FPGA CONTROLLED RF PULSE GENERATOR

FOR TEACHING MRI

A Thesis

by

MARIAM NIDA USMANI

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2021

Major Subject: Electrical Engineering

ABSTRACT

Currently, many researchers in MRI are focused on creating low-cost MR setups using off-the-shelf components that are comparable in performance to existing MR systems. A problem these setups face is phase instability due to hardware components not synchronized perfectly. To overcome this, systems with pulse generation and echo digitization on a single hardware platform are gaining popularity.

With FPGAs gaining traction in recent years, it is unsurprising to find them being incorporated in low-cost MR setups today. This is because of their suitability in highly precise applications along with occupying smaller chip areas, consuming lesser power and keeping equipment cost lower than their analog counterparts. Given the current scenario of virtual and hybrid classes, FPGAs also make it possible to get hands-on experience in building working MR setups using off-the-shelf components at home, which promotes learning while being socially distant. Hardware platforms such as the Red Pitaya are well-suited for this purpose, which has an Artix-7 FPGA coupled with a dual-core ARM Cortex A9 processor, and DACs and ADCs all housed under the same chassis.

This thesis aims to build and test an RF pulse generator on an FPGA using a Red Pitaya board. The pulse generator is integrated into a tabletop MR setup and its phase stability determined using a Pentek high-speed digitizer. The entire process has been documented in a manual attached to this document along with all source codes used. Finally, a discussion has been initiated regarding the inclusion of a working digitizer within Pitaya itself, alongside the pulse generator.

DEDICATION

I dedicate this thesis to my parents, my sister, and my grandparents for constantly encouraging me to pursue my interests and strive for excellence.

I thank my mother for being there for me during the highs and lows of life and motivate me towards my goals, and my father for helping me reason out problems and refine my thinking as an engineer.

To my sister – thank you for all your love and support and adding to my happiness.

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Wright, and my committee members, Dr. Gratz, Dr. Hur and Dr. Ji for their invaluable guidance, support and learning opportunities throughout the course of this research.

Thanks also goes out to Dr. Hur and ETID for teaching opportunities provided during my time as a GAT for ESET 369. I offer my sincere appreciation to all friends and colleagues at MRSL and the department faculty and staff of ECEN and ETID for making my time at Texas A&M University a great experience.

Finally, my heartfelt thanks to my family for their encouragement, in particular my parents and sister for their patience and love.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Dr. Steven M. Wright, Chair, Dr. Paul V. Gratz, and Dr. Jim Ji of the Department of Electrical and Computer Engineering and Dr. Byul Hur of the Department of Engineering Technology and Industrial Distribution.

The SD image for STEMLab 125-14 used in this research (latest stable) has been created by the Red Pitaya team; it is documented at https://redpitaya.readthedocs.io/en/latest/quickStart/SDcard/SDcard.html

All Vivado projects were created by adapting and modifying Anton’s project directory and code files from his GitHub page: https://github.com/apotocnik/redpitaya_guide

The server and client programs were constructed by modifying codes by Marco Marchini on his gist page: https://gist.github.com/marcom04/22860f1168330605cac3c448982b0393

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was partially supported by a summer 2020 research scholarship from the Department of Electrical and Computer Engineering, Texas A&M University.

NOMENCLATURE

FPGA	Field Programmable Gate Array
CLB	Complex Logic Block
PLL	Phase Locked Loop
HDL	Hardware Description Language
FF	Flip Flop
MUX	Multiplexer
MRI	Magnetic Resonance Imaging
SNR	Signal to Noise Ratio
STEM	Science, Technology, Engineering and Management
FFC	Fast Fourier Tracking
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
DAC	Digital to Analog Convertor
RF	Radio Frequency
SoC	System on Chip
GPIO	General Purpose Input/Output
NMR	Nuclear Magnetic Resonance
MR	Magnetic Resonance
PSG	Pulse Sequence Generator
PC	Personal Computer
TAMU	Texas A&M University
MRSL	Magnetic Resonance Systems Lab
ADC	Analog to Digital Convertor

AXI	Advanced eXtensible Interface
AMBA	Advanced Microcontroller Bus Architecture
DSO	Digital Storage Oscilloscope
MIT	Massachusetts Institute of Technology
OCRA	Open-source Console for Real-time Acquisition
DDS	Direct Digital Synthesizer
BRAM	Block Random Access Memory
SE	Spin Echo
FID	Free Induction Decay
IP	Intellectual Property
μs	Microsecond
ns	Nanosecond
TXG	Transmission Gate
RXG	Receive Gate
RTL	Register Transfer Level
rad	radian
Hz	Hertz
MHz	Megahertz
T	Tesla
LUT	Lookup Table
PA	Power Amplifier
LPF	Lowpass Filter
Vpp	Peak-to-peak Voltage

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Advancements in reconfigurable logic devices like Field Programmable Gate Arrays (FPGAs) in recent times has made far-ranging changes to existing devices. It has become essential to modernize existing and/or obsolete equipment with reconfigurable FPGAs [1]. This is expected, given their low cost and size coupled with high reliability and accuracy. Moreover, their ability to replicate complex analog circuit designs continues to improve dramatically.

This makes these devices highly suited in prototyping and developing digital logic to address a wide spectrum of medical imaging modalities as well as medical devices [2] such as Magnetic Resonance Imaging (MRI), which traditionally uses complex analog circuits. From an educational viewpoint, it is equally important to make them accessible and easy to understand for students in Science, Technology, Engineering and Management (STEM) programs to promote research and development.

This project aims to build and test a simple MRI interface along with the underlying digital logic for a pulse sequence generator implemented on Red Pitaya, an inexpensive FPGA development board. The resulting circuit and interface serve as a base for future MR experiments and logic/processing upgrades. To demonstrate its working, the fully configured Pitaya board was integrated into an RF followed by an MR setup, one at a time, and its phase stability quantified. Attached to this thesis are four sets of codes – the server and client programs, three RTL scripts, one testbench/simulation script and two MATLAB programs. The first three sets of code pertain to setting up the pulse generator and verifying its functionality. The MATLAB codes are intended for processing data digitized in the MR setup. Finally, a manual walking through the design and testing process of a simple pulse generator has been attached to this document.

2. BACKGROUND

2.1 FPGAs – a brief history

The idea of programmable logic came into being with the introduction of XC157 by Motorola in 1969 [3]. This is the first mask-programmed gate array to ever exist, with 12 intra-connected gates and uncommitted inputs and outputs that could be interconnected according to the engineer's custom design. Over the years, this design was improved upon for speed, power consumption and throughput leading to smaller devices that could be reconfigured just like EEPROM (electrically erasable Programmable Read Only Memory). These were called Field Programmable Gate Arrays (FPGAs), the first of which was the XC2064 chip invented by Ross Freeman and Bernard Vonderschmitt, co-founders of Xilinx Inc[4]. It was produced using a 2 µm fabrication process, housed 800 gates and sold for $55 in 1985.

With fabrication techniques reaching the nanometer scale, it is now possible to etch millions of gates on to the same chip space. This has allowed FPGAs to reach a level of sophistication, making it viable to build complex circuits consisting of phase locked loops (PLLs) and communication interfaces[5].

The current trend in reconfigurable logic is to integrate FPGAs with processor cores to form a System on Chip (SoC). This pairing creates a powerful computing platform with greater system reliability, low power needs and low latency, all on the same chip dimensions or smaller.

2.2 Digital circuits on an FPGA

The modern FPGA architecture comprises of arrays of I/O lines, configurable logic blocks (CLBs) and routing channels. A very basic CLB will have one lookup table (LUT), one flip flop (FF) and one multiplexer (MUX). By manipulating the I/O lines and routing channels CLBs can be set to specific configuration – this is the basis of all logic circuits today.

To allow for this manipulation a set of languages called hardware description languages (HDLs) have been created, the most popular being VHDL and Verilog. The idea behind HDLs is to come up with a logic design and codify it. This codified logic is then converted into a bitstream/gateware file which is flashed on to the FPGA fabric. It can be thought of as 'activating' the FPGA with your logic design, effectively creating an electrically erasable digital circuit.

This conversion is done with the help of software such as Xilinx Vivado or Intel Quartus Prime. To be clear, they offer features that go well beyond HDL code editing and bitstream/gateware generation. These include logic simulators, which test a logic design before generating its corresponding bitstream. Also, both have a large library of IP (Intellectual Property) cores, which are basically reusable pieces of codified logic. This encourages FPGA designers to break down their logic design into smaller, manageable portions that can be created and tested independently, and interfaced with IP cores to build the desired digital circuit.

2.3 Advancements in MRI

MRI is a non-invasive, radiation-free imaging modality that is carried out in the RF range of the electromagnetic spectrum, i.e., 2-200 MHz. This makes it quite appealing to use by doctors as well as researchers. Because of this, MRI has been a witness to several technological leaps and bounds ever since it was introduced as a diagnostic tool in the 1980s.

Improvements in RF coil design, pulse sequences, magnets and image processing have all contributed to making MRIs a crucial part of patient studies. These include the adoption of phased array coils on the receive side [6], using superconducting magnets for better field homogeneity and higher field strengths [7], the addition of gradient pulses to image specific regions of the patient [8] and better algorithms for image reconstruction using multidimensional Fourier transforms [9].

One focus of most MR-related research continues to be improved SNR and system stability along with lower running/maintenance cost and patient scan time. To this end, MR systems continue to be miniaturized by replacing existing components such as magnets [10-15] with smaller ones demonstrating equal or better performance than the former. This enhances the portability of MRI; many portable MR scanners have been designed [16-24] and studied [25] or improved [26-27] in recent times. It is hoped that by simplifying the system workflow in this manner, MRI becomes more accessible and affordable to the public [28].

2.4 Phase Instability in MR Systems

Phase stability refers to the condition where an MR echo from each pulse sequence repetition possesses the same amplitude and phase information, given that the pulse sequence parameters, external magnetic field (B_0) and other environmental factors like temperature remain constant. As one can imagine, MR systems are prone to phase instability, making the generated echoes somewhat unpredictable in location, amplitude, and phase. It is crucial to have a system with predictable phase, especially when MR sequences with phase encoding gradients are used.

Several ways have been proposed to overcome this. One includes digitizing the second half of the last RF pulse (the RF180 in a Spin Echo sequence) and using the accumulated phase data to post-process the collected echo data. This has been shown to work by Ogier et al. at MRSL when investigating improvements in low-field MRI at 0.06 T [29].

A method called Fast-Field Tracking (FFC) has been proposed to correct for phase instabilities and ghosting. This involves varying B_0 field in fixed steps throughout the duration of an MR scan. Because of this method, reconstructed MR images can be corrected for phase and artifacts anywhere between a few seconds to a couple of minutes, depending on the image size [30].

Yet another technique is to synchronize the digitizer with the transmit such that echo data is collected precisely after all RF pulses in a sequence have been transmitted. To do this, trigger pulses or gating signals may be used for activating and deactivating the digitizer. The catch to this method is that the pulse generator and digitizer both possess operating frequencies much higher than the frequencies of transmitted and received RF pulses and exist on the same device. The current MR tabletop setup for ECEN 463/763 (Magnetic Resonance Engineering) implements this using an NI chassis with a 14-bit PXI 5412 DAC card at 100 MSa/s for RF output, a 14-bit PXI 5122 ADC card at 100 MHz to digitize the receive, and an 8-channel PXI 6733 analog output card for slow-speed digital I/O and generating gradients [39]. This chassis is combined with RF frontend components outlined in the paper and gives a phase stable and reliable MR system.

2.5 Pulse generation and Programmable Logic

Modern MRI experiments are time-sensitive and extremely precise. Depending on the amount of data acquired, they can also place high demands on processing power as well. Due to the specialized and complex nature of circuitry involved, it also becomes a costly endeavor.

Further, pulse sequences, which are comprised of RF and gradient pulses, have been traditionally created with closed source programmers in mind, making the concept of MRI appear esoteric. Efforts in recent times seek to make pulse sequence programming open source – a notable example being Pulseq [31]. It is a MATLAB toolbox/translator that generates MR pulses on several legacy MR scanners – Siemens, Bruker and GE – all with the same lines of code. Pulseq was successfully translated for use with a Varian scanner in 2018 [32]; to this day, it is used as a pulse programmer for ECEN 411 – Introduction to MRI/MRS in Magnetic Resonance Systems Lab, Texas A&M University (TAMU).

A renewed interest in MRI is being witnessed not only due to open-source pulse programmers, but also with the rise of programmable logic, namely FPGAs. Their high reliability, high processing power and low cost has appealed to researchers as well as students especially in NMR at low fields. They have not only allowed for creating and miniaturizing MRI setups to tabletop scanners with off-the-shelf components [33] but have also proven to be capable of high-volume data processing [34] and image reconstruction using SENSE, a parallel MRI algorithm [35].

Coming back to MRI, FPGAs also allow for open-source pulse programmers to be developed [36]. This is a big achievement which adds to the appeal of FPGAs – they not only bring down machinery cost and space requirements, but also give beginners a chance to understand, build and appreciate the practical working of MRI.

One such initiative is OPENCORE NMR [37], a project that continues to be actively updated at the time of writing this thesis. It provides a console, and data processing software as well as VHDL modules that can be assembled as a logic design to run on an Altera/Intel Cyclone III (65 nm technology node) or a Cyclone V (28 nm technology node) FPGA. The FPGA chip, DDSes, DACs and ADCs are all hand-soldered and assembled to form the spectrometer on a single board. While this is a good approach, it is mainly aimed towards researchers well-versed with MR systems, RF modulation, VHDL modules and DDS. This may present difficulties for students with no prior experience in MR development or FPGA programming to follow and understand. For this reason, the author chose a development board with the above components already assembled on it – the Red Pitaya STEMLab 125-14.

2.6 Red Pitaya – an RF/digital workbench in a handheld chassis

MRI is highly specific when it comes to timing. RF transmission and acquisition must be done with extreme precision; any deviation from the preset timing parameters will lead to errors and/or failed experiments. Hence, coupling this with the Red Pitaya development board is a good idea. It is housed inside an aluminum chassis of dimensions 11×7×2.5 cm, which makes it the size of a wallet. Several papers have been published demonstrating the capabilities of this handheld device for RF applications, especially as a Nuclear Magnetic Resonance (NMR) spectrometer [40][38]. With a clock frequency of 125 MHz, it is quite reliable when it comes to setting timed events, which is crucial in MRI. This has been demonstrated to work in the OCRA project conducted at MIT [38], which created a series of server and client codes in C and Python to control the existing bitstream-programmed FPGA fabric. This system demonstrated better phase stability than that of the existing Medusa console, which is promising.

What makes the proposed system in this thesis different is the focus – OCRA developed an interface (set of server and client programs) to control the existing FPGA logic (underlying circuitry). The latter has remained more of less fixed, with limited guidance on how to verify its functionality before running its corresponding bitstream on the Red Pitaya.

Here we leveraged the FPGA portion of Red Pitaya's STEMLab 125-14 board for timed MR pulse events, integrating them with a much simpler version of a server-client pair, and provided a manual along with well-commented source codes for future development. In other words, this project allows for flexibility in both digital logic (underlying circuitry) as well as the interface (server and client).

The focus of this research work is to build FPGA logic from the ground up, add an interface to it, and integrate it to an existing MR system consisting of a fully assembled RF frontend. In this study, a Pentek digitizer was used to quantify phase stability of the pulse generator and determine its efficacy for future MR experiments. Finally, the option of building a digitizer within the Pitaya has been explored as well and is discussed under 'Future Work'.

All codes utilized in the study are kept simple enough to understand and deploy, and are attached as appendices A through D. A manual has been created for students to build their own pulse generator – it is attached to this thesis as Appendix E. Finally, Appendix F correlates the pulse generators created with the MR phenomenon.

3. METHODOLOGY

3.1 Goals

This thesis work utilizes the Red Pitaya STEMLab 125-14, a digital hardware platform costing under \$400 with a clock frequency of 125 MHz. It has two 14-bit, 125 MHz digital-to-analog convertors (DACs) with a memory depth of 16k samples. It also contains two 125 MSa/s RF output channels and extended general purpose input-output (GPIO) lines. These components are housed within an aluminum chassis of dimensions 11×7×2.5 cm, which is basically the size of a wallet. Of particular interest is the Xilinx Zynq 7010 SoC it houses, which has an Artix-7 programmable logic (28 nm technology node) integrated with a dual-core ARM Cortex A9 processor.

The focus of this research is to upgrade the existing MRI teaching setup for ECEN 463/763 (Magnetic Resonance Engineering) at TAMU that comprises of an NI chassis with a 14-bit PXI 5412 DAC card at 100 MSa/s for RF output, a 14-bit PXI 5122 ADC card at 100 MHz to digitize the receive, and an 8-channel PXI 6733 DAC card for slow-speed digital I/O and generating gradients [39]. The upgraded MRI setup is more compact and less expensive than the existing one.

While Pitaya is capable of digitizing data at 125 MS/s with analog input channels IN1 and IN2, it was decided to keep it strictly as a pulse generator and not use it as a digitizer as well. Digitizing entails the use of an ADC to sample incoming RF echoes. ADC-sampled data can be stored in a soft Block Random Access Memory (BRAM) on the FPGA and sent out to the server via a 32-bit AXI GPIO register. The issue arises when server attempts to read this data off the AXI GPIO's memory address. BRAM data updates every 8 ns in memory, which is too fast for the server to read and transmit to the client before the AXI GPIO register is overwritten with a fresh value. Creating a digitizer that works will require a better understanding of BRAM specifications and AMBA AXI4 interface protocol, which is beyond the scope of this research. Hence the choice of a standalone digitizer has been made in place of digitizing within Pitaya during acquisition.

Apart from pulse and GPIO timing and number of repetitions, the frequency of sine pulses, generated by a direct digital synthesizer (DDS), should be adjustable in the MR frequency range – from a few kHz up to a few hundred MHz. To accomplish this, a server program must be built that can run on top of the FPGA fabric within Pitaya and establish a socket connection with a client program running on another PC. Through this client, we intend to allow user control over RF pulse generation.

3.1.1 Hardware side

On the hardware side, our upgrade swaps out the DAC and ADC cards with a much smaller Red Pitaya board and a high-speed digitizer respectively, which simplifies and miniaturizes the MR setup. To achieve this, a pulse sequence generator (PSG) needs to be built from the ground up, tested, added to the FPGA fabric of Red Pitaya and integrated to an MR system. A PSG is a highly accurate computer that accepts timing inputs from a user and generates real-time outputs to drive MRI experiments, namely trigger signals and sine wave pulses. All the timing inputs correspond to spin echo (SE) pulse parameters; hence the RF and trigger outputs must resemble the transmit side and gating signals of an SE as closely as possible.

3.1.2 Software/interface side

On the software/interface side, VIs (programs running in LabVIEW) will be replaced by a command line interface (CLI) – facilitated by a Python client program on a PC and a C server program executable running on top of the FPGA fabric of Pitaya. Basically, the client will establish a socket connection with the server and transfer pulse parameters to it. The server will convert these parameters to appropriate values and send them over to the digital circuit via memory mapped I/O. In short, VIs shall be swapped out for a more stripped-down approach to device interfacing with a server-client pair of programs.

3.1.3 Testing and phase stability

As part of the testing process, the digital circuit's functionality must be verified using simulations running on a testbench code. Since Vivado 2020.1 is to be used in this project, its XSim tool is a good choice for testbenching and simulations. This is to be done before bitstream generation. After bitstream generation, analog and GPIO outputs generated by the device will be tested using various oscilloscopes.

Finally, the circuit and interface will be incorporated into two MR setups, their receive channel will be digitized for a certain number of repetitions, and raw data from both setups shall be processed in MATLAB. The goal is to extract the resulting echo from raw data, line them up and ensure the setups are phase stable.

All program codes pertaining to designing and testing the PSG, server and client, and data processing shall be attached as appendices to this document. A detailed manual explaining the steps needed to control pulses and GPIOs generated off the Pitaya board is also to be attached as an appendix to this document.

3.2 RF Pulse Parameters

The timed events are set by the following spin echo (SE) pulse parameters:

RF90 (in µs)	–	time used to tip magnetic spins by 90°
Off_time (in µs)	–	time between RF90 end and RF180 start when RF output is a flatline
RF180 (in µs)	–	time used to tip magnetic spins by 180°
TR (in µs)	–	time after which all pulse events repeat
Acquisition (in µs)	–	time after RF180 end up to when MR echo is expected to form

Repetitions	–	number of times the SE pulse sequence repeats
Frequency (in MHz)	–	frequency of RF pulses

Each RF pulse is directly tied to the NMR phenomenon, which is governed by the equations for Larmor frequency f_{Larmor} and tip angle α given below.

$$f_{Larmor} = \bar{\gamma} B_0 \tag{1}$$

$$\alpha = \gamma B_1' t \tag{2}$$

where $\bar{\gamma}$ = gyromagnetic ratio of sample in Hz T^{-1} H

γ = gyromagnetic ratio of sample in rad s^{-1} T^{-1}

B_1'= 0.5 × B field of the solenoid/RF coil

B_0 = B field of the magnet

Note that $\bar{\gamma}$ and γ indicate the same thing – gyromagnetic ratio. They differ only in units; γ is in Hz/T and $\bar{\gamma}$ is in (rad/s)/T. The former value is used to calculate Larmor frequency in Hz, while the latter formula is utilized in tip angle computation (in radians).

Basically, f_{Larmor} is the frequency at which magnetic spins will precess when placed inside the permanent magnet. These spins will precess in the longitudinal direction and the net magnetization will remain constant; for this reason, no current is induced in the solenoid and no signal is picked up.

To generate a signal (called a Free Induction Decay or FID) from the sample, we need to ‘tip’ the spins in the transverse plane, where the solenoid is located. In other words, the spins must be tipped by α = 90°. This is done using the RF90 pulse, which is basically a sine wave of duration corresponding to α_{90} and frequency f_{Larmor}.

Note that the 90° tipped spins will undergo rapid dephasing due to T_2* effects. This leads to signal loss, the spins recover back to longitudinal by T_1 recovery, and the amount of detected signal in the transverse/solenoid plane diminishes. To counteract this, a Spin Echo sequence will excite the sample with a second RF pulse – the RF180. This pulse possesses a pulse length corresponding to α_{180} (when the magnetization/spin is 'tipped' by 180°) and frequency f_{Larmor}.

The timed events are all tied to when the RF pulse and TXG/RXG signals are activated and deactivated, and for how long. Further, the frequency of the RF pulse sinusoid must be customizable – handy when performing frequency sweeps. Finally, we are dealing with a Spin Echo (SE) pulse sequence here – it comprises of two RF pulses generated after a certain amount of timed delay. Figure 1 explains this sequence in the most basic form, with the timed events labeled appropriately.

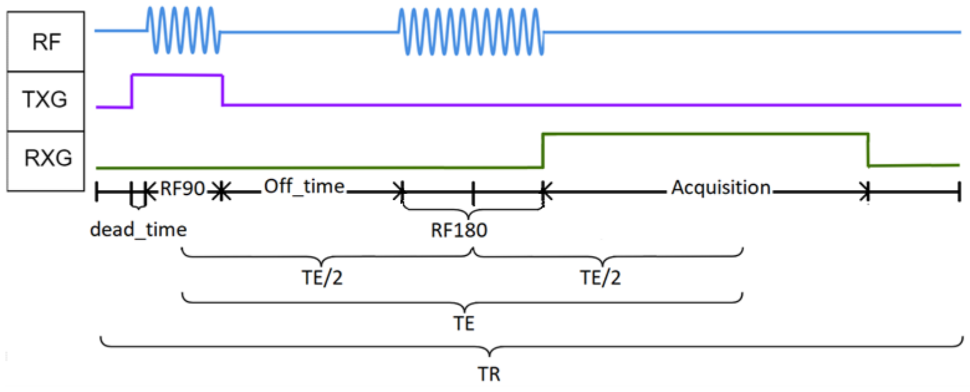


Figure 1: Timing Diagram for Pulse States - Spin Echo (SE)

TXG and RXG in the timing diagram correspond to GPIO lines programmed in Vivado. These are pins DIO0_N and DIO7_N respectively.

There is a value called dead_time mentioned in the timing diagram – this is simply a preset delay between the rising edge of TXG and the beginning of RF90. In our project, this is maintained at a preset value of 100 µs everywhere unless stated otherwise. Next, TR (i.e., repetition time) is inclusive of all timed events – dead_time, RF90, Off_time, RF180, Acquisition, along with time when all RF and gating pulses are kept deactivated. Completion of one TR means that one repetition has been completed and the same pulse sequence repeats till all Repetitions have been exhausted.

Also note the presence of TE – the echo time. In SE, this is the distance between the center of RF90 and center of the MR echo. This parameter is not an input to the console – it merely indicates the location where the center of MR echo will show up. It is used to make sure the Acquisition time input is large enough to allow digitizing the echo; if it includes the echo center, the digitizer is capturing the right RF data from the receive side.

Figure 1 does not correspond to an actual SE sequence. A complete SE sequence will have two TXG pulses – one for RF90 and one for RF180, and the RXG pulse/Acquisition will begin from the center of RF180. The TXG used in this research is different because it is utilized by Pentek to digitize the receive. After receiving the positive edge of TXG, Pentek is set to digitize the receive side of MR2 for 20 ms and re-triggers in the next TR, when another TXG pulse is generated on the GPIO. Finally, Acquisition is kept right after the end of RF180, so that if the receive is digitized using this, only the MR echo is digitized and not the transmitted pulses.

3.3 Server and Client Pair

The client program client_program.py establishes a socket connection to the server via Ethernet. Pitaya has its own IP address – using this, our client connects to server and timed events can now be inputted. The client takes these from the user and transfers to the server for further processing, along with saving a copy of the user inputs in a timestamped text file in the same folder it is located in.

The server is a gcc-compiled executable version of server_program.c. It accepts all these values and converts RF90, Off_time, RF180, TR and Acquisition, all inputs in the microsecond scale, to number of cycles. This is done by multiplying them with 125 – the clock frequency of the

Pitaya in MHz. These values determine the number of cycles at which an RF pulse should turn on, when the trigger pulses are activated/deactivated and for how long etc. As for frequency of each pulse, that is computed using equation 3 – this is explained in section 3.7. The result is the variable Frequency.

These computed values, along with Repetitions, are all sent via specific memory addresses in the device to the digital circuit on our FPGA fabric. As described in the next section, these memory addresses allow for AXI GPIOs to read these computed values and pass them on to the rest of the digital circuit. Please refer to Appendix A for the server and client programs used in this project.

After computing and sending out all 7 pulse parameters to the digital circuit, the server waits till (TR*Repetitions) time has passed. Once that happens, it resets all computed values stored in its memory mapped I/O to zeros and waits for a second. After this, it closes the socket connection and exits back to the PuTTY console. To launch a new MR scan, we simply run the executable again and send fresh parameters from the Python program, and the process repeats.

3.4 AXI GPIOs

The role of an AXI GPIO IP is to map a maximum of two registers to device memory. Effectively, it implements memory mapped I/O so that both the CPU and FPGA fabric of the SoC can access the same address space for read/write operations. In this project, all pulse parameters are sent as 32-bit outputs in the block design – this means that the value for each parameter is read from its assigned memory location from the device and processed by rest of the bitstream-programmed digital circuit.

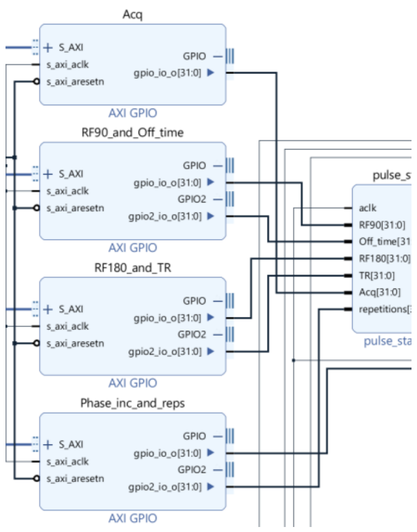
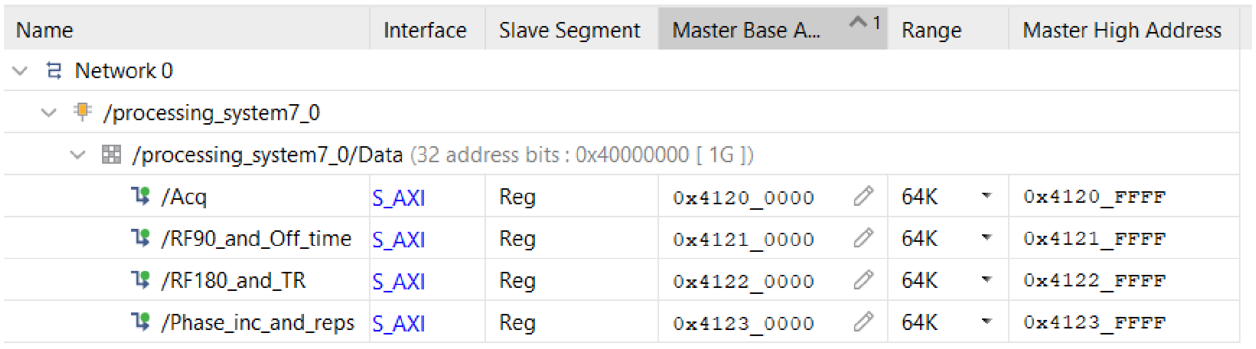


Figure 2: AXI GPIO IPs in system.bd block design

As seen in the figure above, 4 AXI GPIO blocks have been used for 7 outputs. These correspond to the seven timing parameters discussed in section 3.2 and are all sent to pulse_state_generator. They are crucial for setting things in motion in the circuit and generating the required RF and trigger pulses corresponding to a spin echo pulse sequence, as will be explained in upcoming sections.

To use them in the circuit, all memory addresses need to be mapped to user space on the device using the auto-assign feature in Vivado’s Address editor – the steps are explained in detail in the attached user manual. When auto-assigned, the following addresses were assigned to the AXI GPIO cores. Note that each GPIO IP is named in order of the register that comes first in the address space – this means that in RF90_and_Off_time, RF90 occupies the address space 0x41210000 ~ 0x41210007 and Off_time occupies 0x41210008 ~ 0x4121000F.



Name	Interface	Slave Segment	Master Base A...	Range	Master High Address
Network 0					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/Acq	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
/RF90_and_Off_time	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
/RF180_and_TR	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
/Phase_inc_and_reps	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF

Figure 3: Address Editor displaying assigned addresses for AXI GPIOs

3.5 Pulse state generation

Relating the pulse parameters to timed events bring us to RTL codes created in Verilog. These codes relate a value 'pulse era' to a specific state value as indicated in the following table. Pulse era can be thought of as a count value that increments by 1 at every positive edge of the 125 MHz clock signal driving the FPGA logic.

