Validation and Verification Document

Ultra-Fast Failure Test Unit

Capstone Project #45

Boran Gungor
Jude Lu
Lukas Lund
Romina Adib
Xingwei Su

Table of Contents

1 Summary

This document includes test descriptions and results for each aspect of our design. We use the results of these tests to verify that all of our requirements have been met, and to validate our design for usability.

For the purpose of this document, verification means ensuring that our system meets the quantitative specifications we set out to achieve. We verify that our system performs as expected, and that all intended functionality has been implemented successfully.

We validate our design using experience-based criteria. Our validation tests ensure that the client's needs have been achieved and that our product provides ease-of-use.

For the sake of readability, we will refer to a Functional Requirement as a FR and a Non-Functional Requirement as a NFR throughout this document.

2 TCP Client Application

2.1 TCP Connection

The rftool receives commands via TCP connection at port 8082 and sends and receives data via port 8081. To replace the RFDC GUI with a simple, more usable piece of software, we began by ensuring we could successfully connect to both these sockets. The following is a rudimentary snippet of code we used to ensure our connection requests were accepted by the rftool which acts as the server.

```
// define the same ports as rftool
#define COMMAND_PORT 8081
#define DATA_PORT    8082

int ConnectTCPserver(void) {
   int sock = 0;
   int valread;

   if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
       printf("\n  Socket creation error \n");
       return -1;
   }

   address.sin_family = AF_INET;
   address.sin_port = htons(COMMAND_PORT);

   if (inet_pton(AF_INET, "192.168.1.3", &address.sin_addr) <= 0) {
       printf("\n  Invalid address/ Address not supported \n");
       return -1;
   }
```

```
    if (connect(sock, (struct sockaddr *)&address, sizeof(address)) < 0) {
        printf("\n  Connection Failed \n");
        return -1;
    }

    return 0;
}
```

After running the above code for both TCP ports, both connection requests are heard by the rftool and accepted.

2.2 Command Interface

Figure 1 shows our application prompting the user to issue commands to the rftool. The purpose of this is to demonstrate that we are able to successfully send commands in the expected format and receive the expected response from the server. In real use, the necessary commands are executed automatically to configure the ADC according to the user's needs.

First, the rftool requires an indication as to whether the command will be a read, a write, or a disconnect command. This is done by issuing a single character command 'r', 'w', or 'd' respectively. Once this step is complete, we must send a valid command, defined in `cmd_interface.c` from the rftool source code.

After issuing a write command, the following command must be a read in order to read the response from the previous command. This is illustrated in Figure 1. The final writes command requests to view the log file: after issuing the command and then issuing an 'r', the log file is returned from the server.

This experiment proves that we are able to successfully send commands to the rftool over a TCP connection and read its response.

```
Read or Write Operation (D for Disconnect):[R/W/D]w
Enter Command:
JtagIdcode
Do you want to sent Comand: JtagIdcode  [Y/N]y
[Socket]Command Sent: JtagIdcode

Read or Write Operation (D for Disconnect):[R/W/D]w
Enter Command:
GetBitstream
Do you want to sent Comand: GetBitstream        [Y/N]y
[Socket]Command Sent: GetBitstream

Read or Write Operation (D for Disconnect):[R/W/D]r
JtagIdcode 43605
GetBitstream 3

Read or Write Operation (D for Disconnect):[R/W/D]w
Enter Command:
GetMemType
Do you want to sent Comand: GetMemType  [Y/N]y
[Socket]Command Sent: GetMemType

Read or Write Operation (D for Disconnect):[R/W/D]r
GetMemType 1
5
GetBitstream 3

Read or Write Operation (D for Disconnect):[R/W/D]w
Enter Command:
GetLog
Do you want to sent Comand: GetLog      [Y/N]y
[Socket]Command Sent: GetLog

Read or Write Operation (D for Disconnect):[R/W/D]r
GetLog metal: error:     | Requested tile (ADC 1) not available in XRFdc_DynamicPLLConfig|metal: error:
   | Requested tile (ADC 2) not available in XRFdc_DynamicPLLConfig|metal: error:     | Requested tile (A
DC 3) not available in XRFdc_DynamicPLLConfig|metal: error:     | Requested tile (DAC 0) not available in
 XRFdc_DynamicPLLConfig|metal: error:     | Requested tile (ADC 1) not available in XRFdc_RestartIPSM|met
al: error:     | Requested tile (ADC 2) not available in XRFdc_RestartIPSM|metal: error:     | Requested
tile (ADC 3) not available in XRFdc_RestartIPSM|metal: error:     | Requested tile (DAC 0) not available
in XRFdc_RestartIPSM
Read or Write Operation (D for Disconnect):[R/W/D]
```

