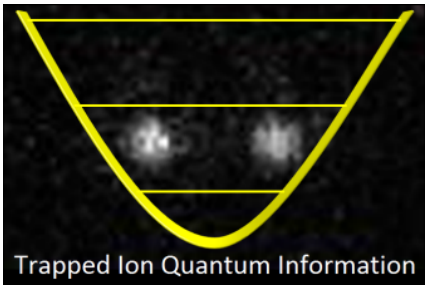

A 16 Channel DDS Implementation For The ZCU216 RFSoC Evaluation Board

Ali Doruk Bekatli, Sevket Baturay

Group Project

Supervisors: Dr. Abdulkadir Akin, Martin Stadler
Principal Investigator: Prof. Jonathan Home

Trapped Ion Quantum Information Group, Institute for Quantum Electronics, ETH Zürich

September 2022

Abstract

Trapped ion experiments often require fast and low noise signal generators with synchronized outputs that can be easily controlled via industry standard interfaces such as Ethernet. Increasing the number of ions in a quantum experiment requires the development of a scalable control system with a high number of analog output channels. This defines a need for certain components: a processor for control and communication, digital logic to generate samples, high-speed Digital-to-Analog Converters (DAC) to generate the signals, and a synchronization methodology between boards and channels. Due to these requirements, the *ZCU216 UltraScale+ RFSoC Evaluation Board* was chosen as it fulfills the requirements and contains 16 9.85 GSPS DACs that support synchronization. The DACs are synchronized using a common clock and an external signal which means that it is possible even to synchronize DACs on different boards.

The goal of this group project was to port an existing Direct Digital Synthesis (DDS) implementation written in Verilog to the ZCU216 platform, implement synchronization of all DAC channels, create an AXI-Lite peripheral along with a C library to be able to control the DDS implementation with the embedded ARM processor, and report the performance results of DAC outputs. Since synchronization imposes certain requirements on the clock and sample frequencies of the system, a sample rate of *9.6 GSPS* was chosen and the DDS logic is clocked at *600 MHz*. Each DAC is connected to an adder that sums the outputs of 4 DDS modules meaning all 16 DACs are capable of outputting the superposition of 4 sinusoidal waveforms of varying frequencies, amplitudes, and phases. The resulting implementation contains 64 individual DDS signal generators and was characterized using SNR, SFDR, and crosstalk measurements at different test frequencies.

Contents

1 Introduction

Trapped ion quantum information processing experiments and many other kinds of experiments involving quantum physics often require programmable signal generators [2]. Such signal generators need to be capable of generating sine waves with arbitrary amplitudes, frequencies and phases while having as little noise at the output side as possible. These criteria necessitate the usage of high sampling rate, low noise DACs and sample generators that are fast enough for the chosen DAC. While it is possible to build such signal generators using discrete components and a self-designed PCB, this presents a major challenge as designing PCBs intended to work with high-speed analog and digital signals is a whole subject of its own.

A simple way of generating sine waves is utilizing an address counter and a lookup table per signal. This approach is very simple and easy to implement but is very inflexible as changing the content of the lookup table on the fly imposes data bandwidth restrictions. Direct Digital Synthesis (DDS)[1] on the other hand is a technique that can be used to create sine waves with arbitrary frequencies, amplitudes and phases. DDS also utilizes a lookup table but replaces the simple address counter with a phase accumulator that has a tuning input. This tuning input can be used to adjust the phase of the signal by adjusting the initial address that will be sent to the LUT and the frequency by adjusting the jump size of the addresses.

The *Xilinx Zynq UltraScale+ RFSoC ZCU216 Evaluation Kit* was chosen as the target platform. The main board features a XCZU49DR-2FFVF1760, a part that combines 16 14-bit, 9.85GSPS RFDACs, 16 14-bit 2.5GSPS ADCs, and a reasonably large Zynq Ultrascale+ FPGA in a single package. The kit completely eliminates the need to design custom hardware out of discrete components as it also contains the CLK104 clock generator board and the XM655 breakout board. In other words, the evaluation kit contains everything necessary to get a signal generation system up and running. Although the RFSoC parts are intended for rather cutting-edge applications such as 5G transceivers and phased array radars, it is more than adequate for our desired quantum control system application.

The Zynq part on the chosen evaluation kit contains 4 Cortex-A53 cores, 2 Cortex-R5F real-time cores, and standard communication interfaces such as Ethernet (RJ45 and SFP28) and USB. Our design implements the sample generation logic on the Programmable Logic (PL) side and the control part on the Programmable Software (PS) side through one of the A53 cores. Utilization of the A53 core simplifies the task of implementing some form of control logic for the sample generators and makes it possible to easily control them over Ethernet.

Since the target platform was already chosen before the start of our group project, our main implementation task was getting accustomed to the RFSoC and then porting the TIQI DDS firmware to the new hardware. We first wrote a crude lookup-table-based sine generator that ran at 9.85GSPS to see what was necessary in terms of clocking and then moved on to porting the DDS code itself. We first had to change the output of the DDS cores to fit the requirements of the DACs (16 samples per clock at 600MHz) and then worked on improving the performance of the DDS code to meet timing requirements. After having managed to get the DDS code working, we then created an AXI-Lite peripheral to control the DDS modules using the ARM cores and lastly implemented multi-tile synchronization.

In the end, a sampling rate of 9.6GSPS was chosen, all timing issues were solved, and synchronization was implemented using only the hardware that came with the evaluation kit.

2 Architecture

The main parts of our design are the ZYNQ Processing System, the RF Converter module, and the DDS Core. The 16 RFDACs of the RFSoC are organized as 4 individually clocked Tiles with each of them containing 4 RFDACs. Since we want to have a defined phase relation between all outputs, multi-tile synchronization had to be implemented. This was rather challenging as synchronization imposes certain requirements on the chosen reference clock signals and available sample rates. The block diagram of the design and the DAC settings that were used can be seen in Figures 1 and 2

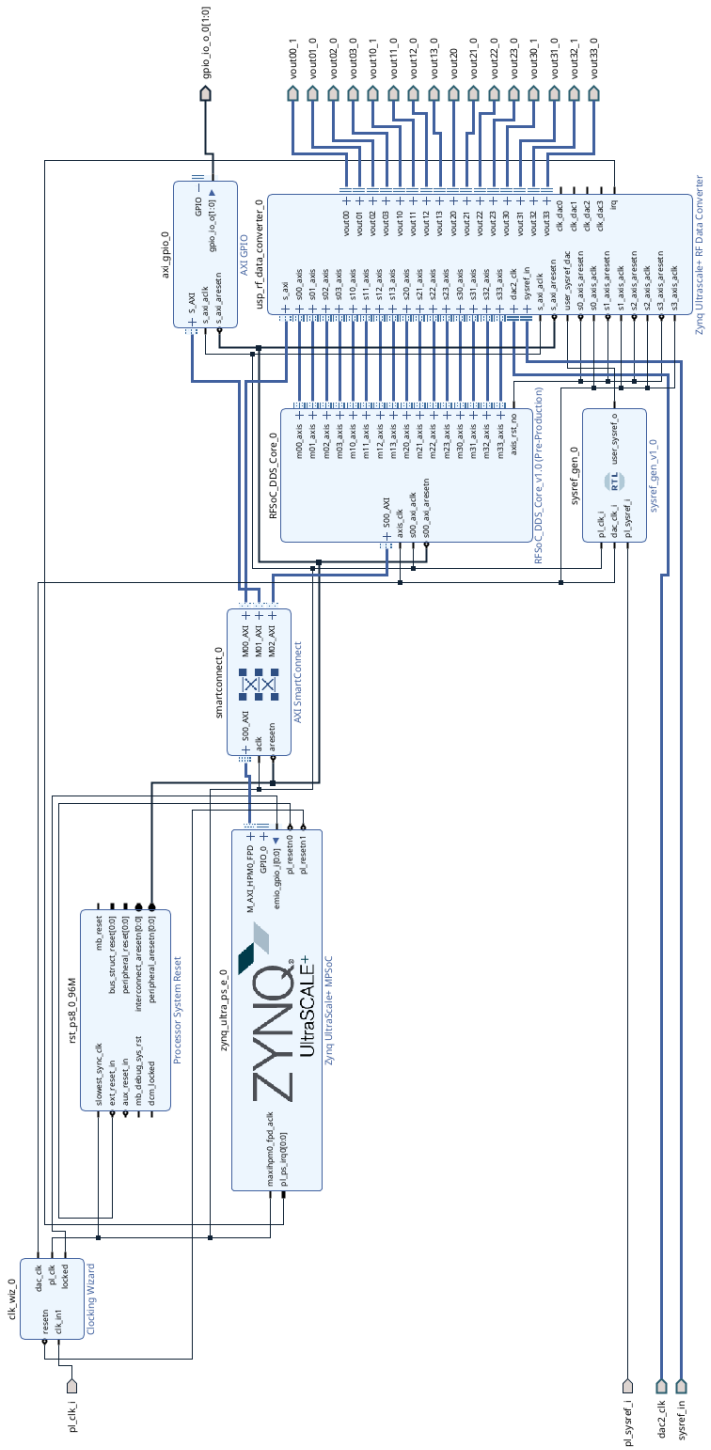

Figure 1: The block diagram of the digital logic. The "RFSoC_DDS_Core IP" block contains the DDS cores, the DDS wrapper and the AXI-Lite interface