Pulse era	State	RF	TXG	RXG
0 ~ dead_time	0	OFF	1	0
dead_time ~ dead_time + RF90	1	ON	1	0
dead_time + RF90 ~ dead_time + RF90 + off_time	2	OFF	0	0
dead_time + RF90 + off_time ~ dead_time + RF90 + off_time + RF180	4	ON	0	0
dead_time + RF90 + off_time + RF180 ~ dead_time + RF90 + off_time + RF180 + Acq	3	OFF	0	1
dead_time + RF90 + off_time + RF180 + Acq ~ TR	2	OFF	0	0

Table 1: Pulse State Generation in Verilog

In the Vivado block design, we have set dead_time equal to 100 µs. Hence, TXG is activated 100 µs before the actual RF pulse, whether it is a 90° or a 180° pulse.

If pulse era becomes greater than TR, its value is reset to zero and the process continues 'Repetitions' number of times. After this, the state generator will remain in state '2'. Meanwhile, the server code is designed to wait till (TR*Repetitions) time has passed. Once that happens, it blanks out the state generator by sending all zeros as pulse parameters to it and waits for a second. After this, fresh parameters can be sent into the state generator for a new MR scan.

3.6 RTL Scripts

Three RTL scripts have been created here – pulse_state_generator, DDS_control and DAC_activator. The first script accepts all 7 AXI GPIO values and increments a counter called pulse_era in a manner discussed in section 4.2 and generates state values corresponding to pulse era in accordance with Table 1. Note that pulse_state_generator also outputs a resync_bit that is fed to DDS. This is important since it clears the phase accumulator when set to '1' and allows phase accumulator to update itself when set to '0'. Hence, resync_bit is set to '1' after an RF pulse has completed generation, i.e., right after state changes from '1' to '2' or '4' to '3'. It is also set to '1' right before the pulse sequence begins or repeats. Without this functionality, the system may fail to generate the first RF90 pulse since resync_bit needs to be asserted high for one clock cycle before DDS can generate any output. At all other times, resync_bit is set to '0'.

DDS_control accepts state values generated by pulse_state_generator (as state) and phase increment from AXI GPIO (as phase_inc_server) and passes on this phase increment value (as p_inc) and a phase offset value of zero (as p_off) to DDS, along with TXG and RXG signals according to Table 1. Their interconnections in the block design are shown in Figure 4.

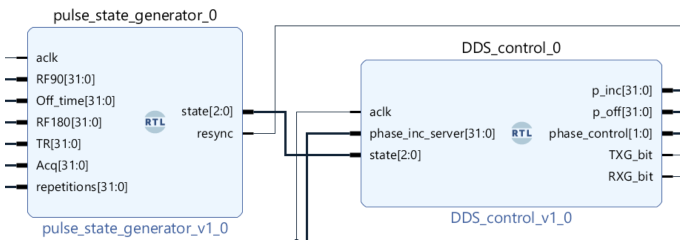


Figure 4: pulse_state_generator and DDS_control – connected

DAC_activator is needed to instruct the Red Pitaya DAC IP core when to generate a DAC output corresponding to RF90 or RF180 and when not to. The output tvalid_out connects directly to s_axis_tvalid of the DAC IP, as seen in the figure below.

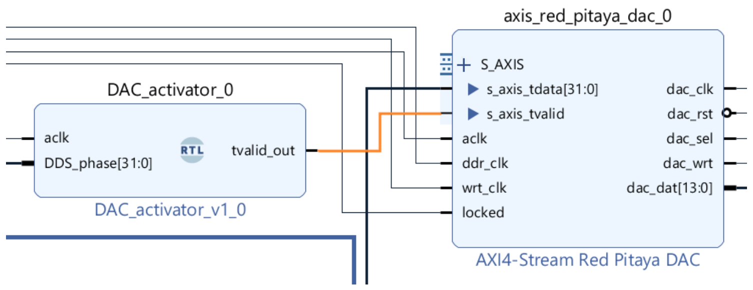


Figure 5: DAC_activator and Pitaya's DAC IP – connected

Its operation is simple – when phase generated by DDS changes, set tvalid_out as HIGH, hence activating the DAC. When phase generated by the DDS remains steady over two clock cycles or is zero, set tvalid_out as LOW, hence deactivating the DAC. The three script files are attached to Appendix B of this document.

3.7 Direct Digital Synthesis

DDS is short for Direct Digital Synthesizer – it is an IP core in Vivado that generates an arbitrary sine waveform that updates by a certain 'phase increment' (determines frequency) after every clock cycle and are initialized by a 'phase offset' (determines starting position of arbitrary wave). It contains a phase accumulator and a sine lookup table, both of which require m_axis_phase_tdata to be provided as input – basically, we need to plug in phase increment and offset to the right locations. This is explained in detail in the following subsection.

By plugging in these values, the phase accumulator is instructed to compute a phase value – this is used to search the sine lookup table. The amplitude corresponding to this phase value is sent out as output. As mentioned in section 3.3, frequency of the sine pulses is governed by formula 3. Formula 4 computes phase offset in degrees, as a function of phase word length.

$$phase\ increment = \frac{f_{out} \times 2^{B_\theta(n)}}{f_{in}} \tag{3}$$

$$phase\ offset, \alpha = \frac{2^{B_\theta(n)} \times \alpha}{360} \tag{4}$$

where f_{out} = desired output frequency of DDS

f_{in} = frequency of clock signal fed into DDS = 125 MHz

$B_\theta(n)$ = phase width (part of DDS properties)

In this project, $B_\theta(n)$ was equal to 30, after setting the DDS IP core according to steps outlined in section 3.2.5 of the Red Pitaya User Manual, attached as Appendix D to this document.

DDS in this project is used to generate sinusoidal pulses that are offset by 0° (i.e., begin at zero phase). For this reason, only phase increment is calculated in the server program and not phase offset, and resync bit is set to zero to clear the phase accumulator of its previous value after a RF pulse has been generated fully.

In fact, phase offset p_off is kept at a constant 0 in the RTL code DDS_control.v itself, whether pulses are being generated or not. Note that phase offset has not been added as an AXI GPIO output in the main project, although it is included in the Red Pitaya User Manual in chapter 3. This is because the author anticipates phase offset being incorporated in future projects involving Pitaya bitstreams for MRI.

Hence, only formula 3 is implemented in the server program, which takes f_{out} from the user, converts it to the corresponding phase increment, and sends this phase increment to the Pitaya. The relevant lines of C code are mentioned on the next page.

```
uint32_t RF90, Off_time, RF180, TR, Acq, reps;
float frequency;
int freq_MHz = 125;
uint32_t phase_inc = 1<<30; //2^30
…
//Memory map pointers to AXI GPIO addresses
```

```
/* get frequency and other inputs from client */
…
float val = 125/frequency;
phase_inc /= val; //final DDS input

*((uint32_t *)(cfg_90_and_Off_time + 0)) = RF90*freq_MHz;
*((uint32_t *)(cfg_90_and_Off_time + 8)) = Off_time*freq_MHz;
*((uint32_t *)(cfg_180_TR + 0)) = RF180*freq_MHz;
*((uint32_t *)(cfg_180_TR + 8)) = TR*freq_MHz;
*((uint32_t *)(cfg_Acq + 0)) = Acq*freq_MHz;
*((uint32_t *)(cfg_Phase_inc_and_reps + 0)) = phase_inc;
*((uint32_t *)(cfg_Phase_inc_and_reps + 8)) = reps;
```

The first line sets an unsigned 32-bit integer value, phase_inc, as 2^{30} using the right shift operator. Next, val, a float value, computes the result for 125/frequency up to 6 places after decimal point, assuming frequency is another float variable obtained from user. Finally, the last line of code divides phase_inc, i.e., 2^{30}, by val to give the result of $2^{30}\times$frequency/125. This way, the C executable computes the appropriate phase_inc value to send to AXI GPIO and transfer by DDS_control to DDS, ready for sine wave generation.

A 32-bit value called s_axis_phase_tdata is outputted by the DDS – this is our digital RF waveform. It is converted to a 14-bit output by AXI4-Stream Red Pitaya DAC – this is the analog waveform that shows up at OUT1 port on the Pitaya board.

3.7.1 m_axis_phase_tdata

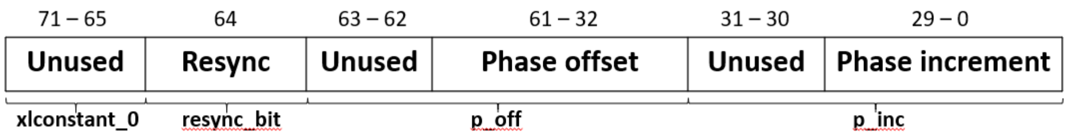


Figure 6: Layout of m_axis_phase_tdata

The DDS IP functions by accepting a 72-bit input called m_axis_phase_tdata, the internal structure of which is given in figure 6. This complete input is sent by concatenating all the values underneath it – p_inc (phase increment), p_off (phase offset), resync_bit and xlconstant_0. Since our phase width is 30 and both phase increment and offset are set to ‘Streaming’, both are 30 bits wide each. However, register memory in the DDS is assigned as multiples of 8, therefore 32 bits are assigned to each of them, with the last two bits from the MSB side left unused. Since p_off is always zero in this project and p_inc is computed keeping 30 bits in mind, there is no risk of bit overflow into the unused part of m_axis_phase_tdata – they will always remain zeros.

As discussed before, resync_bit is set by pulse_state_generator as 0 or 1 as per requirement. xlconstant_0 is simply a 7-bit constant value that is all zeros – this is sent in to fill out the input to complete the required 72-bit input length.

3.8 Block design for pulse sequence generator (PSG)

The complete block design for PSG is shown in the figure on the next page. A Verilog wrapper is generated for it and set as top module. Finally, a bitstream corresponding to the block design was generated in Vivado, transferred to Pitaya and activated using the cat command. This process forms our PSG digital circuit and is controlled by the server and client programs as shown in Figure 7. The TXG and RXG lines are allotted the GPIO output connections indicated in Figure 8. These and RF output (OUT1) are tested in the next section.

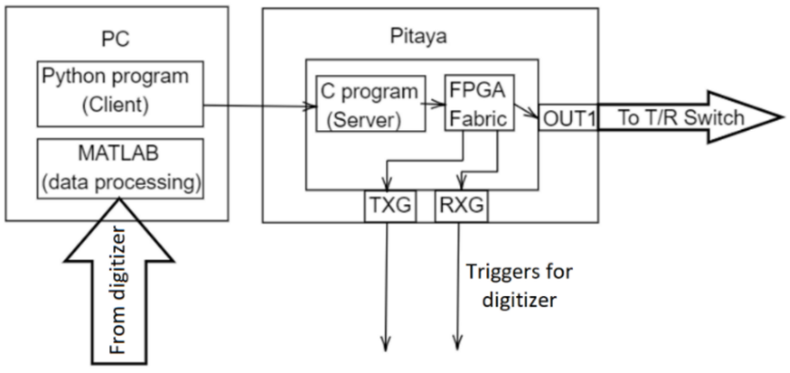


Figure 7: Connecting PSG (housed in FPGA fabric) with I/O and other devices

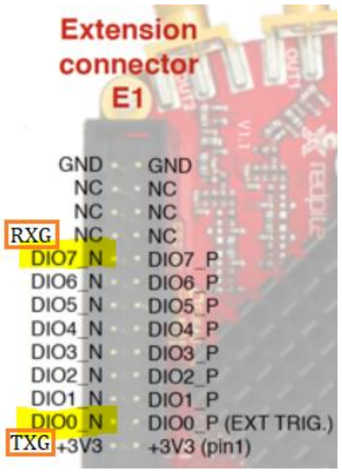


Figure 8: TXG and RXG connections on E1 of Pitaya – highlighted. Modified from Extension — Red Pitaya 0.97 documentation[41]

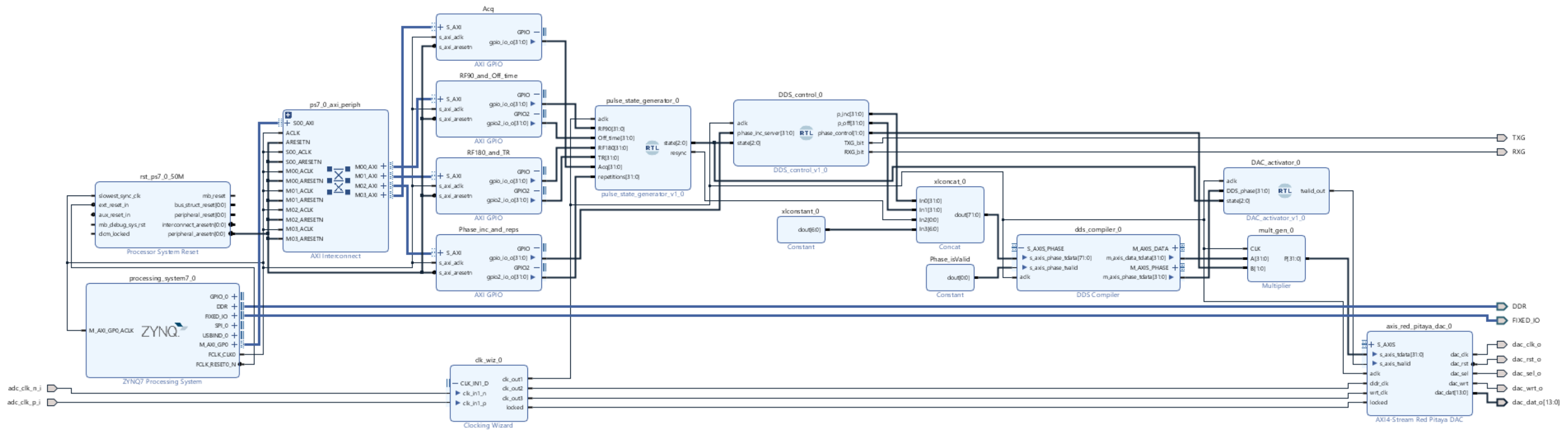


Figure 9: Complete block design for PSG

3.9 Testing and setups

Three setups were used in this research. The first setup was completely digital – it used Vivado's XSim tool to evaluate the functionality of the digital circuit and verify that it works before bitstream creation. A testbench code was created and used for the purpose – it is attached to the appendix of this document.

The second one tested pulse and trigger generation after the bitstream was created and activated the FPGA fabric, and the server-client program pair was set up correctly. The output was observed on three different oscilloscopes for RF delays and responsiveness and ensuring that it matched all pulse parameters.

The last one was an experiment run on an MR setup containing a 4.7 T Varian magnet. This setup involved connecting the Pitaya and PC to an RF frontend comprising of a power amplifier, a match and tune circuit, a passive T/R switch, one preamplifier and one digitizer. Pitaya was set to run a pulse sequence twenty times. The digitizer was triggered either by TXG or RXG for each repetition as explained later, and the resulting data was saved to PC and processed in MATLAB. The purpose behind this was to extract echoes and check for phase stability – whether the RF pulses and echoes from multiple repetitions aligned at the same locations or not.

3.9.1 XSim Testing – Simulations

The FPGA logic is verified through simulations running in Vivado's XSim tool. The testbench code for the same is provided as an appendix. This is done after each RTL code is created and added to the block design described in figure 7.

Basically, all the blocks highlighted below were selected and a hierarchy called hier_0 was created from it. Once created, the block clk_wiz_0 was added to the hierarchy as well.

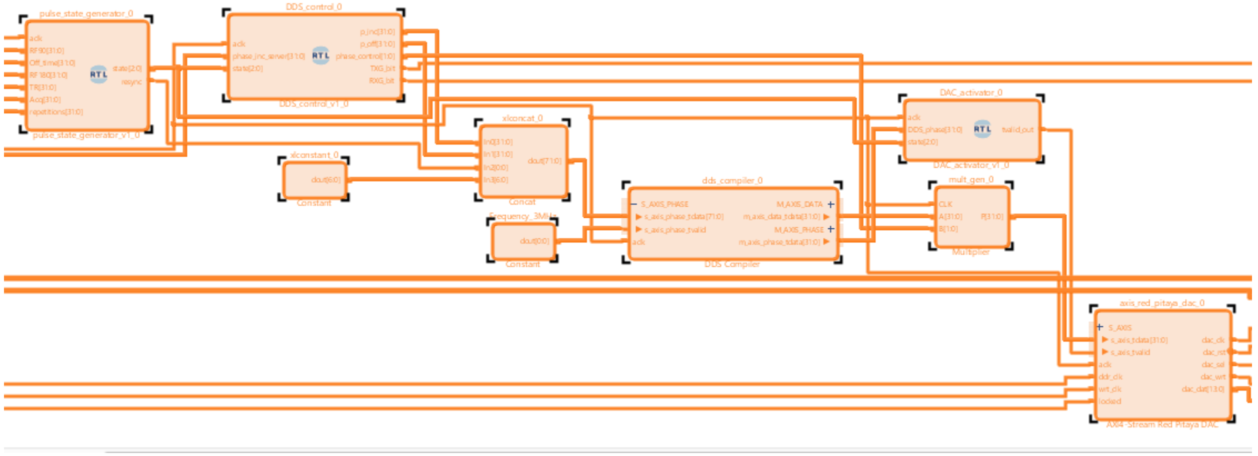


Figure 10: Creating a hierarchy from selected blocks

A new block diagram called design_test was created in the same project and hier_0 was added to it. After auto-generating input and output pins for hier_0, the hierarchy was ungrouped in design_test.

Five multiplier IP cores were added to design_test. All of them were set as constant coefficient multipliers with input width set to 32 bits wide and coefficient being a constant integer value of 125. The output width was set at custom values of 31 as output MSB and 0 as output LSB. They were connected and all IPs and ports were named to match the block design given below. The blocks design was saved, an HDL wrapper was created from design_test.bd, and the resulting design_test_wrapper.v file was set as ‘top module’ under Sources in Vivado. Note that these multiplier IPs are not part of the original system.bd block design. In fact, they can be replaced with appropriate lines of code in the testbench file using a set of 5 wires, just like phase offset is computed from frequency by the testbench using phase_val_1 and phase_val_2 as explained later.

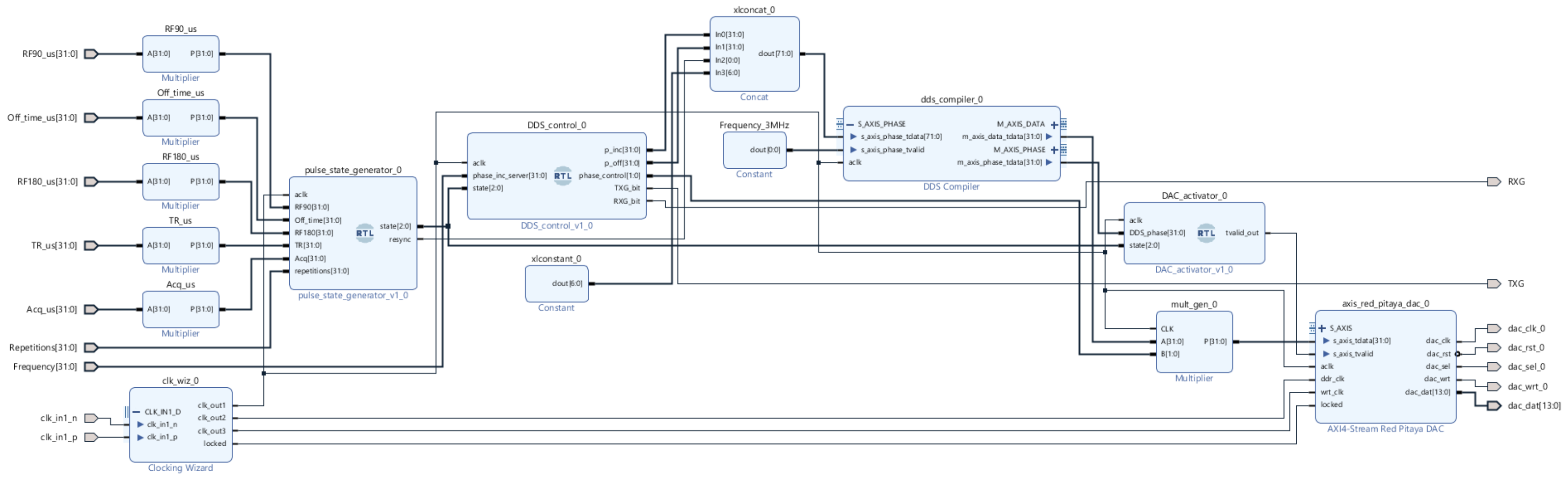


Figure 11: Complete block design for design_test.bd

Next, the simulation code file provided in appendix D (design_test_TB.v) was set as ‘top’ module under Simulation Sources in Vivado. It has two differential clock inputs clk_in1_n and clk_in1_p that are set such that they toggle every 4 ns. They replicate two one-bit signals of 50% duty cycle that repeat every 8 ns – in other words, two differential clock inputs of 125 MHz each. This feeds into the Clocking Wizard IP which generates the required clock signals to drive all components of the device under test (DUT).

The testbench code also computes phase increment (indicated by phase_val_2) from Frequency_Hz. Remember that the original block design does not compute phase increment – this is calculated and provided by the server program itself. Hence the lines of code below use two 32-bit regs phase_part_1 and phase_part_2 to replicate the functionality of phase_inc and val used in the C server code; only the language differs.

```
phase_part_1 = Frequency_Hz * (1<<30);
phase_part_2 = phase_part_1/125000000;
```

The remaining testbench code simply matches the relevant regs and wires with ports of the design_test_wrapper DUT (device under test) and sets the seven parameters in the following manner, with each sequence executed once for the stipulated duration, one after the other.

After the functionality is verified in XSim, the bitstream file (a .bit file) is generated in Vivado. This is copied to the /tmp directory of Pitaya and activated using the cat command.

3.9.2 RF1 Setup – PC, Pitaya and oscilloscope

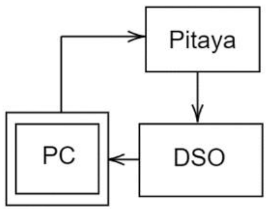


Figure 12: Setup for RF1 – pulse generation and triggering on Pitaya

After activating Pitaya’s FPGA with our bitstream, the server executable is created and executed on the Pitaya. Inputs are set in the client, and the resulting outputs checked for time lag, correctness to timing inputs and initial phase offset of RF pulses. Four different DSOs (digital storage oscilloscopes) are used for this, each possessing a different sampling rate and number of input and trigger/GPIO channels.

The first oscilloscope used is an Agilent DSO3062 to check the time lag between GPIO lines and RF output. For this test, the bitstream is modified such that dead_time equals zero. This will cause TXG to be set as HIGH at the same time as RF90 begins instead of being HIGH 100 μs before RF90 is generated. The check will be performed by inputting parameters for sequence #1 and observing the time difference between the positive edge of TXG and the start of RF90 on the oscilloscope. They are also listed in the table below describing the properties, pulse parameters and tests performed with the digitizing instruments (DSOs or oscilloscopes).

Next, the RF and GPIO (TXG and RXG) outputs will be observed on a BitScope DSO. Finally, to observe the beginning and ending phases of the RF90 and RF180 pulses and check if their output frequency matches the requested one, a PicoScope 3206 DSO is used. The RF output will be observed after inputting timing parameters mentioned in Table 3.

3.9.3 MR2 Setup – 4.7 T magnet and 2H sample

This setup uses the magnet of a legacy 4.7 T Varian scanner, which comes with its own match and tune circuit. The Pitaya’s output (OUT1) is connected to a 32 dB Minicircuits power amplifier (PA), as shown in the diagram. This connects to a T/R switch followed by a match and tune and a solenoid wrapped around a test tube filled with heavy water which contains 2H. This connects to the rest of the MR setup as shown in Figure 13.

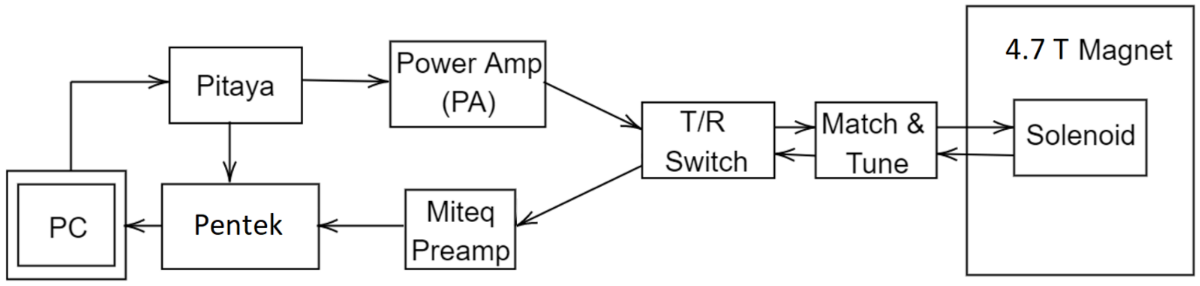


Figure 13: Setup for MR2 – 4.7 T magnet and 2H sample

Here, γ = gyromagnetic ratio of 2H in Hz T^{-1} = 6.536×10^6 Hz T^{-1}

γ = gyromagnetic ratio of 2H in rad s^{-1} T^{-1} = $2\pi \times 6.536 \times 10^6$ rad s^{-1} T^{-1}

B_0 = B field of the magnet = 4.7 T

This gives the Larmor frequency of this sample as $\gamma \times B_0$ = 30.72 MHz. Hence, the frequency input shall be set as 30.72 MHz on the client side, and the remaining pulse parameters from table 2.

Apart from the PC, Pitaya and digitizer, all components of the workstation are completely analog. The PA connects to a 30.72 MHz match and tune network and a solenoid wrapped around a test tube containing 2H. The solenoid and match and tune is placed inside a 4.7 T permanent magnet – all the other components are placed outside the magnet. The solenoid, T/R switch and match-and-tune network are part of the receive channel. The echo travels this path into a Miteq AU1579 preamplifier which boosts the received signal level. The RF pulses generated by Pitaya follow the transmit path to reach the heavy water sample housed in the solenoid and an echo is sent out on the receive path.

At the end of the acquisition/receive path, we place the Pentek digitizer. It has two analog input channels and one external trigger, and a bandwidth of 200 MHz. This makes it possible for Pentek to capture an analog signal upon an external trigger and record it as raw data for further analysis. It will capture and record signals on the receive channel after the preamp stage based on trigger pulses sent out by Pitaya directly to it.

Pitaya sends out two +3.3 V trigger pulse signals, TXG and RXG, on separate GPIO lines – this is described in the manual. Pentek is triggered by the TXG pulse such that digitization of the receive side begins on its rising edge. Using a LabVIEW program, an acquisition period of 20 ms is set – this instructs Pentek to digitize all signals from the receive side up to 20 ms after a rising edge is observed on TXG. Since Pitaya will send out 20 repetitions, Pentek will acquire data 20 times and append data from each repetition in a new column of a .dat file.

3.10 Data processing for MR2 Setup

This is another step carried out on setup MR2, which described in subsection 3.9.3. The raw data generated by Pentek are files with the extension .dat. It is processed using two MATLAB scripts – ReadPentek.m and display_Pentek_RX.m. These scripts are attached to Appendix D of this document.

The goal of these scripts is to read raw data from its file and plot its graph, correct for DC offset, perform a Fourier Transform on it and plot the frequency response, identify the peak frequency level and crop out 25 kHz to its left and right, and display the result. All 20 echo peaks of Pentek data are then superimposed on top of one another on the same graph using MATLAB’s ‘hold on’ feature.

4. IMPLEMENTATION

4.1 Parameters used

For simulations, the following pulse sequences were used. The code for the same is provided in Appendix C – Testbench/Simulation Code.

Sequence	#1	#2	#3	#4	#5
Duration (in µs)	800	5	800	5	800
Frequency (in MHz)	2.55	0	10	0	30.72
RF90 (in µs)	5	0	4	0	1
Off_time (in µs)	10	0	4	0	4
RF180 (in µs)	10	0	8	0	2
Acquisition (in µs)	20	0	15	0	10
TR (in µs)	180	0	150	0	130
Repetitions	3	0	4	0	5

Table 2: Parameters used in XSim simulation

As seen above, all values are set in accordance with their units. The reason for choosing 2.55 MHz for sequence #1 and 30.72 MHz for sequence #5 is to compare these against the pulses generated by the actual device after bitstream activation. Moreover, these are the Larmor frequencies of 1H at 0.06 T and 2H at 4.7 T – this is discussed in more detail in the next subsection. 10 MHz in sequence #3 is an arbitrary frequency chosen to demonstrate the generation of pulses over a range of frequencies – this is not a Larmor frequency intended to be used in this project. Finally, Acquisition and TR are smaller than the typical values sent in to

Sequences #2 and #3 have all parameters set to zero – the purpose is to demonstrate the 'blanking out' requirement of our digital circuit. As a final check, these were toggled OFF to show the circuit indeed does not function correctly without a blanking sequence sent in before a fresh sequence.

In the testbench code, '_us' suffix after a reg indicates that it is a parameter in microseconds, and '_Hz' indicates that it is a parameter in Hz. Pound symbols (#) are used to set delays while the testbench controls the DUT. Since the time scale used is in nanoseconds (indicated by `timescale 1ns / 1ps), all delays are set in nanoseconds. With each delay, the DUT is allowed to run on a fixed set of inputs for the duration of that delay. After the delay, the inputs are overwritten with the next set of values, ready to be accepted by the DUT and the relevant output is generated. After 2.410 milliseconds, the testbench concludes its operation as it reaches the $finish command.

Table 3 details all the pulse parameters and properties of the RF/MR testing performed.