Figure 1: Command verification

2.3 Data Interface

Figure 2 demonstrates the operation of the TCP connection over the data port. In order to read ADC data from memory, we issue the `ReadDataFromMemory` command. After following a similar w/r protocol as in the command interface, raw data is read back from the onboard memory.

```
steven@ub:~/Documents/SocketTest$ ./client
Data Interface for ZCU111
Selected PORT# = 8082 & IP address(Default) = 192.168.1.3
Read or Write Operation (D for Disconnect):[R/W/D]d
[Socket]Command Sent: disconnect

Data Interface for ZCU111
Selected PORT# = 8082 & IP address(Default) = 192.168.1.3
Read or Write Operation (D for Disconnect):[R/W/D]w
Enter Command:
ReadDataFromMemory 0 0 32768 1
Do you want to sent Comand: ReadDataFromMemory 0 0 32768 1      [Y/N]y
[Socket]Command Sent: ReadDataFromMemory 0 0 32768 1

Read or Write Operation (D for Disconnect):[R/W/D]r
0C
04
FFFFFFEC
24
24
FFFFFFFC
14
FFFFFFE4
FFFFFFF4
FFFFFFEC
2C
3C
14
FFFFFFD4
FFFFFFD4
FFFFFFF4
FFFFFFEC
1C
FFFFFFE4
FFFFFFE4
FFFFFFF4
FFFFFFE4
FFFFFFF4
FFFFFFDC
14
FFFFFFDC
0C
34
FFFFFFDC
```

Figure 2: Data interface verification

2.4 Validation of TCP Client Application

Our simple and elegant application provides all the benefits and functionality of Xilinx's RF Data Converter tool in the most efficient, user-friendly way which will enable researchers to easily automate their testing procedures. By eliminating the need to use Xilinx's general-purpose user interface, users will be able to run a simple program with optional input parameters defining ADC configurations. This will allow for easy integration of our tool into automation scripts that can be used for accelerating research and improving user experience.

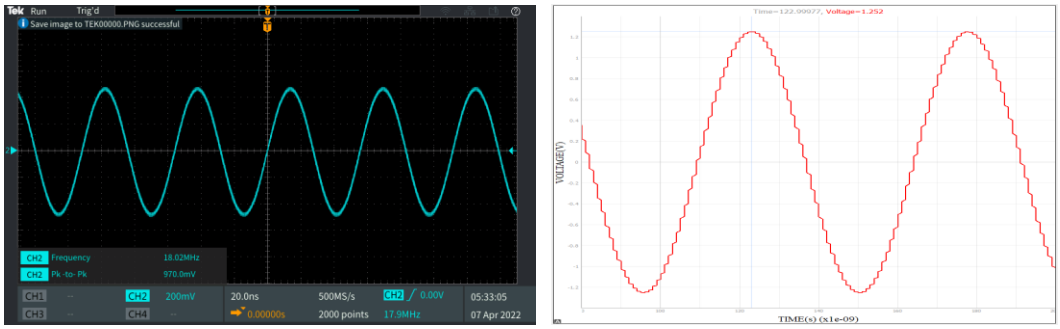
3 Signal Processing

To verify that we are able to measure an analog voltage signal and convert it to the digital domain (**FR2** and **FR3**) we run several tests with varying signal frequencies. All tests are run using an ADC sampling frequency of 1 GSPS as per **NFR1**.

3.1.1 High-Frequency Analog Signal Generation Test

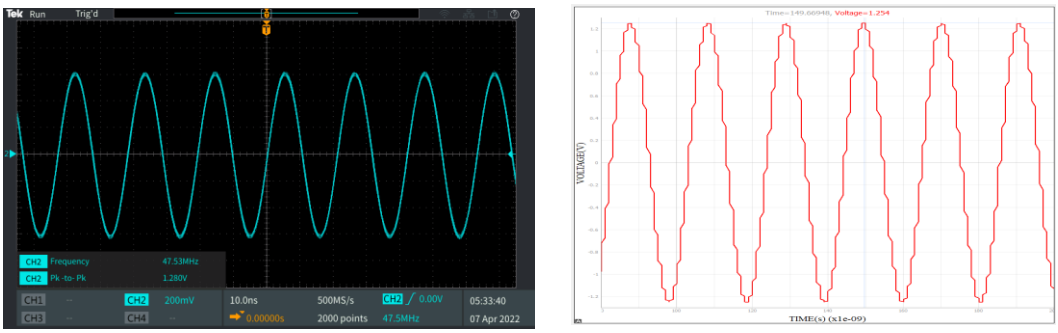
Though it is not a formal requirement for our project, our client has specified that the signal they wish to sample has a frequency of 250 MHz. To ensure that our design is configurable and portable, we have implemented a solution which can measure any signal frequency between up to 500 MHz. Unfortunately, due to limited lab equipment we are unable to generate such a high frequency signal for testing. As a solution, we use the DAC capability of our board to achieve a higher frequency signal. To ensure thorough testing, we first verify that we are able to successfully generate such a signal.

We view the signal generated by the DAC using an oscilloscope and our GUI. To check that the signal is being successfully generated at the desired frequency, we connect the DAC output to the oscilloscope. We verify that signal frequency and peak-to-peak voltage match between the oscilloscope waveform and the waveform plotted using our GUI.



