final FSM,



1. S0 - Wait state. All three modes branch from here. For the *Configuration* mode, the FSM stays in the S0 state. For the *Output* mode, `read_data` must go HIGH. For the *Sample Write* mode, `data_ready` should go HIGH.
2. S1 - In this state, the write cycle starts and $\overline{CS}$ is lowered and new address is placed.
3. S2 - $\overline{WE}$ is lowered and write cycle started.
4. S3 - Wait for 6 cycles for write to happen.
5. S4 - Write cycle ends.
6. S5 - Begin the read sequence, $\overline{CS}$ is lowered and new address is placed.
7. S6 - $\overline{OE}$ is lowered and read cycle started.
8. S7 - Wait for certain cycles. This wait period is generally the sample period minus the *SRAM Read Offset* (described above).
9. S8 - End the reading cycle.

3.2.6 TopLevel

Finally, a wrapper brings together all these modules into a final system. This is implemented in `https://github.com/Abhin02/EDL/blob/master/uart_receiver/TopLevel.vhd`. Additionally, we've added a couple of testing modules, one for the UART Receiver (`https://github.com/Abhin02/EDL/blob/master/uart_receiver/Testbench.vhd`) and one for the whole system `https://github.com/Abhin02/EDL/blob/master/uart_receiver/Testbench_Complete.vhd`. Additionally, this script generates input data to the system - `https://github.com/Abhin02/EDL/blob/master/uart_receiver/gen_uart.py`.

4 Performance Evaluation

4.1 Details of Prototype

For our final demo, we plan to show Binary PAM transmission and sinusoidal waves transmission. A PAM signa/sinusoidal is generated in GNURadio and given as an input to our UART Sender block. This block sends the data to a USB port using `pyserial`. This input is given to the AFE 7070 EVM and it converts the digital signal to analog and modulates it using Local Oscillator frequency. On the receiving end is an RTL-SDR Dongle connected to the Laptop which easily interfaces with GNURadio and we can view the baseband signal. Next we have made a decoder block which will decode the baseband signal and give the output as the original signal.

4.1.1 Sinusoidal Waves Transmission

1. We will be sending a linear combination of different sinusoidal waves via the GNURadio block. We will be able to see the peaks at the corresponding sinusoidal frequencies on either side of the local oscillator frequency.
2. We will also show the corresponding time domain sinusoidal wave on the 0808DAC circuit.

4.1.2 PAM Transmission and Reception

PAM Signal Generation The signal input here, is assumed to be from a vector source. This can be a string, an audio file, or essentially any array. This is passed through a PAM Transmitter block which is made using the procedure given here. As shown in the diagram below, the Vector Source is used to generate the ASCII code bytes. The Packed to Unpacked block performs the actual parallel-to-serial conversion. After this we have to polarize the signal and offset it such that it has negative values too. This is done using the multiply and add blocks. Other blocks in the diagram are just for generalizations like whether we want the bits to be interpreted as inverted etc. This converts the ASCII values to float. Next we use an Interpolating IIR filter which is the pulse shaping filter. It can be either `rect`, `rcf`,`rrcf` etc. The final output is the PAM signal to be transmitted. This has been implemented in `https://github.com/Abhin02/EDL/blob/master/BPSK_UART.grc`.
PAM Signal Reception The reception module uses the RTL-SDR Dongle Receiver block along with a tuner slider. This output is then decoded using a custom block we are currently developing in `https://github.com/Abhin02/EDL/tree/master/gr-pam_receiver`. This is partially complete and we hope to show a decoded PAM output in the final demonstration.

4.2 Testing Results

4.2.1 Test Results for AFE

After the basic settings of the AFE7070 and supporting peripherals are done as mentioned above we can have our system working. Without sending any data through the FPGA and just providing the local oscillator of 1 GHz frequency to the AFE7070 EVM gives us the following expected output.