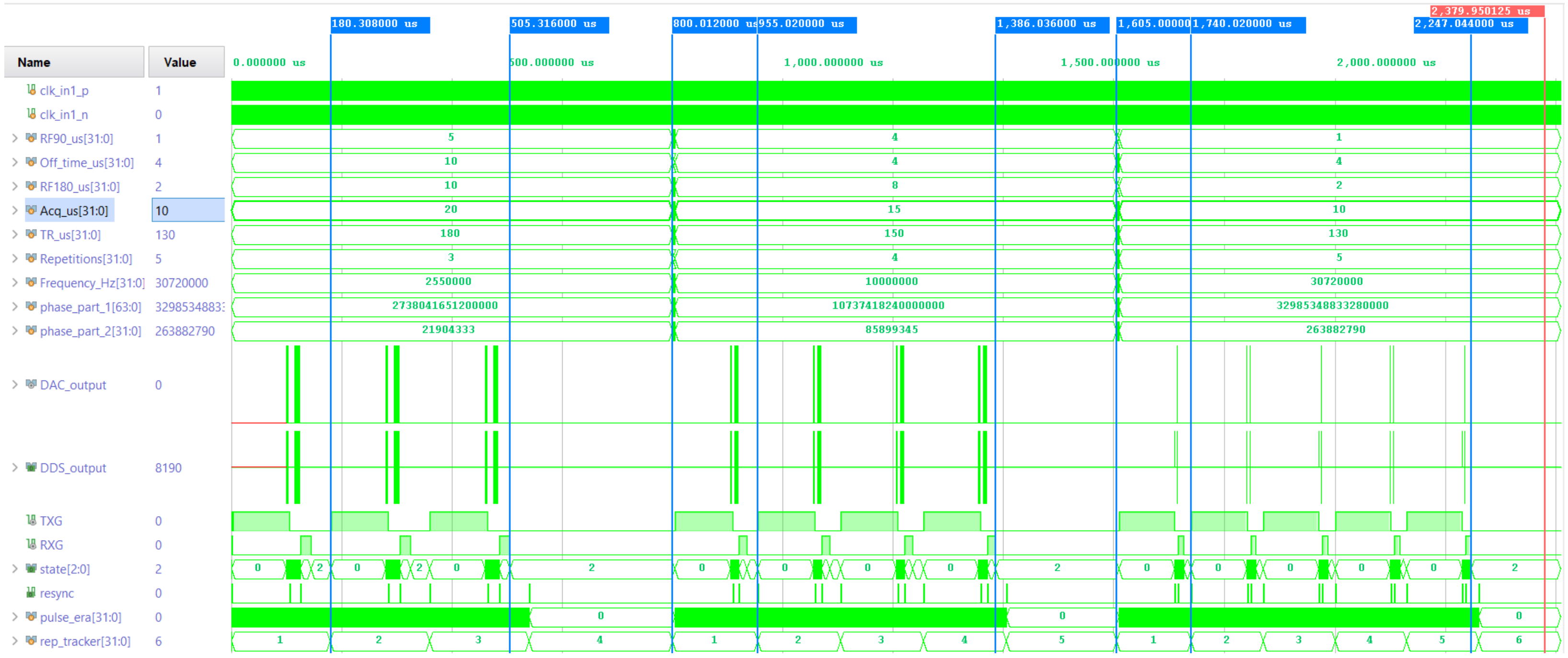
Instrument	**Agilent**	**BitScope**	**Picoscope**	**Pentek**
Sampling Rate	1 GSa/s	5 MSa/s	50 MSa/s	200 MSa/s
What to test	Time lag	Transmit	Transmit	Receive
Test setup	RF1	RF1	RF1	MR2
Magnet Used	X	X	X	4.7 T
Test sample	X	X	X	2H
RF90 (in us)	5	70	70	600
Off_time (in us)	10	300	300	6,500
RF180 (in us)	10	140	140	1,200
Acq (in us)	20	200	200	20,000
TR (in us)	180	10^6	10^6	5.21×10^6
Repetitions	3	24	24	20
Frequency (in MHz)	2.55	2.55	2.55, 10, 30.72	30.72

Table 3: Details on RF/MR testing

4.2 Waveforms and results

4.2.1 Simulated – before bitstream generation

Following the method in subsection 3.9.1, all five sequences were executed correctly by our DUT in XSim. The complete waveform plot is shown below, containing inputs and outputs for all sequences.



Name	Value
clk_in1_p	1
clk_in1_n	0
RF90_us[31:0]	1
Off_time_us[31:0]	4
RF180_us[31:0]	2
Acq_us[31:0]	10
TR_us[31:0]	130
Repetitions[31:0]	5
Frequency_Hz[31:0]	30720000
phase_part_1[63:0]	3298534883:
phase_part_2[31:0]	263882790
DAC_output	0
DDS_output	8190
TXG	0
RXG	0
state[2:0]	2
resync	0
pulse_era[31:0]	0
rep_tracker[31:0]	6

Figure 14: Simulation results for testbench code

As expected, TXG was set HIGH by the circuit at the beginning of every TR and stayed like this for 100 µs, after which it was set LOW. Right after TXG was set to LOW, an RF sine pulse was activated by the circuit for RF90_us number of microseconds and then deactivated. All RF and GPIO outputs remained in LOW for Off_time_us number of microseconds. After that, another RF sine pulse was activated for RF180_us number of microseconds and deactivated. Finally, RXG was set HIGH right after RF180 generation was completed for Acq_us number of microseconds and then set to LOW. Once TR_us number of microseconds passed, the pulse generation process continued 'Repetitions' number of times.

To verify the timing, markers were placed at specific locations of the plot – three at positive edges of TXG, three at negative edges of RXG and two at preset delays set using # symbol in the testbench code. The results are tabulated below. Note that #800_µs refers to a preset delay event of 800 µs – the suffix _µs indicates this value is in microseconds and refers to the Verilog line of #800000, or 800,000 ns. The subscript at the end of an event without # refers to its sequence number w.r.t. Table 2.

Chain of timed events	**Expected time marker**	**Simulated time marker**
TR_1 × 1	180.000 µs	180.308 µs
(TR_1 × 2) + (dead_time + $RF90_1$ + Off_time_1 + $RF180_1$ + Acq_1)	505.000 µs	505.316 µs
#800_µs	800.000 µs	800.012 µs
#800_µs + #5_µs + TR_3	955.000 µs	955.020 µs
#800_us + #5_us + (TR_3 × 3) + (dead_time + $RF90_3$ + Off_time_3 + $RF180_3$ + Acq_3)	1386.000 µs	1386.036 µs
#800_µs + #5_µs + #800_µs	1605.000 µs	1605.000 µs
#800_µs + #5_µs+ #800_µs + #5_µs + (TR_5 × 1)	1740.000 µs	1740.020 µs
#800 + #5 + #800 + #5 + (TR_5 × 4) + (dead_time + $RF90_5$ + Off_time_5 + $RF180_5$ + Acq_5)	2247.000 µs	2247.044 µs

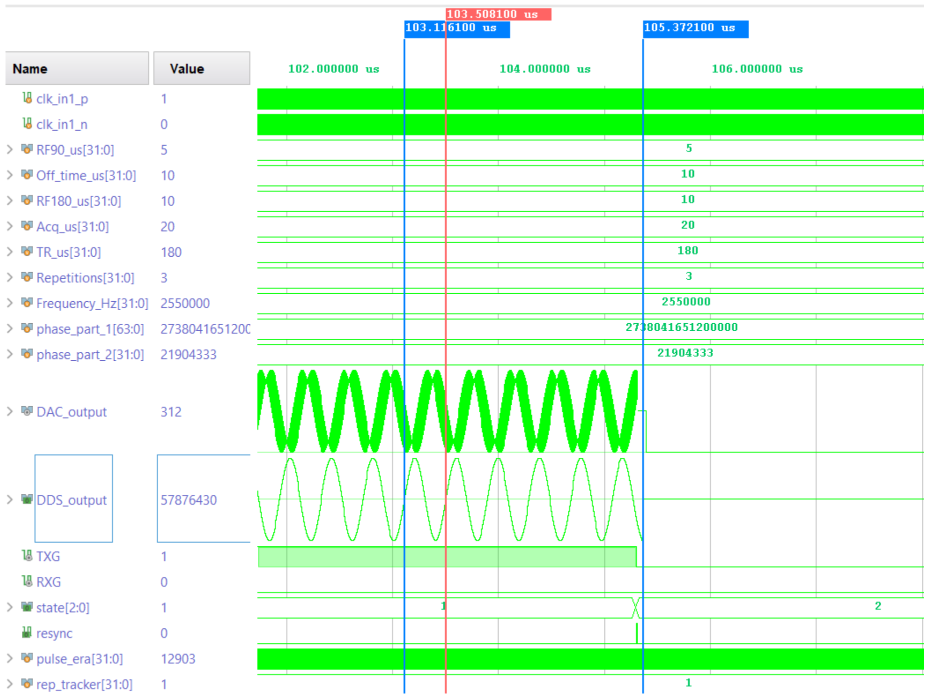
Table 4: Comparing time markers in Figure 14 with expected values

From this table, the largest difference between simulated and expected time events is 0.316 μs; the difference is less than 1 μs. Hence, the RF output, TXG and RXG of each sequence indeed matches up in terms of overall timing for pulses to start and stop/ turn HIGH and LOW.

The TXG and RXG outputs seem to be correctly generated – this is verified later by zooming in to one part of the graph. The number of times the pulse sequence repeats is also correct. This indicates a digital circuit with its output(s) matching the timing parameters discussed previously.

4.2.1.1 Sequence #1

Next, the graph was zoomed in to the end of the first RF180 pulse of sequence #1 to check if our testbench code computed the correct phase increment, and whether the generated RF pulse was of the right frequency. For this, we have the set frequency (input) = 2,550,000 Hz = 2.55 MHz and calculated phase increment = 21904333.21. Since phase increment fed into the DDS is an integer value, all digits after the decimal place would ideally be discarded, i.e., it must be 21904333.



Name	Value
clk_in1_p	1
clk_in1_n	0
RF90_us[31:0]	5
Off_time_us[31:0]	10
RF180_us[31:0]	10
Acq_us[31:0]	20
TR_us[31:0]	180
Repetitions[31:0]	3
Frequency_Hz[31:0]	2550000
phase_part_1[63:0]	273804165120
phase_part_2[31:0]	21904333
DAC_output	312
DDS_output	57876430
TXG	1
RXG	0
state[2:0]	1
resync	0
pulse_era[31:0]	12903
rep_tracker[31:0]	1

Figure 15: Check for phase increment and frequency – sequence #1

The testbench-computed phase increment is phase_part_2; from the above figure, we see that this was computed as expected. The simulated frequency is calculated by finding the difference between the red marker and the first blue one and inverting this difference, which is 10^6/(103.5081-103.1161) = 2.5510 MHz. This is very close to the set frequency value.

Next, the graph was zoomed in further to observe the delay between state change from 4 (RF ON) to 3 (Acq ON) and the time resync, DDS and DAC respond to this state change. To double-check, frequency was calculated again in this graph based on markers – this time in the nanosecond scale.

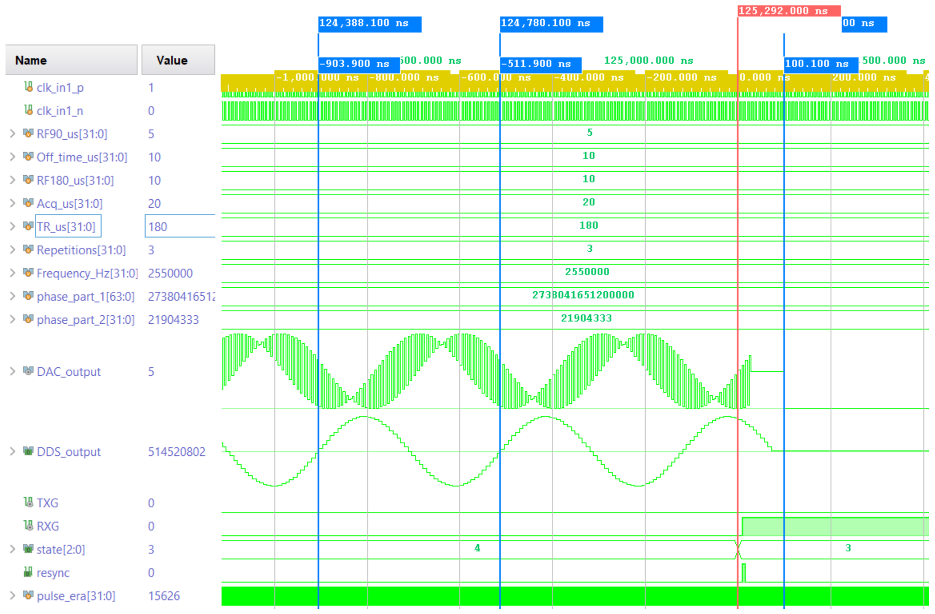


Figure 16: Observing frequency and state change – sequence #1

The simulated frequency here is $10^9/(124780.1 - 124388.1) = 2.5510$ MHz. This is the same frequency as before. To quantify the time delay mentioned previously, the graph was zoomed into further as shown below.

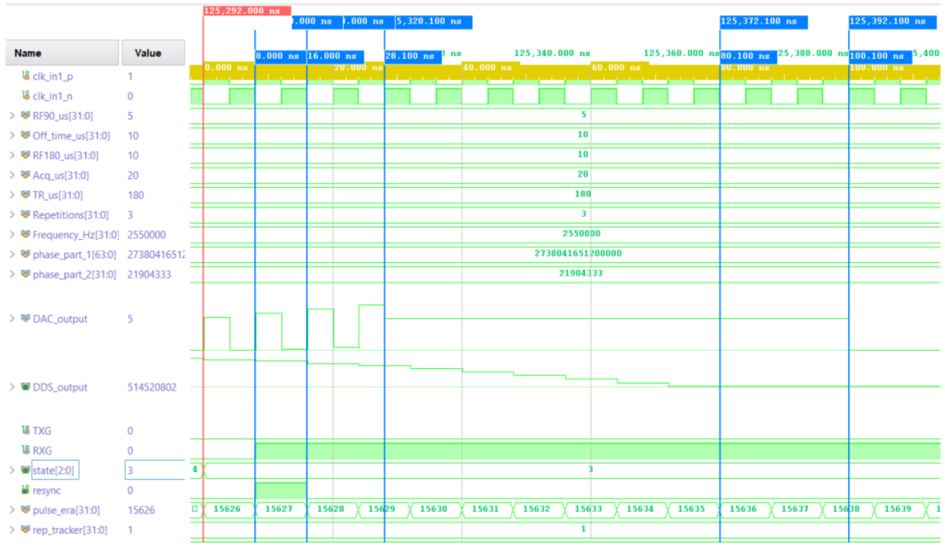


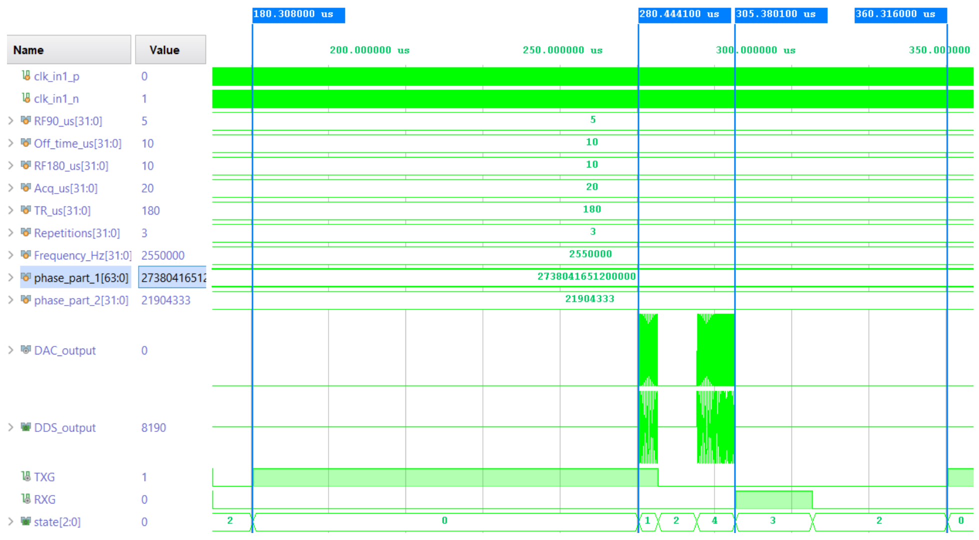
Figure 17: Zoomed in – state transition

We see that resync goes HIGH 8 ns after the state transition, or one clock cycle after changing states from 4 to 3. This matches our ideal, since resync is programmed to turn on one clock cycle after a state transition from that of non-zero phase increment (pulse generation) to zero increment (flatline). As expected, it is asserted for 1 clock cycle (8 ns) and then goes LOW.

DAC output appears to remain a steady non-zero value 28.1 ns after the state change, or 12.1 ns after resync is de-asserted. DDS takes a little longer (80.1 ns from state change, or 64.1 ns after de-asserting resync) to settle down to its ‘steady flatline’ level, i.e., 8190. This value of 8190 shows up whenever no pulse is created by DDS and remains steady if no phase increment is fed to it. DAC appears to translate this steady flatline value to a zero in its output every time – hence this is not an issue.

In this case, DAC generates a steady zero 100.1 ns after state change, or 84.1 ns after resync is de-asserted. We assume the longest delay, i.e., DAC output flatlines 100.1 ns later than expected. This comes down to 0.1001 us delay, which is negligible, considering our system needs to be precise only up to a micrometer.

Next, the graph was zoomed out to view the second TR of sequence #1. To check for timings, four markers were places – one at the rising edge its TXG, one right before RF90 was generated, one at the rising edge of RXG and one at the rising edge of the next TXG pulse. The first rising TXG edge indicates start of the second TR; the second rising edge indicated the end of this TR and start of the next one. The timings from this figure were then tabulated in Table 3.



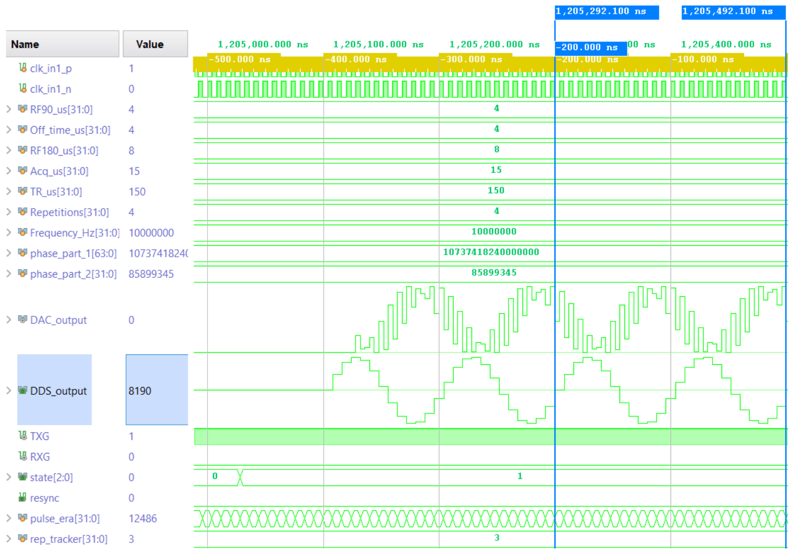
Name	Value
clk_in1_p	0
clk_in1_n	1
RF90_us[31:0]	5
Off_time_us[31:0]	10
RF180_us[31:0]	10
Acq_us[31:0]	20
TR_us[31:0]	180
Repetitions[31:0]	3
Frequency_Hz[31:0]	2550000
phase_part_1[63:0]	27380416512
phase_part_2[31:0]	21904333
DAC_output	0
DDS_output	8190
TXG	1
RXG	0
state[2:0]	0

Figure 18: Timings in second TR of sequence #1

Chain of timed events	Expected time marker	Simulated time marker
$TR_1 \times 1$	180.000 µs	180.308 µs
$(TR_1 \times 1)$ + dead_time	280.000 µs	280.4441 µs
$(TR_1 \times 1) + (\text{dead\_time} + RF90_1 + \text{Off\_time}_1 + RF180_1)$	305.000 µs	305.3801 µs
$TR_1 \times 2$	360.000 µs	360.316 µs

Table 5: Comparing time markers in Figure 18 with expected values

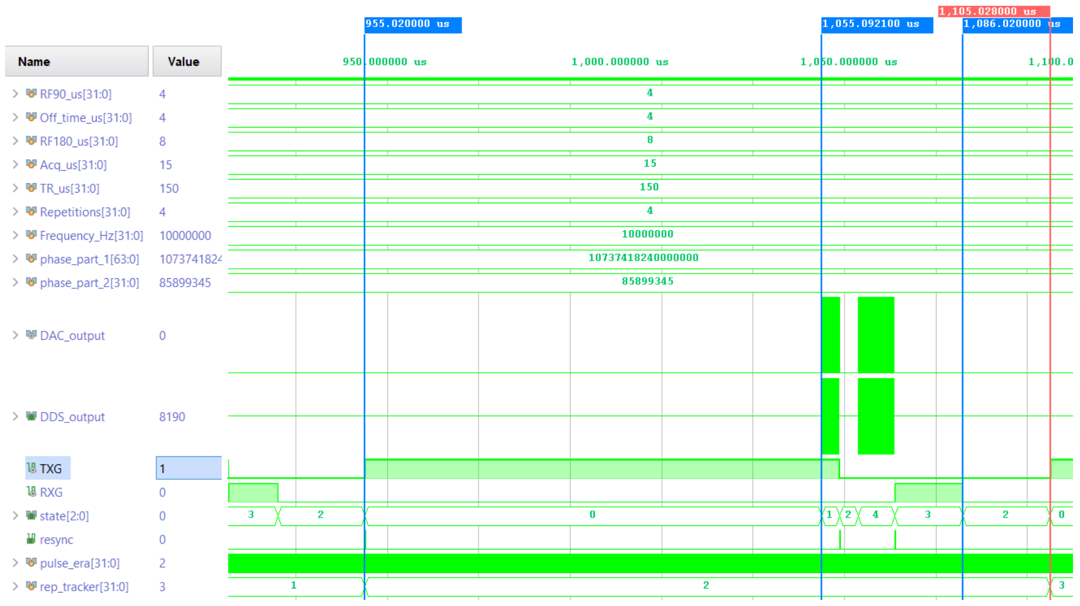
4.2.1.2 Sequence #3



Name	Value
clk_in1_p	1
clk_in1_n	0
RF90_us[31:0]	4
Off_time_us[31:0]	4
RF180_us[31:0]	8
Acq_us[31:0]	15
TR_us[31:0]	150
Repetitions[31:0]	4
Frequency_Hz[31:0]	10000000
phase_part_1[63:0]	10737418240
phase_part_2[31:0]	85899345
DAC_output	0
DDS_output	8190
TXG	1
RXG	0
state[2:0]	0
resync	0
pulse_era[31:0]	12486
rep_tracker[31:0]	3

Figure 19: Sequence #3 – start of RF90

Figure 18 shows the beginning of RF90 for the third TR in sequence #3. It has the set frequency (input) = 10,000,000 Hz = 10 MHz and calculated phase increment = 85899345.92. Discarding digits beyond the decimal point gives 85899345 – this matches the computed phase increment. This phase increment value gives a frequency of 9.99 MHz – quite close to the requested 10 MHz. Next, the simulated frequency is $2\times10^9/(1205492.1-1205292.1)$ = 10 MHz – this is the same as our set frequency value. Note that this value is multiplied by 2 because the difference between two sine waves was used in the calculation.



Name	Value
RF90_us[31:0]	4
Off_time_us[31:0]	4
RF180_us[31:0]	8
Acq_us[31:0]	15
TR_us[31:0]	150
Repetitions[31:0]	4
Frequency_Hz[31:0]	10000000
phase_part_1[63:0]	107374182
phase_part_2[31:0]	85899345
DAC_output	0
DDS_output	8190
TXG	1
RXG	0
state[2:0]	0
resync	0
pulse_era[31:0]	2
rep_tracker[31:0]	3

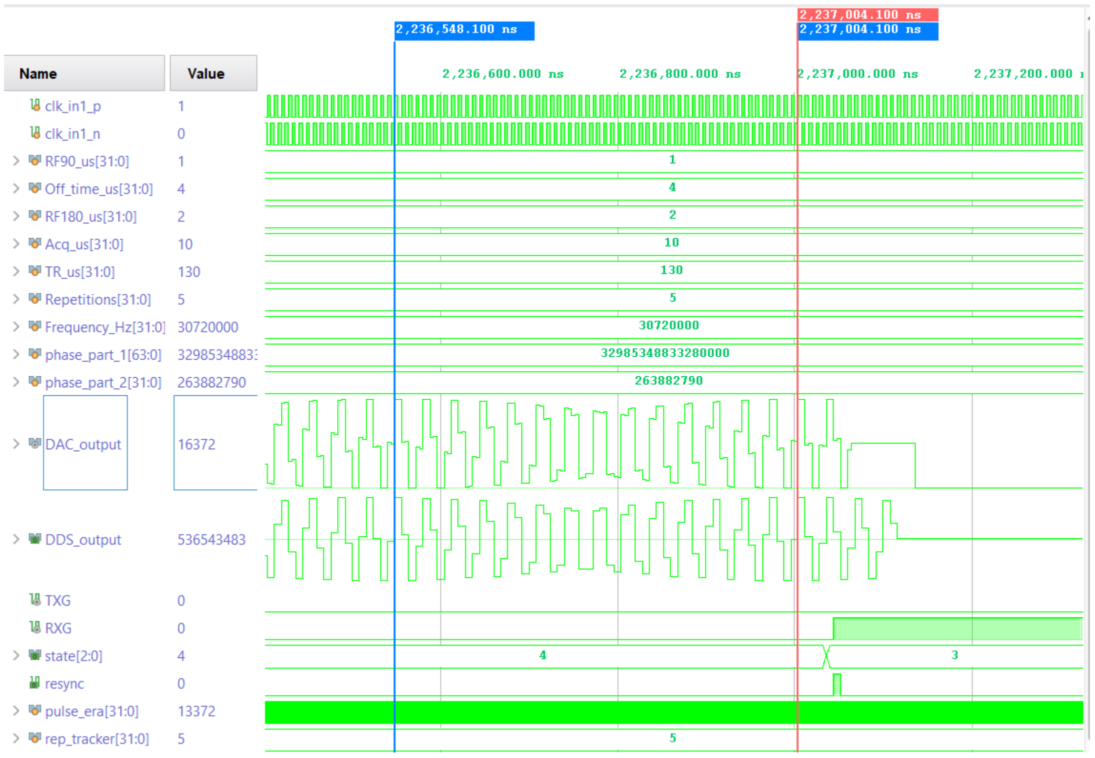
Figure 20: Timings for third TR – sequence #3

Chain of timed events	Expected time marker	Simulated time marker
#800_us + #5_us + (TR_3 × 1)	955.000 µs	955.020 µs
#800_us + #5_us + (TR_3 × 1) + dead_time	1055.000 µs	1055.0921 µs
#800_us + #5_us + (TR_3 × 1) + (dead_time + $RF90_3$ + Off_time_3 + $RF180_3$)	1086.000 µs	1086.020 µs
#800_us + #5_us + (TR_3 × 2)	1105.000 µs	1105.028 µs

Table 6: Comparing time markers in Figure 20 with expected values

As seen from values in tables 3 and 4, the largest deviation from ideal time markers is 0.4441 us for sequence #1 and 0.316 us for sequence #3.

4.2.1.3 Sequence #5



Name	Value
clk_in1_p	1
clk_in1_n	0
RF90_us[31:0]	1
Off_time_us[31:0]	4
RF180_us[31:0]	2
Acq_us[31:0]	10
TR_us[31:0]	130
Repetitions[31:0]	5
Frequency_Hz[31:0]	30720000
phase_part_1[63:0]	32985348833
phase_part_2[31:0]	263882790
DAC_output	16372
DDS_output	536543483
TXG	0
RXG	0
state[2:0]	4
resync	0
pulse_era[31:0]	13372
rep_tracker[31:0]	5

Figure 21: Sequence #5 – end of RF180 of last TR

Here, the graph is zoomed to the end of the last RF180 pulse in the last TR for sequence #5. The set frequency (input) equals 30,720,000 Hz = 30.72 MHz and calculated phase increment = 263882790.7. Discarding digits beyond the decimal point gives 263882790 – this matches the computed phase increment. This phase increment value gives a frequency of 30.719 MHz – quite close to the requested 30.72 MHz. Next, the simulated frequency is 14×10^9/(2237004.1-2236548.1) = 30.701 MHz – this is slightly different from our set frequency value, but close nonetheless. It is tricky to verify the correctness of this pulse frequency in simulations alone because lesser number of samples are used to represent a sine wave of this frequency as compared to a 2.55 MHz or 10 MHz wave.

In figures 16 and 19, the DAC output appears to be a rectangular wave modulated with a sine function – notice how it rises for one clock cycle and 'dips' every clock cycle. In figure 21, the DAC output not only appears like a random waveform rather than a sine, but the DDS output does not resemble a 30.72 MHz sine at all. The reason for these outputs not being consistent with sine waves is due to two factors – frequency resolution of the DDS and the behavior of the DAC IP created by Pavel.

Frequency resolution is given by $f_{clk}/2^{B\theta}$, with f_{clk} being the clock frequency of the system, 125 MHz and Bθ being the phase width, i.e., 30. This gives a frequency resolution of $125 \times 10^6/2^{30}$ = 0.1164 Hz. This corresponds to 8.58 ns – a sine wave with time period comparable to this value will not appear precise in simulations. That is why 30.72 MHz has a DDS output appearing random instead of a repeatable sinusoid.

Pavel's DAC IP works by toggling between two values for every non-zero DDS input value. This is the expected behavior of the DAC, as seen in figures 16 and 19. An analog output stage consisting of a lowpass filter (LPF) exists between the 14-bit DAC output port of our digital circuit and the SMA connector of OUT1 that causes the DAC IP output to 'smoothen out' into a clean sinusoid as seen in oscilloscope results after bitstream generation.

4.2.1.4 'Blanking out' after a scan

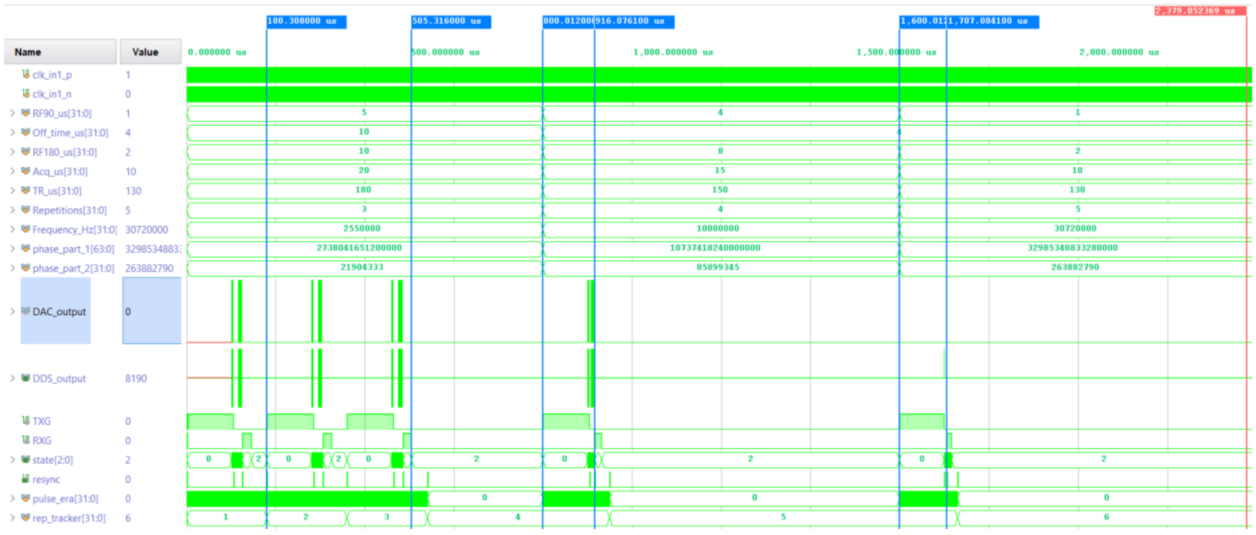


Figure 22: Simulation results without blank-out sequences

It was mentioned previously that requesting a new set of pulses without ‘blanking out’, i.e., setting all parameters to zero, will cause the circuit to behave unexpectedly. To demonstrate this, all blocks of testbench code after 800 us # delays up to 5 us # delays were commented out, essentially suppressing sequences #2 and #4. The result is shown above – rep_tracker is not reset after accepting a new set of inputs, causing the corresponding sequence to run less than the requested number of repetitions. Hence, it is important to perform a ‘blank out’ operation on the inputs every time a new sequence of RF and GPIO pulses must be generated.