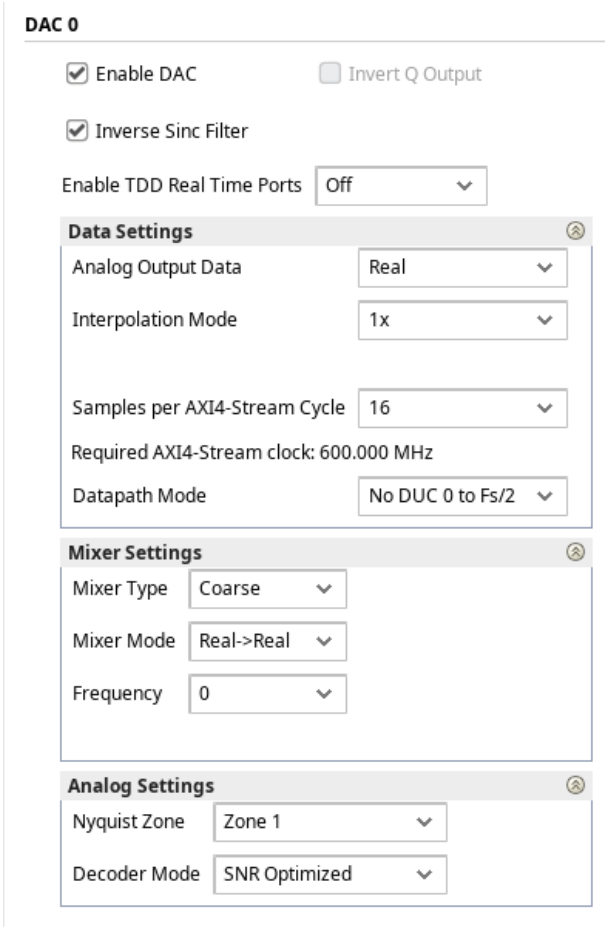

Figure 2: DAC settings

2.1 Hardware Structure

The structure of the digital logic is rather straightforward and is centered around the Advanced eXtensible Interface (AXI) bus. The AXI is a very common bus protocol, especially in projects utilizing Xilinx parts. The RF data converter block, the DDS cores, and the GPIOs are all controlled by the ARM cores through AXI.

Since there are multiple ICs on the CLK104 board, a MUX chip on the CLK104 has been used to connect the data outputs of these ICs to the processor, necessitating the usage of a GPIO IP to control the MUX through 2-bits in order to read data from the selected IC. Even though this is not required for the CLK104 board to function, we implemented it during the early stages of the project to verify that the writes to the CLK104 were successful.

The RF data converter IP does not function unless it has been reset properly which made it necessary to implement a proper reset solution. The entirety of the digital logic (including the clocking wizard/MMCM) cannot be reset simultaneously as the MMCM needs time to lock and generate an input signal. This problem was solved by using separate reset signals for the MMCM and the rest of the logic. The processor sets up the CLK104 board, resets the MMCM, waits for it to lock to the input signal, and then resets the rest of the logic. Lastly, since the reset signals are synchronous to PL_CLK, the DDS IP contains *xpm_cdc_sync_rst* reset signal synchronizers to handle the Cross Domain Crossing (CDC) from PL_CLK to the DAC clock.

2.2 Clocking

First of all, the Tiles and the logic driving the DACs need to be driven using the same clock signal (PL_CLK) either directly or using a PLL/MMCM. Then, a free-running reference signal (PL_SYSREF) with a frequency smaller than 10MHz needs to be supplied for synchronization. This reference clock also needs to be an integer submultiple of PL_CLK and the DAC clock that's generated from it. Due to these stringent requirements, we've opted for a sample rate of 9.6GSPS, 600 MHz for the DAC clocks, 120MHz for PL_CLK and 4 MHZ for PL_SYSREF. The clocking settings of each tile can be seen in Figure 3.

Tile Clocking Settings

Tile	Sampling Rate (GSPS)	Max Fs (GSPS)	PLL	Reference Clock (MHz)	PLL Ref Clock (MHz)	Ref Clock Divider	Fabric Clock (MHz)	Clock Out (MHz)	Clock Source	Distribute Clock
ADC 224	2.0	2.500	☐	2000.000	-	1	0.0	250.000	Tile224	Off
ADC 225	2.0	2.500	☐	2000.000	-	1	0.0	31.250	Tile225	Off
ADC 226	2.0	2.500	☐	2000.000	-	1	0.0	31.250	Tile226	Off
ADC 227	2.0	2.500	☐	2000.000	-	1	0.0	31.250	Tile227	Off
DAC 228	9.6	10.000	☑	240.000	240.0	1	600.000	600.000	Tile230	Off
DAC 229	9.6	10.000	☑	240.000	240.0	1	600.000	600.000	Tile230	Off
DAC 230	9.6	10.000	☑	240.000	240.0	1	600.000	600.000	Tile230	Input Refclk
DAC 231	9.6	10.000	☑	240.000	240.0	1	600.000	600.000	Tile230	Off

Figure 3: Tile clocking settings

The analog and digital DAC reference clocks(240 MHz), the input signal for the MMCM(pl_clk_i in Figure 3, 120 MHz), and the synchronization signal(4 MHz) are all generated by the LMK04828 IC on the CLK104 board. The CLK104 board contains a 10 MHz Temperature Controlled Crystal Oscillator(TCXO) and a separate external reference input. We used an external clock source in our experiments as the RFSoC will most likely be used with an external clock source in an actual trapped ion experiment. The RF PLLs on the CLK104 aren't used in our design as our design utilizes the internal PLLs of the RFDACs to generate the required clock signals from the 240 MHz reference clocks.

Furthermore, an MMCM is used to generate the clock signals for the DACs and the DDS cores. The MMCM generates the 600 MHz clock signal for the AXI Stream interfaces of the RF data converter and the DDS IP and the 120 MHz clock signal for the rest of the logic. The MMCM generates a separate 120 MHz signal even though its input signal also has a frequency of 120 MHz because clocking the logic using the MMCM's input signal instead of a signal generated by the MMCM increases routing difficulty and caused timing violations. The SYSREF signal is sampled using two FFs clocked by the 120 MHz and the 600 MHz signal as explained in the RF Data Converter user guide.[4]. The RF-Converter generates a 9.6 GHz clock internally. Figure 4 illustrates how all clock and reference signals are generated and distributed.

The longest path of the design is contained within the lookup table of one of the DDS modules and has a delay of 1.666 ns.