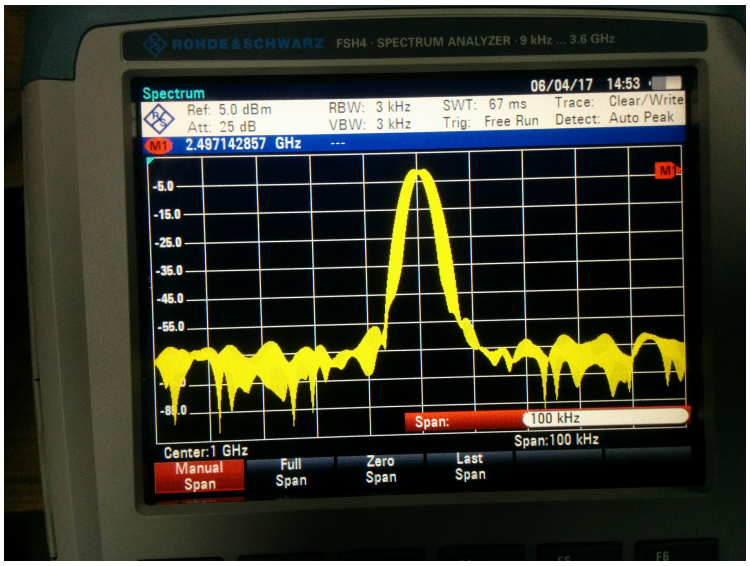


Figure 5: Local Oscillator Signal

Now we give the samples of the signal $sin(2\pi * f_0 t) + sin(4\pi * f_0 t)$ where f_0 is 10 kHz we get the following output. This is the correct output since we expect peaks at 10 kHz and 20 kHz on either side of the local oscillator frequency along with LO feedthrough at 1GHz.

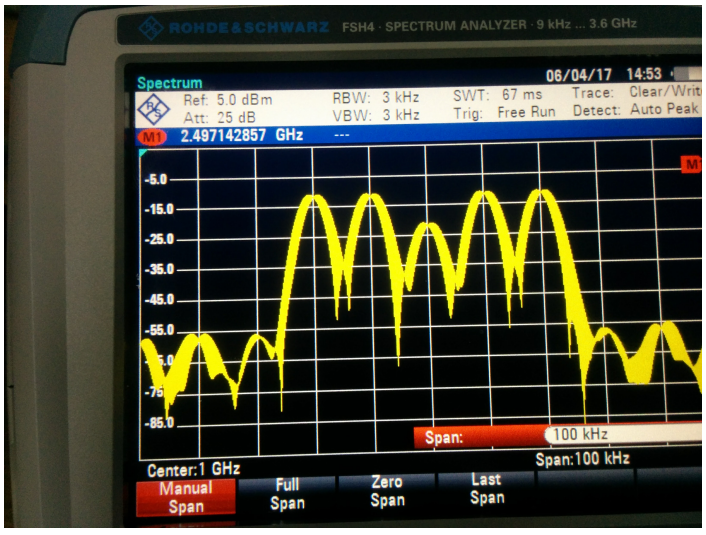


Figure 6: Two sinusoids with LO feedthrough

Now using the offset correction features of the AFE7070 we can reduce the LO

feed-through. This happens digitally i.e it happens before the digital signal is converted to the analog signal. As a result we get the output with suppressed carrier.