Overall, this set of simulations shows that all RTL codes were written and integrated properly into the system. Notice that small values of RF90_us, Off_time_us, RF180_us and Acq_us were chosen. This was done on purpose to allow simulation results to be generated quickly and without overloading the computer. Although the DAC output did not represent a sine waveform here, it was later verified that sine pulses were indeed generated in the indicated green areas of the waveform.

4.2.2 After bitstream – Testing with PC inputs

To check for time lag between GPIO lines and OUT1, dead_time was temporarily set to zero, the corresponding bitstream generated, and TXG and OUT1 from the bitstream-programmed Pitaya were monitored with an Agilent oscilloscope. As seen in figure 7, the time lag between the two was 84 ns. In other words, all frequencies beyond 11.9 MHz have their RF channel lagging from the GPIO by at least one clock cycle. However, this lag remains constant, making it a predictable and quantifiable lag with every repetition or parameter set.

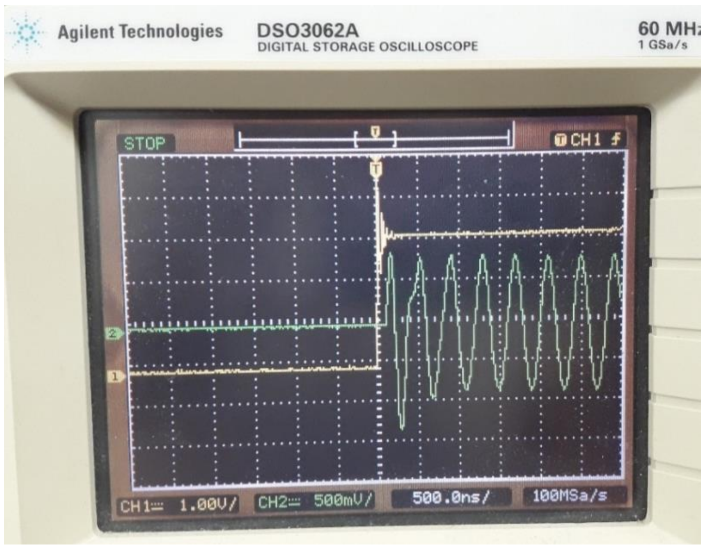
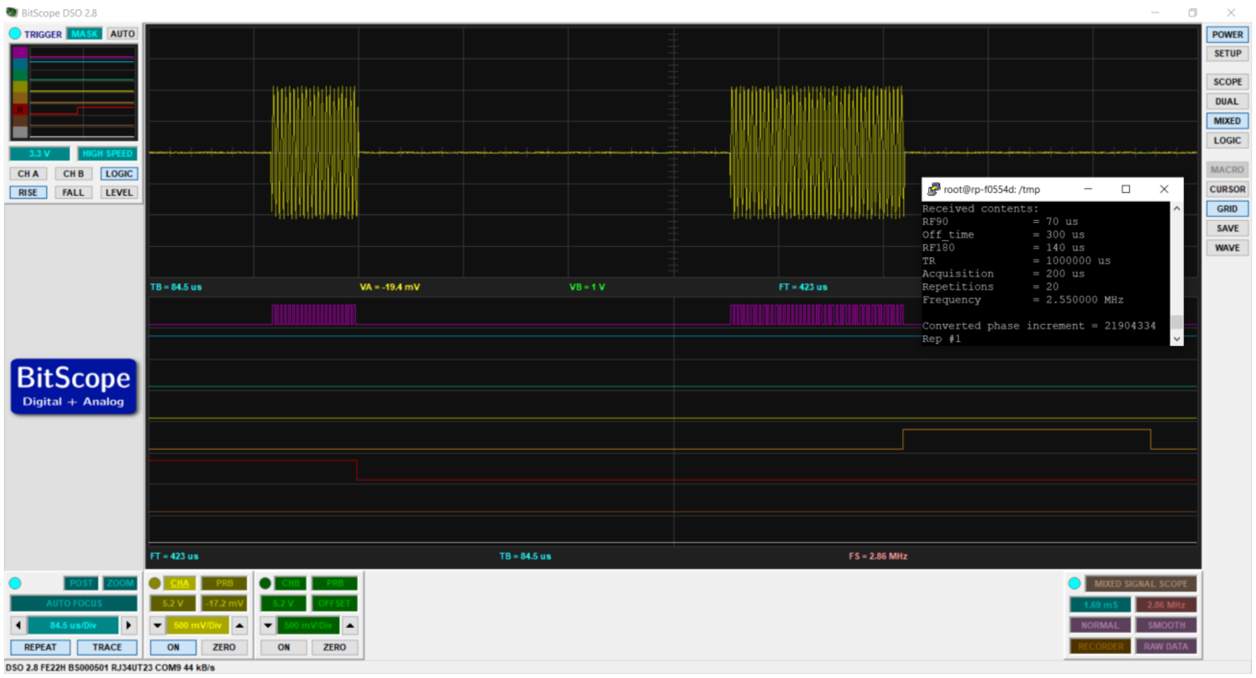


Figure 23: Lag between TXG and RF90 (keeping dead_time = 0)

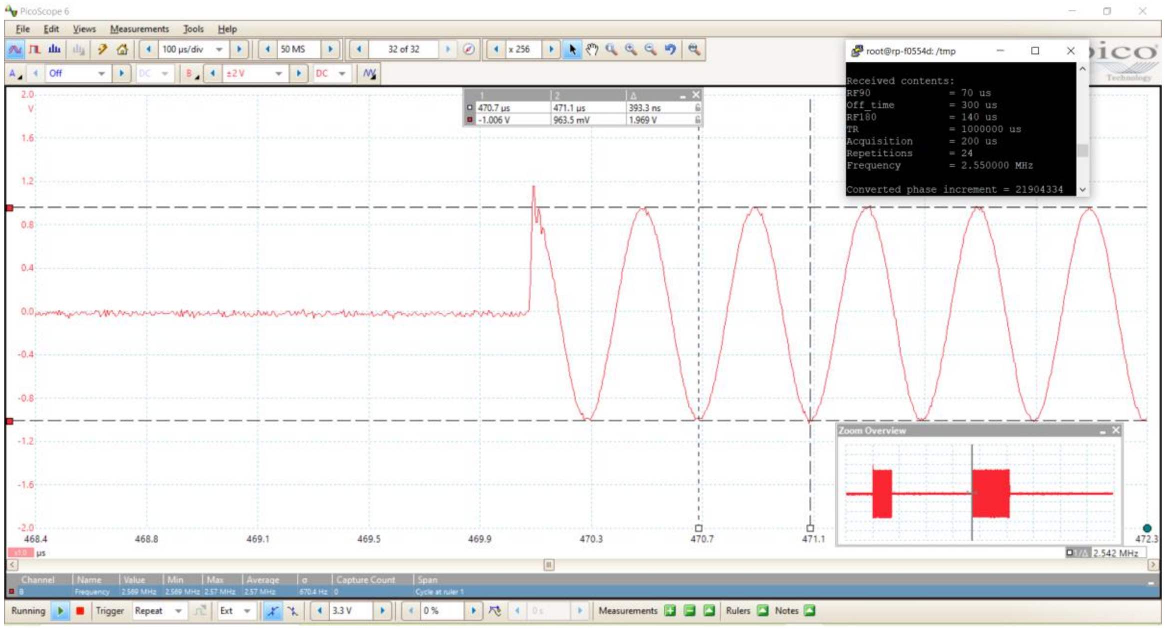
Next, OUT1 channel and GPIO lines were recorded over a BitScope DSO, as seen in figure 8. OUT1 IS the yellow waveform, the red and orange lines in the bottom half corresponded to TXG and RXG respectively.



```
Received contents:
RF90                = 70 us
Off_time            = 300 us
RF180               = 140 us
TR                  = 1000000 us
Acquisition         = 200 us
Repetitions         = 20
Frequency           = 2.550000 MHz

Converted phase increment = 21904334
Rep #1
```

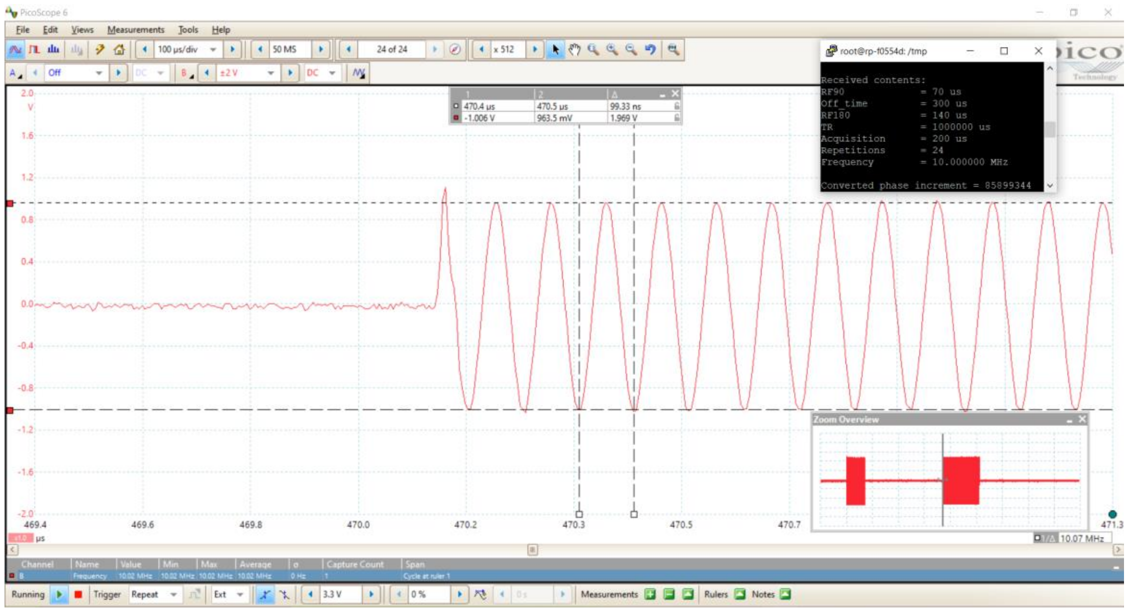
Figure 24: Pitaya outputs (OUT1, TX, RXG) captured with BitScope DSO

Next, to observe frequency and amplitude, the PicoScope was used. TXG was used as a trigger signal for this, giving the results below. In all output waveforms, the voltage is found to be 1.969 Vpp.



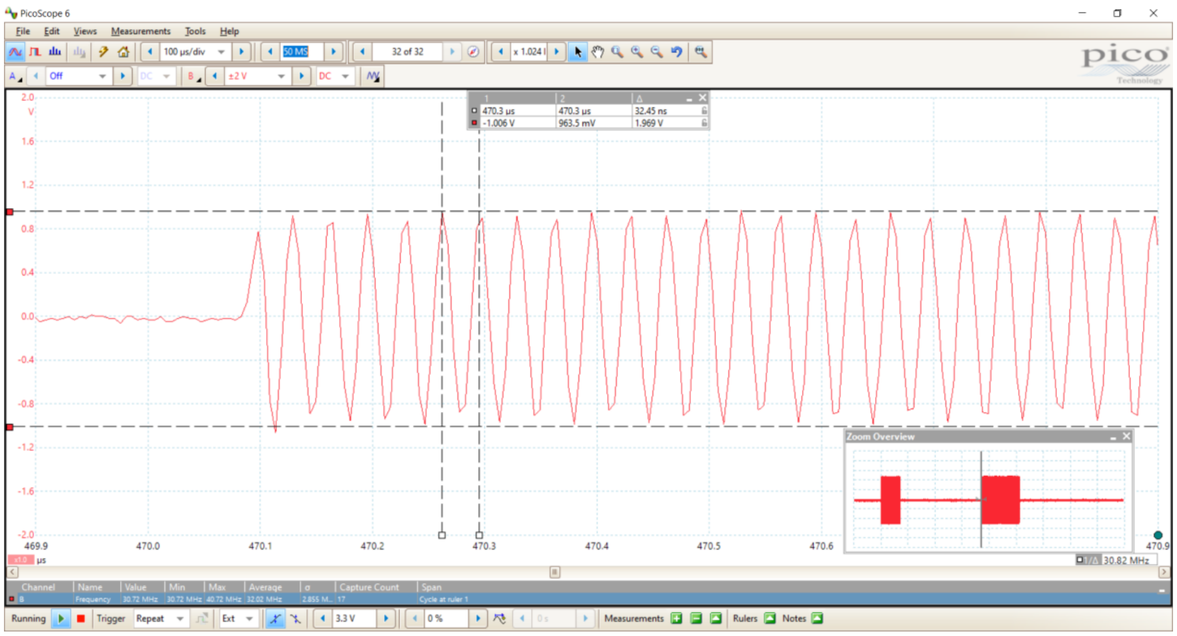
```
root@rp-f0554d: /tmp

Received contents:
RF90                = 70 us
Off_time            = 300 us
RF180               = 140 us
TR                  = 1000000 us
Acquisition         = 200 us
Repetitions         = 24
Frequency           = 2.550000 MHz

Converted phase increment = 21904334
```

Figure 25: Pitaya RF output for 2.55 MHz captured with PicoScope



```
root@rp-f0554d: /tmp

Received contents:
RF90                = 70 us
Off_time            = 300 us
RF180               = 140 us
TR                  = 1000000 us
Acquisition         = 200 us
Repetitions         = 24
Frequency           = 10.000000 MHz

Converted phase increment = 85899344
```

Figure 26: Pitaya RF output for 10 MHz captured with PicoScope



	1	2	Δ
☐	470.3 µs	470.3 µs	32.45 ns
🟥	-1.006 V	963.5 mV	1.969 V

Channel	Name	Value	Min	Max	Average	σ	Capture Count	Span
B	Frequency	30.72 MHz	30.72 MHz	40.72 MHz	32.02 MHz	2.855 M...	17	Cycle at ruler 1

Figure 27: Pitaya RF output for 30.72 MHz captured with PicoScope

Since PicoScope has a sampling rate of 50 MSa/s, it under-samples the 30.72 MHz signal, making it appear jagged instead of sinusoidal. However, it is noted in later sections that the Pentek digitizes this perfectly, giving an echo of 30.72 MHz.

When looking at the starting phase of all RF90 and RF180 pulses, all three sequences for PicoScope have their RF90 and RF180 starting at the same phase value. This indicates that overall, the output matches the input parameters, and the pulse generator is phase stable.

This demonstrated that Pitaya was now fully programmed for pulse generation and triggering acquisition events and could be integrated with a PC and RF frontends to create a working MR setup for spin echoes.

4.2.3 Testing on MR setup

In this setup, Pitaya was connected to an MR setup with a 4.7 T permanent magnet and the receive side was analyzed using a Pentek digitizer with TXG as the triggering signal. This was done to capture the transmitted pulses as well as the echo generated by the 2H sample.

Using the MATLAB codes attached to the appendix of this document, the captured data of all 20 TRs were superimposed on top of each other. The superimposed data was zoomed in to check how well the RF90s and RF180s overlap.

There is a 125 ns difference between the time when the RF90 is expected to show up and the actual sequences. In other words, each RF90 starts 125 ns after the 100 us time step in our waveform. Within the RF90 pulses itself there is a jitter of 5 ns. Upon zooming in the RF90 end and ringdown, there is a 50 ns delay between expected and actual RF90 end, and a delay of 125 ns between RF90 end and the beginning of ringdown. Jitter at RF90 end and ringdown stays the same, at 5 ns.

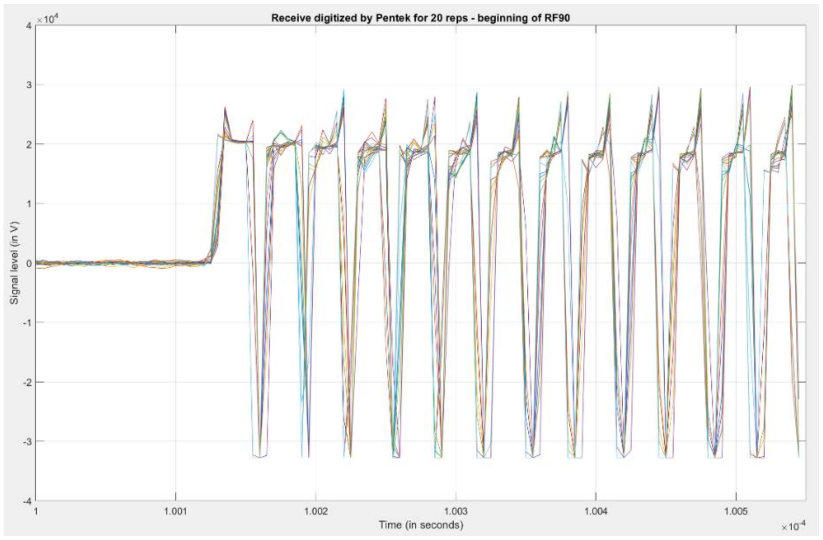


Figure 28: Beginning of RF90 – 20 acquisitions superimposed, zoomed in

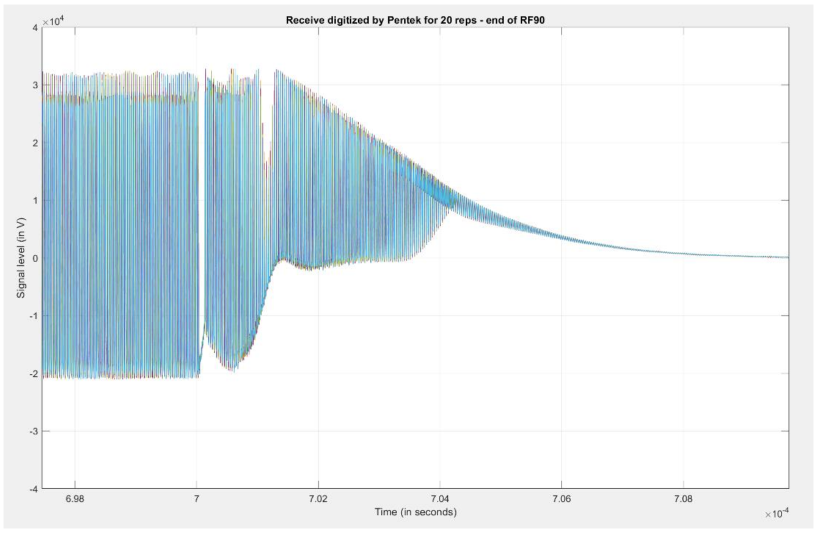


Figure 29: End of RF90 – 20 acquisitions superimposed – zoomed out

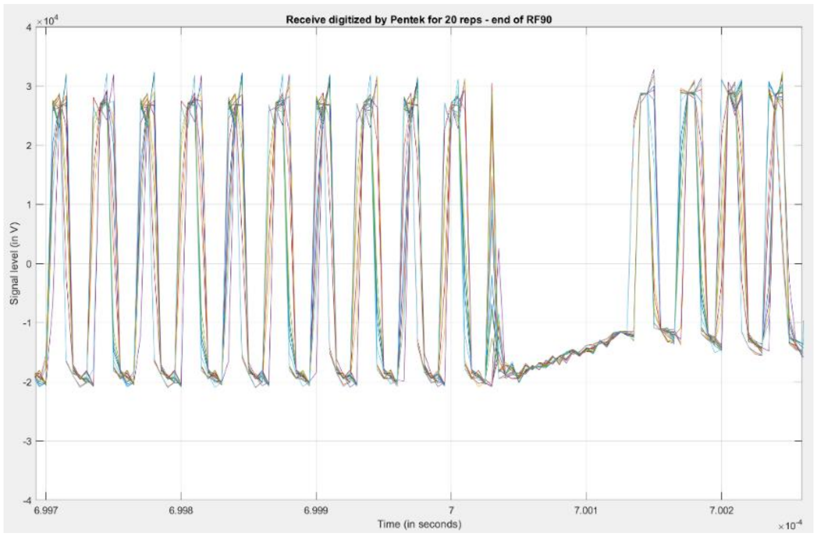


Figure 30: End of RF90 and ringdown, zoomed in

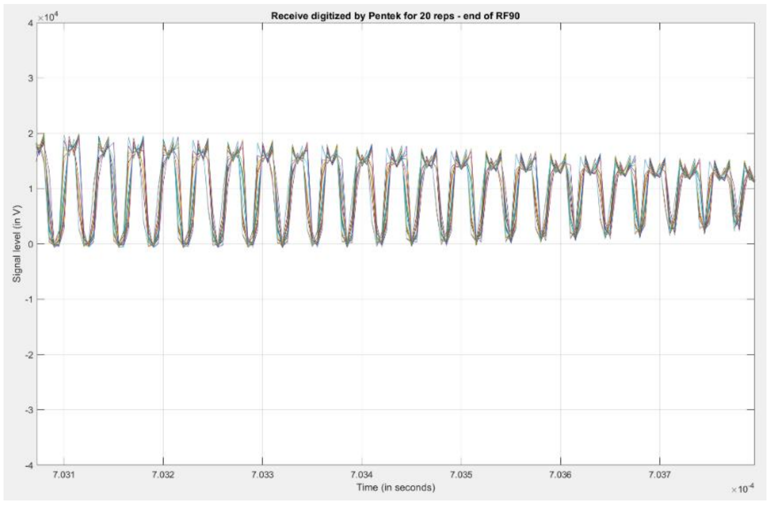


Figure 31: End of RF90 – pure ringdown, zoomed in

The beginning and end of RF180 were analyzed as well. RF180 starts 35 ns before expected time and end 5 ns before expected time. The jitter is very close to the value before, at 6 ns. The delay between RF180 end and its ringdown is 90 ns - less than that of RF90 and its ringdown. This shows that all the RF pulses generated are phase stable.

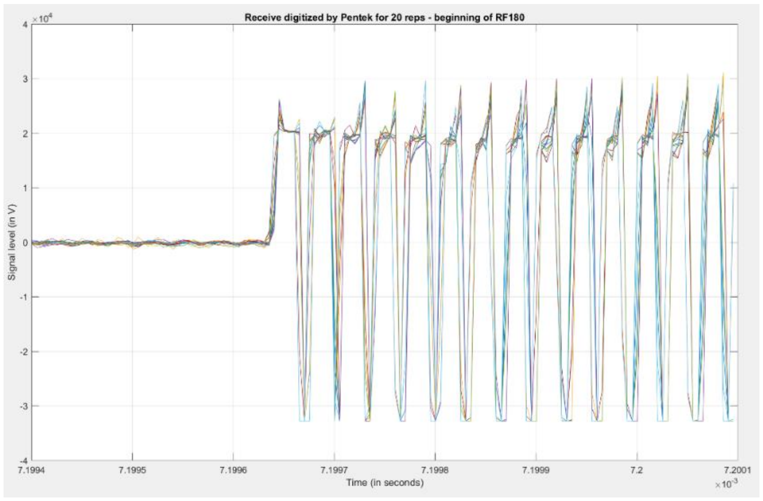


Figure 32: Beginning of RF180, zoomed in

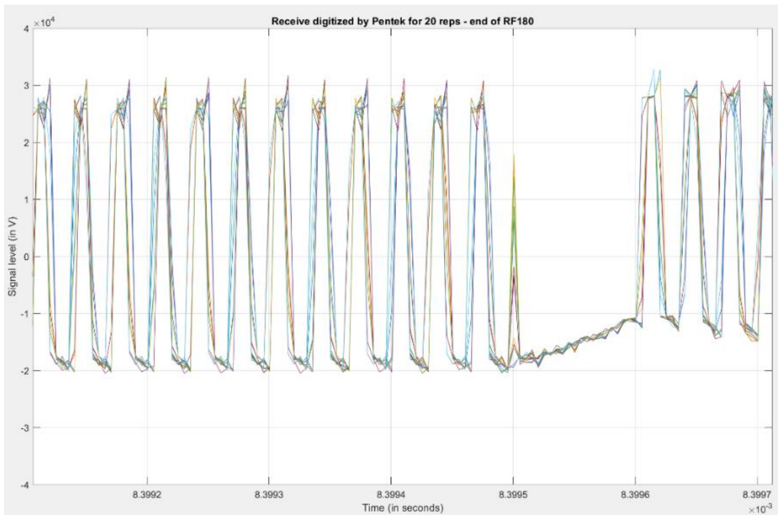


Figure 33: End of RF180 and ringdown, zoomed in

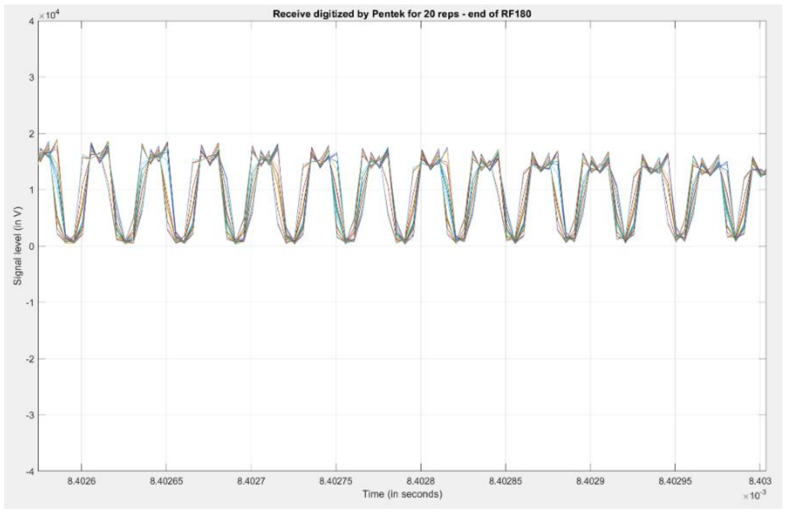


Figure 34: End of RF180 – pure ringdown, zoomed in

Finally, all data between 13.2 ms and 17.2 ms was cropped out. Based on inputted parameters, 15.2 ms is where the echo peak will show up. This was Fourier Transformed to give its spectrum. As seen in the figures, the magnitudes of all spectra superimposed perfectly on top of each other at Larmor frequency, with a distinct peak showing up for all the acquisitions at 30.7226 MHz. The phases, however, do not seem to align well. At 30.7229 MHz, it is seen that the phases vary from 1.2 radians to 2.2 radians, which is a large phase deviation between acquisitions.

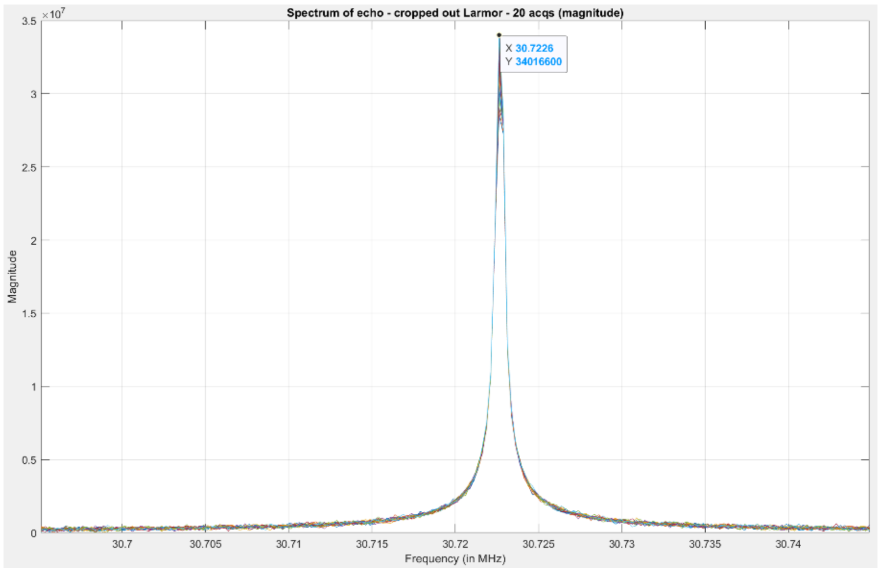


Figure 35: Magnitude of spectra acquired with Pentek at Larmor

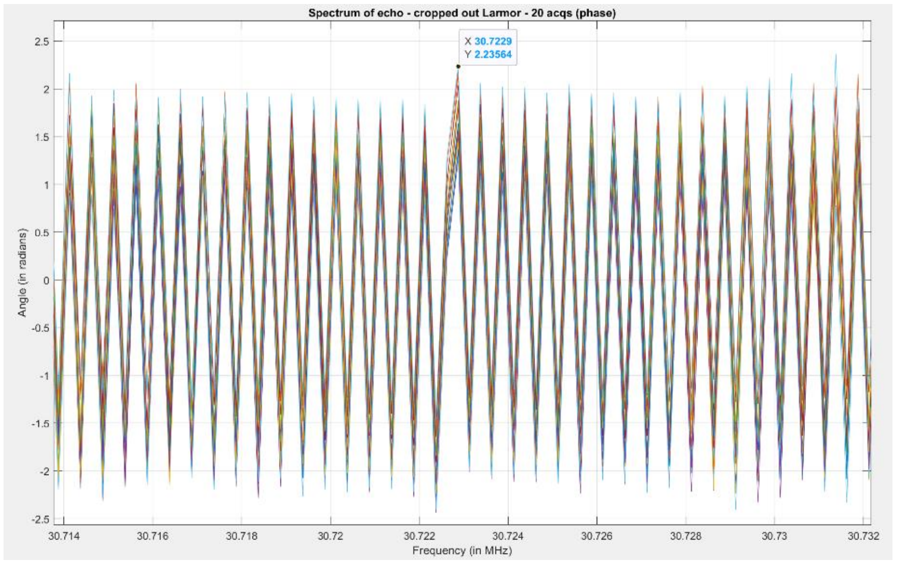


Figure 36: Phase of spectra acquired with Pentek – zoomed at Larmor

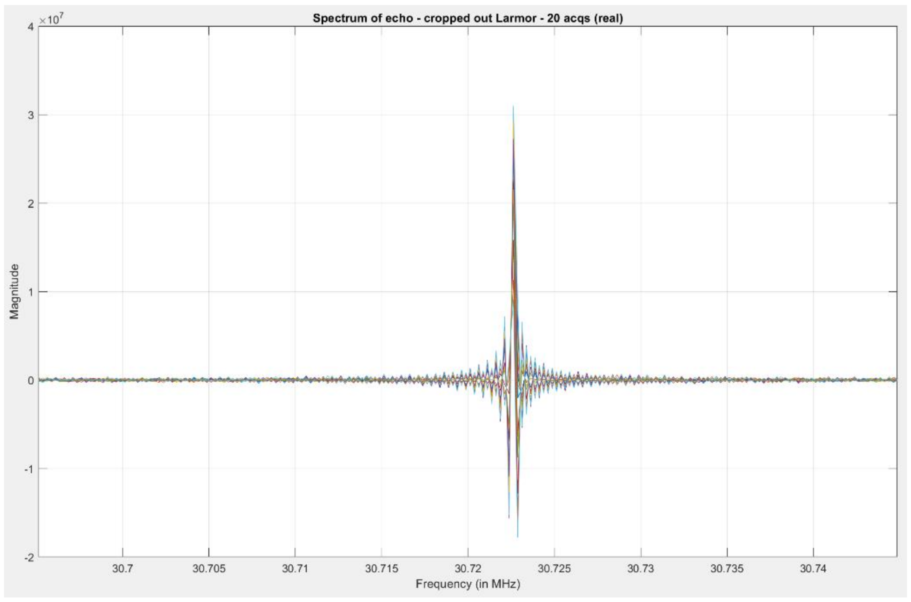


Figure 37: Real part of spectra acquired with Pentek – zoomed at Larmor

A close inspection of the real part of echo spectra further confirms the phase instability of the setup since they do not align well. It is very likely the echoes are not phase stable due to Pentek not being perfectly synchronized with Pitaya. Because of this, it is not possible to create a phase stable system with MR2. If this setup is to generate phase stable echoes, then Pentek must be synchronized with Pitaya, preferably with a shared clock signal.