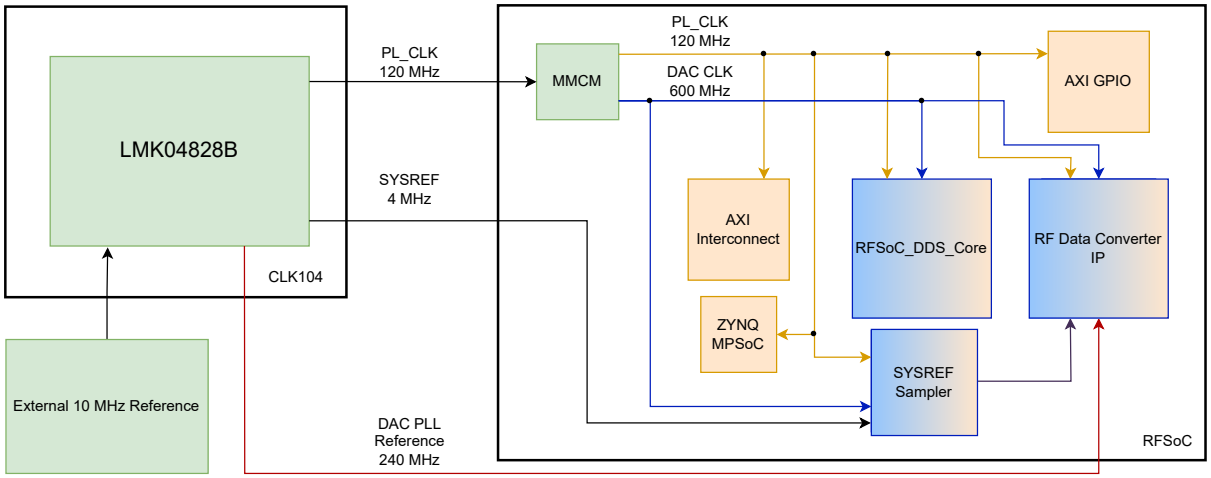

Figure 4: Block diagram of the clock distribution system

2.2.1 Timing Issues

The DDS implementation was originally intended to run at a clock frequency much lower than 600 MHz with fewer total DDS cores and only 2 DDS modules per core, as it was designed for 1.0 GSPS DAC channels [2]. Therefore, resolving timing violations to support the 9.6 GSPS DACs took up the majority of the time we spent on the project. According to the timing reports, nearly all paths are dominated by routing delays as opposed to logic delays due to the very high utilization of the design.

First of all, specific synthesis and implementation strategies had to be chosen. The DDS IP and the RF Data Converter are synthesized using the *Flow_PerfOptimizedHigh* strategy due to their high clock frequency whereas the rest of the design is synthesized using the *Flow_AlternateRoutability* strategy to ease routing. *Phys_opt_design* is also run after placement and routing, which are run with the *Explore* and *NoTimingRelaxation* strategies respectively. The code had to be adapted as well, many pipeline stages were inserted in the DDS modules along with some code refactoring. Furthermore, a pipeline stage had to be added to the output of the 14 to 16-bit sample converter and the reset signals of the DDS cores were pipelined as well to solve the last remaining timing issues. Although these pipeline stages didn't actually shorten any critical paths, they ease the task of the router which solved the timing issues as the longest paths are dominated by routing delays as previously mentioned.

2.3 The DDS IP

The original TIQI DDS code contains a wrapper that instantiates all submodules and includes an instruction decoder that inputs 40-bit instructions over UART. Each instruction contains fields that indicate the type of the instruction, the parameter the instruction should modify and the desired value, and the number of the DDS core the instruction should act upon. Several modifications to the original wrapper were made and an AXI-Lite peripheral was added. The instruction width has also been increased to 41 bits to accommodate the increased number of DDS modules.

The AXI-Lite peripheral has two 32-bit registers. A finite state machine (FSM) monitors the registers, assembles the 41-bit instruction from the contents of the two registers after the registers have been written to, and sends the instruction to the DDS wrapper. The DDS wrapper instantiates DDS cores that contain 4 DDS modules and a sample adder that combines the outputs of the DDS modules. As the RFSoC has 16 DACs and each DAC has its own DDS core, the final design contains a total of 64 DDS modules. The DACs are 14 bit but the RF Data converter has a 16-bit data path so the 14-bit samples have to be converted to 16-bit MSB aligned samples, which is also done by the wrapper. Each DAC channel of the RF Converter receives 16 consecutive samples (256-bits) at 600 MHz to generate analog signals at 9.6 GSPS. Therefore, 16 DACs receive 256x16-bits at 600 MHz from the *dds_core_wrapper*. The structure of the *dds_core_wrapper* is illustrated in Figure 5

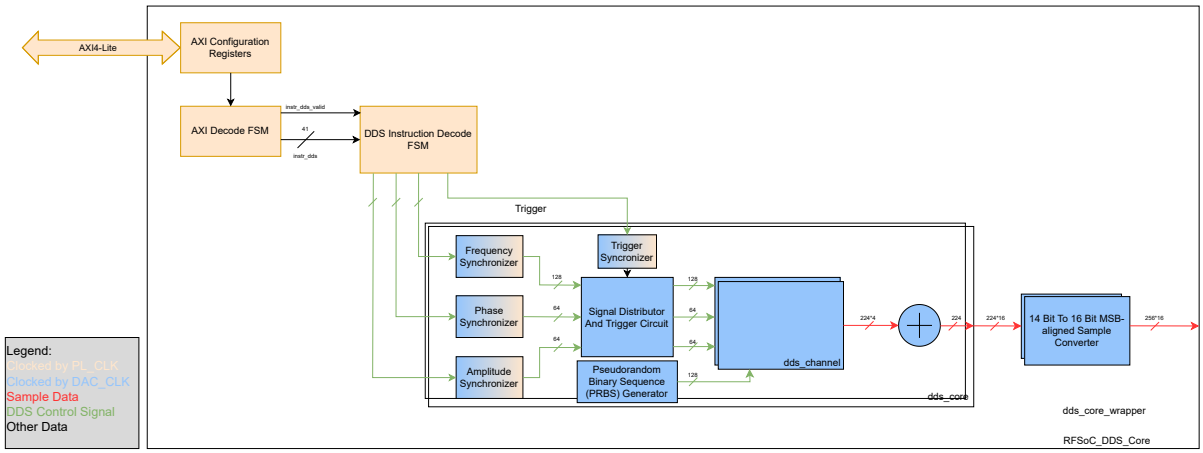

Figure 5: The block diagram of DDS IP

Each DDS module has a 32-bit frequency input, a 16-bit amplitude input, and a 16-bit phase input. The wrapper decodes instructions and updates the contents of the registers that are connected to the inputs of the DDS cores. The DDS cores sample their inputs when they receive a trigger signal from the wrapper and the registers that sample the inputs are connected to the individual DDS modules. The DDS cores contain *xpm_cdc* synchronizers for the parameter inputs and the trigger signals. Unfortunately, we had to use multiple single-bit synchronizers instead of a proper handshake-based solution as the increased logic utilization caused increased routing difficulty and multiple timing violations. Nevertheless, the DDS cores run at 600 Mhz whereas the instruction decoder runs at 120 MHz and the cores sample the data only when a trigger pulse has been received so the design still works without issues.

2.4 Implementation Results

As can be expected, the design is large and utilizes one-third of available LUTs and BRAMs. Although there is a fair bit of free resources on the chip left, the logic is quite spread out due to the vertical alignment of the RFDACs and we've observed that even small changes cause timing violations. As such, future additions to the PL side of the design for any kind of signal processing purposes may increase routing complexity and would require further improvements to the DDS core.

It can be observed in Figure 6 that the DDS cores utilize the highest amount of resources while the RF Data Converter IP also uses a relatively less but still considerably high amount of resources. The rest of the logic on the other hand is much more lightweight in terms of resource utilization as expected. The overall utilization of the design and a post route illustration can be seen in Figures 7 and 8 respectively.