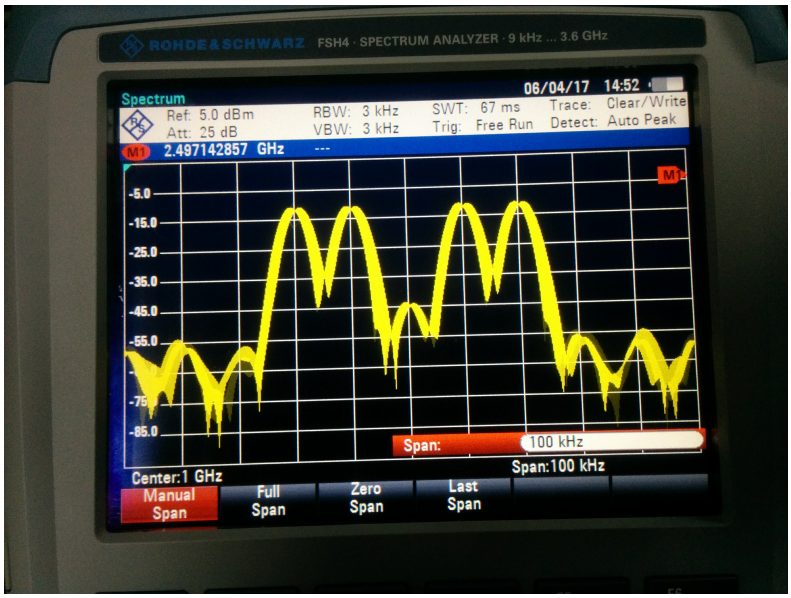


Figure 7: Two sinusoids with suppressed LO

Now we give the samples of the signal $sin(2\pi * f_0 t) + sin(4\pi * f_0 t) + sin(6\pi * f_0 t)$ where f_0 is 10 kHz we get the following output. This is the correct output since we expect peaks at 10 kHz, 20 kHz and 30kHz on either side of the local oscillator frequency along with LO feed-through at 1GHz. We have suppressed the LO feed-through here.

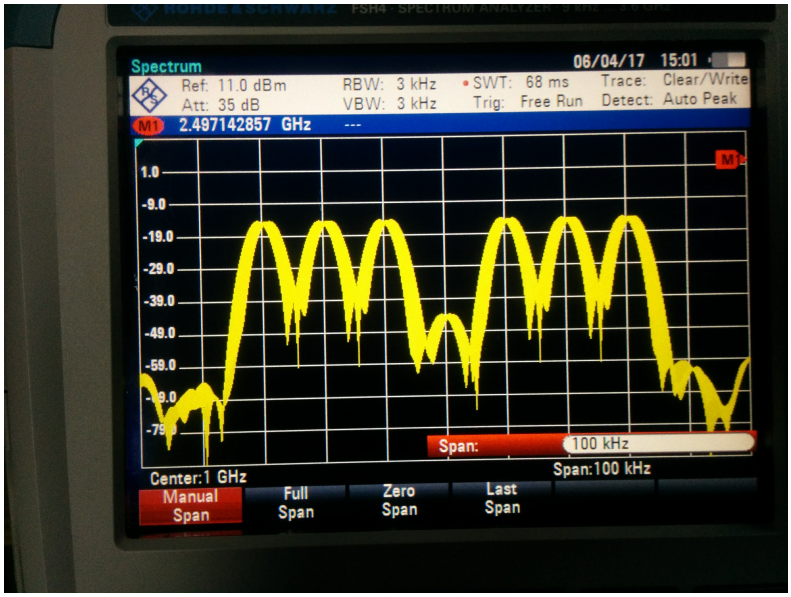


Figure 8: Three sinusoids with suppressed LO

Now to test the BPSK module, we gave the following string as the input to the

BPSK Transmitter:- "Karan Kalpesh Abhin". The image below shows the spectrum of the transmitted signal of the GNURadio software.

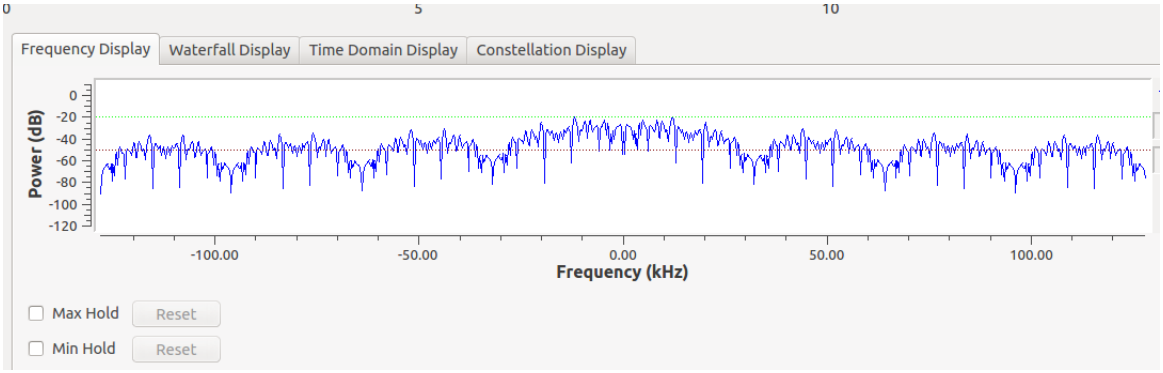


Figure 9: Frequency spectrum on GNURadio

The image below shows the transmitted spectrum by the AFE7070 EVM. We clearly see that this is the correct result since the entire spectrum is exactly the same as we transmitted from the GNURadio block and it is upconverted the Local Oscillator frequency.