However, figures 28-34 indicate that the RF pulse generation (and hence Pitaya) is phase stable since the RF90 and RF180 in each repetition superimpose well in the time domain. Therefore, the complete system is controllable with a Pitaya configured with the author’s digital circuit and interface, and capable of producing echoes with a high-speed digitizer. While the RF pulses are phase stable, the MR echoes are not phase stable because the digitizer is not synchronized with the pulse generator.

4.3 Future work

Future developments of this project may involve digitization performed on the Pitaya itself using BRAM IP cores as part of the digital circuit. This is quite beneficial as far as phase stability of the echoes (and hence the entire system) is concerned since this allows for the digitizer and pulse generator to coexist side by side on the same hardware platform and be perfectly synchronized. With improved synchronization comes phase stability of MR echoes, which is highly desirable. The author notes that this functionality is not trivial to implement and requires sound knowledge of memories and TCP/IP to ensure correct implementation of data storage and integration with AX14-Stream Red Pitaya ADC, an IP core that converts RF input to 14-bit digital output, ready to be stored/transferred elsewhere for further processing.

Coming to data processing, this feature can be implemented upon successful deployment of digitization on Pitaya itself. Although this will make the circuit more complex, this will greatly reduce equipment cost since an external PC and digitizer will no longer be required for data processing – simply generate RF pulses and triggers from Pitaya, digitize the receive, and process the raw data to obtain the echo information. The final echo information may be streamed out to the client program which will save it on the client PC.

It will be beneficial to add gradient pulse generation to this system as well. Since Pitaya has only two RF output channels, OUT1 and OUT2, out of which the former is already used for RF pulses, that leaves only OUT2 as a gradient pulse channel. This in conjunction with a gradient coil can be used to create projection images of samples, just like the ones created with the MR hardware in ECEN 463/763 using a planar gradient coil. To generate a complete 2D image, one RF channel and three gradient channels are required – one for slice select, one for frequency encode/readout and one for phase encode. This can be done by using two Pitaya boards at the same time – this system will have 4 output channels which is sufficient for the task. For this to work, it is important to ensure both Pitaya boards are perfectly synchronized.

5. CONCLUSION

The program codes in Verilog, C and Python, and the manual created in this project are intended to guide students and researchers towards digital logic deployment in MR instrumentation. By incorporating them in MR setups, students will gain hands-on experience in RF pulse generation via digital circuits and appreciate its importance in MR applications. More importantly, it will encourage them to formulate solutions to existing MR and RF problems in terms of programmable logic and develop skills in FPGA prototyping. The digital circuit design, server and client programs and testbench code serve as a baseline for researchers to create and add more features to the circuit and interface for future MR experiments.

It is meant to act as system upgrade to the existing MR setup used in ECEN 463/763 – Magnetic Resonance Engineering, at Texas A&M University, along with setting a baseline for researchers to utilize programmable logic in signal generation through a Pitaya development board. The results indicate that this signal generator is well suited for the task, seeing that its output matches user inputs well, is easy to control and assemble, has phase stable outputs and can be expanded to build more sophisticated setups for MRI experiments.

REFERENCES

[1] C. Thompson, "Upgrading obsolete integrated circuits using Field Programmable Gate Arrays (FPGA)," in *2014 IEEE AUTOTEST*, 2014-09-01 2014: IEEE, doi: 10.1109/autest.2014.6935173.

[2] M. M. Balas, "A New Generation of Biomedical Equipment: FPGA," in *Advances in Intelligent Analysis of Medical Data and Decision Support Systems*: Springer International Publishing, 2013, pp. 235-246.

[3] *Motorola Semiconductor Data Book*, Fourth Edition. Motorola Inc. 1969. p. IC-73.

[4] *XCELL*, Issue 32, Second Quarter, Xilinx Inc. 1999. p. 4. Accessed on: June 11, 2021 [Online]. Available: https://www.xilinx.com/publications/archives/xcell/Xcell32.pdf

[5] J. J. Rodriguez-Andina, M. J. Moure, and M. D. Valdes, "Features, Design Tools, and Application Domains of FPGAs", *IEEE Transactions on Industrial Electronics,* vol. 54, no. 4, pp. 1810–1823, 2007.

[6] Kaufman et al., "Switchable MRI RF Coil Array with Individual Coils Having Different and Overlapping Fields of View", U.S Patent 4881034, Nov. 14, 1989. Accessed on June 12, 2021 [Online]. Available: https://patents.google.com/patent/US4881034.

[7] Wood, A. "Magnetic venture: the story of Oxford Instruments", *Oxford University Press,* ISBN 0- 19-924108-2, 2001.

[8] P. Mansfield and A. A. Maudsley, "Planar spin imaging by NMR", *Journal of Magnetic Resonance (1969)*, vol. 27, no. 1, pp. 101–119, 1977.

[9] A. Kumar, D. Welti, and R. R. Ernst, "NMR Fourier zeugmatography", *Journal of Magnetic Resonance (1969)*, vol. 18, no. 1, pp. 69–83, 1975.

[10] J.E. Berlien, "Engineering MR Technology for Low-Cost Portable Device Design", Honors and Undergraduate Research, Texas A&M University, College Station, TX, USA May 2016. Accessed on: June 29, 2021 [Online]. Available: https://hdl.handle.net/1969.1/167845.

[11] I. Kang, "A portable, low-cost, 3D-printed main magnetic field system for magnetic imaging," *2017 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2017, pp. 3533-3536, doi: 10.1109/EMBC.2017.8037619.

[12] Z. H. Ren, W. C. Mu, and S. Y. Huang, "Design and Optimization of a Ring-Pair Permanent Magnet Array for Head Imaging in a Low-Field Portable MRI System", *IEEE Transactions on Magnetics*, vol. 55, no. 1, pp. 1–8, 2019.

[13] S. Huang, Z. H. Ren, S. Obruchkov, J. Gong, R. Dykstra, and W. Yu, "Portable Low-Cost MRI System Based on Permanent Magnets/Magnet Arrays", *Investigative Magnetic Resonance Imaging,* vol. 23, no. 3, p. 179, 2019.

[14] G. Moresi and R. Magin, "Miniature permanent magnet for table-top NMR", *Concepts in Magnetic Resonance,* vol. 19B, no. 1, pp. 35–43, 2003.

[15] C. W. Windt, H. Soltner, D. V. Dusschoten and P. Blumler, "A portable Halbach magnet that can be opened and closed without force: The NMR-CUFF," *Journal of Magnetic Resonance,* vol. 208, pp. 27-33, 2011.

[16] C.Z. Cooley, "Portable low-cost magnetic resonance imaging", Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, Aug. 2014. Accessed on: Jun. 29, 2021. [Online]. Available: https://dspace.mit.edu/handle/1721.1/93060

[17] C. Z. Cooley, J. P. Stockmann, B. D. Armstrong, M. Sarracanie, M. H. Lev, M. S. Rosen, and L. L. Wald, "Two-dimensional imaging in a lightweight portable MRI scanner without gradient coils", *Magnetic Resonance in Medicine*, vol. 73, no. 2, pp. 872–883, 2015.

[18] M. Sarracanie, C. D. Lapierre, N. Salameh, D. E. J. Waddington, T. Witzel, and M. S. Rosen, "Low-Cost High-Performance MRI", *Scientific Reports*, vol. 5, no. 1, p. 15177, 2015.

[19] N. R. Routley and K. J. Carlton, "The HALO system—a light weight portable imaging system", *Magnetic Resonance Imaging*, vol. 22, no. 8, pp. 1145–1151, 2004.

[20] Z. H. Ren, S. Obruchkov, D. W. Lu, R. Dykstra, and S. Y. Huang, "A low-field portable magnetic resonance imaging system for head imaging", 2017.

[21] F. J. Mateen, C. Z. Cooley, J. P. Stockmann, D. R. Rice, A. C. Vogel, and L. L. Wald, "Low-field portable brain MRI in CNS demyelinating disease", *Multiple Sclerosis and Related Disorders*, vol. 51, p. 102903, 2021.

[22] K.-M. Lei, D. Ha, Y.-Q. Song, R. M. Westervelt, R. Martins, P.-I. Mak, and D. Ham, "Portable NMR with Parallelism", *Analytical Chemistry*, vol. 92, no. 2, pp. 2112–2120, 2020.

[23] W.-H. Chang, J.-H. Chen, and L.-P. Hwang, "Single-sided mobile NMR with a Halbach magnet", *Magnetic Resonance Imaging*, vol. 24, no. 8, pp. 1095–1102, 2006.

[24] T. Kimura, Y. Geya, Y. Terada, K. Kose, T. Haishi, H. Gemma, and Y. Sekozawa, "Development of a mobile magnetic resonance imaging system for outdoor tree measurements", Review of Scientific Instruments, vol. 82, no. 5, p. 053704, 2011.

[25] L. L. Wald, P. C. Mcdaniel, T. Witzel, J. P. Stockmann, and C. Z. Cooley, "Low-cost and portable MRI", *Journal of Magnetic Resonance Imaging*, vol. 52, no. 3, pp. 686–696, 2020.

[26] D. W. Lu and S. Y. Huang, "A TSVD-based approach for flexible spatial encoding strategy in portable Magnetic Resonance Imaging (MRI) system", 2017.

[27] M.S. Wijchers, "Image Reconstuction in MRI: The Possibilities of Portable Low-cost MRI Scanners", M.S. thesis, Delft University of Technology, Mekelweg 5, 2628 CD Delft, Netherlands, Aug. 2016. Accessed on: June 29, 2021 [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid:0a0685fc-ce0a-4580-a355-978810005b71.

[28] T. O'Reilly and A. Webb, "Deconstructing and reconstructing MRI hardware", *Journal of Magnetic Resonance*, vol. 306, pp. 134–138, 2019.

[29] S. E. Ogier, “Improvements in Low Field MRI”, Honors and Undergraduate Research, Texas A&M University, College Station, TX, USA May 2013. Accessed on: June 12, 2021 [Online]. Available: https://hdl.handle.net/1969.1/148860.

[30] L. M. Broche, P. J. Ross, G. R. Davies, and D. J. Lurie, “Simple algorithm for the correction of MRI image artefacts due to random phase fluctuations”, *Magnetic Resonance Imaging*, vol. 44, pp. 55–59, 2017.

[31] K. J. Layton *et al.*, "Pulseq: A rapid and hardware-independent pulse sequence prototyping framework," *Magnetic Resonance in Medicine,* vol. 77, no. 4, pp. 1544-1552, 2017-04-01 2017, doi: 10.1002/mrm.26235.

[32] C. C. Bauer, "Adaptation of Open Source Pulseq Pulse Sequence Writing Toolbox for Varian Compatibility", M.S. thesis, Texas A&M University, College Station, TX, USA, Dec. 2018. Accessed on: May. 4, 2021 [Online]. Available: https://oaktrust.library.tamu.edu/bitstream/handle/1969.1/174608/BAUER-THESIS-2018.pdf?sequence=1&isAllowed=y

[33] C. A. Michal, "A low-cost multi-channel software-defined radio-based NMR spectrometer and ultra-affordable digital pulse programmer," *Concepts in Magnetic Resonance Part B: Magnetic Resonance Engineering,* vol. 48B, no. 3, p. e21401, 2018-07-01 2018, doi: 10.1002/cmr.b.21401.

[34] L. Li and A. M. Wyrwicz, "Design of an MR image processing module on an FPGA chip," *Journal of Magnetic Resonance,* vol. 255, pp. 51-58, 2015-06-01 2015, doi: 10.1016/j.jmr.2015.03.007.

[35] S. A. Qazi, M. F. Siddiqui, J. Jacob Wikner, and H. Omer, “ASIC modelling of SENSE for parallel MRI”, *Computers in Biology and Medicine*, vol. 109, pp. 53–61, 2019.

[36] H.-Y. Chen, Y. Kim, P. Nath, and C. Hilty, "An ultra-low cost NMR device with arbitrary pulse programming," *Journal of Magnetic Resonance,* vol. 255, pp. 100-105, 2015-06-01 2015, doi: 10.1016/j.jmr.2015.02.011.

[37] K. Takeda, "OPENCORE NMR: Open-source core modules for implementing an integrated FPGA-based NMR spectrometer," *Journal of Magnetic Resonance,* vol. 192, no. 2, pp. 218-229, 2008-06-01 2008, doi: 10.1016/j.jmr.2008.02.019

[38] S. Anand, "OCRA: A Low-Cost, Open-Source FPGA-Based MRI Console Capable of Real-Time Control," M.E. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, Sept. 2018. Accessed on: May. 1, 2021. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/121619/1098041021-MIT.pdf?sequence=1&isAllowed=y

[39] S. M. Wright, M. P. Mcdougall, and J. C. Bosshard, "A desktop imaging system for teaching MR engineering," in *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*, 2010-08-01 2010: IEEE, doi: 10.1109/iembs.2010.5627156.

[40] J. B. W. Webber and P. Demin, "Credit-card sized field and benchtop NMR relaxometers using field programmable gate arrays," *Magnetic Resonance Imaging,* vol. 56, pp. 45-51, 2019-02-01 2019, doi: 10.1016/j.mri.2018.09.018.

[41] "3.1.1.2.2. Extension — Red Pitaya 0.97 documentation", Redpitaya.readthedocs.io, 2021. Accessed on: August 27, 2021 [Online] Available: https://redpitaya.readthedocs.io/en/latest/developerGuide/hardware/125-14/extent.html

APPENDIX A

SERVER AND CLIENT PROGRAMS

1. server_program.c

```c
#include <sys/socket.h>
#include <arpa/inet.h> //inet_addr
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>    //write
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>

#pragma pack(1)

typedef struct payload_t {
/* NOTE: All these are inputs are 32 bits long and in microseconds, taken from the
Python client. We get these from the client and send to bitstream-configured GPIO
- the first stage of our digital circuit */

    uint32_t RF90;
    uint32_t Off_time;
    uint32_t RF180;
    uint32_t TR;
    uint32_t Acq;
    uint32_t reps;
    float frequency;
} payload;

#pragma pack()

int main(int argc, char** argv)
{
    //Step 1: Initialize socket stuff
    int PORT = 2300;
    int BUFFSIZE = 512;
    char buff[BUFFSIZE];
    int ssock, csock;
    int nread;
    struct sockaddr_in client;
    int clilen = sizeof(client);
    //Step 2: Initialize bitstream control stuff
    int fd;
    char *name = "/dev/mem";
    void *cfg_90_and_Off_time, *cfg_180_TR, *cfg_Phase_inc_and_reps, *cfg_Acq; /*
pointers to address/memory locations used by bitstream's GPIO*/
    uint32_t phase_inc = 1<<30; //fed to DDS
    uint32_t RF90, Off_time, RF180, TR, Acq, reps; float frequency; //we get thes
e from the Python client
    int freq_MHz = 125; // 125 MHz - multiply with pulse timing parameters

   if((fd = open(name, O_RDWR)) < 0)
  {
    perror("open");
    return EXIT_FAILURE;
  }

  /* Mapping to addresses specified by bitstream's AXI GPIO
  (Please refer to your block design's 'Address Editor' in Vivado for this) */
  cfg_90_and_Off_time = mmap(NULL, sysconf(_SC_PAGESIZE),PROT_READ|PROT_WRITE, MA
P_SHARED, fd, 0x41210000);
  cfg_180_TR = mmap(NULL, sysconf(_SC_PAGESIZE),PROT_READ|PROT_WRITE, MAP_SHARED,
 fd, 0x41220000);
  cfg_Phase_inc_and_reps = mmap(NULL, sysconf(_SC_PAGESIZE),PROT_READ|PROT_WRITE,
 MAP_SHARED, fd, 0x41230000);
  cfg_Acq = mmap(NULL, sysconf(_SC_PAGESIZE),PROT_READ|PROT_WRITE, MAP_SHARED, fd
, 0x41200000);

// Creating and binding a socket
    struct sockaddr_in server;

    if ((ssock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("ERROR: Socket creation failed\n");
        exit(1);
    }
    printf("Socket created\n");

    bzero((char *) &server, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(PORT);
    if (bind(ssock, (struct sockaddr *)&server , sizeof(server)) < 0)
    {
        printf("ERROR: Bind failed\n");
        exit(1);
    }
    printf("Bind done\n");
    listen(ssock, 3);
    printf("Server listening on port %d\n", PORT);

    while (1)
    {
        csock = accept(ssock, (struct sockaddr *)&client, &clilen);
        if (csock < 0)
        {
            printf("Error: accept() failed\n");
            continue;
        }

        printf("Accepted connection from %s\n", inet_ntoa(client.sin_addr));

        //Accepted socket connection. Now preparing to receive inputs from client
        bzero(buff, BUFFSIZE);
        while ((nread=read(csock, buff, BUFFSIZE)) > 0)
        {
            printf("Received %d bytes\n", nread);
            payload *p = (payload*) buff;

            //Print received inputs from Python client
            printf("\nReceived contents:\n");
            printf("RF90 \t\t = %d us\n", p->RF90);
            printf("Off_time \t = %d us\n", p->Off_time);
            printf("RF180 \t\t = %d us\n", p->RF180);
            printf("TR \t\t = %d us\n", p->TR);
            printf("Acquisition\t = %d us\n", p->Acq);
            printf("Repetitions\t = %d\n", p->reps);
            printf("Frequency\t = %0.6f MHz\n", p->frequency);

            RF90 = p->RF90;
            Off_time = p->Off_time;
            RF180 = p->RF180;
            TR = p->TR;
            Acq = p->Acq;
            reps = p->reps;
            frequency = p->frequency;
        }

    float val = 125/frequency;
    phase_inc /= val; //converting phase increment to final DDS value
    printf("\nConverted phase increment = %d", phase_inc);

    *((uint32_t *)(cfg_90_and_Off_time + 0)) = RF90*freq_MHz;   // convert RF90 t
o number of clocks
    *((uint32_t *)(cfg_90_and_Off_time + 8)) = Off_time*freq_MHz;   // convert Of
f_time to number of clocks
    *((uint32_t *)(cfg_180_TR + 0)) = RF180*freq_MHz;   // convert RF180 to numbe
r of clocks
    *((uint32_t *)(cfg_180_TR + 8)) = TR*freq_MHz;   // convert TR to number of c
locks
    *((uint32_t *)(cfg_Acq + 0)) = Acq*freq_MHz;   // send acquisition
    *((uint32_t *)(cfg_Phase_inc_and_reps + 0)) = phase_inc;   // send phase incr
ement
    *((uint32_t *)(cfg_Phase_inc_and_reps + 8)) = reps;   // send repetitions


    for(int count_reps= 1; count_reps <= reps; count_reps ++){
        printf("\nRep #%d", count_reps);
        sleep(TR/1000000); //delay in seconds
    }
    printf("\nNow blanking out the pulse parameters...");
    *((uint32_t *)(cfg_90_and_Off_time + 0)) = 0;   // convert RF90 to number of
clocks
    *((uint32_t *)(cfg_90_and_Off_time + 8)) = 0;   // convert Off_time to number
 of clocks
    *((uint32_t *)(cfg_180_TR + 0)) = 0;   // convert RF180 to number of clocks
    *((uint32_t *)(cfg_180_TR + 8)) = 0;   // convert TR to number of clocks
    *((uint32_t *)(cfg_Phase_inc_and_reps + 0)) = 0;   // send phase increment
    *((uint32_t *)(cfg_Phase_inc_and_reps + 8)) = 0;   // send repetitions
    sleep(1);
    printf("\nOperation complete.");
    sleep(1);
    printf("\nClosing connection to client\n");
    sleep(1);
    printf("-----------------------------\n");
    sleep(1);
    close(csock);
    return 0;
    }
}
```

2. client_program.py

```
#!/usr/bin/env python

""" client.py - Echo client for sending/receiving C-like structs via socket
References:
- Ctypes fundamental data types:
https://docs.python.org/2/library/ctypes.html#ctypes-fundamental-data-types-2
- Ctypes structures:
https://docs.python.org/2/library/ctypes.html#structures-and-unions
- Sockets: https://docs.python.org/2/howto/sockets.html
"""

import socket
import sys
from datetime import datetime #to name the MR output file
from ctypes import *

""" This class defines a C-like struct """
class Payload(Structure):
    _fields_ = [("RF90", c_uint32), ("Off_time", c_uint32),
                ("RF180", c_uint32), ("TR", c_uint32),
                ("Acq", c_uint32), ("Reps", c_uint32),
                ("Frequency", c_float)]

def main():
    redpitaya = "169.254.217.146"
    #redpitaya = "169.254.156.79"
    server_addr = (redpitaya, 2300)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        s.connect(server_addr)
        print ("Connected to %s" % repr(server_addr))
    except:
        print ("ERROR: Connection to %s refused" % repr(server_addr))
        sys.exit(1)

    try:
            #Send data to server
            print ("")
            payload_out = Payload(140,12895,280,5210000,20000,20,30.72)
            print("Sending these params:")
            print("RF90 \t\t = %d us \nOff_time \t = %d us \nRF180 \t\t = %d
us \nTR \t\t\t = %d us"
                  "\nAcquisition\t = %d us\nRepetitions\t = %d \nFrequency\t
= %.6f MHz\n"
                                                  % (payload_out.RF90,
                                                     payload_out.Off_time,
                                                     payload_out.RF180,
                                                     payload_out.TR,
                                                     payload_out.Acq,
                                                     payload_out.Reps,
                                                     payload_out.Frequency
                                                     ))
            nsent = s.send(payload_out)
            # Alternative: s.sendall(...): continues to send data until
```

```
either
            # all data has been sent or an error occurs. No return value.
            print ("Sent %d bytes" % nsent)
            ct = datetime.now()
            stamp = ct.strftime("%b_%d_%Y_%I_%M_%S_%p")
            filename = "MR_data_" + stamp + ".txt"
            with open(filename, "a") as txt_file:
                txt_file.write('Sent out the following:\n')
                txt_file.write('RF90 = %d us\n' % (payload_out.RF90))
                txt_file.write('Off_time = %d us\n' % (payload_out.Off_time))
                txt_file.write('RF180 = %d us\n' % (payload_out.RF180))
                txt_file.write('TR = %d us\n' % (payload_out.TR))
                txt_file.write('Acquisition = %d us\n' % (payload_out.Acq))
                txt_file.write('Repetitions = %d\n' % (payload_out.Reps))
                txt_file.write('Frequency = %.6f MHz\n' %
(payload_out.Frequency))
                txt_file.write("\n")
    finally:
        print ("Closing socket")
        s.close()

if __name__ == "__main__":
    main()
```

APPENDIX B

RTL SCRIPTS

1. pulse_state_generator.v

```
`timescale 1ns / 1ps

module pulse_state_generator(
    input aclk,
    // inputs from user
    input [31:0] RF90, [31:0] Off_time, [31:0] RF180, [31:0] TR,
[31:0] Acq,
    input [31:0] repetitions,
    output [2:0] state,
    output resync
    );

reg resync_bit = 0;
reg [31:0] pulse_era;
reg [31:0] dead_time = 12500; //100 us before RF90 or RF180 begins -
for TXG
reg [2:0] state_val = 2;
reg [31:0] rep_tracker = 0;

always @ (posedge aclk)
begin
    resync_bit <= 0;
    if (repetitions == 0) begin
        rep_tracker <= 1;
        pulse_era <= 0;
        state_val <= 2;//flatlines for TXG and RF OUT
    end
    else if (rep_tracker > repetitions) begin
        pulse_era <= 0;
        state_val <= 2; //flatlines for TXG and RF OUT
    end
    else if (pulse_era<dead_time)
        state_val <= 0; //TXG is ON
    else if (pulse_era<(dead_time+RF90))
        state_val <= 1; //RF90 and TXG is ON
    else if (pulse_era == (dead_time+RF90 + 1))
        resync_bit <= 1;
    else if (pulse_era<(dead_time+RF90+Off_time))
        state_val <= 2; //TXG and RF OUT are OFF
    else if (pulse_era<(dead_time+RF90+Off_time+RF180))
        state_val <= 4; //RF180 is ON, TXG is OFF
    else if (pulse_era == (dead_time+RF90+Off_time+RF180 + 1))
        resync_bit <= 1;
    else if (pulse_era<(dead_time+RF90+Off_time+RF180+Acq))
        state_val <= 3; //RXG is ON
    else if (pulse_era<TR)
        state_val <= 2; //flatlines for everything
    else begin
        pulse_era <= 0;     // Reset counter
        resync_bit <= 1;
        rep_tracker <= rep_tracker + 1; //next cycle
        end
    pulse_era = pulse_era + 1;  // increment counter
end

assign state = state_val;
assign resync = resync_bit;

endmodule
```

2. DDS_control.v

```
`timescale 1ns / 1ps

module DDS_control(
    input aclk,
    input [31:0] phase_inc_server,
    input [2:0] state,
    output [31:0] p_inc, //phase increment
    output [31:0] p_off, //phase offset
    output [1:0] phase_control,
    output reg TXG_bit, RXG_bit
    );

reg [31:0] res, offset; //offset is redundant here - not used just yet
reg phase_val_bit = 0;

always @ (posedge aclk)
case (state)
0   : begin //dead time - turn on TXG
        offset <= 0;
        phase_val_bit <= 0;
        res <= 0;
        TXG_bit <= 1;
        RXG_bit <= 0;
      end
1    : begin     //RF90 and TXG
        offset <= 0;     //0 degree phase shift //2<<28; //180 degree
phase shift
        phase_val_bit <= 1;
        res <= phase_inc_server;
        TXG_bit <= 1;
        RXG_bit <= 0;
      end
2    : begin     //Off_time and end of acq
        offset <= 0;
        phase_val_bit <= 0;
        res <= 0;        //flatline
        TXG_bit <= 0;
        RXG_bit <= 0;
      end
3    : begin     //Acquisition
        offset <= 0;
        phase_val_bit <= 0;
        res <= 0;        //flatline
        TXG_bit <= 0;
        RXG_bit <= 1;
      end
4    : begin     //RF180 with TXG being OFF
        offset <= 0;     //0 degree phase shift //2<<28; //180 degree
phase shift
        phase_val_bit <= 1;
        res <= phase_inc_server;
        TXG_bit <= 0;
        RXG_bit <= 0;
      end
endcase

assign p_inc = res;
assign p_off = offset;
assign phase_control = phase_val_bit;

endmodule
```

3. DAC_activator.v

```
`timescale 1ns / 1ps
module DAC_activator(
    input aclk,
    input wire [31:0] DDS_phase,
    output tvalid_out
    );
reg bit = 0;
reg [31:0] temp = 0;
always @ (posedge aclk)
begin
    if ((temp == DDS_phase)||(DDS_phase == 0))
        bit <= 0;
    else begin
        bit <= 1;
        temp <= DDS_phase;
        end
end
assign tvalid_out = bit;
endmodule
```

APPENDIX C

TESTBENCH/SIMULATION CODE

1. design_test_TB.v

```
`timescale 1ns / 1ps //resolution of XSim simulation
`define time_period 8 //time period of 125 MHz clock, i.e., 8 ns

module design_TB();

//Set all inputs as regs and all outputs as wires

  reg [31:0]Acq_us;
  reg [31:0]Frequency_Hz;
  reg [31:0]Off_time_us;
  reg [31:0]RF180_us;
  reg [31:0]RF90_us;
  wire RXG;
  reg [31:0]Repetitions;
  reg [31:0]TR_us;
  wire TXG;
  reg clk_in1_n = 1;
  reg clk_in1_p = 0;
  wire dac_clk_0;
  wire [13:0]dac_dat;
  wire dac_rst_0;
  wire dac_sel_0;
  wire dac_wrt_0;
  //Below two wires are meant to calculate phase increment from
frequency.
  //Basically, this testbench performs the same arithmetic done by the
server to obtain phase increment.
  reg [63:0] phase_part_1;// = Frequency_Hz * (2<<30);
  reg [31:0] phase_part_2;// = phase_part_1/125000000;

design_test_wrapper DUT
/* connecting previously defined regs and wires to ports in DUT
instance of design_test_wrapper */
   (.Acq_us(Acq_us),
    .Frequency(phase_part_2),
    .Off_time_us(Off_time_us),
    .RF180_us(RF180_us),
    .RF90_us(RF90_us),
    .RXG(RXG),
    .Repetitions(Repetitions),
    .TR_us(TR_us),
    .TXG(TXG),
    .clk_in1_n(clk_in1_n),
    .clk_in1_p(clk_in1_p),
    .dac_clk_0(dac_clk_0),
    .dac_dat(dac_dat),
    .dac_rst_0(dac_rst_0),
    .dac_sel_0(dac_sel_0),
    .dac_wrt_0(dac_wrt_0)
    );

/* Now sending inputs to design_test_wrapper (a subsection of system.bd
- the system block design) */

initial begin
//Toggle below set of inputs every 4 ns
forever #(`time_period/2) begin
  clk_in1_n = ~clk_in1_n;
  clk_in1_p = ~clk_in1_p;
end
end

initial begin
//Configure below sets of inputs once

//Enter first set of pulse parameters - sequence #1
  Frequency_Hz = 2550000; //2.55 MHz
  phase_part_1 = Frequency_Hz * (1<<30);
  phase_part_2 = phase_part_1/125000000;
  RF90_us <= 5;
  Off_time_us <= 10;
  RF180_us <= 10;
  Acq_us <= 20;
  TR_us <= 180;
  Repetitions <= 3;
#800000 //wait for 800 us

//Blanking out pulse generator - sequence #2
  Frequency_Hz = 0; //0 MHz
  phase_part_1 = Frequency_Hz * (1<<30);
  phase_part_2 = phase_part_1/125000000;
  RF90_us <= 0;
  Off_time_us <= 0;
  RF180_us <= 0;
  Acq_us <= 0;
  TR_us <= 0;
  Repetitions <= 0;
#5000 //wait for 5 us

//Enter second set of pulse parameters - sequence #3
  Frequency_Hz = 10000000; //10 MHz
  phase_part_1 = Frequency_Hz * (1<<30);
  phase_part_2 = phase_part_1/125000000;
  RF90_us <= 4;
  Off_time_us <= 4;
  RF180_us <= 8;
  Acq_us <= 15;
  TR_us <= 150;
  Repetitions <= 4;
#800000 //wait for 800 us

//Blanking out pulse generator - sequence #4
  Frequency_Hz = 0;
  phase_part_1 = Frequency_Hz * (1<<30);
  phase_part_2 = phase_part_1/125000000;
  RF90_us <= 0;
  Off_time_us <= 0;
  RF180_us <= 0;
  Acq_us <= 0;
  TR_us <= 0;
  Repetitions <= 0;
#5000 //wait for 5 us

//Enter third set of pulse parameters - sequence #5
  Frequency_Hz = 30720000; //30.72 MHz
  phase_part_1 = Frequency_Hz * (1<<30);
  phase_part_2 = phase_part_1/125000000;
  RF90_us <= 1;
  Off_time_us <= 4;
  RF180_us <= 2;
  Acq_us <= 10;
  TR_us <= 130;
  Repetitions <= 5;

#800000 //wait for 800 us
 $finish; //simulation has completed
end

endmodule
```