Name	CLB Registers (850560)	CLB LUTs (425280)	CARRY8 (53160)	F7 Muxes (212640)	F8 Muxes (106320)	CLB (53160)	LUT as Logic (425280)	LUT as Memory (213600)	Block RAM Tile (1080)	DSPs (4272)
main	251294	126040	14634	136	3	29670	125525	515	256	1024
dac_inst (DAC_wrapper)	251294	126040	14634	136	3	29670	125525	515	256	1024
DAC_i (DAC)	251294	126040	14634	136	3	29670	125525	515	256	1024
axi_gpio_0 (DAC_axi_gpio_0_0)	49	41	0	0	0	9	41	0	0	0
clk_wiz_0 (DAC_clk_wiz_0_1)	0	0	0	0	0	0	0	0	0	0
RFSoC_DDS_Core_0 (DAC_RFSoC_DDS_Core_0_0)	245621	119429	14592	32	0	28532	118917	512	256	1024
inst (DAC_RFSoC_DDS_Core_0_0_RFSoC_DDS_Core_v1_0)	245621	119429	14592	32	0	28532	118917	512	256	1024
dds_core_wrapper_inst (DAC_RFSoC_DDS_Core_0_0_dd	245252	117991	14592	0	0	28468	117479	512	256	1024
channel[0].gen_dds.dds_core_inst (DAC_RFSoC_DDS_	14674	7361	912	0	0	1971	7329	32	16	64
channel[1].gen_dds.dds_core_inst (DAC_RFSoC_DDS_	14603	7357	912	0	0	1828	7325	32	16	64
channel[2].gen_dds.dds_core_inst (DAC_RFSoC_DDS_	14603	7370	912	0	0	1967	7338	32	16	64
channel[3].gen_dds.dds_core_inst (DAC_RFSoC_DDS_	14634	7349	912	0	0	2034	7317	32	16	64
channel[4].gen_dds.dds_core_inst (DAC_RFSoC_DDS_	14639	7339	912	0	0	1950	7307	32	16	64
channel[5].gen_dds.dds_core_inst (DAC_RFSoC_DDS_	14669	7411	912	0	0	1889	7379	32	16	64
channel[6].gen_dds.dds_core_inst (DAC_RFSoC_DDS_	14606	7348	912	0	0	1941	7316	32	16	64
channel[7].gen_dds.dds_core_inst (DAC_RFSoC_DDS_	14603	7353	912	0	0	2025	7321	32	16	64
channel[8].gen_dds.dds_core_inst (DAC_RFSoC_DDS_	14603	7383	912	0	0	2118	7351	32	16	64
channel[9].gen_dds.dds_core_inst (DAC_RFSoC_DDS_	14603	7364	912	0	0	1773	7332	32	16	64
channel[10].gen_dds.dds_core_inst (DAC_RFSoC_DD	14603	7352	912	0	0	1722	7320	32	16	64
channel[11].gen_dds.dds_core_inst (DAC_RFSoC_DD	14603	7381	912	0	0	1996	7349	32	16	64
channel[12].gen_dds.dds_core_inst (DAC_RFSoC_DD	14612	7392	912	0	0	1938	7360	32	16	64
channel[13].gen_dds.dds_core_inst (DAC_RFSoC_DD	14603	7422	912	0	0	1970	7390	32	16	64
channel[14].gen_dds.dds_core_inst (DAC_RFSoC_DD	14612	7401	912	0	0	1956	7369	32	16	64
channel[15].gen_dds.dds_core_inst (DAC_RFSoC_DD	14603	7384	912	0	0	2000	7352	32	16	64
go_synchonizer_handshake (DAC_RFSoC_DDS_Core_	42	2	0	0	0	26	2	0	0	0
xpm_cdc_sync_rst_inst (DAC_RFSoC_DDS_Core_0_0_	4	0	0	0	0	1	0	0	0	0
RFSoC_DDS_Core_v1_0_S00_AXI_inst (DAC_RFSoC_DDS_	369	1438	0	32	0	708	1438	0	0	0
rst_ps8_0_96M (DAC_rst_ps8_0_96M_0)	35	17	0	0	0	10	16	1	0	0
smartconnect_0 (DAC_smartconnect_0_0)	847	838	0	0	0	213	837	1	0	0
sysref_gen_0 (DAC_sysref_gen_0_0)	4	0	0	0	0	3	0	0	0	0
usp_rf_data_converter_0 (DAC_usp_rf_data_converter_0_0)	4738	5715	42	104	3	1165	5714	1	0	0
zynq_ultra_ps_e_0 (DAC_zynq_ultra_ps_e_0_0)	0	0	0	0	0	0	0	0	0	0

Figure 6: Resource utilization table

Resource	Utilization	Available	Utilization %
LUT	126040	425280	29.64
LUTRAM	515	213600	0.24
FF	251294	850560	29.54
BRAM	256	1080	23.70
DSP	1024	4272	23.97
IO	7	408	1.72
BUFG	5	696	0.72
MMCM	1	8	12.50

Figure 7: Summarized resource utilization table

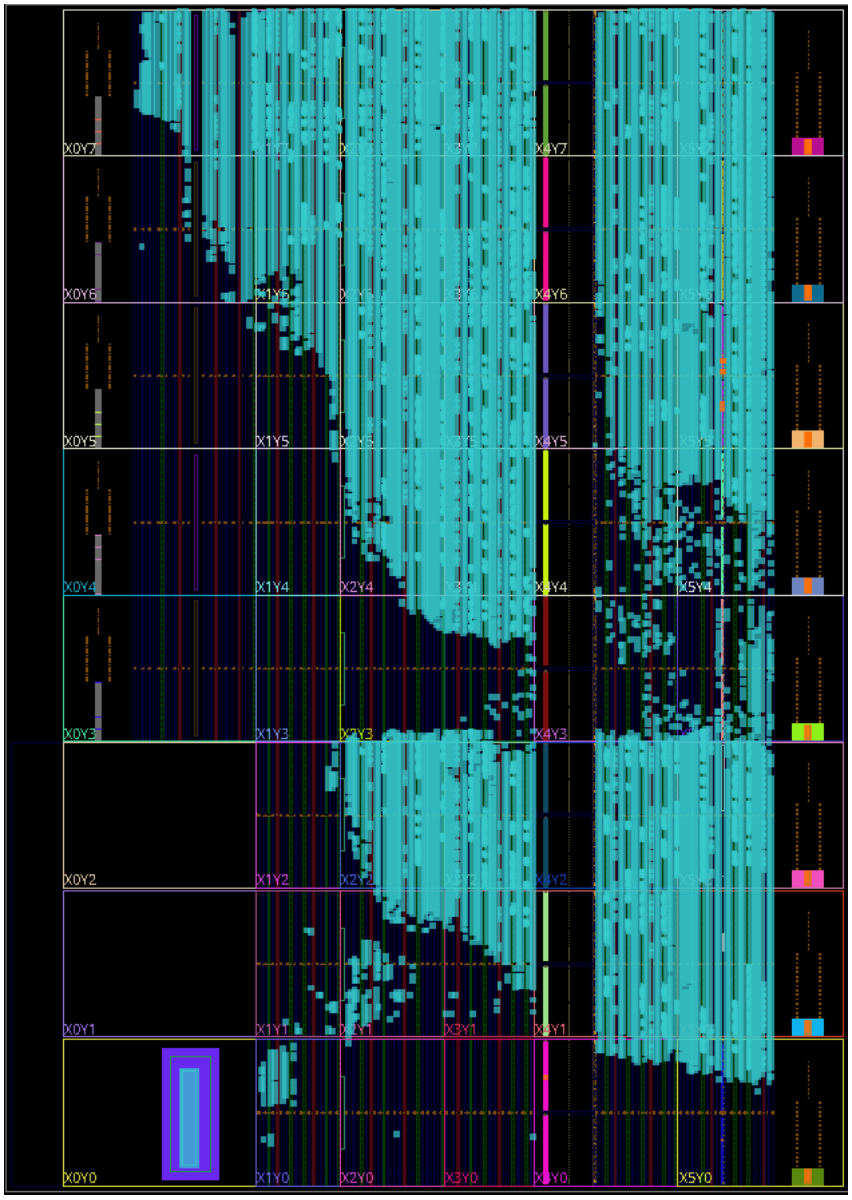

Figure 8: The fully routed design

2.5 The Enclosure

As for the hardware itself, we used the 19in 2U high rackmount Qufast Antelope housing developed by the Engineering Unit of Quantum Center at ETHZ. The case offers enough space for the boards and the power supply brick along with enough mounting holes for every DAC and ADC of the RFSoC. The enclosure still has some empty space, so it is possible to expand the system using additional FMC cards, analog filters, or RF mixers. The enclosure itself and a simplified power distribution schematic can be seen in Figures 9, 10, and 11.