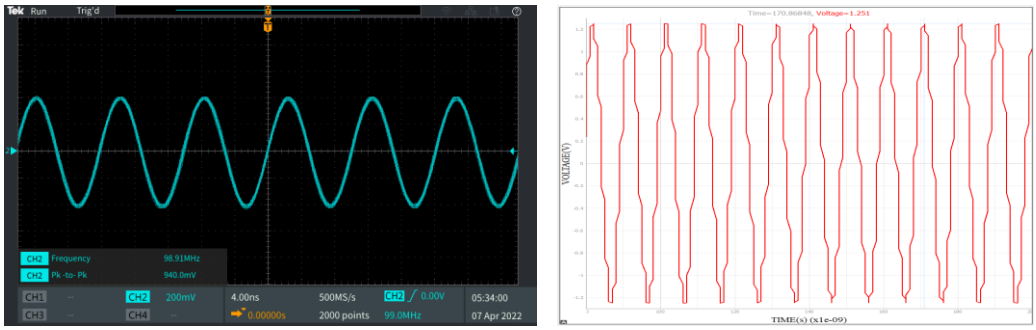
(a) displayed on an oscilloscope (b) using our GUI

Figure 3: 20 MHz signal generated using DAC



(a) displayed on an oscilloscope (b) using our GUI

Figure 4: 50 MHz signal generated using DAC



(a) displayed on an oscilloscope (b) using our GUI

Figure 5: 100 MHz signal generated using DAC

3.1.2 Low-Frequency Analog Sampling Test

We test that we are able to sample different low-frequency signals at different voltage levels. The signals are generated using a signal generator in the ECE lab. Again, we use an oscilloscope to verify that we are sampling and reconstructing the signal accurately.

As shown in Figure 6, label 1 is the ZCU111 evaluation board and the XM500 ADC/DAC board connected with two FMC LPC ports. Label 2 is the signal generator that is used to generate the low frequency signal for testing. The signal is input to the ADC by connecting a wire to the SMA port on the ADC. Label 3 is the oscilloscope.

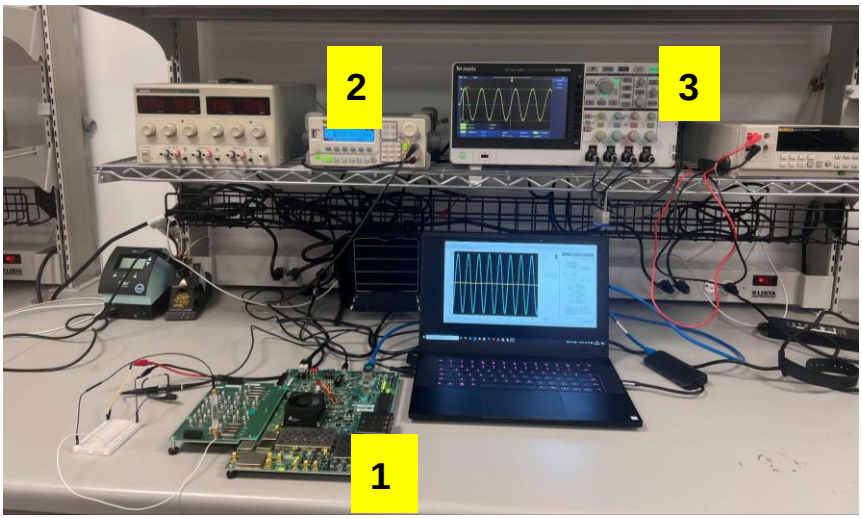
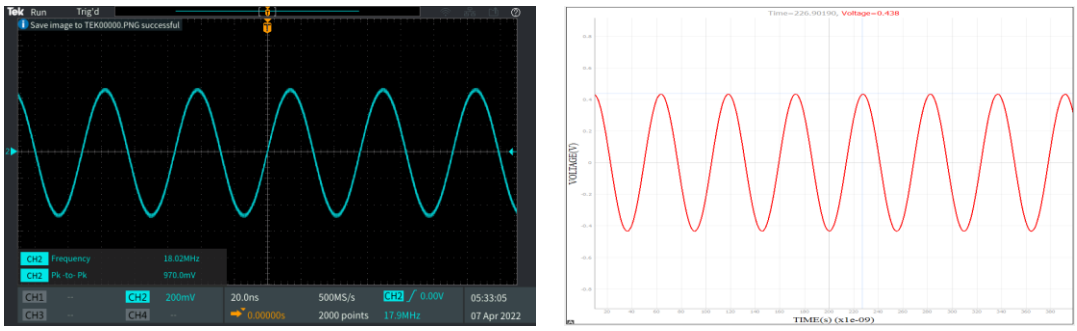


Figure 6: Hardware setup for testing

Given that we were able to connect a signal generator to our system, we can verify that we have satisfied **FR1**. The signal types are the same, and therefore we can safely assume that if one is compatible, the other will be as well.

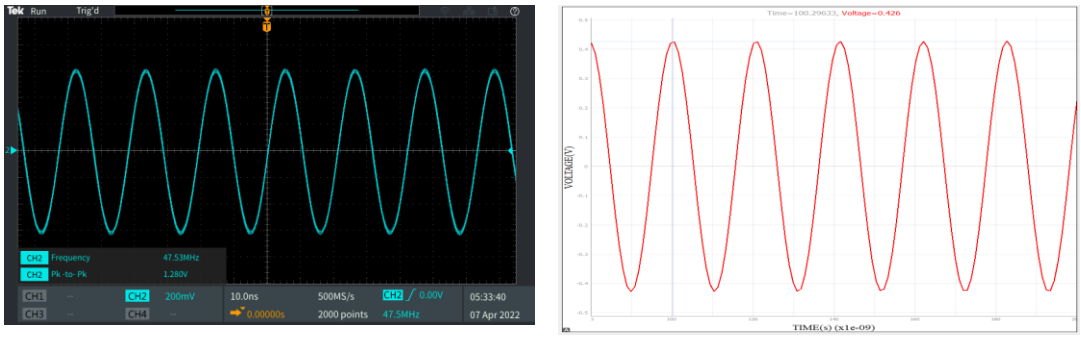
Signals with frequencies of 20 MHz, 50 MHz, and 100 MHz are generated and observed.