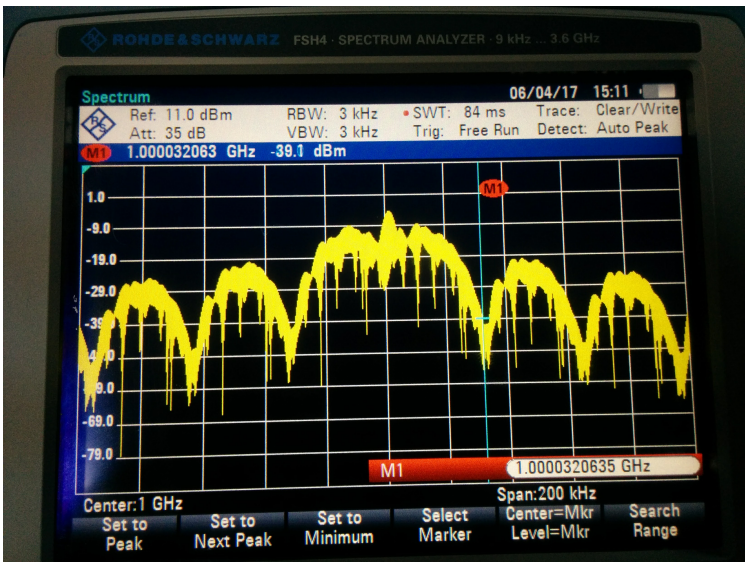


Figure 10: Frequency spectrum on Spectrum Analyzer

4.2.2 Test Results for DAC0808

When we provide samples of a 50 kHz sine wave to the 0808 DAC circuit e get the following output on the oscilloscope :