Figure 9: Outside view of fully assembled enclosure

Figure 10: Inside view of the fully assembled enclosure

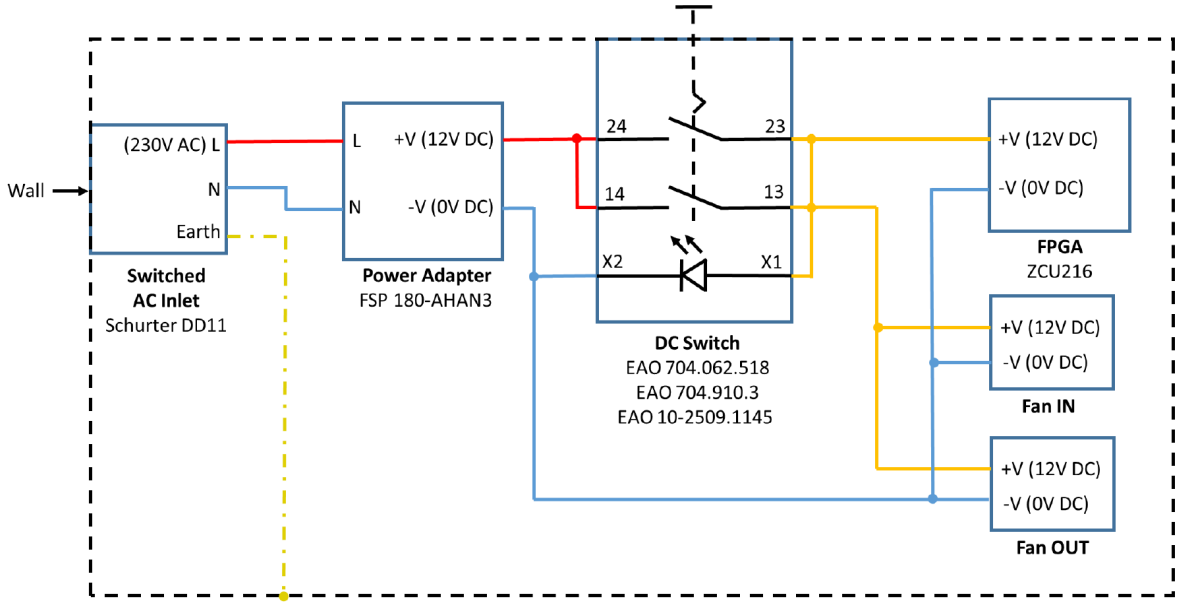

Figure 11: Power distribution schematic of the enclosure

3 Software

The current version of the software is quite minimal in terms of functionality and is intended to just demonstrate the capabilities of the hardware. It first initializes the CLK104, resets the PL using the procedure previously mentioned, initializes the *XRFdc* library, synchronizes the tiles using it, and configures 4 DACs in different tiles to generate the same frequency.

In our software implementation, we first reset the Mixed-Mode Clock Manager (MMCM), which is responsible for the clock inputs of other modules. Once the "locked" output of the MMCM changes to a logical 1, we conclude that the MMCM has been reset successfully and producing the clock signals. It is important that we reset MMCM before we reset the programmable logic, since MMCM is responsible for the clock signal of other modules and other modules have synchronous reset inputs.

The source file *clk104.c* contains certain functions that can be used to control the CLK104 board. Normally, the *XRFclk* library from Xilinx is enough to achieve this but we experienced that it wasn't using the GPIO to control the SPI read MUX, breaking register readouts from the chips on the board. To mitigate this, we wrote our minimal library that initializes the GPIO along with the *XRFclk* library and uses it control to the MUX before issuing the *XRFclk* readout command.

The XRFclk library comes with numerous configurations for the LMk04828 and the LMX2594 chips. During the first stages of the project, we used one of the supplied configurations but then had to modify it to be able to use an external reference signal. Since this only required changing the value of a single register, it was possible to modify one of the existing configurations by just referring to the datasheet of the LMK04828 and changing a single value. After having verified the functionality of the DDS IP we wanted to implement MTS and needed a specific combination of reference frequencies that isn't produced by any of the preexisting configurations. We used Texas Instruments' TICSPRO software to modify one of the configurations that is distributed with the RFSoC Data Converter Evalution Tool to generate a new configuration that fulfills our requirements.

The file *dac.c* on the other hand is responsible for setting up the *XRFdc* library that is used to read the status of the RF Data Converter IP and setting up Multi-Tile Synchronization (MTS). It contains two functions. The first one initializes the Metal API that's required by *XRFdc*, then initializes *XRFdc*, and sets up MTS. The other function prints out the statuses of all enabled DACs. The last relevant source file *dds.c* contains functions to control the DDS IP. The first two functions are used to control the frequency, phase, and amplitude of the desired DDS module. The third one is used to send trigger signals. Writes to the DDS control registers are achieved by writing data to the DDS IP's base address. An illustration of the structure of the software and the address map of the system can be seen in Figures 12 and 13 respectively.

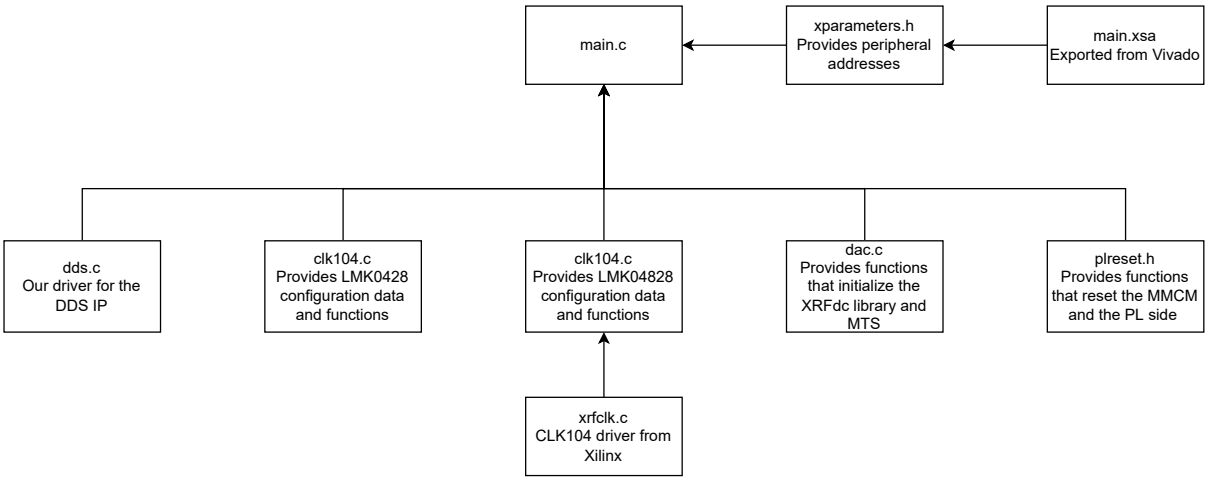

Figure 12: Software block diagram

Figure 13: The address map of the system

4 Measurements & Characterization

As previously mentioned, the noise characteristics of signal generators are quite important for precise control of trapped ion experiments. After having finished the implementation of the DDS system and the software, we performed numerous measurements at 4 different output frequencies (100 MHz, 250 MHz, 500 MHz, 1 GHz) using a spectrum analyzer to be able to characterize the behavior of the system in different use cases. We used our measurements to calculate the SNR and SFDR of the DACs and investigate the crosstalk between DACs. The 10 MHz reference output of a TPI-1002-A signal generator was used as the reference clock of the CLK104 board since our goal was to investigate how the RFSoC would behave in a realistic use case. In addition, we verified the synchronization of the signals from multiple channels.

Our measurements were done using a Keysight FieldFox N9913B Handheld Microwave Analyzer. While doing the measurements, we noticed that the measured noise floor could consistently be lowered by reducing the Resolution Bandwidth (RBW) which indicated that we were measuring the noise floor of the spectrum analyzer itself. This was the case even at very low values such as 1KHz which would make measurements impractically slow which is why we chose a relatively low RBW that . Therefore, we suspect our measurements were limited by the noise floor of our spectrum analyzer.