We observe that the waveform seen in our GUI matches the waveform seen in the oscilloscope. The peak-to-peak voltage and the signal frequency match. This test verifies that we are able to measure analog voltage (**FR2**) and convert it to the digital domain (**FR3**).



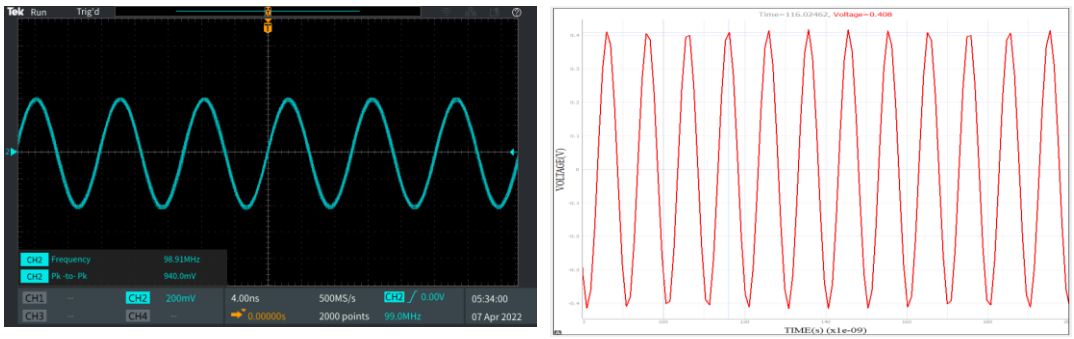
(a) displayed on an oscilloscope (b) using our GUI

Figure 7: 20 MHz signal sampled by ADC



(a) displayed on an oscilloscope (b) using our GUI

Figure 8: 50 MHz signal sampled by ADC



(a) displayed on an oscilloscope (b) using our GUI

Figure 9: 100 MHz signal sampled by ADC

3.2 High-Frequency Analog Sampling Test

We cannot use the oscilloscope in the UBC lab for the high frequency test because it is limited to 100 MHz frequency. Due to these limitations, we used our data processing program to analyse the signal that is being read from the ADC. The ADC samples the high-frequency signal generated by the DAC. Signals at 250 MHz, 400 MHz, and 500 MHz frequency are generated. We measured the frequency of the signal and the peak-to-peak voltage using our data processing program. These tests showed that our ADC successfully samples high frequency signals.