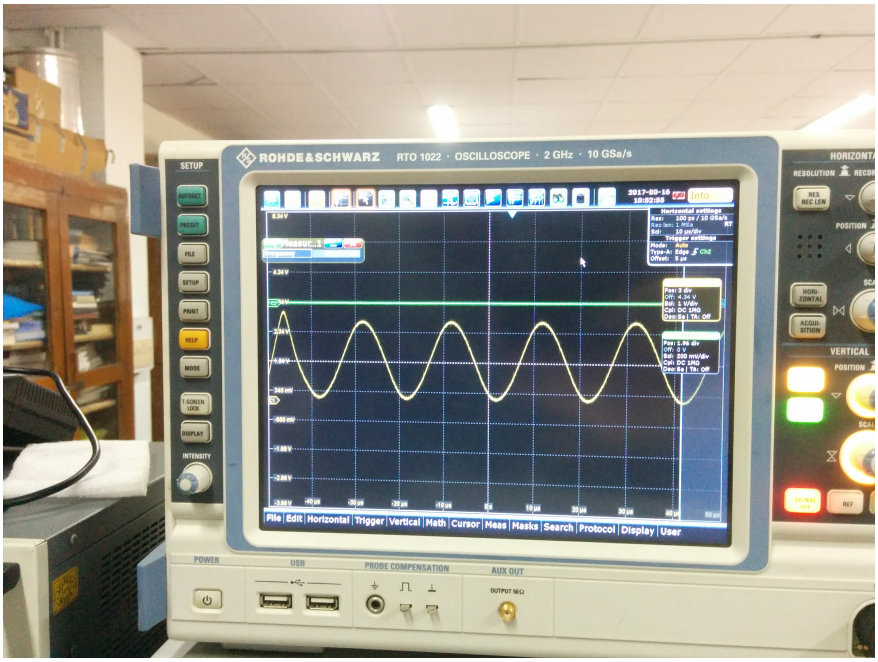


Figure 11: Sinusoidal wave on oscilloscope

4.3 Problems faced, Limitations and Lessons learned

4.3.1 AFE7070 and DAC Circuit

The problems faced during the entire project were as follows:-

1. **TIVA Software Setup** -
   - We could not find proper guidelines for installing the Code Composer Studio required for the Tiva microcontrollers in Ubuntu. We managed to get the software installed in Windows OS for the time being.
   - Even the WEL RA's were not aware of the proper procedure to do the same in Ubuntu. Even after we managed to get the CCS installed in Ubuntu we were not able to run the previous project folder due to its incompatibility.
2. **Insufficient support for TIVA** -
   - The last year's EDL group worked on the Tiva^{TM} TM4C123GH6PM Microcontroller. To enhance the speed of data transfer, the WEL RA's shifted to the Tiva^{TM} TM4C1294NCPDT Microcontroller making use of the additional features like Ethernet.
   - The base code that they provided us was thus for this particular microcontroller. There is not enough documentation and forums available for the TIVA TM4C129NCPDT.
   - The WEL RA's suggested us to make use of the RTOS and there were unnecessary delays due to the operating system. As a result we faced

many difficulties while trying to implement the timers on this microcontrollers. We were not able to get accurate data for any frequency.

3. **Figuring the AFE settings** -
   - The AFE7070 EVM is a very complicated module to understand. It provides a variety of features with multiple ways of executing them.
   - We were not able to segregate the features that we actually require and how to use each of them.
   - Also we were confused between the AFE7070 IC and AFE7071 IC. The NCO and LVDS output functions are not available with the AFE7071 but the AFE7070 has both functions.
4. **AFE7070 Software Setup** -
   - In order to make use of the AFE7070 EVM, we had to install the propriety Texas Instruments software for AFE7070. This software is available only for Windows OS.
   - For the same purpose it was required to install USB drivers. It was mentioned in the datasheet that a pop-up screen would appear while installation from where we could follow the instructions to install the necessary drivers. But this was true only for the Windows 10 OS.
   - As a result we had to manually install the drivers on Windows 8.1 OS and there were no proper guidelines for doing the same. To install the drivers run the files `ftdibus.sys` and `ftser2k.sys`.
5. **TSW1400 software setup** -
   - In order to test the AFE7070 EVM. we decided to follow the basic test procedure outlined in the datasheet. For this we required TSW1400 board.
   - The TSW1400 is a high speed data capture and pattern generator board used to evaluate performances of a wide range of TI high-speed digital-to-analog converters. Luckily this was available in WEL. Now in order to use this module we again needed the propriety Texas Instruments software for TSW1400.
   - When we connected the setup as mentioned and tried to create a pattern we got the following error:- "JTAG Chain broken error". We were not able to resolve this error and as a result could not the send the data to AFE 7070 EVM.
6. **Dependency on the FPGA code for testing the AFE7070 EVM** -
   - With the option of using TSW1400 for testing the AFE7070 ruled out, we shifted to Arduino. We generated samples of the data via Arduino and gave it as an input to the AFE7070.
   - But we were not able to match the output sampling rate of the Arduino with the input sampling rate of AFE7070. This was because there was

a 10 MHz on-board DACCLK on AFE7070 and the Arduino could not produce samples at such a high rate.

- As a result there was a dependency on the FPGA code to provide the correct samples in order to test the AFE7070.

7. **Bypassing the CDCM7005** -
    - The CDCM7005 can lock to one of two reference clock inputs (PRI_REF and SEC_REF). The SEC_REF is an on-board 10MHz crystal oscillator. The PRI_REF can be provided externally through the CLKIO SMA connector.
    - Along with this, it also synchronizes a VCXO (voltage controlled crystal oscillator) or VCO (voltage controlled oscillator) frequency to one of the two reference clocks. But we found out quite late, that this feature is not really required in our project.
    - There is no proper explanation of how the Phase Locked Loop in the CDCM7005 works and hence there was no way to figure the relationship between input and output frequencies.
    - We decided to use the PRI_REF reference clock which would enable us to provide any desired clock input to the DACCLKP/N without using the PLL of CDCM7005.
    - We also figured out that we don't need the CLKIO to synchronize the input data with the clock as both of them are given by the same FPGA and this could be done in software. Thus by using the single differential DDR mode we could completely bypass the CDCM7005 IC.
8. **Incorrect DACCLK** -
    - Now to bypass the CDCM7005, we put the CDCM7005 in the buffer mode through the software and made the necessary changes to use the PRI_REF. But the changes were not reflected on the EVM.
    - We tried to check the signal obtained at the DACCLK pins and found out that it was not what was expected.
    - As a result we had to solder two wires directly on the AFE7070 IC of the EVM to provide the DACCLKP/N.
9. **IQ modulation** -
    - The data-sheet of AFE7070 mentions that it does quadrature modulation. We then started working assuming that it indeed does so.
    - But it turned that no such thing was actually happening. We even tried doing the same on a new AFE7070EVM that was available in the WEL. But we got similar results.
    - We consulted our Faculty mentor on this. With his suggestions we send many sample signals one of which was $sin\theta$ in the I phase and $-sin\theta$ in the Q phase. The output we got on the spectrum analyzer was almost