The measurements were done at 3 different frequencies, 260 MHz, 520 MHz, and 1 GHz using the 10 MHz-1 GHz RF Baluns on the XM655 breakout board as presented in figures 14, 15, and 16. For each frequency, we did one measurement between 0-3GHz with an RBW of 30KHz and another measurement in a $\pm 3\%$ interval centered around the target frequency with an RBW of 10KHz.

4.1 Single Tone Measurements

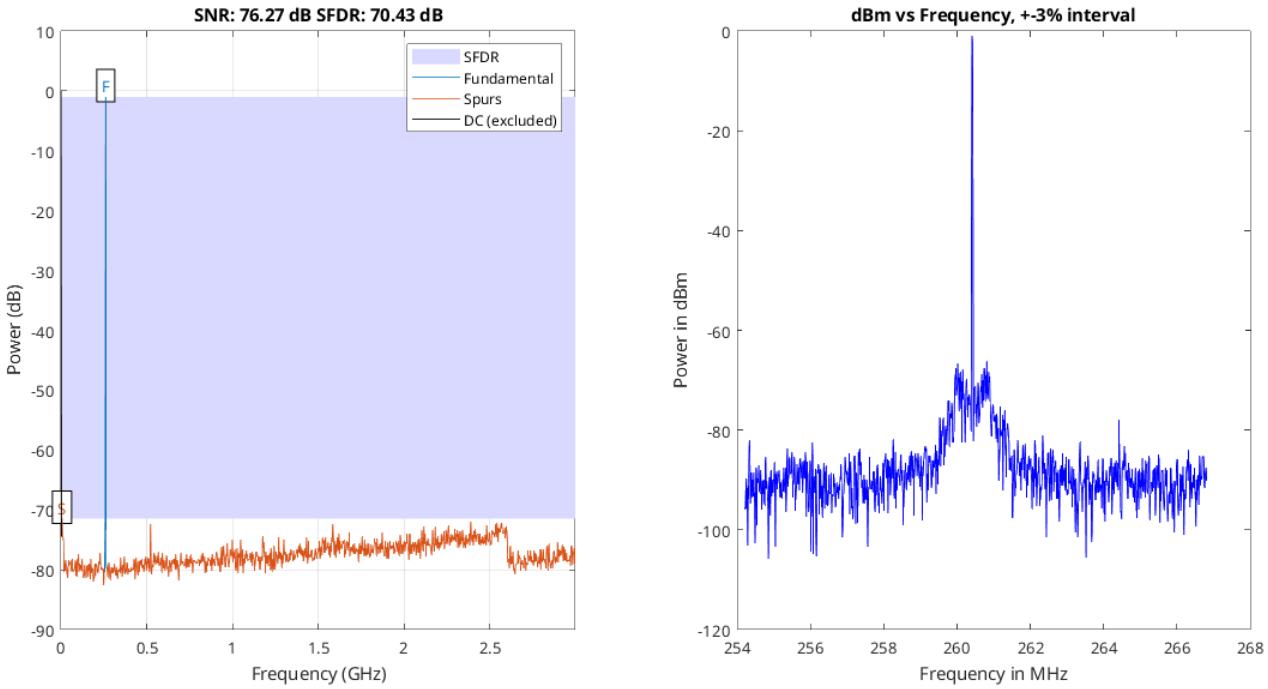

Figure 14: Measurement results for the 260 MHz test signal

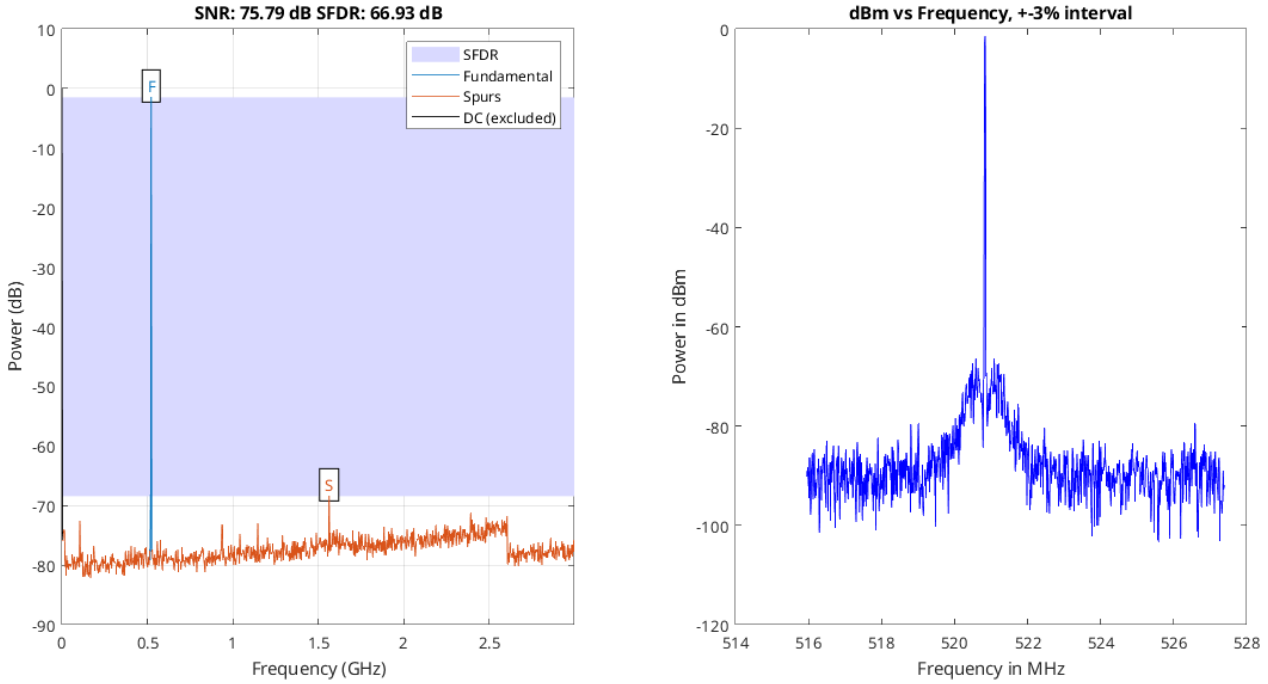

Figure 15: Measurement results for the 520 MHz test signal

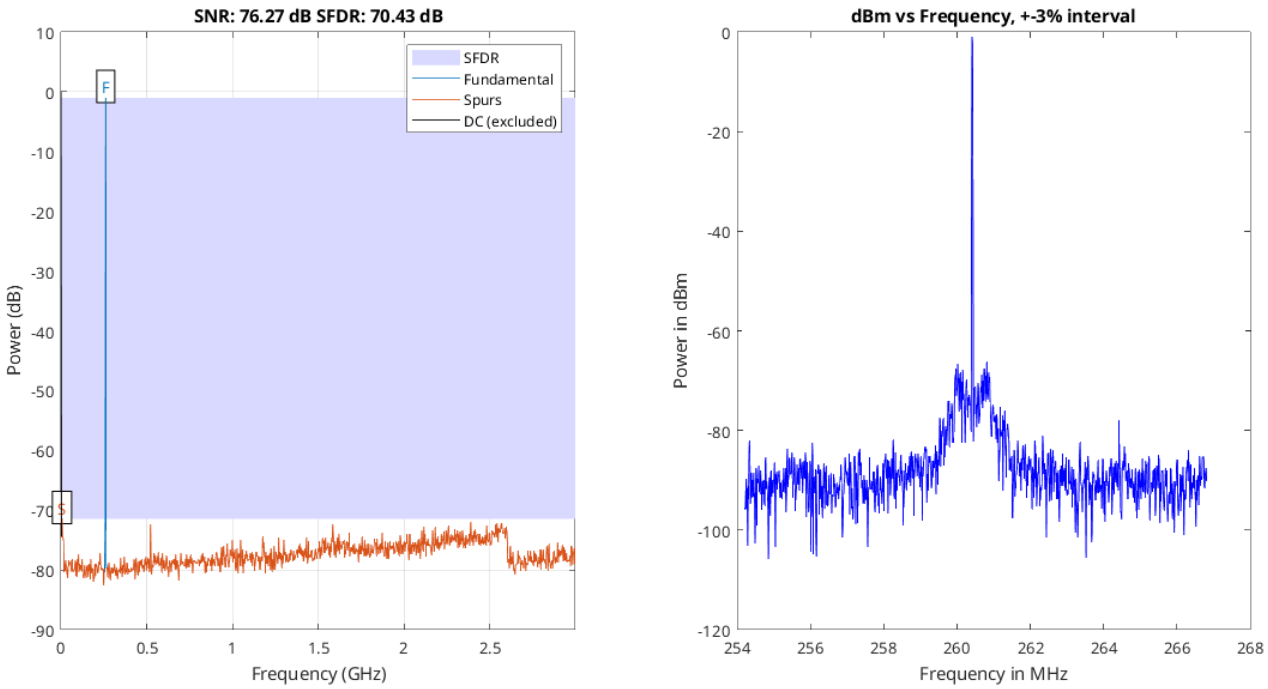

Figure 16: Measurement results for the 1 GHz test signal

It can be observed that the SNR is slightly inversely proportional to the frequency of the test signal whereas the inverse proportionality is much more significant for the SFDR. Despite this, the SNR doesn't fall below 75 dB and the lowest SFDR if 54.33dB. Even though the measurements were limited by the measurement equipment, it can be seen that the RFSoC offers very impressive noise characteristics.