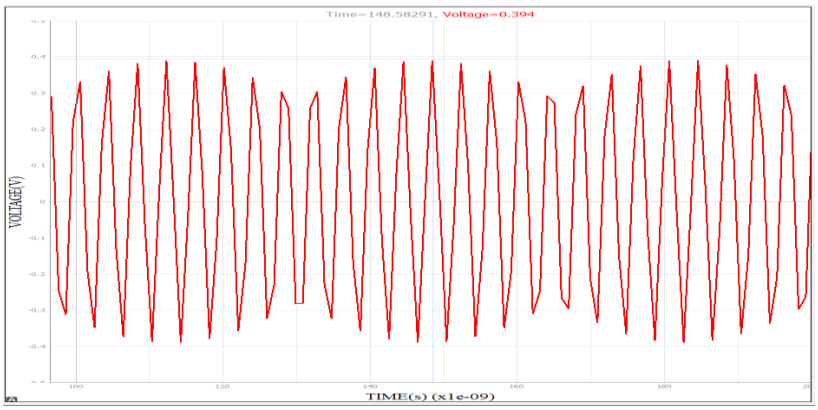


Figure 10: 250 MHz signal sampled by ADC

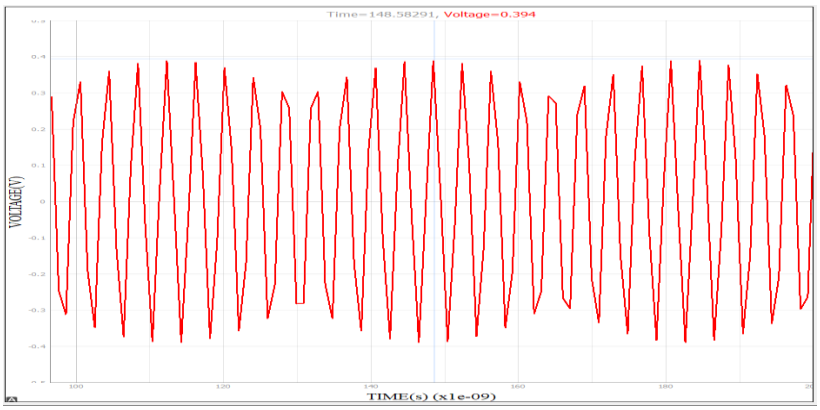


Figure 11: 400 MHz signal sampled by ADC

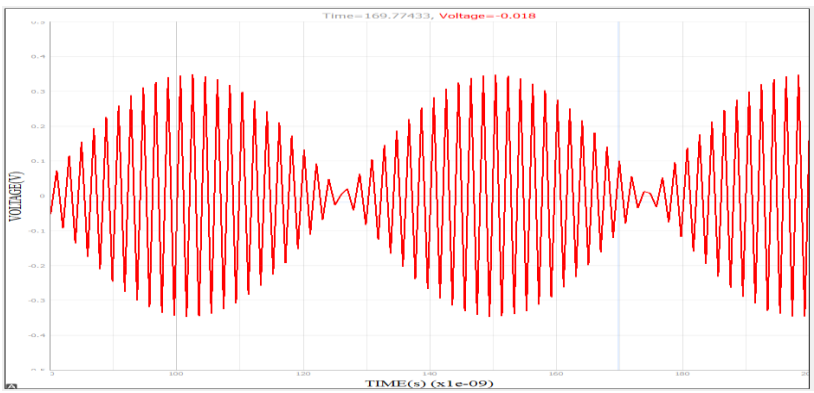


Figure 12: 500 MHz signal sampled by ADC

3.2.1 No Filter Test

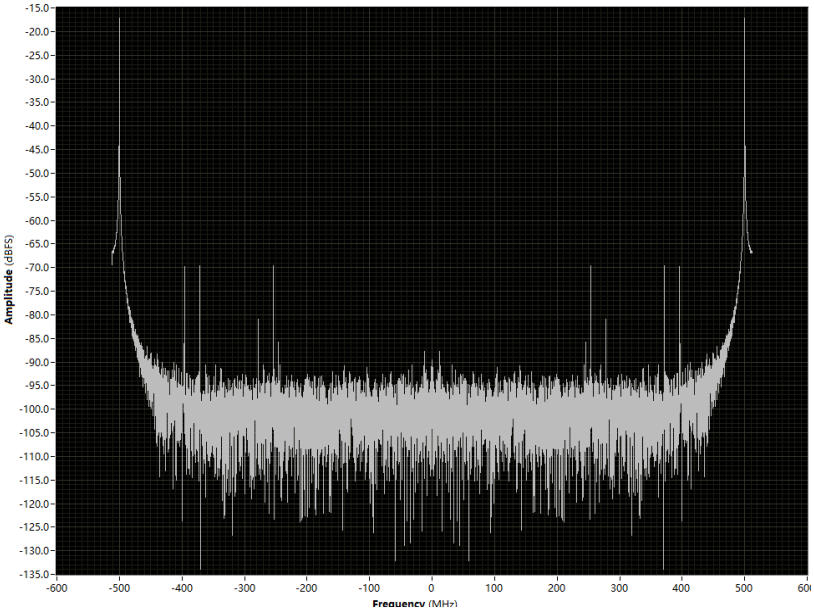


Figure 13: ADC sampled data using no filter in the frequency domain

Figure 13 shows the signal sampled without using any filters. Without any filters, we observe noise in the frequency range above 500 MHz. We wish to attenuate this noise in order to prevent aliasing.

3.2.2 Low-Pass Filter Test

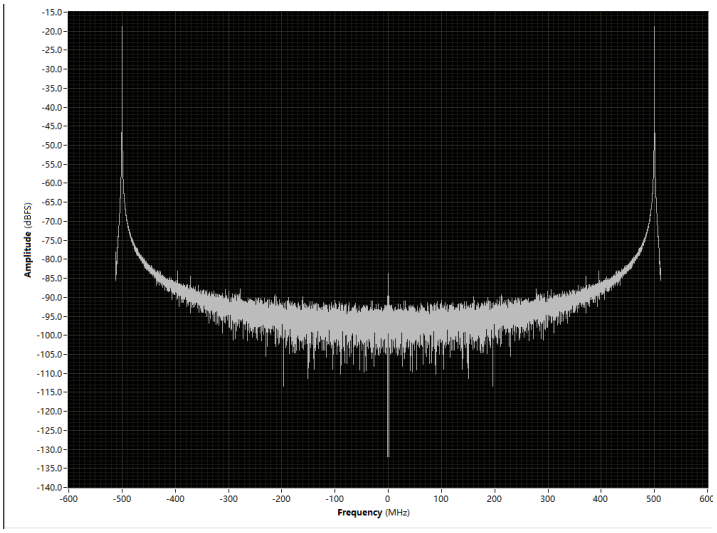


Figure 14: ADC sampled data with a 1300 MHz low-pass filter

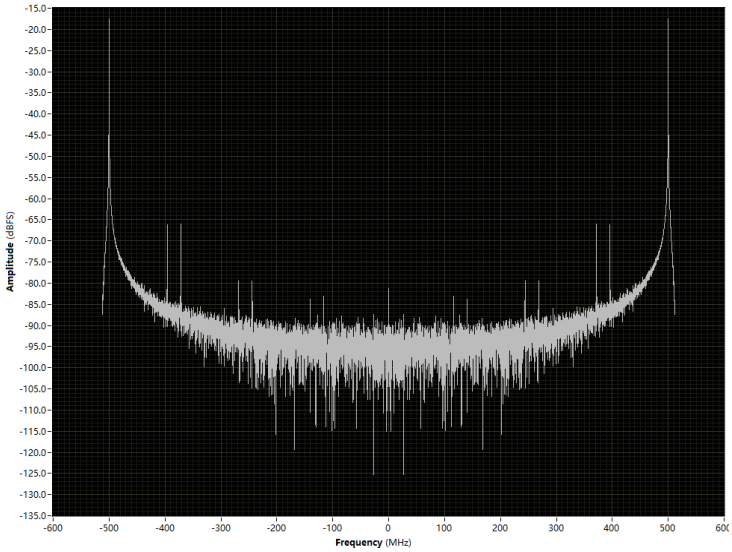


Figure 15: ADC sampled data with a 2500 MHz low-pass filter

As shown in the figures above, the 2500 MHz filter eliminates far less noise than the 1300 MHz filter. The is because with a sampling rate of 1 GHz, the Nyquist Frequency is 500 MHz, so with the higher cut-off point of the 2500 MHz filter, all noise in the frequency range 500 MHz – 25000 MHz will be aliased, resulting in the low-frequency noise observed in Figure 15.