close to zero. As a result we concluded that the IQ modulation was not happening correctly.

4.3.2 FPGA & GNURadio

Due to the limited time frame, the system has a few flaws. Nevertheless, it has beaten the original proposed specification of 100 KSPS by roughly 50 times! (System works up to 5 MSPS).

1. **Real-Time** - The FPGA needs to be RESET and restarted every time a different signal is to be sent via GNURadio. Hence this system is not exactly real-time. Nevertheless, the setup does not take more than half a minute and is very convenient to use via GNURadio. This makes it quite usable for the Communication Lab.
2. **Sequence of Steps** - It's often imperative to follow a fixed sequence of steps (switch changes, as described in *Running The System* to get the system working. The system needs to be made more robust to bad usage.
3. **Configuration** - This step can be easily avoided after some changes to the FSM structure. The idea is to send the configuration details along with the data. In the current implementation, switches have to be changed in a particular way to correctly configure the FPGA.

A few problems were faced, since this was a reasonably sized VHDL digital design for the stipulated time period. Primarily,

1. **Debouncing** - Earlier, the system was not working well due to the absence of a delay in the UART receiver's debouncing. It was showing random behaviour, and accepting a different number of samples on each run.
2. **Difficulty in Debugging** - The only available port for quick debugging was the LED port. This often lead to several iterations of changing code and burning it onto the FPGA to properly debug the system.
3. **GNURadio** - Since Out of Tree Modules in GNURadio inherently expect C++ code, there was some tweaking required to ensure it runs Python-only code.