APPENDIX D

MATLAB CODES

1. ReadPentek.m

```
function [datamat] = ReadPentek(filename, samprate, at, nacq)

% Pentek Recon Code

fid = fopen(strcat(filename, '.dat'), 'r');
data = fread(fid, 'int16');
fclose(fid);
sizeacq = samprate*at;
timeline = linspace(0,at,sizeacq)';
idata = data(1:2:end);
qdata = data(2:2:end);
dataformed = idata + 1j*qdata;
ns = length(dataformed)/nacq;
datamat = reshape(dataformed, [ns nacq]);
end
```

2. display_Pentek_RX.m

```
%% Crop out the echo data - 2 ms before and 2 ms after center
clc; close all; clear all
file_handle = ReadPentek('5-25-
2021\05242021_Mariam30r73MHz_3_ch2',200e6,70e-3,20);
[rows, cols] = size(file_handle);
X = linspace(0, 70e-3, rows)';
for a = 1:cols
minValue = 13.2e-3;
maxValue = 17.2e-3;
indexesInRange = X >= minValue & X <= maxValue;
Data2 = [X(indexesInRange) file_handle(indexesInRange,a)];
% plot(Data2(:,1),real(Data2(:,2)));
%    title("Receive digitized by Pentek for 20 reps - 4 ms echo
data");
%    xlabel("Time (in seconds)");
%    ylabel("Signal level (in V)");
%    xlim([minValue maxValue])
%    ylim([-0.5e4 0.5e4])
%    hold on
%    grid on
fs = 200e6;
spec = fftshift(fft(fftshift(Data2(:,2))));
n = length(Data2(:,2));
freq = (-(n-1)/2:(n-1)/2)*(fs/n);
%   figure; plot(freq * 1e-6, abs(spec));
%    title("Spectrum of echo data - no DC correction");
%    xlabel("Frequency (in MHz)");
%    grid on
dccorr = mean(Data2(end-5000:end,2));
Data2(:,2) = Data2(:,2) - dccorr;
%% Now take FT of cropped signal data
fs = 200e6;
spec = fftshift(fft(fftshift(Data2(:,2))));
n = length(Data2(:,2));
freq = (-(n-1)/2:(n-1)/2)*(fs/n);
%  plot(freq * 1e-6, abs(spec));
%    title("Spectrum of echo data - with DC correction - 20 acqs");
%    xlabel("Frequency (in MHz)");
%    hold on
%    grid on
%% Brickwall - crop 50 kHz around center frequency
freq = freq*1e-6;
center_point = 30.72;
 [~, lower_lim]= min(abs(freq-(center_point-0.025)));
 [~, upper_lim]= min(abs(freq-(center_point+0.025)));
cropped_spec = spec(lower_lim:upper_lim);
cropped_freq = freq(lower_lim:upper_lim);

plot(cropped_freq, real(cropped_spec));
    title("Spectrum of echo - cropped out Larmor - 20 acqs (real)");
    xlabel("Frequency (in MHz)");
    ylabel("Magnitude");
    hold on
    grid on
    xlim([min(cropped_freq) max(cropped_freq)])

% plot(cropped_freq, abs(cropped_spec));
%     title("Spectrum of echo - cropped out Larmor - 20 acqs
(magnitude)");
%     xlabel("Frequency (in MHz)");
%     hold on
%     grid on
%     xlim([min(cropped_freq) max(cropped_freq)])

    plot(cropped_freq, angle(cropped_spec));
    title("Spectrum of echo - cropped out Larmor - 20 acqs (phase)");
    %xlabel("Frequency (in MHz)");
    hold on
    grid on
    xlim([min(cropped_freq) max(cropped_freq)])
sig = max(abs(cropped_spec));
noise = mean(abs(cropped_spec(1:end/4)));
SNR = sig/noise

x = 1:lower_lim-1;
y = upper_lim+1:length(spec);
spec(x) = 0;
spec(y) = 0;
end
```

APPENDIX E

Following is a user manual created for students to build and test projects for the Red Pitaya development board in MRSL.

Programming the Red Pitaya for MR Pulse Sequences

Magnetic Resonance Systems Lab
Texas A&M University

By: Mariam Nida Usmani
June 2021

Page

1 Introduction

This document serves as a starting point for those new to an FPGA programming environment and MRI. It should allow you to create a project on a Windows machine, deploy the same on a Red Pitaya board and build a working MR system centered around it.

Basically, this is a guide on how to design and build a digital circuit on FPGA fabric and create an interface around it for user control.

1.1 Red Pitaya

The Red Pitaya board used at TAMU-MRSL consists of a Zynq 7010 FPGA chip. The environment used to access this is a custom-built Ubuntu OS, called STEMlab 125-14.
The latest OS version during the time of writing this manual is 1.04.

1.2 Project Overview

In general, every Red Pitaya project has three components:
1. a bitstream file (.BIT)
2. server program (compiled from a .c file)
3. client program (a Python script)

Out of all these, the bitstream file is the most important component. This directly configures the Zynq 7010 FPGA housed within Red Pitaya.

Server and client programs provide a way to interact with a bitstream-activated FPGA. While server runs on top of the FPGA, the client runs on an independent PC (the 'client PC'). This client connects to the server using Ethernet sockets and provides an interface for the PC used to control the FPGA system. Depending on the interface, it may be a CLI (command line interface), a GUI (graphical user interface), or a combination of both.

This guide helps set up a bitstream controlled with a Python CLI. This may be replaced with a Python GUI using editors such as QT Designer, but GUIs are beyond the scope of this manual.

1.3 Prerequisites

1.3.1 Hardware

To get started with FPGA programming on Pitaya, you will need the following digital components:

1. A Red Pitaya development board (STEMlab 125-14)
2. A Windows 10 PC
3. A blank micro SD card (min. 4 GB)
4. A 5V/3A micro USB cable (preferably with a wall socket plug)

NOTE: For hardware testing, you will also need an oscilloscope.

1.3.2 Software

This manual walks through the installation and setup for the following programs:
1. Vivado
2. PuTTY
3. WinSCP
4. PyCharm

A few other software may be required to control the oscilloscope – they are not covered in this manual.

Finally, the following zipped folders contain all programs required to get started with:
a) MRI experiments (chapter 3):
`mr_experiment_SE.zip`

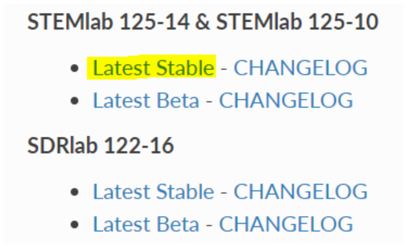
b) Your first Vivado project (chapter 4):
`base_project.zip`

2 Getting Started

2.1 Micro SD Card

The first step is to download the zipped SD image of the STEMLab 125-14 OS from this link and unzip it:
https://downloads.redpitaya.com/downloads/STEMlab-125-1x/STEMlab_125-xx_OS_1.04-7_stable.img.zip

Insert the blank SD card into a card reader in the Windows PC. If the above link gives an error, go to https://redpitaya.readthedocs.io/en/latest/quickStart/SDcard/SDcard.html and click on the highlighted option.


STEMlab 125-14 & STEMlab 125-10

- Latest Stable - CHANGELOG
- Latest Beta - CHANGELOG

SDRlab 122-16

- Latest Stable - CHANGELOG
- Latest Beta - CHANGELOG

Next, download Win32 Disk Imager from the link below. Install and run the program.
https://sourceforge.net/projects/win32diskimager/

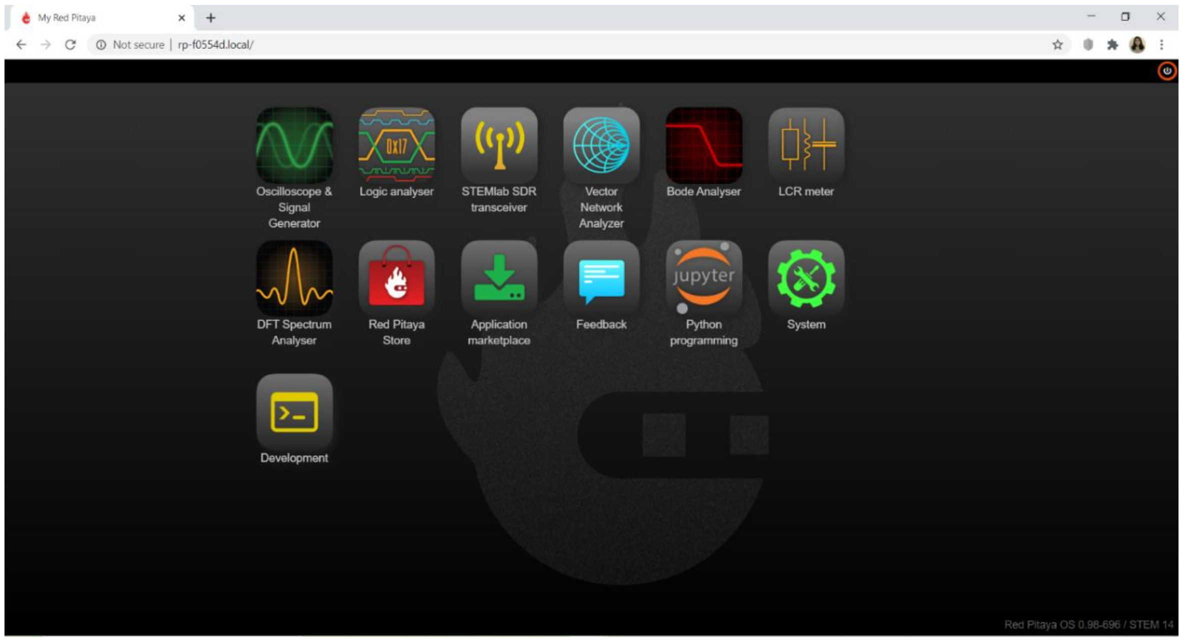
Under 'Image File', select the path to the .img file of STEMlab OS. Make sure the drive letter under 'Device' corresponds to the SD card you inserted. Once all this is confirmed, hit 'Write'.

The SD image will take a few minutes to be written, after which a 'Write Successful' dialog box appears. Safely remove this microSD card from the PC and move on to the next section.

2.2 Red Pitaya Machine

Insert the microSD card previously prepared into its slot in the Red Pitaya development board. Plug the microUSB cable to the ‘PWR’ socket on the board. Be careful to connect to ‘PWR’ socket only and not ‘CONS’ socket. Failure to do so may result in damage to the board.
Now power on the machine. You will initially see a green LED lit up; this confirms the power supply. Momentarily, a blue LED should also light up. This confirms access to the SD card.
Finally, let a few more seconds pass, and you should see a red LED blinking, in a ‘heartbeat’ pattern (two flashes at a time). This confirms that the development board is fully loaded, and ready for use.

Now connect the Red Pitaya board to your Windows PC using an Ethernet cable. Wait for 30 seconds and open a web browser.
Type the address pasted on the back of the board (of the form `rp-fxxxx.local/`), and press Enter. If successful, the following web page opens:



NOTE: If the web page is unreachable, try disconnecting the PC from its WiFi network and try again.

Now go to System>Network manager and note the address under ‘Wired connection status’. This is the IP address of the board (not to be confused with the webpage; that is simply its URL) and will be of the form `abc.def.ghi.jkl`.

2.3 Windows PC - Basic Setup

Essentially, Vivado, PuTTY, WinSCP and PyCharm will be installed and configured on the Windows PC.

2.3.1 Vivado

NOTE: All Verilog codes in this manual have been tested using version 2020.1 of Xilinx Vivado; hence this installation guide is for that specific version. You might want to install a more recent version instead – the steps will remain same.

Go to the webpage below, and search for 'Vivado Design Suite - HLx Editions - 2020.1'
https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools.html

Click on the link 'Xilinx Unified Installer 2020.1: Windows Self Extracting Web Installer'. You will be redirected to a Xilinx Sign-In page. If you do not possess one already, go ahead and create a Xilinx account. If you already have one, just enter your credentials and sign in.
Fill up the required fields in the Name and Address Verification form and click on 'Download'. An executable file should start downloading now.

Launch the downloaded executable and follow the instructions in the installation wizard. Make sure to select 'Vivado' when prompted about the product, and 'Vivado HL WebPACK' as the edition to be installed. Keep everything else as default and continue with the installation.
This step will take anywhere between half an hour to several hours, depending on your Internet connection. Once fully downloaded and installed, Vivado is ready for use. Follow the setup wizard, keeping all the default fields and check boxes.

2.3.2 PuTTY

Go to this webpage and download the 64-bit MSI package file:
https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
Click on the downloaded file to launch the setup wizard. Follow the installation steps that show up, keeping everything as default.

2.3.3 WinSCP

Scroll down the linked webpage below and press 'Download WinSCP'. Launch the downloaded executable and follow the setup wizard.
https://winscp.net/eng/download.php

Now that WinSCP has been installed, it must be configured to ensure a working development environment. Launch WinSCP; a login window opens. For host name, type in Pitaya's IP address. Enter `root` as both username and password, then press Save. Keep the site name as 'Red Pitaya' and press OK. Now press the Login button; a warning window might pop up. Simply press Yes and continue.

Retype the password as root and press OK. Navigate from root to `/tmp` in Pitaya's window – this is where the bitstream and C files will be stored and activated/executed.

2.3.4 PyCharm

First download Python for Windows from https://www.python.org/downloads/. Select 'Add Python X.X to PATH' and press 'Install Now'.

After that you can download PyCharm Community version from the link below and follow its installation procedure. We will use PyCharm to launch our user console.
https://www.jetbrains.com/pycharm/download/#section=windows

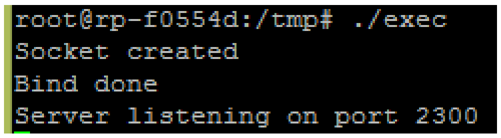
2.4 Deploying a Prebuilt Project

Open WinSCP and log in to Pitaya. Within WinSCP navigate to the directory where `mr_experiment_SE` was extracted. From there, move to `\tmp\pulse_sequence_generator\pulse_sequence_generator.runs\impl_1` – you should now see a file named `system_wrapper.bit`. Drag it to the `/tmp` directory of Pitaya.

Now go to `mr_experiment_SE\Server-client pair` in both WinSCP and File Explorer. Drag `server_program.c` to Pitaya's /tmp directory and open `client_program.py` in File Explorer.
Within WinSCP open a PuTTY terminal, log in with 'root' as password and type in the following commands:

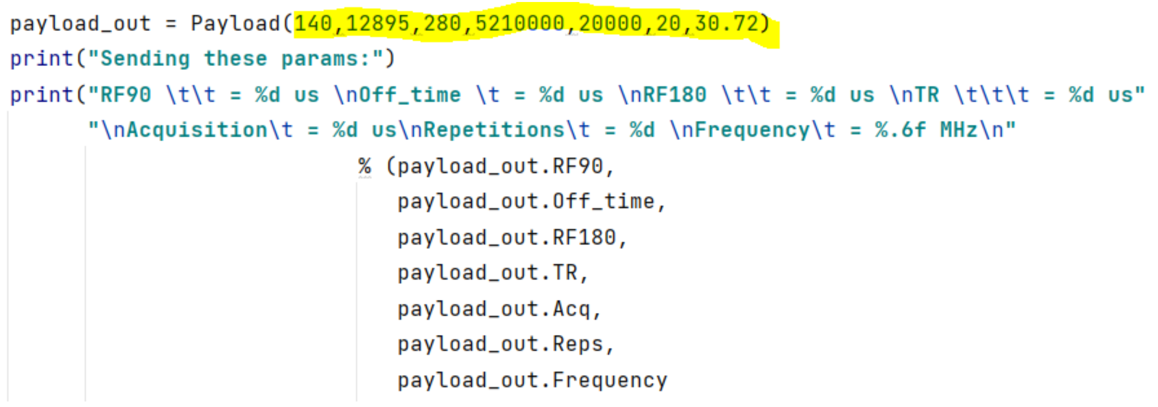
```
cd /tmp
cat system_wrapper.bit > /dev/xdevcfg
chmod u+x server_program.c
gcc -o exec server_program.c
./exec
```

You will be greeted with this PuTTY screen:



```
root@rp-f0554d:/tmp# ./exec
Socket created
Bind done
Server listening on port 2300
```

Now open `client_program.py` in PyCharm. In the code, note the numbers highlighted below – these are parameters you can set for this pulse generator. The second print explains what each parameter is, along with their units (if any).



```
payload_out = Payload(140,12895,280,5210000,20000,20,30.72)
print("Sending these params:")
print("RF90 \t\t = %d us \nOff_time \t = %d us \nRF180 \t\t = %d us \nTR \t\t\t = %d us"
      "\nAcquisition\t = %d us\nRepetitions\t = %d \nFrequency\t = %.6f MHz\n"
                              % (payload_out.RF90,
                                 payload_out.Off_time,
                                 payload_out.RF180,
                                 payload_out.TR,
                                 payload_out.Acq,
                                 payload_out.Reps,
                                 payload_out.Frequency
```

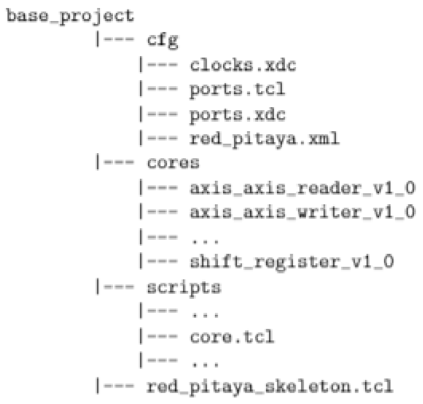
Go to Run>Run>client_program. Upon success you will note a timestamped .txt file created in the same folder as `client_program`. This is a record of all parameters set in the current pulse sequence – useful for future reference.

3 Your First Vivado Project

In this section, we start with a 'skeleton' Vivado project with all the Pitaya-specific IP cores and config files included and build on it. Afterwards, we add an interface on top of it and deploy a complete set of bitstream, server and client programs together.

3.1 A new project

To make things simple, you would have been provided a zipped folder named `base_project.zip` with all the required directories. Go ahead and unzip this folder. Once unzipped, you will note the following directory structure:



```
base_project
        |--- cfg
              |--- clocks.xdc
              |--- ports.tcl
              |--- ports.xdc
              |--- red_pitaya.xml
        |--- cores
              |--- axis_axis_reader_v1_0
              |--- axis_axis_writer_v1_0
              |--- ...
              |--- shift_register_v1_0
        |--- scripts
              |--- ...
              |--- core.tcl
              |--- ...
        |--- red_pitaya_skeleton.tcl
```

Do not alter this directory structure. Any changes made to it at this stage may result in Vivado projects not compiling due to errors.

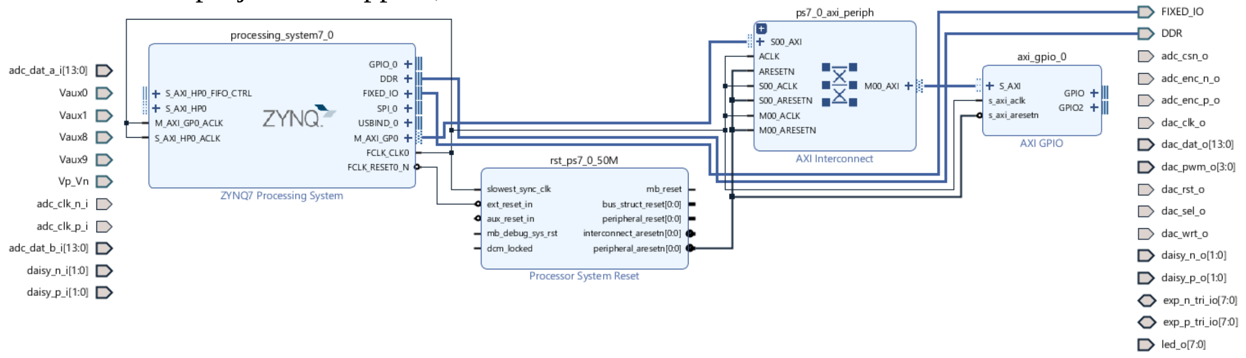
Launch Vivado. Once opened, go to the Tcl console, and type the following set of commands:

```
cd
cd /full/path/to/base_project
```

```
source red_pitaya_skeleton.tcl
```

<u>NOTE:</u> If you copy-paste the path address to base_project in console for the second command, make sure to change all back slashes ('\') to forward slashes ('/') before hitting enter. Otherwise Tcl will throw up an error.

The last command takes about 6 minutes to execute on an Intel i7 machine (9th gen) with 16 GB RAM. After executing, click on 'Block Diagram' under the Project Manager menu. A block layout of the skeleton project will appear, such as the one below.



Also, the directory will now look like this:

```
base_project
        |--- cfg
        |       |--- clocks.xdc
        |       |--- ports.tcl
        |       |--- ports.xdc
        |       |--- red_pitaya.xml
        |--- cores
        |       |--- axis_axis_reader_v1_0
        |       |--- axis_axis_writer_v1_0
        |       |--- ...
        |       |--- shift_register_v1_0
        |--- scripts
        |       |--- ...
        |       |--- core.tcl
        |       |--- ...
        |--- tmp
        |       |--- cores
        |       |--- your_first_project
        |--- red_pitaya_skeleton.tcl
```

Now you can go ahead and add more blocks (formally called IP cores) the block design. This can be using the provided IP cores by Xilinx and Pavel, as well as the ones created by your own Verilog and/or XDC files.

3.2 The IP cores

There are several IP cores available for us to use. Here, we will focus on the once already available in Vivado, and the ones built using the source command (see section 4.1). The latter cores are custom-made for the Red Pitaya board. Out of all these cores, a few are required to build a bitstream for an MR project. The essential ones are already present in the 'system' diagram of `your_first_project`. Below are all the cores that need to be added and/or re-customized to our needs.

3.2.1 Processing System

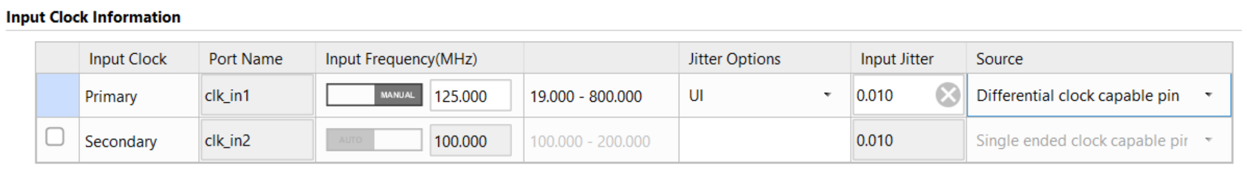
Under Diagram, double-click on `processing_system7_0`. This opens the Re-customize IP window. Go to PS-PL Configuration>HP Slave AXI Interface. Uncheck the square box next to S AXI HP0 interface. Click OK to save this change and exit the window.

3.2.2 AXI Interconnect

Double-click `ps7_0_axi_periph`. Under Top Level Settings, select 3 from the drop-down menu next to Number of Master Interfaces. By doing this, we ensure that 3 AXI GPIOs can be connected to our block design – its importance will be explained in a bit. Press OK to customize this core and save changes.

3.2.3 Clocking Wizard

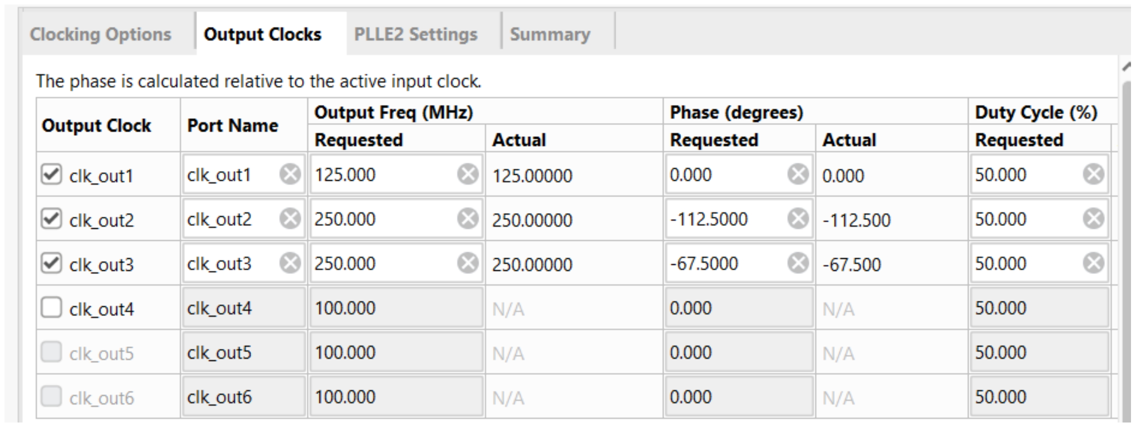
Notice that this IP core is absent from our block design. To add it, right-click on an empty spot in the Diagram window and click 'Add IP'. In the search bar, type Clocking Wizard. One option will show up in search results – double-click on that. Double-click on `clk_wiz_0`. Under Clocking Options> Primitive select PLL. Scroll down and set your Input Clock Information just like the screenshot below.



Input Clock Information

	Input Clock	Port Name	Input Frequency(MHz)			Jitter Options	Input Jitter	Source
	Primary	clk_in1	MANUAL	125.000	19.000 - 800.000	UI	0.010	Differential clock capable pin
☐	Secondary	clk_in2	AUTO	100.000	100.000 - 200.000		0.010	Single ended clock capable pir

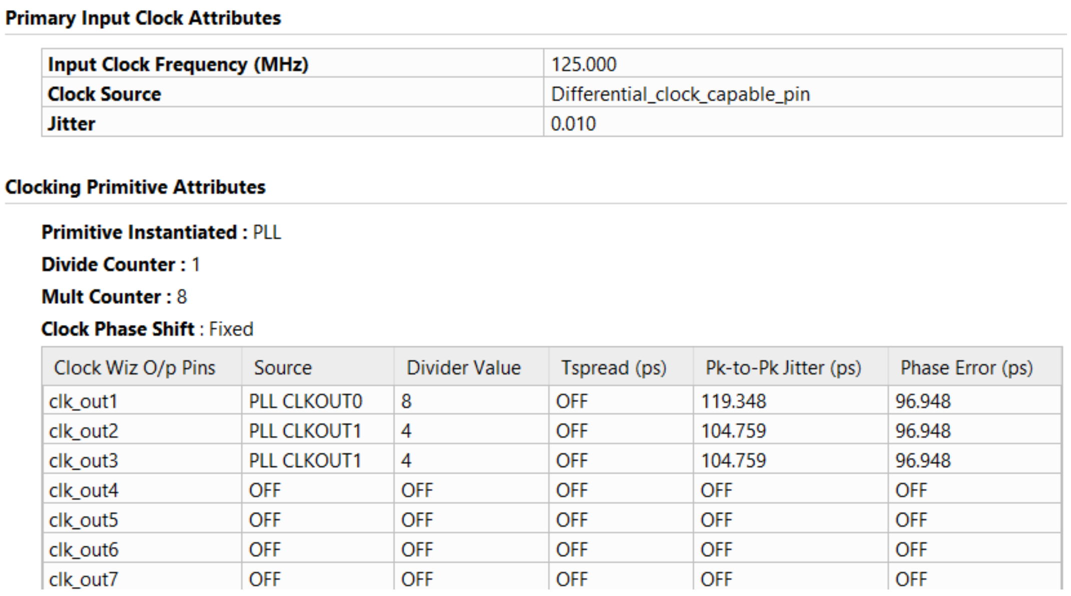
In the same menu, go to Output Clocks page and set the output clocks identical to this screenshot:



The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)
		Requested	Actual	Requested	Actual	Requested
✓ clk_out1	clk_out1	125.000	125.00000	0.000	0.000	50.000
✓ clk_out2	clk_out2	250.000	250.00000	-112.5000	-112.500	50.000
✓ clk_out3	clk_out3	250.000	250.00000	-67.5000	-67.500	50.000
☐ clk_out4	clk_out4	100.000	N/A	0.000	N/A	50.000
☐ clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000
☐ clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000

Scroll down and deselect 'reset' under Enable Optional Inputs/ Outputs for MMCM/PLL. Now go to 'Summary' page and make sure it is identical to the snap below.