4.2 Comparison With Xilinx's Figures

Xilinx provides their own performance analysis in the RFSoC's datasheet[3], the table belonging to the part used in our project can be seen in Figure 17. Although the datasheet doesn't contain any SNR measurements, it includes SFDR measurements which are a logical point of reference for our own measurements. Our design uses the 20 mA low noise mode so the first column is of relevance to us. Most of the measurements in the datasheet are done at wireless communication frequencies so a direct comparison isn't possible. Nevertheless, the closest frequency to our measurements on the table is 240 MHz and the SFDR given for this frequency is 71-78 dBc whereas we measured an SFDR of 70.43 dBc in our 260 MHz measurement, which is almost in the range given in the datasheet.

It can also be seen in our and Xilinx's measurements that the SFDR is inversely proportional to the frequency of the test signal. With this in mind, we can qualitatively compare the two sets of measurements. For instance, we measured an SFDR of 66.93 dBc at 520 MHz which is comparable to the SFDR given for 5.9 GHz in the datasheet, 66.9 dBc. Comparing our 1 GHz measurement with Xilinx's 1.9 GHz measurement we again observe our measurement of 54.33 dbC is much lower than Xilinx's 74.5 dBc.

To sum up the comparison, the datasheet contains higher SFDR than our measurements, especially at frequencies larger than 260 MHz. This could be caused by our measurement methodology or the length of the DDS cores' lookup tables although the latter is very unlikely to have caused larger spurs. Xilinx doesn't state their measurement hardware in the datasheet but it is clearly not the ADCs on RFSoC, as they can measure up to 2.5 GSPS, which would not allow the measurement of 4.9 or 5.9 GHz signals presented in the datasheet. It is possible that Xilinx used better measurement hardware than ours. Our lower SFDR values might have been caused by parasitic effects caused by the RF Balun or cables connected to it but since this accurately represents the typical use case of this device in an experiment. We still are of the opinion that our measurements are valid considering that we got similar dBc results for the 240 MHz signal.

Table 139: **RF-DAC Mode Performance Benchmark for ZU4xDR Devices** *(cont'd)*

Symbol	Parameter	Comments/Conditions[1, 2]	20 mA Low Noise Mode			20 mA High Linearity Mode			Units
			Min	Typ[3]	Max	Min	Typ[3]	Max	
SFDR[4]	Spurious free dynamic range excluding second- and third-order harmonic distortion and Fs/2	F_{OUT} = 240 MHz CW at 0 dBFS	71.0	78.0	–	78.3	80.4	–	dBc
		F_{OUT} = 1.9 GHz CW at 0 dBFS	–	74.5	–	–	79.8	–	dBc
		F_{OUT} = 2.4 GHz CW at 0 dBFS	–	73.6	–	–	78.7	–	dBc
		F_{OUT} = 3.5 GHz CW at 0 dBFS, F_S = 9 GS/s (E/I speed grade)	65.1	71.8	–	70.5	77.7	–	dBc
		F_{OUT} = 3.5 GHz CW at 0 dBFS, F_S = 9 GS/s (M speed grade)	64.9	71.8	–	69.8	77.7	–	dBc
		F_{OUT} = 4.9 GHz CW at 0 dBFS, F_S = 8.5 GS/s	61.2	68.0	–	65.7	73.8	–	dBc
		F_{OUT} = 5.9 GHz CW at 0 dBFS	–	66.9	–	–	72.3	–	dBc
		F_{OUT} = 240 MHz CW at –10 dBFS	–	73.5	–	–	79.4	–	dBc
		F_{OUT} = 1.9 GHz CW at –10 dBFS	–	70.8	–	–	78.1	–	dBc
		F_{OUT} = 2.4 GHz CW at –10 dBFS	–	70.0	–	–	77.7	–	dBc
		F_{OUT} = 3.5 GHz CW at –10 dBFS, F_S = 9 GS/s (E/I speed grade)	60.9	66.8	–	71.6	75.9	–	dBc
		F_{OUT} = 3.5 GHz CW at –10 dBFS, F_S = 9 GS/s (M speed grade)	60.6	66.8	–	67.0	75.9	–	dBc
		F_{OUT} = 4.9 GHz CW at –10 dBFS, F_S = 8.5 GS/s	57.3	63.8	–	66.2	70.4	–	dBc
		F_{OUT} = 5.9 GHz CW at –10 dBFS	–	62.7	–	–	69.0	–	dBc

Figure 17: SFDR measurements given in the datasheet

4.3 Crosstalk Measurements

To measure crosstalk between DACs within same and different tiles, we connected the spectrum analyzer to the output of one DAC and made it generate 0 V DC, and made another DAC in the same tile or one in a different tile to generate a test frequency. We performed 4 measurements 18 at test frequencies of 250 MHz and 1 GHz using an RBW of 30 KHz between 0-3 GHz.