Ideally, we would use an analog filter with a cut-off equal to the Nyquist Frequency, since any higher frequency will be aliased, reducing the signal to noise ratio (SNR). Due to our limited selection of anti-aliasing filters, and the possibility of increasing sampling rate in the future, the 1300 MHz filter is the most suitable for our project. We conclude that **FR4** has been met, as we are able to eliminate aliasing with the use of the 1300 MHz filter.

3.2.2 Band-Pass Filter Test

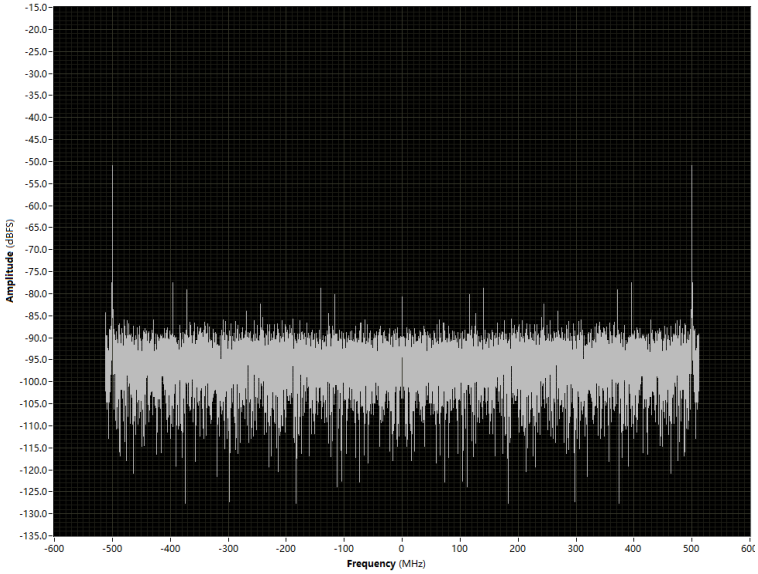


Figure 16: ADC sampled data with 3000-4300 MHz Band-Pass filter

As shown in figure 16, the 500 MHz signal is outside the filter bandwidth and has been significantly attenuated by the filter. So we think 3000–4300 MHz bandpass filters are not suitable for use in this project.

4 Data Storage Module Test

4.1 Test Descriptions

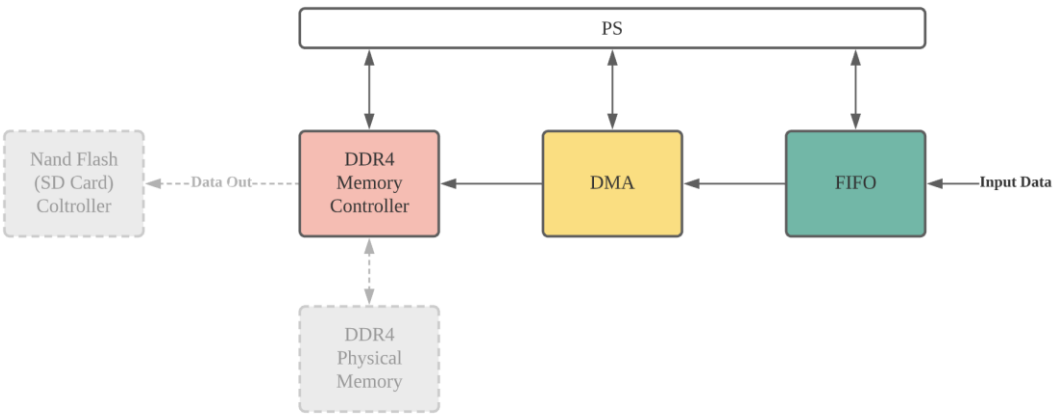


Figure 17: Digital System Block Diagram

Figure 17 is the overall digital system block diagram. However, for the Alinx board, we were focused on getting the DDR4 and FIFO modules working. We have a Top.sv file that includes

all testing modules. Theoretically if the system performs as predicted, there will be made up input data generated by the processor system entering the FIFO, then written to DDR4 physical memory.

To ensure the module behaviour is as expected, we used an IP block ILA (Integrated Logical Analyzer so that signals are forwarded to Vivado to generate a waveform. This IP core allows us to analyse logic in real time as it is occurring on our board, therefore we can conclude any waveforms generated are valid both in simulation and synthesis.

After this series of tests is complete, we will verify the Data Storage Module's capability of storing data into DDR4 memory.