Primary Input Clock Attributes

Input Clock Frequency (MHz)	125.000
Clock Source	Differential_clock_capable_pin
Jitter	0.010

Clocking Primitive Attributes

Primitive Instantiated : PLL
Divide Counter : 1
Mult Counter : 8
Clock Phase Shift : Fixed

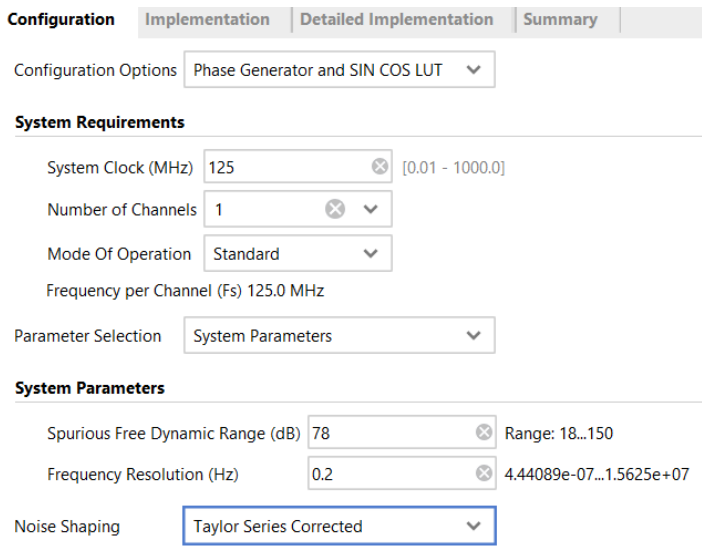
Clock Wiz O/p Pins	Source	Divider Value	Tspread (ps)	Pk-to-Pk Jitter (ps)	Phase Error (ps)
clk_out1	PLL CLKOUT0	8	OFF	119.348	96.948
clk_out2	PLL CLKOUT1	4	OFF	104.759	96.948
clk_out3	PLL CLKOUT1	4	OFF	104.759	96.948
clk_out4	OFF	OFF	OFF	OFF	OFF
clk_out5	OFF	OFF	OFF	OFF	OFF
clk_out6	OFF	OFF	OFF	OFF	OFF
clk_out7	OFF	OFF	OFF	OFF	OFF

3.2.4 AXI GPIO

We need 3 of these cores. One is already present in our diagram, `axi_gpio_0`. Single-click this and hit CTRL+C followed by CTRL+V two times. We now have 3 AXI GPIO cores awaiting connections.

3.2.5 DDS Compiler

This is the heart of our system – it is responsible for generating RF pulses. Just like Clocking Wizard this IP core needs to be added. Right-click on an empty spot in Diagram again and search for DDS Compiler. When found, select it to add to the diagram. Double-click to customize. Refer to this screenshot for setting the Configuration page:



Configuration Options: Phase Generator and SIN COS LUT

System Requirements

System Clock (MHz): 125 [0.01 - 1000.0]
Number of Channels: 1
Mode Of Operation: Standard
Frequency per Channel (Fs) 125.0 MHz

Parameter Selection: System Parameters

System Parameters

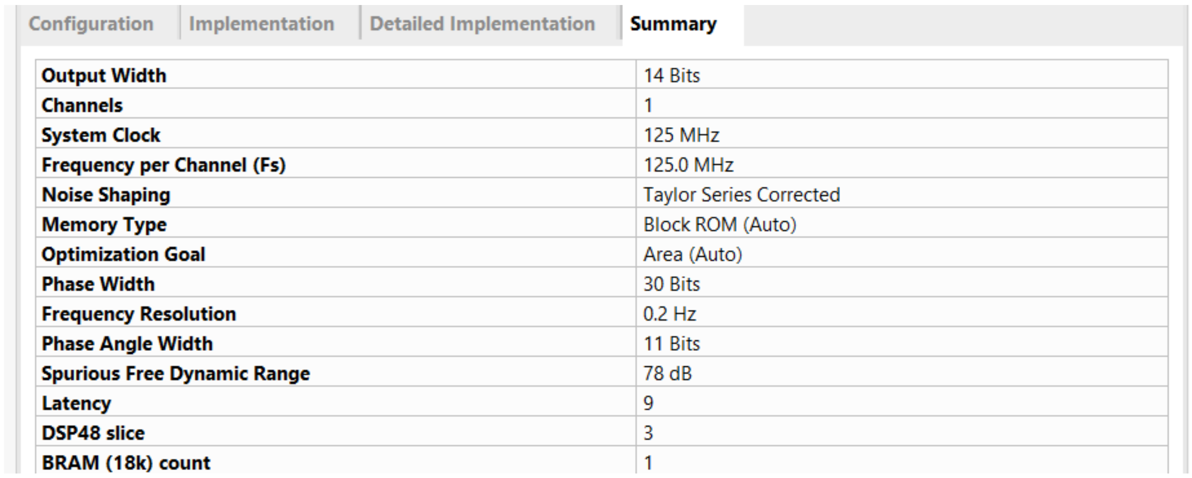
Spurious Free Dynamic Range (dB): 78 Range: 18...150
Frequency Resolution (Hz): 0.2 4.44089e-07...1.5625e+07

Noise Shaping: Taylor Series Corrected

Under Implementation, set the following:

```
Phase Increment Programmability: Streaming (with Resync checked)
Phase Offset Programmability: Streaming
```

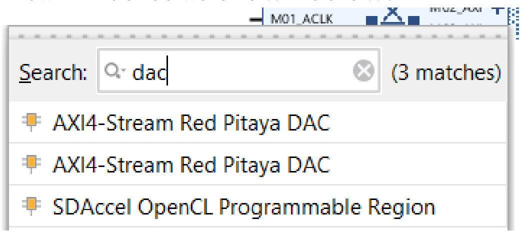
Your summary page should now look like this. Press OK to customize and exit the menu.



Output Width	14 Bits
Channels	1
System Clock	125 MHz
Frequency per Channel (Fs)	125.0 MHz
Noise Shaping	Taylor Series Corrected
Memory Type	Block ROM (Auto)
Optimization Goal	Area (Auto)
Phase Width	30 Bits
Frequency Resolution	0.2 Hz
Phase Angle Width	11 Bits
Spurious Free Dynamic Range	78 dB
Latency	9
DSP48 slice	3
BRAM (18k) count	1

3.2.6 AXI4-Stream Red Pitaya DAC

This is a Pitaya-specific core created by Pavel Demin. Use the same steps as used before for Clocking Wizard to add AXI4-Stream Red Pitaya DAC to the block design. Note that searching for 'DAC' results in two identical IP cores as shown below.



AXI4-Stream Red Pitaya DAC
AXI4-Stream Red Pitaya DAC
SDAccel OpenCL Programmable Region

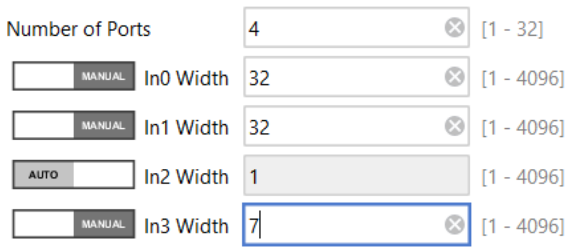
That is because two versions of the DAC IP core exist. Select both, one at a time. Two blocks will show up – retain the core with `wrt_clk` pin present. Delete the DAC core without this pin by left clicking and pressing Delete on the keyboard.

3.2.7 Constant

We need two Constant IP cores in this project. Right-click and add this IP core to the block design. Left-click on it and copy-paste it again. In the Re-customize window for `xlconstant_0`, set Const width as 7 and Const val as 0. Press OK and save changes (Ctrl+S).

3.2.8 Concat

Right-click and add this IP core to the block design. Double-click on it, enter Number of Ports as 4 and press Enter after setting the port widths as below.



Number of Ports 4 [1 - 32]

MANUAL In0 Width 32 [1 - 4096]

MANUAL In1 Width 32 [1 - 4096]

AUTO In2 Width 1 [1 - 4096]

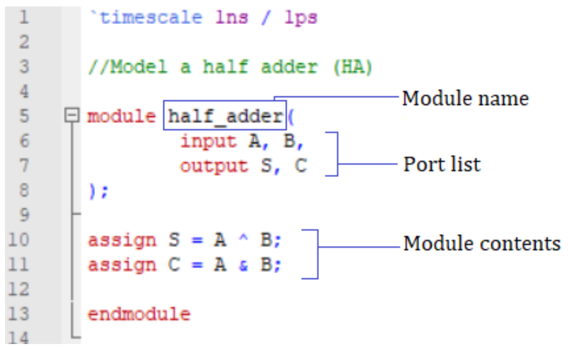
MANUAL In3 Width 7 [1 - 4096]

3.3 Your own RTL script(s)

MRI requires three parameters to be specified – timing, pulse frequency and number of pulses. To this end, two RTL scripts will be created – a pulse state generator and a DDS controller.

3.3.1 Parts of an RTL script

The diagram below illustrates the basic parts of an RTL script in Verilog – a module name, a port list and module contents.



```
`timescale 1ns / 1ps

//Model a half adder (HA)

module half_adder(
            input A, B,
            output S, C
);

assign S = A ^ B;
assign C = A & B;

endmodule
```

Vivado generates the first two in its Source Wizard; this leaves us with only the module contents to draft out. The next section explains the basic functionality we need for RF pulse generation.

3.3.2 Deciding on the functionality

The first step is to decide on the states. To keep it simple, we consider two states – 'RF ON' (corresponds to Pitaya generating a sine RF waveform) and 'RF OFF' (corresponds to Pitaya staying inactive). It is recommended that the functionality be tabulated – one has been provided below for generating RF pulses indefinitely.

Pulse Era	State Values	RF	TXG	RXG
0 ~ off_cycles	0	OFF	0	1
off_cycles ~ off_cycles + on_cycles	1	ON	1	0

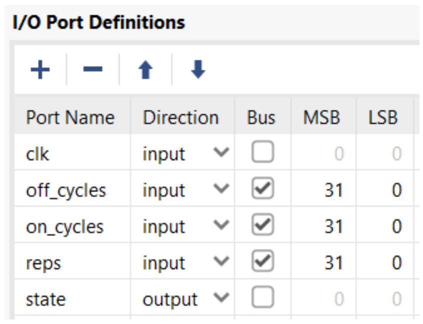
RF OFF is assigned a state value 0 and RF ON has a state value 1. Next, we have `off_cycles` and `on_cycles`. These indicate the number of clock cycles for which a state value must be retained. In other words, they decide the pulse era.

TXG (short for transmit gating) and RXG (short for receive gating) are two external 3.3 V GPIO pins we will be controlling, with 1 corresponding to a logical high and 0 corresponding to a logical low. NOTE: This is completely unrelated to state values; do not confuse logical high and low with that.

Finally, the states will cycle for a certain number of times. Let us denote this with ‘rep’. A variable in our RTL code will keep track of how many times the states have been cycled through. Once it has cycled through states 0 and 1 ‘rep’ number of times, no more pulses must be generated. In other words, we keep it in state 0 until a new set of parameters are fed in.
Having a layout in mind, we proceed towards building a pulse sate generator – aptly named `state_generator`.

3.3.3 state_generator

Go to Flow Navigator>Project Manager>Add Sources. Select Add or Create Design Sources and hit Next. Click on Create File, leave the file type as Verilog and name it as `state_generator`, keeping the location local to project. Click Finish. Set the I/O Ports as shown below. Press OK after this.



Port Name	Direction	Bus	MSB	LSB
clk	input	☐	0	0
off_cycles	input	✓	31	0
on_cycles	input	✓	31	0
reps	input	✓	31	0
state	output	☐	0	0

This file will now show up under Sources>Design Sources. Double-click on it. Notice the contents of the file – it includes a commented section followed by these lines:

```
module state_generator(
    input clk,
    input [31:0] on_cycles,
    input [31:0] off_cycles,
    input [31:0] reps,
    output state
    );
endmodule
```

To translate the previous table to Verilog an always block is required. Go ahead and type in this code in between `);` and `endmodule`.

```
reg [31:0] pulse_era = 0, rep_tracker = 0;

always @ (posedge clk)
begin
```

```
    if (reps == 0) begin
         rep_tracker <= 0;
         pulse_era <= 0;
         end
     else if ((pulse_era < off_cycles)||(rep_tracker == reps))
     begin
         state <= 0;
         pulse_era <= pulse_era + 1;
     end
     else if (pulse_era < off_cycles + on_cycles)
     begin
         state <= 1;
         pulse_era <= pulse_era + 1;
     end

     else begin
         pulse_era <= 0;
         rep_tracker <= rep_tracker + 1;
         end
end
```

The `pulse_era` register is important – it keeps track of the number of clock cycles passed. We use another register `rep_tracker` to check if the requested number of reps has been completed or not. Comparing this with inputs `off_cycles` and `on_cycles` helps determine the correct state of the generator. In other words, we have created a complex Moore finite state machine (FSM).

For this to work without errors, the port declaration for `state` needs to be modified. Go to the port list of `state_generator` and type `reg` between `output` and `state`. This creates a register that retains the value of `state` within the always block. Save this file.

3.3.4 DDS_controller

Using the generated states, we need to create digital RF pulses off DDS Compiler. It generates a sine waveform of specified frequency using two inputs – phase increment and phase offset. Both were set to 'streaming' in its Re-Customize IP window – the intention was to create RF pulses as per user specifications.

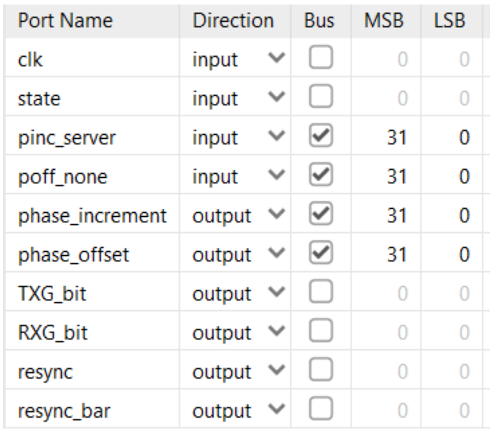
Basically, a DDS (short for Direct Digital Synthesizer) accepts a phase input, looks up the sine value corresponding to it and outputs the same. Vivado's DDS Compiler is designed such that it accepts a phase increment and a phase offset and generates a sine wave matching these parameters. Phase increment decides how fast the wave amplitude changes with each clock cycle, while phase offset determines the starting angle value for which the sine wave is generated.

$$phase\ increment, p_{inc} = \frac{f_{out} \times 2^{B_{\theta}(n)}}{f_{in}} \quad (1)$$

$$phase\ offset, p_{off} = \frac{2^{B_{\theta}(n)} \times \alpha}{360} \quad (2)$$

In this chapter, we will keep phase offset at 0. However, bear in mind that this can be altered in the bitstream later as per requirement. To control phase increment and offset, another script will act as an intermediary between `state_generator` and DDS Compiler – we name it `DDS_controller`.

Just like `state_generator`, go to Flow Navigator>Project Manager>Add Sources. Select Add or Create Design Sources and hit Next. Click on Create File, leave the file type as Verilog and name it as `DDS_controller`, keeping the location local to project. Click Finish.
Set the I/O Ports as shown below. Press OK after this.



Port Name	Direction	Bus	MSB	LSB
clk	input	☐	0	0
state	input	☐	0	0
pinc_server	input	✓	31	0
poff_none	input	✓	31	0
phase_increment	output	✓	31	0
phase_offset	output	✓	31	0
TXG_bit	output	☐	0	0
RXG_bit	output	☐	0	0
resync	output	☐	0	0
resync_bar	output	☐	0	0

Once again, we need an always block and a few registers. Go ahead and copy-paste the below code snippet between `);` and `endmodule`:

```
always @ (posedge clk)
 case (state)
     0: begin
            phase_increment <= 0;
            phase_offset <= 0;
            TXG_bit <= 0;
            RXG_bit <= 1;
            resync <= 1;
            resync_bar <= 0;
        end
     1: begin
            phase_increment <= pinc_server;
            phase_offset <= poff_none;
            TXG_bit <= 1;
            RXG_bit <= 0;
            resync <= 0;
            resync_bar <= 1;
        end
 endcase
```

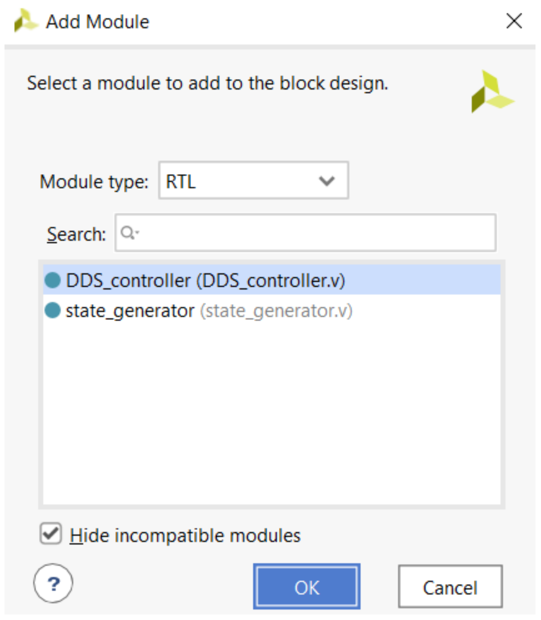
Finally, modify its port list to match this screenshot. Save this file.

```
module DDS_controller(
    input clk,
    input state,
    input [31:0] pinc_server,
    input [31:0] poff_none,
    output reg [31:0] phase_increment,
    output reg [31:0] phase_offset,
    output reg TXG_bit,
    output reg RXG_bit,
    output reg resync,
    output reg resync_bar
    );
```

TIP: If you need to add a new pin to the RTL modules or any functionality for that matter, be sure to add these changes directly to the design source files and not instances; changes to the latter might not be recognized by Vivado. To do this simply go to Sources>Design Sources>your_RTL_script_file_name.v and save your edits there. Make sure to hit 'Run Synthesis' under Flow Navigator>Synthesis to reflect the changes everywhere.

3.4 Block design – clean up and unify

Go to the Diagram window and delete pins `daisy_n_0` and `daisy_p_0`. Right-click on an empty spot and select 'Add Module'. Both your modules should show up like the figure below. If not, go back to the Verilog source files, correct any syntax errors, save them, and try again.


Add Module

Select a module to add to the block design.

Module type: RTL

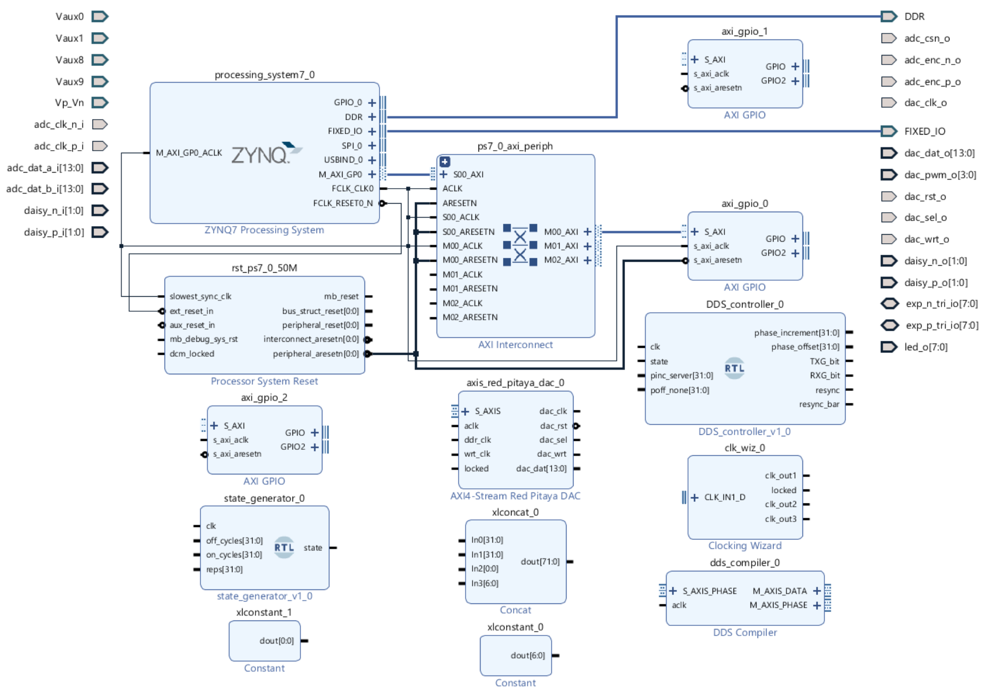
Search:

DDS_controller (DDS_controller.v)

state_generator (state_generator.v)

☑ Hide incompatible modules

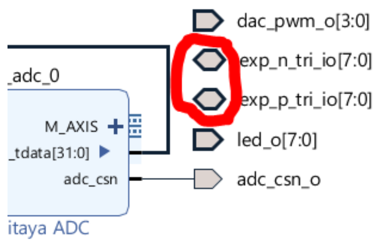
One by one select a module and press OK. Both will show up as blocks in the diagram with 'RTL' printed on them. The overall diagram will now look like this:



3.5 Programming GPIOs

Now that the logic for TXG and RXG has been written out in Verilog and all blocks connected, we need to select and activate physical GPIO pins on Pitaya's extension connectors. It is useful in, say, generating trigger pulses to drive other devices in the MR system.

Click on 'Open Block Design'. Your block design shows up in the Diagram window. Note the pins `exp_n_tri_io[7:0]` and `exp_p_tri_io[7:0]` as shown in the figure below. These pins will be siphoned off for our purposes.



In the TCL console type these commands.

```
create_bd_port -dir O TXG
create_bd_port -dir O RXG
```

Two output ports named `TXG` and `RXG` will be generated and added to the diagram; we will connect these shortly.

Go to the link indicated below and decide on your pins of interest.
https://redpitaya.readthedocs.io/en/latest/developerGuide/125-14/extent.html

In this tutorial, we pick pin 4 for TXG with these details from the developer guide:

```
Pin #:                          4
Description:                    DIO0_N
FPGA pin number:                G18
FPGA pin description:           IO_L16N_T2_35
Voltage levels:                 3.3V
```

Similarly, we pick pin 18 for RXG with these details from the developer guide:

```
Pin #:                          18
Description:                    DIO7_N
FPGA pin number:                M15
FPGA pin description:           IO_L23N_T3_35
Voltage levels:                 3.3V
```

Having decided on the pins, navigate to Sources> Constraints> constrs_1> ports.xdc. Comment out the lines with 'G18' and 'M15' in it using the # symbol – these are the pins we will siphon off as TXG and RXG respectively. Go to the bottom of the XDC file and type the following commands. Save the XDC file.

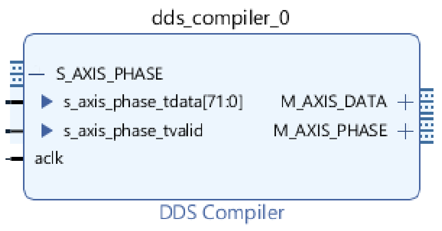
```
## Custom
set_property IOSTANDARD LVCMOS33 [get_ports TXG]
set_property SLEW FAST [get_ports TXG]
set_property DRIVE 8 [get_ports TXG]
set_property PACKAGE_PIN G18 [get_ports TXG]

set_property IOSTANDARD LVCMOS33 [get_ports RXG]
set_property SLEW FAST [get_ports RXG]
set_property DRIVE 8 [get_ports RXG]
set_property PACKAGE_PIN M15 [get_ports RXG]
```

3.6 Connecting the blocks together

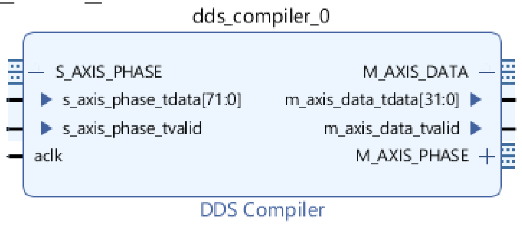
Go to Flow Navigator> IP Integrator> Open Block Design.

3.6.1 Pins and ports

Note that some blocks in the diagram have '+' symbols in place of where you would expect a pin. These are called 'interface pins', typically prefixed with `M_AXIS` or `S_AXIS`. Some of the connections require actual 'pins' and not the 'interface pin' and vice versa. Below is what a channel looks like with a '+' symbol (`M_AXIS_DATA` and `M_AXIS_PHASE`), and individual ports with a '-' symbol next to their channel (`S_AXIS_PHASE`).



To toggle between 'interface pin' and 'pin', simply mouse over to the '-' or '+' symbol. The cursor turns into either two upward chevrons or downward chevrons. Left click once to toggle. Below is the result after a toggle at `M_AXIS_DATA`:

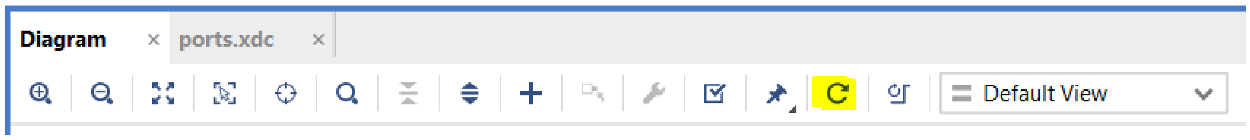


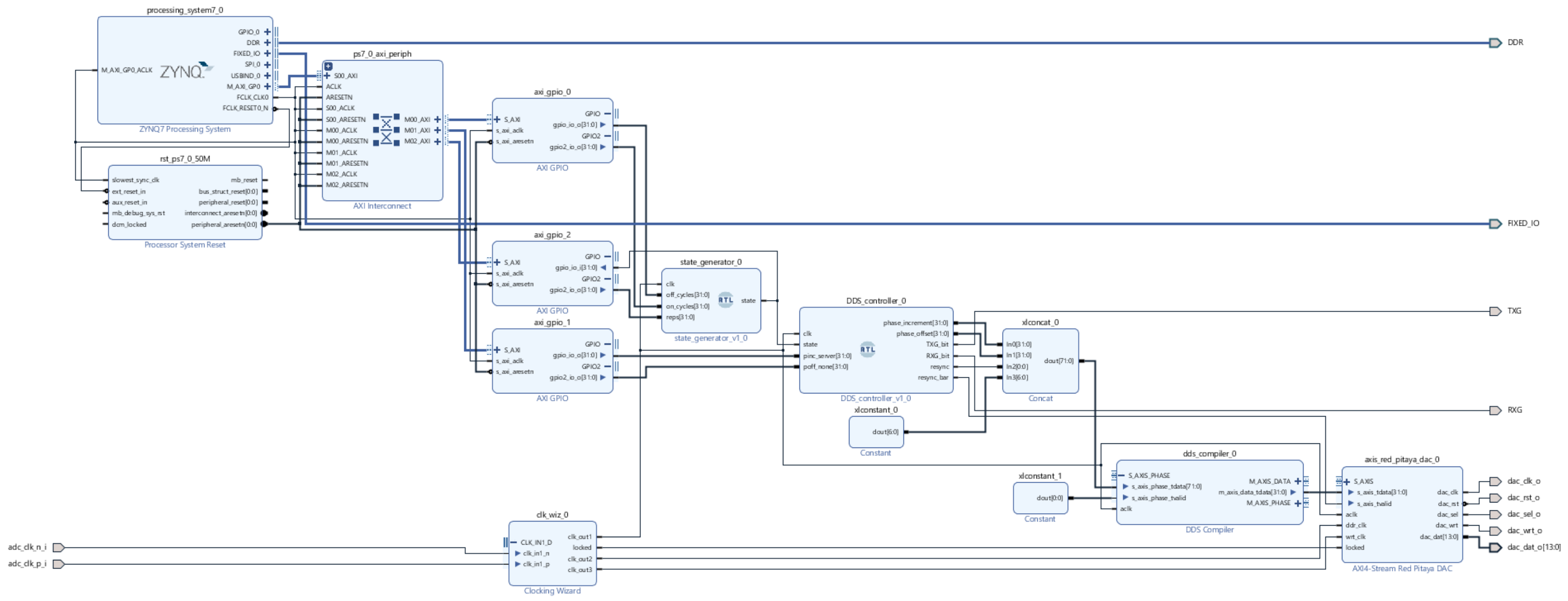
Toggle `M_AXIS_DATA` back to 'interface pin' for now.

3.6.2 The actual connections

Now that all the blocks, pins and ports are present, we can go ahead and connect them all up. In Diagram, left click on `clk_out1` of `clk_wiz_0`, release and left click on `clk` pin of `state_generator_0`. This makes one connection. Similarly, make all connections between pins, interface pins and ports to match the block diagram in the next page. Delete all unconnected ports from the diagram.

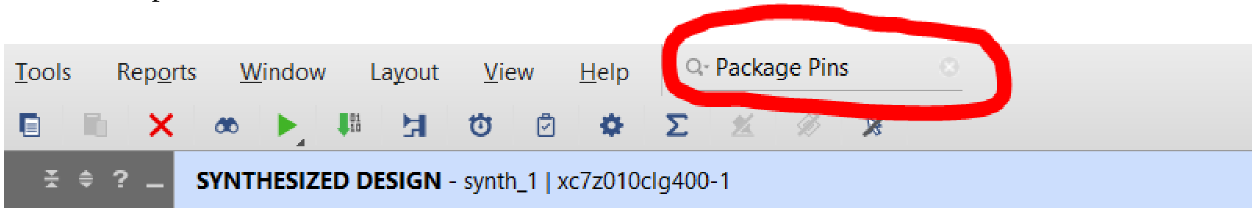
Your diagram may not exactly look like this – this is expected. If all the cores have the exact same connections as above, their placement does not matter. You can further clean up the diagram by pressing the 'Regenerate Layout' button in the diagram window – the one that is highlighted below.



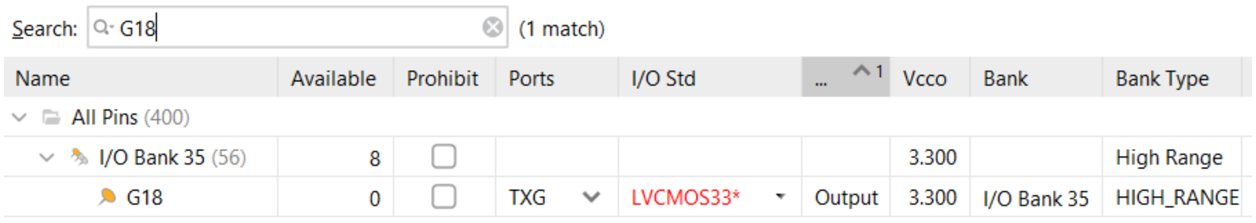


processing_system7_0
M_AXI_GP0_ACLK
ZYNQ
GPIO_0
DDR
FIXED_IO
SPI_0
USBIND_0
M_AXI_GP0
FCLK_CLK0
FCLK_RESET0_N
ZYNQ7 Processing System
ps7_0_axi_periph
S00_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
M01_ACLK
M01_ARESETN
M02_ACLK
M02_ARESETN
M00_AXI
M01_AXI
M02_AXI
AXI Interconnect
rst_ps7_0_50M
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
axi_gpio_0
S_AXI
s_axi_aclk
s_axi_aresetn
GPIO
gpio_io_o[31:0]
GPIO2
gpio2_io_o[31:0]
AXI GPIO
axi_gpio_2
S_AXI
s_axi_aclk
s_axi_aresetn
GPIO
gpio_io_i[31:0]
GPIO2
gpio2_io_o[31:0]
AXI GPIO
axi_gpio_1
S_AXI
s_axi_aclk
s_axi_aresetn
GPIO
gpio_io_o[31:0]
GPIO2
gpio2_io_o[31:0]
AXI GPIO
state_generator_0
clk
off_cycles[31:0]
on_cycles[31:0]
reps[31:0]
RTL
state
state_generator_v1_0
DDS_controller_0
clk
state
pinc_server[31:0]
poff_none[31:0]
RTL
phase_increment[31:0]
phase_offset[31:0]
TXG_bit
RXG_bit
resync
resync_bar
DDS_controller_v1_0
xlconstant_0
dout[6:0]
Constant
xlconcat_0
In0[31:0]
In1[31:0]
In2[0:0]
In3[6:0]
dout[71:0]
Concat
xlconstant_1
dout[0:0]
Constant
dds_compiler_0
S_AXIS_PHASE
s_axis_phase_tdata[71:0]
s_axis_phase_tvalid
aclk
M_AXIS_DATA
m_axis_data_tdata[31:0]
M_AXIS_PHASE
DDS Compiler
axis_red_pitaya_dac_0
S_AXIS
s_axis_tdata[31:0]
s_axis_tvalid
aclk
ddr_clk
wrt_clk
locked
dac_clk
dac_rst
dac_sel
dac_wrt
dac_dat[13:0]
AXI4-Stream Red Pitaya DAC
clk_wiz_0
CLK_IN1_D
clk_in1_n
clk_in1_p
clk_out1
locked
clk_out2
clk_out3
Clocking Wizard
adc_clk_n_i
adc_clk_p_i
DDR
FIXED_IO
TXG
RXG
dac_clk_o
dac_rst_o
dac_sel_o
dac_wrt_o
dac_dat_o[13:0]

Finally, go to Flow Navigator > Synthesis and hit 'Run Synthesis'. Press OK. A window with launch run critical messages might show up – simply press OK. Once the synthesis is complete, click on 'Open Synthesized Design'. Next, type 'Package Pins' in the search bar as indicated below, and press Enter.