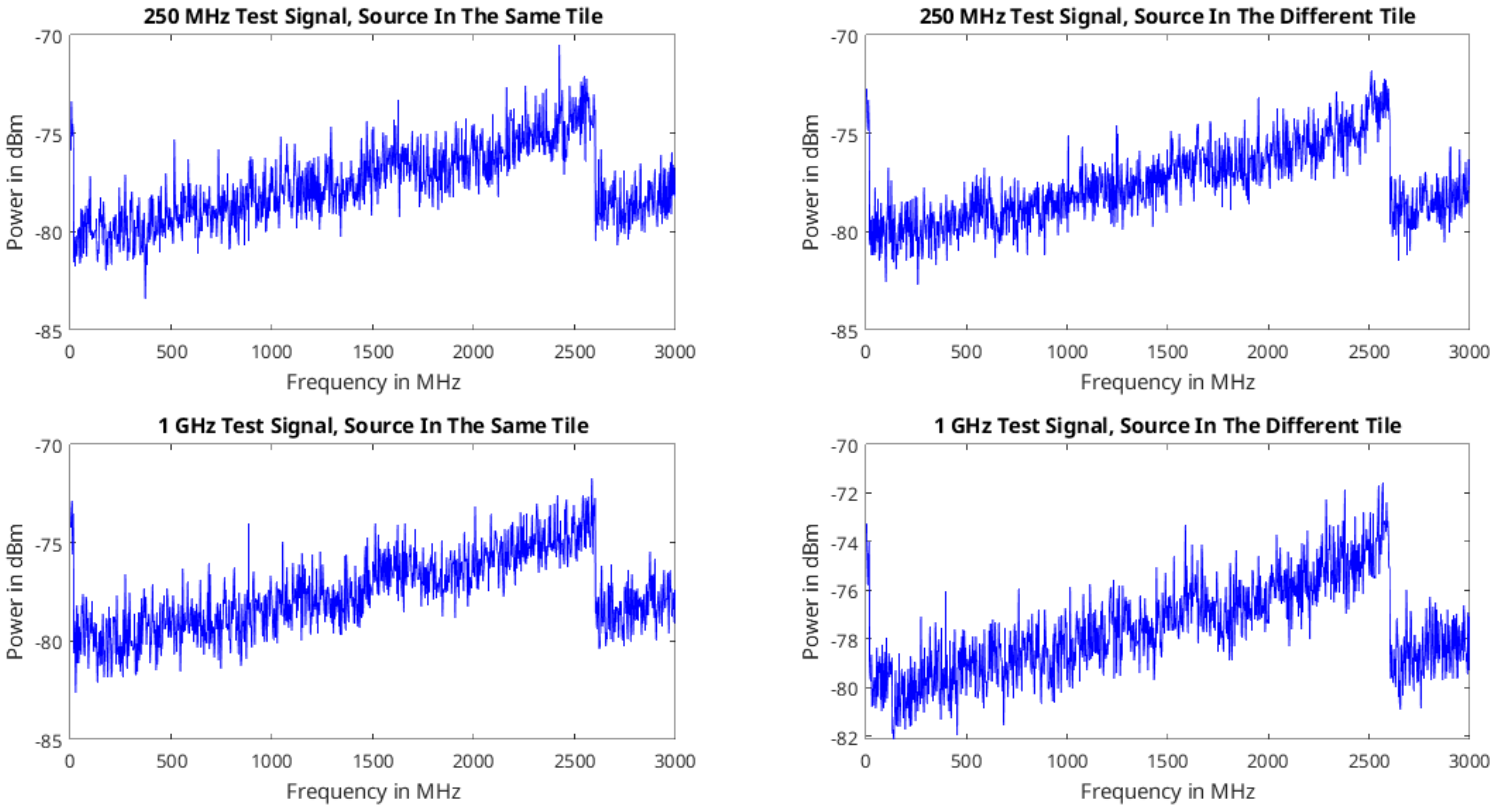

Figure 18: Crosstalk measurements

Apart from some very minor harmonics that can be seen when a 250 MHz test signal is being generated by a DAC in the same tile at around 500 MHz and 2.5 GHz, the noise generated by other DACs is negligible or too low to be measured by our equipment. Unfortunately we were not able to determine the cause of the drop-off at around 2.5 GHz, it is present in the single tone measurements as well and we've observed it even the spectrum analyzer was not connected to any source. Therefore, it's likely an issue with our spectrum analyzer.

4.4 Synchronization Results

As previously mentioned, synchronization is essential for the intended use case of our signal generator. To verify that Multi-Tile Synchronization (MTS) was working, we made 4 DACs in different Tiles generate signals with the same frequency and phase while measuring the generated signals using an oscilloscope. Time domain alignment of the signals can be seem in Figure 19. Without MTS, different tiles do not have a constant phase relation and we've observed that the phase between signals generated by DACs in different tiles changes every time the device is power cycled even though the DDS cores were instructed to generate signals with the same phase. Therefore, MTS has to be implemented for every use case that requires more than 4 DACs.

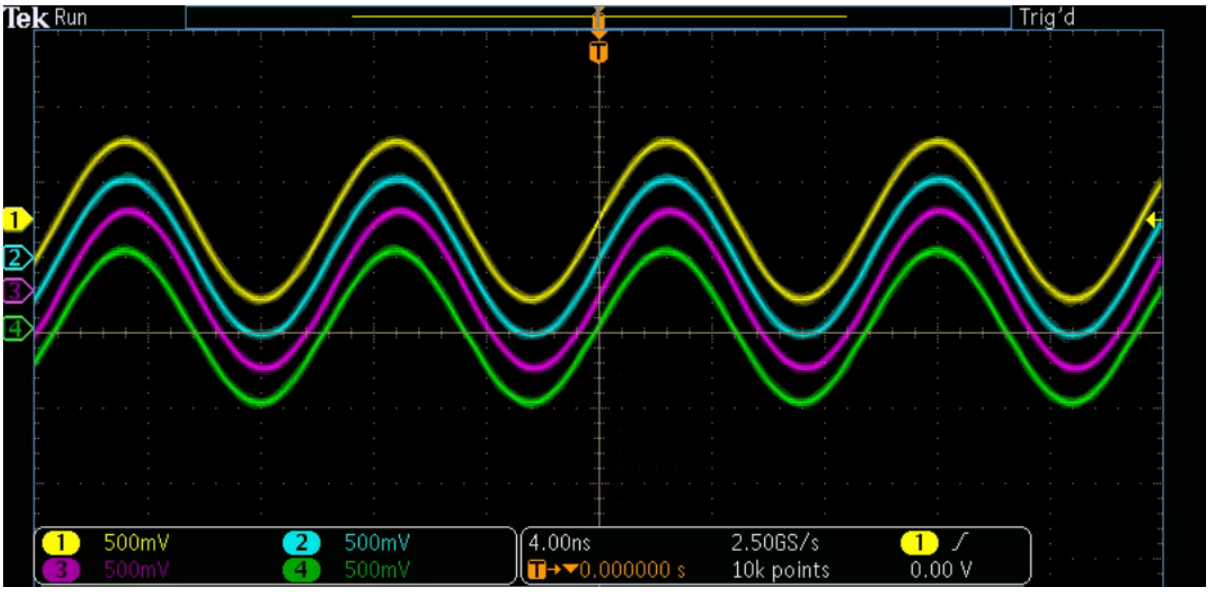

Figure 19: Time domain measurement of the synchronization test signals

5 Conclusion

5.1 Future Outlook

One possible addition to the current design would be to improve the software side. In the current system, the ARM processor controls DDS modules with the given settings embedded in the code, but the user should be able to change them from a PC during run time. The RFSoC contains an Ethernet interface that can easily be used to implement a network control mechanism. Since doing this on bare metal would be challenging, porting the existing code to Linux would be the logical next step before attempting to implement PC/network control.

The ADCs are another important feature of the RFSoC but they were beyond the scope of this project. In an actual experiment some form of measurement will be necessary so the current system can be improved by utilizing the ADCs for signal acquisition. In addition, both the DACs and the ADCs require the usage of baluns to generate/acquire single-ended signals. However, the XM655 board that is supplied with the evaluation kit only contains 16 baluns and to make matters worse, most of those baluns are made for frequencies higher than 1 GHz as they are intended for wireless communication applications. Since the XM655 only has 4 10 MHz - 1 GHz baluns, any trapped ion experiment requiring the usage of a total of more than 4 DACs and/or ADCs will likely require the creation of a custom, purpose built PCB that contains a more appropriate selection of baluns.

Lastly, it is possible to use MTS to synchronize different chips as mentioned previously. This would require the distribution of the same reference clock and SYSREF signals to all devices that need to be synchronized. This functionality wasn't verified during this project as we had one evaluation kit at our disposal but verifying multi-board synchronization would be useful as it would demonstrate the scalability of the RFSoC.

5.2 Conclusion

A 16 Channel DDS system is implemented on high-end RFSoC FPGA, and the signal characterization is performed. The ZCU216 demonstrated quite impressive noise characteristics in our measurements and is in general a versatile platform for controlling quantum physics experiments. The RFSoC includes high number of on-chip AD/DA converters and makes cost-effective system implementation possible. The FPGA on the ZCU216 board is one of the largest on the market. In addition, the evaluation board eliminates the need to design an FPGA board suitable for multi-channel high-frequency signaling. To conclude, this platform has significant potential to be used in experimental quantum physics experiments where high sampling rate DACs/ADCs and high-speed signal processing are needed.

References

[1] Analog Devices. “Fundamentals of Direct Digital Synthesis (DDS)”. In: (2008), pp. 1–9. URL: https://www.analog.com/media/en/training-seminars/tutorials/MT-085.pdf.

[2] Vlad Negnevitsky. “Feedback-stabilised quantum states in a mixed-species ion system”. en. Doctoral Thesis. Zurich: ETH Zurich, 2018-09-27. DOI: 10.3929/ethz-b-000295923.

[3] Xilinx. “Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926)”. In: (2022), pp. 147–151. URL: https://docs.xilinx.com/r/en-US/ds926-zynq-ultrascale-plus-rfsoc.

[4] Xilinx. “Zynq UltraScale+ RFSoC RF Data Converter v2.4 Gen 1/2/3 LogiCORE IP Product Guide”. In: (2020), pp. 174–176. URL: https://docs.xilinx.com/v/u/2.4-English/pg269-rf-data-converter.