4.2 AXKU040 Data Storage Test

4.2.1 AXKU040 FIFO Test

Since our ADC will be running at a ultra-fast sampling speed, we need some kind of buffer to get every sample data recorded. FIFO is a solution for buffering the data before sending it to the memory module. It buffers the data into logic gates and releases using rules of "First in first out". In our design, since our ADC has sampling resolution of 8 bit, we will shift the ADC data before sending it to FIFO so that the data size matches the 64-bit data width of our memory module.

We used a "FIFO Generator" IP Core to build a FIFO block. This block has a 64-bit data input to match with DDR4 memory data width, and a 2048 depth. The depth is the number of input data samples we can fit into our FIFO before we need to begin emptying it. However, due to the limitation of IP blocks from Xilinx, the real depth of this module is 2047. To verify the functionality of this core, we set up a state machine which fills in the FIFO with incrementing data (0, 1, 2...etc.) when the FIFO is empty. Once the FIFO is full, we read data from the FIFO. The Empty and Full signal will indicate the status of our FIFO module.

The last two signals in Figures 18 and 19 are the state parameter. 1 means "idle" and 2 means "at work".

FIFO Empty & Fill in Data

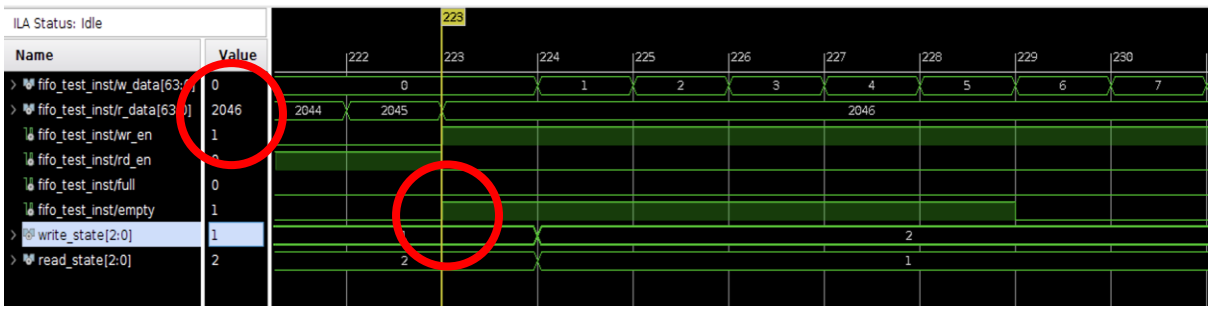


Figure 18: FIFO empty signal

In the waveform above (Figure 18), we can see that when the read data hits 2046 all of the space inside FIFO has been cleared out, so the **empty signal** becomes high. At this stage, this module will go into the **write state(write_state = 2)**, where it begins to write data from 0 to 2046.

FIFO Full & Release Data

Figure 19: FIFO full signal

When the write data hits 2046, the **full signal** goes high. Then the module will enter the **read state (read_state = 2)**. It will reset the w_data counter to 0 and then proceed to read from the FIFO.

From the graph, we can see that there is latency between the start of the write and read enable signal until the data arrives. In this case (64-bit width and 2047 depth), the latency is around 6 clock cycles.

4.2.2 AXKU040 DDR4 Test

We used the 'DDR4 SDRAM(MIG)' IP core to generate a memory controller. In the top level file, we instantiate mem_test.sv, axi_master.sv and the IP core for the DDR4. The AXI master allows the processing system to communicate with the FIFO and DDR4. The AXI protocol illustrated in the figure below (DDR_TEST Block). We have yet to implement a DMA controller into this test.

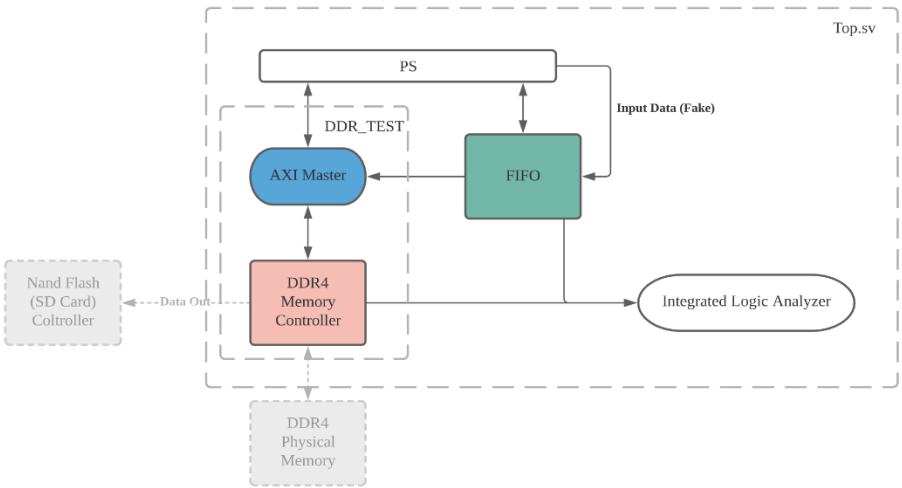


Figure 20: DDR4 Block Diagram

The module mem_test will send a burst start address to axi_master and increment the data by 1 when written to the next address. Here, the burst data width is 64-bits, and each burst has 128-bit length. The burst start address is sequentially given by the mem_test module. In Figure 21, we can see that **once the previous write request is finished**, the new burst start address is b2d400 and the data is 0h515(last three digits). Since the burst length is 128 which is equal to 0h80, the last data for this burst instruction will be 0h595 which is correct compared to the second part of Figure 21(burst finish = 1 @ 0h595).