5 Conclusions and Future Work

5.1 PCB Design

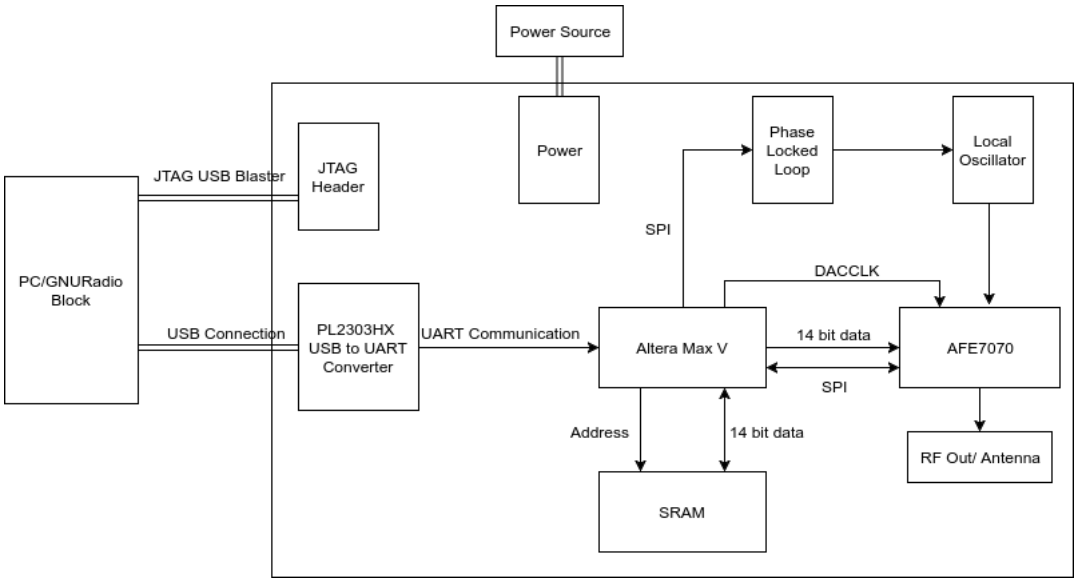


Figure 12: PCB

The above figure shows a block diagram of how the final PCB can be designed. It has just three external connections, one of which is a power source. This is put in because although rest of the circuit can be powered by the PC USB port, we are not sure of the power that will be required in the Phase Locked Loop. Another connection is the JTAG programmer, this would be required for one-time programming of the FPGA and debugging. The only other connection is the USB connection for the Data transfer. This would be used for both the initial signal transfer and later for the control signals to be sent to the AFE for offset correction. In the final PCB, to change the settings of the AFE, as shown in the figure SPI communication will need to be implemented on the FPGA so as to change the values in the corresponding registers in the AFE. Finally, to remove the dependency on an RF Signal generator, a PLL circuit needs to be designed which takes an input from the 50 MHz crystal oscillator present on the Krypton board developed in WEL Lab which is included in the final circuit.

5.2 AFE7070

One could explore more clock modes of the AFE7070. For this more aspects of the CDCM7005 chip which is present on the AFE7070 EVM need to be explored. Other than the above, figuring out exactly how the IQ modulation in the AFE works is an important aspect.

5.3 FPGA

This section describes the new features and improvements needed to get the system into production (for the Communication Lab). It also mentions some implementation guidelines.

5.3.1 Improvements

This section only outlines the improvements to the current implementation, and no new features.

1. **Code Review & Improvement** - Before any further work is done on the project, the code needs a thorough review and clean up. A lot of FSMs have unnecessary states that are slightly degrading the performance of the system. Since the `SMC` module has become so big, it would be worthwhile to split the it into a `Sample_Clock` module as described in the Design section.
2. **Testing Infrastructure** - The code has not been extensively tested in simulation and needs better Testbenches to do this well. While the code does work on the circuit, it's often hard to debug modifications to it due to the lack of simulation. The Testbenches could be improved by modelling an SRAM and providing more realistic inputs.
3. **Sequence of Steps** - The code could be converted into a real-time system only with small modifications to the `SMC` FSM. Additionally, there needs to be an investigation to figure out the exact reason for restarting the system each time and work on fixing it. This can be accelerated by good simulation infrastructure.

5.3.2 New Features

This section outlines the potential new features to extend this system for the final product.

1. **SPI** - In the final system, we wish to communicate with the AFE7070 chip and configure it via GNURadio. There needs to be a module in GNURadio to accept SPI configuration data via the GNURadio-UART block and relay it to the AFE7070 chip using an SPI implementation. This would ideally be a new module in `TopLevel`. It's very essential to make this real-time and the AFE7070 should be re-configurable for offset correction without restarting the FPGA.
   This calls for a full understanding of the configuration details for the AFE7070.
2. **No On-Board Configuration** - It would be convinient to completely control the FPGA system via GNURadio, without touching any switches onboard. This can be done by having a central controller unit, which connects all modules. This unit should be the only entity receiving UART data and it should decode it to decide the FPGA's mode of operation as well as pass the data whenever needed.

This also calls for a well-defined communication protocol between GNURadio and the FPGA. Since large amounts of data are not being sent, UART is a good, easy-to-implement choice.

3. **On-Board Sampling Clocks** - It would be convinient to use the FPGA clock to sample the digital signals in the AFE7070. This has been somewhat completed in the current implementation (inside the `TopLevel` module, but needs checking since it's not working well. For now, a constant frequency clock is being sent. This frequency should be adjustable to the sampling rate. (in case the rate is not a divisor of clock frequency, use the guidelines given in design of UART Receiver).

5.4 Conclusion

From our prototypes, we are confident that this can be used in the communication lab and for other similar applications.
We hope that this work is taken up in the summer and completed to make a final working product. This would involve implementing most of the *Future Work* outlined above. We will be ready to provide any help required. Kindly send us an email with all the authors CCed, since the work was divided.