The same idea will apply to the third part of Figure 21. This function also has error detection. Data read out of the FIFO at any address is compared to the data intended to be written at that address. If these are not equal, an error signal will be set and remain high for the rest of the time. As we can see from the waveform, the error signal is always 0. Which means data has accurately been written to memory.

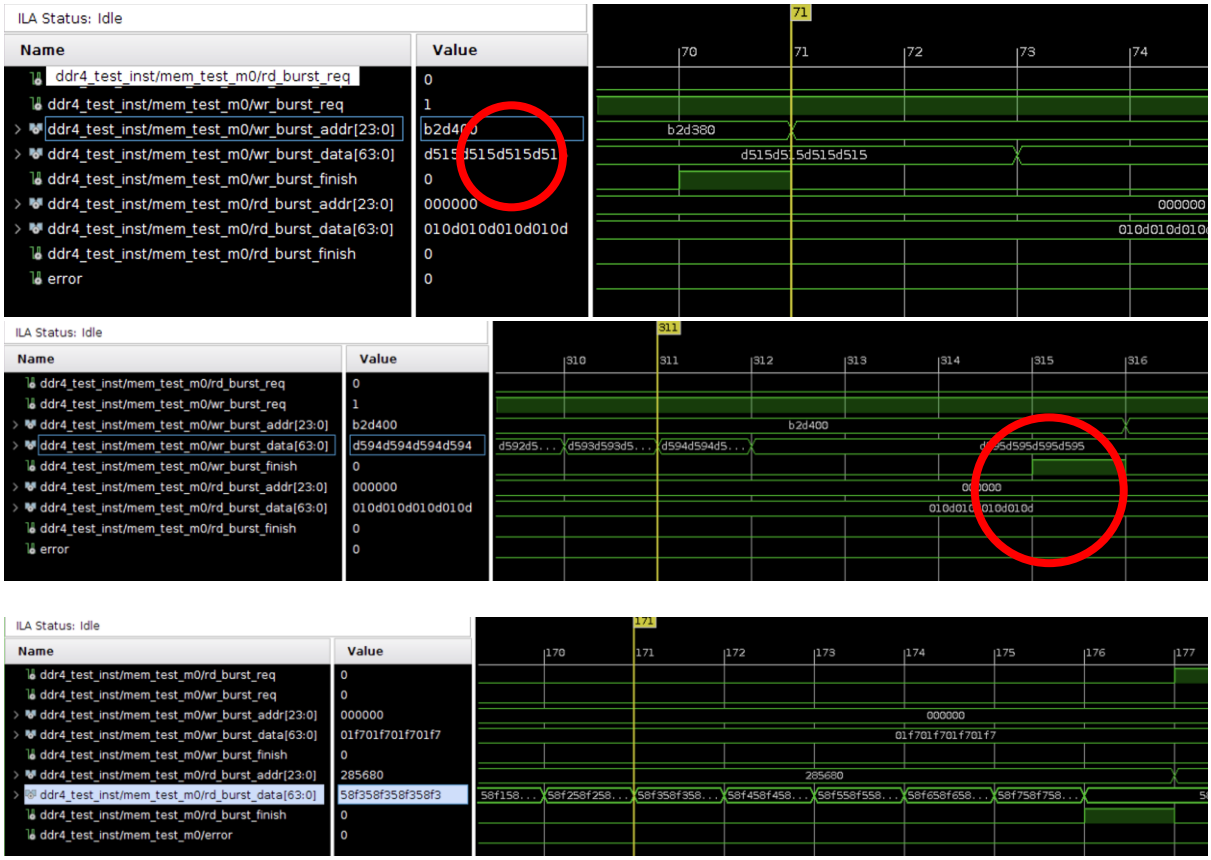


Figure 21: Process of DDR4 read&write

4.2.3 Merged FIFO & DDR4 Test

Figure 22 is the test result of the Top.sv module. It connects the FIFO output to the DDR4 input. Our Alinx sub team is still working on it since the data read from the DDR4 is not correct. The bottom wave is the state signal which clearly shows the state transition is happening. More work is needed to complete this test.

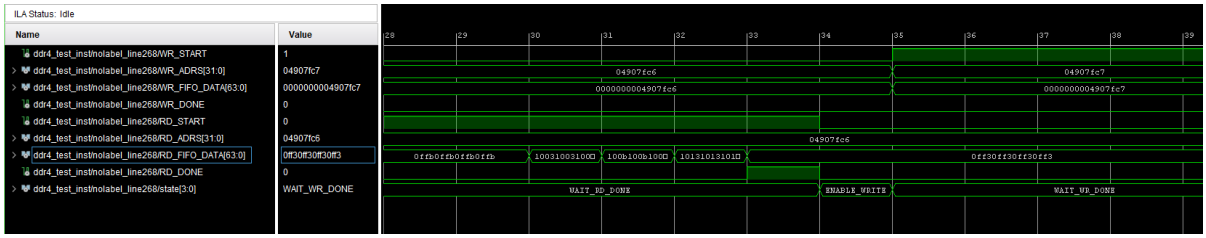


Figure 22: Process of FIFO and DDR