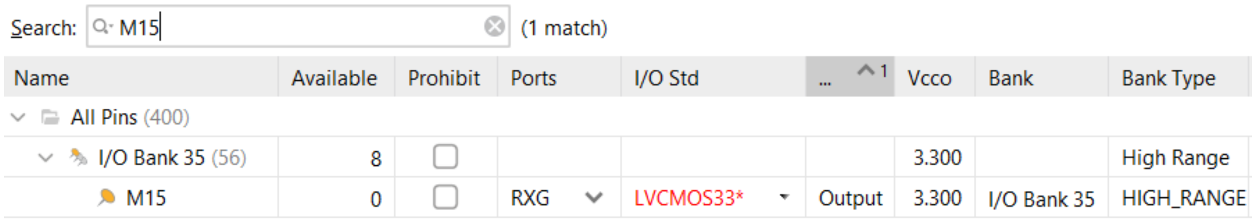
A sub-window named Package Pins will show up at the bottom. Use its search tool to find G18. In this instance, it is present in I/O bank 35. Now open the drop-down menu of I/O Bank 35 and look at the 'Ports' field of M15. Ensure that G18 is set like this.



Search: G18 (1 match)

Name	Available	Prohibit	Ports	I/O Std	...	Vcco	Bank	Bank Type
All Pins (400)								
I/O Bank 35 (56)	8	☐				3.300		High Range
G18	0	☐	TXG	LVCMOS33*	Output	3.300	I/O Bank 35	HIGH_RANGE

Repeat the same for M15. Ensure it is set like this. Save everything (Ctrl+S).



Search: M15 (1 match)

Name	Available	Prohibit	Ports	I/O Std	...	Vcco	Bank	Bank Type
All Pins (400)								
I/O Bank 35 (56)	8	☐				3.300		High Range
M15	0	☐	RXG	LVCMOS33*	Output	3.300	I/O Bank 35	HIGH_RANGE

3.7 The bitstream file

Click 'Generate Bitstream' under Flow Navigator>Program and Debug. This will take a few minutes. Once the bitstream has been successfully generated, navigate to the folder with the Vivado project file (a .xpr file). It is present under xxx.runs/impl_1/

The bitstream is name system_wrapper.bit. This will program the FPGA with the digital design we have built in Vivado.

3.8 Bitstream Testing

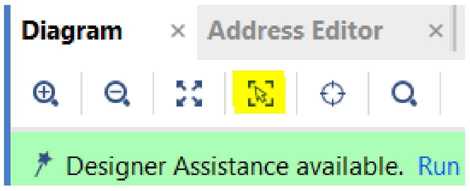
Testbenches are extremely useful for testing out functionality before programming it to an FPGA. It is important to know the parts that make up a testbench. The following diagram explains it in a little more detail.

Red Pitaya – Run simulations on cores/module script

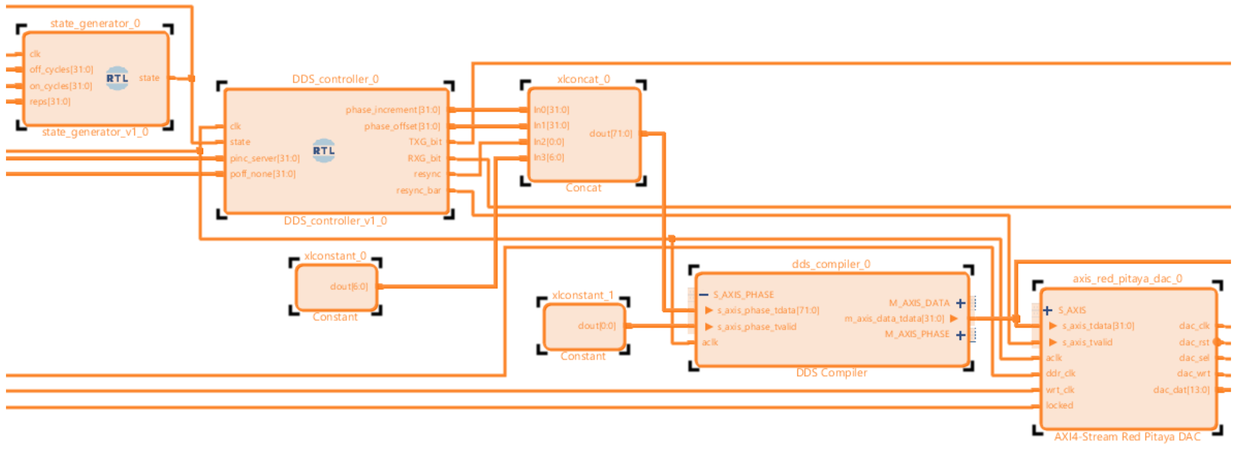
This guide allows for simulations to be run on the existing block design and/or .v module file, before the actual bitstream file is generated and activated in Red Pitaya.
It is very useful to check Verilog modules before integrating them into bigger block designs.

NOTE: Hierarchies
As you develop your block design, it will become increasingly complex, with many IP cores and connections cluttering the screen. There is a way to deal with the clutter - group each set of IP cores under a 'hierarchy', preferably by functionality. We will use a hierarchy to section off a portion of the block design that requires testing.

Go to IP Integrator > Open Block Design. Select the blocks whose functionality you want to simulate using the Select Area button in the Diagram window (see highlighted button below)



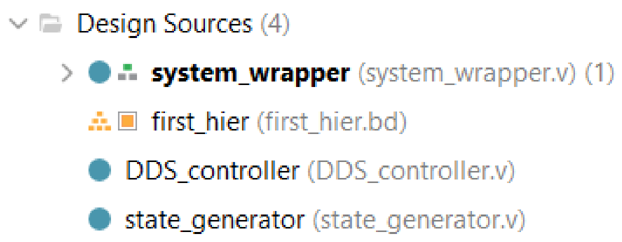
Use this button to select the following blocks:



Right-click on the selected set of blocks and click on Create Hierarchy from the drop-down menu. Leave the cell name as hier_0 and press OK. This will cause all the blocks to be housed under a single block called hier_0; this is our hierarchy.

Drag and drop `clk_wiz_0` in `hier_0`. We now have the complete set of blocks to be testbenched. Right-click on this newly created hierarchy and select Copy. Go to IP Integrator > Create Block Design, and give the block design an appropriate name, e.g., `first_hier`. In the Diagram workspace right-click anywhere, select Paste, and the hierarchy will be copied.

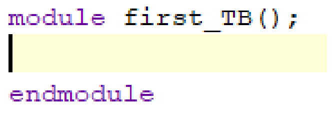
To complete the block design, ports must be added. For this, simply right click on the hierarchy block again, and select Make external. This will generate all the required input and output ports. Under ‘Design Sources’, the new block design will show up as a .bd file (see figure 1). Right click it and select ‘Create HDL wrapper’. Select ‘Let Vivado manage wrapper and auto-update’.


Design Sources (4)
system_wrapper (system_wrapper.v) (1)
first_hier (first_hier.bd)
DDS_controller (DDS_controller.v)
state_generator (state_generator.v)

Once complete, the process will generate a .v file corresponding to the block design. Right-click this and select ‘Set as top’.

Go back to the previous block design, right-click on the hierarchy block and select Ungroup Hierarchy. This will revert the block design to what it was before the hierarchy.
Now the testbench code will be created. Go the Flow Navigator > Add Sources and select Add or create simulation sources. Click on Create file, give an appropriate name (e.g. `first_TB`), and hit Finish. When prompted for module inputs and outputs, do not specify any.
The new simulation source (a .v file) will show up under Sources > Simulation sources. Expand this menu, and click on this .v file.

Go to Design Sources, and double click on the `first_hier_wrapper.v` file (source code). Copy all lines from module `first_hier_wrapper` up to `output state_0`.
Go to the file `first_TB.v` and in between the module definition and `endmodule` (in the highlighted yellow portion below), paste the previously copied lines of code.


```
module first_TB();

endmodule
```

After the paste, rearrange the lines such that the ones with `input` and `output` keywords are placed between the two lines having `module` keywords. Change all `input` keywords to `reg` and `output` keywords to `wire`.

Go to the line with `module first_hier_wrapper` and change it to `first_hier_wrapper DUT`. In the port list, have all ports with a . prefix, followed by the same input/output name in brackets. It should match the diagram below.

```
first_hier_wrapper DUT
   (.RXG_0(RXG_0),
    .TXG_0(TXG_0),
    .adc_clk_n_i_0(adc_clk_n_i_0),
    .adc_clk_p_i_0(adc_clk_p_i_0),
    .dac_clk_o_0(dac_clk_o_0),
    .dac_dat_o_0(dac_dat_o_0),
    .dac_rst_o_0(dac_rst_o_0),
    .dac_sel_o_0(dac_sel_o_0),
    .dac_wrt_o_0(dac_wrt_o_0),
    .led_o_0(led_o_0),
    .off_cycles_0(off_cycles_0),
    .on_cycles_0(on_cycles_0),
    .pinc_server_0(pinc_server_0),
    .poff_none_0(poff_none_0),
    .state_0(state_0),
    .reps_0(reps_0));
```

Since our hierarchy is sequential, it needs to be driven via a clock signal. Red Pitaya has a clock signal of 125 MHz – this makes the time period 8 ns. Go the top of the file and type in `` `define time_period 8 ``.

We have two differential adc_clk_i pins for the clock signal input – `adc_clk_n_i_0` and `adc_clk_p_i_0`. These need to be initialized to make Clocking Wizard work. In the same file, go to their reg definitions, set `adc_clk_n_i_0` as 0 and `adc_clk_p_i_0` as 1. Next, the clock signal needs to toggle at 50% duty cycle – or half the time period. Go to the end of the port list and type in the following:

```
initial
begin
forever #(`time_period/2) begin
adc_clk_n_i_0 <= ~adc_clk_n_i_0; //toggle every 4 ns
adc_clk_p_i_0 <= ~adc_clk_p_i_0; //toggle every 4 ns
end
end
```

After this, append the following lines of code to the end of the testbench file, right before `endmodule`:

```
initial begin

//Enter first set of input parameters
  off_cycles_0 <= 20; //RF OFF for 20 clock cycles
  on_cycles_0  <= 40; //RF ON for 40 clock cycles
  pinc_server_0 <= 25769804; //3 MHz
  poff_none_0 <= 0; //0 degrees
  reps_0 <= 3;      //run sequence 3 times
```

```
    #5000         //retain above values for 5 us

    //Blankout sequence begins - to reset pulse generator
      off_cycles_0 <= 0;
      on_cycles_0 <= 0;
      pinc_server_0 <= 0;
      poff_none_0 <= 0 ;
      reps_0 <= 0;
    #5000         //retain above values for 5 us

    //Enter second set of input parameters
      off_cycles_0 <= 100; //RF OFF for 100 clock cycles
      on_cycles_0 <= 30;   //RF ON for 30 clock cycles
      pinc_server_0 <= 85899346; //10 MHz
      reps_0 <= 4;  //run sequence 4 times
      poff_none_0 <= 268435456; //90 degrees
    #5000         //retain above values for 5 us

    //Blankout sequence begins - to reset pulse generator
      off_cycles_0 <= 0;
      on_cycles_0 <= 0;
      pinc_server_0 <= 0;
      reps_0 <= 0;
      poff_none_0 <= 0;
    #5000         //retain above values for 5 us

    //Enter third set of input parameters
      off_cycles_0 <= 50;  //RF OFF for 50 clock cycles
      on_cycles_0 <= 50;   //RF ON for 50 clock cycles
      pinc_server_0 <= 128849019; //15 MHz
      reps_0 <= 5;   //run sequence 5 times
      poff_none_0 <= 536870912; //180 degrees
    #5000         //retain above values for 5 us

      $finish;   //completed the simulation
    end
```

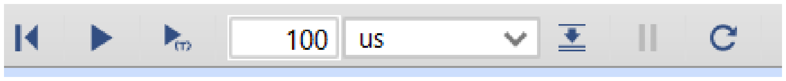
Set the testbench code `first_TB.v` as top. Go to Flow Navigator>Simulation>Run Simulation>Run Behavioral Simulation and a waveform plot will show up. Open it and note that the inputs are flatlines – this is expected, since these were not set in the testbench code! The clock input (if present) would be running continuously regardless – this was already set in the code.

If you would like to see an analog signal on a multi-bit output (such as that of DDS or DAC), follow the steps below.
Go to the simulation wave window and right-click on `dac_dat_o_0[13:0]`. Go to Waveform Style> Analog Settings. Under Analog Settings, make sure to set the interpolation style as 'hold'.

Select 'Apply'. Next, right-click on `dac_dat_o_0[13:0]` again and go to Radix > Signed Decimal.

To display DDS output on the waveform window, go to Scope > first_TB > DUT > first_hier_i > hier_0 and right-click on `dds_compiler_0`. Select 'Add to Wave Window'. This causes all the inputs and outputs of DDS Compiler to show up on the simulation wave window. Right-click on `m_axis_data_tdata[31:0]` and follow the same steps mentioned in the previous paragraph to set it to analog with 'hold' interpolation style, with radix as 'Signed Decimal'.

Finally, hit the play button (the one with an inverted omega subscript), as shown in the figure. Make sure to set the time duration in the text box before pressing the button.



The simulation will now run for the stipulated amount of time, generating a waveform corresponding to the wrapper code (DUT, or device under test). After the simulation is complete, it is advised to set `system_wrapper.v` under Sources>Design Sources back as top. Without this, a bitstream file corresponding to 'system' cannot be generated.

4 User Interface

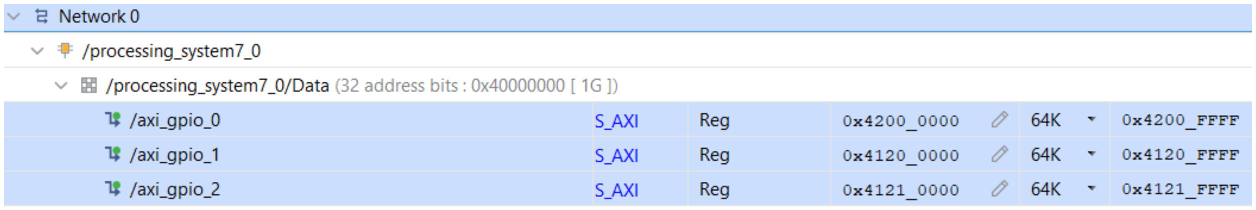
Now that the design has been testbenched and bitstream generated, we need to build an interface around it. There are two parts to it – the server (resides on Pitaya) and the client (resides on the PC).

4.1 Server

The idea behind our server is to make our custom design available for user control. For this to work, the server must correctly map to AXI GPIO addresses of the bitstream-activated FPGA. Also, it needs to establish a socket connection with the PC to accept user inputs remotely.

4.1.1 AXI GPIO addresses

These need to be set. Go to Flow Navigator>IP Integrator and in the search box type Address Editor. Note that `axi_gpio_0` is assigned while the other two AXI GPIOs are unassigned. Right click on Network 0 and press 'Assign All'. Ensure the ranges for all three are 64K; they should now look like the menu below. We will incorporate these addresses in the C code.



Network 0

/processing_system7_0

/processing_system7_0/Data (32 address bits : 0x40000000 [1G])

/axi_gpio_0	S_AXI	Reg	0x4200_0000	64K	0x4200_FFFF
/axi_gpio_1	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
/axi_gpio_2	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF

4.1.2 C program

This server code integrates with the AXI GPIO addresses set previously. Simply copy-paste this into a C file and save it to the PC as `first_server_program.c`.

```
#include <sys/socket.h>
#include <arpa/inet.h> //inet_addr
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>    //write
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>

#pragma pack(1)

typedef struct payload_t {
    //NOTE: All these inputs are presumed to be in microseconds.
    //we get these from the Python client, and send to bitstream
 GPIO
    uint32_t Off_time;
    uint32_t On_time;
    uint32_t Reps;
    float frequency;
} payload;

#pragma pack()

int main(int argc, char** argv)
{
    //Step 1: Initialize socket stuff
    int PORT = 2300;
    int BUFFSIZE = 512;
    char buff[BUFFSIZE];
    int ssock, csock;
    int nread;
    struct sockaddr_in client;
    int clilen = sizeof(client);

    //Step 2: Initialize bitstream control stuff
    int fd;
    char *name = "/dev/mem";
    void *cfg_Off_On_time, *cfg_pinc_poff, *cfg_state_reps; /*po
inters to address/memory locations used by bitstream's GPIO*/
    uint32_t phase_inc = 1<<30; //phase increment - fed to DDS
    uint32_t Off_time, On_time, Reps; float frequency; //we get
this from the Python client
    int freq_MHz = 125; // 125 MHz - multiply with pulse timing
params

   if((fd = open(name, O_RDWR)) < 0)
  {
    perror("open");
    return EXIT_FAILURE;
  }

    /* Mapping to addresses specified by bitstream's AXI GPIO
    (Please refer to your block design's 'Address Editor' in Viv
ado for this) */
    cfg_Off_On_time = mmap(NULL, sysconf(_SC_PAGESIZE),PROT_READ
|PROT_WRITE, MAP_SHARED, fd, 0x42000000);
    cfg_pinc_poff = mmap(NULL, sysconf(_SC_PAGESIZE),PROT_READ|P
ROT_WRITE, MAP_SHARED, fd, 0x41200000);
    cfg_state_reps = mmap(NULL, sysconf(_SC_PAGESIZE),PROT_READ|
PROT_WRITE, MAP_SHARED, fd, 0x41210000);

    struct sockaddr_in server;

    if ((ssock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("ERROR: Socket creation failed\n");
        exit(1);
    }
    printf("Socket created\n");

    bzero((char *) &server, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(PORT);
    if (bind(ssock, (struct sockaddr *)&server , sizeof(server))
 < 0)
    {
        printf("ERROR: Bind failed\n");
        exit(1);
    }
    printf("Bind done\n");
    listen(ssock, 3);
    printf("Server listening on port %d\n", PORT);

    while (1)
    {
        csock = accept(ssock, (struct sockaddr *)&client, &clile
n);
        if (csock < 0)
        {
            printf("Error: accept() failed\n");
            continue;
        }

        printf("Accepted connection from %s\n", inet_ntoa(client
.sin_addr));
        bzero(buff, BUFFSIZE);
        while ((nread=read(csock, buff, BUFFSIZE)) > 0)
        {
            printf("Received %d bytes\n", nread);
            payload *p = (payload*) buff;

            //Print received inputs from Python client
            printf("\nReceived contents:\n");
            printf("Off_time \t = %d us\n", p->Off_time);
            printf("On_time \t = %d us\n", p->On_time);
            printf("Repetitions\t = %d\n", p->Reps);
            printf("Frequency\t = %0.6f MHz\n", p->frequency);

            Off_time = p->Off_time;
            On_time = p->On_time;
            Reps = p->Reps;
            frequency = p->frequency;
        }

    float val = 125/frequency;
    phase_inc /= val; //converting phase increment to final DDS
value
    printf("\nConverted phase increment = %d", phase_inc);

    *((uint32_t *)(cfg_Off_On_time + 0)) = Off_time*freq_MHz;
// convert Off_time to number of clocks
    *((uint32_t *)(cfg_Off_On_time + 8)) = On_time*freq_MHz;
// convert On_time to number of clocks
    *((uint32_t *)(cfg_pinc_poff + 0)) = phase_inc;
// send phase increment
    *((uint32_t *)(cfg_pinc_poff + 8)) = 0;
// send phase offset
    *((uint32_t *)(cfg_state_reps + 8)) = Reps;
// send repetitions

    int total_time = (Off_time + On_time) *Reps;
    total_time /= 1000000; //convert from microseconds to second
s
    printf("\nTime to complete pulse sequence = %d seconds", tot
al_time);
    sleep(total_time);

    printf("\nNow blanking out the pulse parameters...");
    *((uint32_t *)(cfg_Off_On_time + 0)) = 0;
    *((uint32_t *)(cfg_Off_On_time + 8)) = 0;
    *((uint32_t *)(cfg_pinc_poff + 0)) = 0;
    *((uint32_t *)(cfg_pinc_poff + 8)) = 0;
    *((uint32_t *)(cfg_state_reps + 8)) = 0;
    sleep(1);                               //blank out for 1
second
    printf("\nOperation complete.");
    sleep(1);
    printf("\nClosing connection to client\n");
    sleep(1);
    printf("----------------------------\n");
    sleep(1);
    close(csock);
    return 0;
    }
}
```

4.2 Client

Now that the server program exists, the client needs to be built. This will allow the PC user to enter inputs and hand them over to server via the socket connection. Copy-paste the code below into a .py file and save it to PC as `first_client_program.py`. Make sure to replace `abc.def.ghi.jkl` with IP address of Red Pitaya. This is listed under System > Network manager.

```
#!/usr/bin/env python

import socket
import sys
from datetime import datetime #to name the MR parameter .txt
file
from ctypes import *

""" This class defines a C-like struct """
class Payload(Structure):
    _fields_ = [("Off_time", c_uint32), ("On_time", c_uint32),
                ("Reps", c_uint32), ("Frequency", c_float)]
```

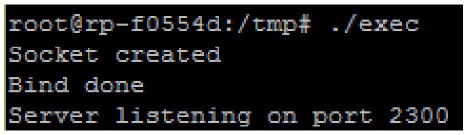
```
def main():
    redpitaya = "abc.def.ghi.jkl" #Replace with Pitaya’s IP
address
    server_addr = (redpitaya, 2300)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        s.connect(server_addr)
        print ("Connected to %s" % repr(server_addr))
    except:
        print ("ERROR: Connection to %s refused" %
repr(server_addr))
        sys.exit(1)

    try:
            #Send data to server
            print ("")
            payload_out = payload_out =
Payload(2000,5000,35,30.72)
            print("Sending these params:")
            print("Off_time \t = %d us \nOn_time \t = %d us"
                  "\nRepetitions\t = %d \nFrequency\t = %.6f
MHz\n"
                                        % (payload_out.Off_time,
                                           payload_out.On_time,
                                           payload_out.Reps,
                                           payload_out.Frequency
                                           ))
            nsent = s.send(payload_out)
            print ("Sent %d bytes" % nsent)
            ct = datetime.now()
            stamp = ct.strftime("%b_%d_%Y_%I_%M_%S_%p")
            filename = "MR_data_" + stamp + ".txt"
            with open(filename, "a") as txt_file:
                txt_file.write('Sent out the following:\n')
                txt_file.write('Off_time = %d us\n' %
(payload_out.Off_time))
                txt_file.write('On_time = %d us\n' %
(payload_out.On_time))
                txt_file.write('Repetitions = %d\n' %
(payload_out.Reps))
                txt_file.write('Frequency = %.6f MHz\n' %
(payload_out.Frequency))
                txt_file.write("\n")
    finally:
        print ("Closing socket")
        s.close()

if __name__ == "__main__":
    main()
```

4.3 Putting them together

Open WinSCP and log in to Red Pitaya. Go to the directory containing `first_server_program.c` and drag-drop it into Pitaya's `/tmp` directory. Next, go to the directory containing `system_wrapper.bit` from the project folder `your_base_directory\base_project\tmp\your_first_project\your_first_project.runs\impl_1\`. Drag-drop `system_wrapper.bit` from here into Pitaya's `/tmp` directory.
Within WinSCP open a PuTTY terminal, log in with 'root' as password and type in the following commands:

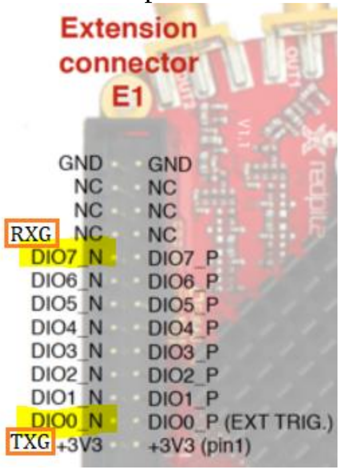
```
cd /tmp
cat system_wrapper.bit > /dev/xdevcfg
chmod u+x first_server_program.c
gcc -o exec first_server_program.c
./exec
```

The second command activates the bitstream. The commands that follow are meant to create a binary executable from the C code and run it. If successful, Terminal will look like this:

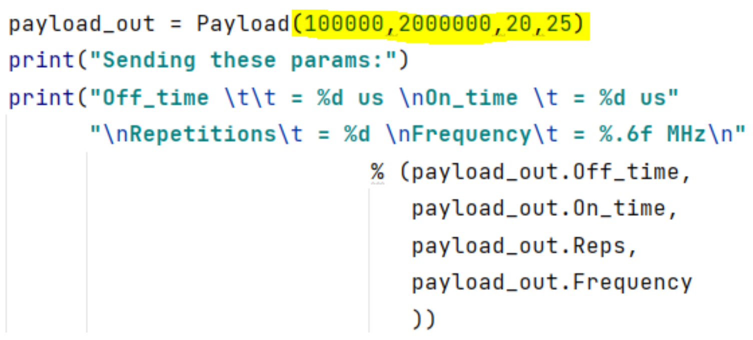


```
root@rp-f0554d:/tmp# ./exec
Socket created
Bind done
Server listening on port 2300
```

Connect OUT1 to an oscilloscope for the RF pulses. The TXG and RXG signals may be read off the indicated GPIO lines – simply hook them up to an oscilloscope to verify.



Now open first_client_program.py in PyCharm. Note the highlighted numbers – these are the parameters you can set for the pulse generator you just built. The second print explains what each parameter is, along with their units (if any).



```
payload_out = Payload(100000,2000000,20,25)
print("Sending these params:")
print("Off_time \t\t = %d us \nOn_time \t = %d us"
      "\nRepetitions\t = %d \nFrequency\t = %.6f MHz\n"
                                 % (payload_out.Off_time,
                                    payload_out.On_time,
                                    payload_out.Reps,
                                    payload_out.Frequency
                                    ))
```

Go to Run>Run>first_client_program. Upon success you will note RF pulse generated from OUT1 of Pitaya (to verify this, hook up OUT1 to an oscilloscope) along with a timestamped .txt file created in the same directory as the Python code. This is a record of all parameters set in the current pulse sequence – very useful for future reference.

5 Version Control

This outlines the process of creating copies of your Vivado project for version control. If a Vivado project runs into errors, it is good to have backup copies like these.

1. Go to File>Project>Write Tcl.
2. Select the output directory of your choice, and name it as `build.tcl` in the output file filed, along with the rest of the directory path. Make sure 'Copy sources to new project' and 'Recreate block designs using Tcl' is checked.
3. Go to the directory where the .tcl file has been generated. Once there, copy the `cores`, `scripts` and `cfg` folders into the same folder as `build.tcl`. If your design has RTL scripts or custom Pitaya cores, copy `tmp` into the folder as well.
4. Open a new Vivado window. In the Tcl console, type the following:
   ```
   cd
   cd \path\to\build.tcl
   source build.tcl
   ```
5. Success! Your project is fully built from the Tcl file.

6 Troubleshooting

• An internal exception has been detected, Vivado may be in unstable state, would you like to exit now?
This shows up randomly when using Vivado. The software seems to trip up after using it for a while. When this happens, be sure to hit Ctrl+S to save all your work, exit Vivado and re-launch Vivado. We don't have a better fix for Vivado's internal exception at this time.

• Unable to login to a PuTTY or WinSCP session of Pitaya, despite being able to log in before
This is common. For some reason, the IP address of the device reconfigures occasionally. This can cause the device to have two, even three possible IP addresses.
If this happens open its webpage (rp-xxxxxx) in a web browser. Go to System > Network manager and note the IP address from there. Use this address to launch PuTTY and/or WinSCP sessions for Pitaya.

• Bus error in PuTTY window
There are two possibilities as to why this happens:
1. The C and/or Python codes from chapter 3 are used with the bitstream from chapter 4
2. The C and/or Python codes from chapter 4 are used with the bitstream from chapter 3

The C and Python programs used in chapter 3 are included in the unzipped folder mr_experiment_SE. Make sure to use this with the bitstream from mr_experiment_SE only.
The C and Python programs in chapter 4 are not included in the unzipped folder base_project. You need to manually copy-paste these into new C and Python files before use.

7 Useful Bash Commands

•`cd`: Change directory command - for moving to a specific directory
Syntax: `cd ⟨path/to/source⟩ ⟨path/to/destination⟩`

•`cp`: Copy command - for copying files or folders
Syntax: `cp ⟨path/to/source⟩ ⟨path/to/destination⟩`

•`rm`: Remove command - for deleting files or folders
Syntax: `rm ⟨path/to/source⟩ ⟨path/to/destination⟩`

•`chmod`: Change mode command - for changing the access permissions to files or folders
Syntax: `chmod ⟨path/to/source⟩ ⟨path/to/destination⟩`

•`cat:` Concatenate command - for sequentially writing files to a standard output
Syntax: `cat ⟨path/to/source⟩ ⟨path/to/destination⟩`

APPENDIX F

CORRELATING PULSE GENERATION WITH MRI

1. Executing MR Pulse Generator – from mr_experiment_SE

The timed events are set by the following spin echo (SE) pulse parameters, along with their units:

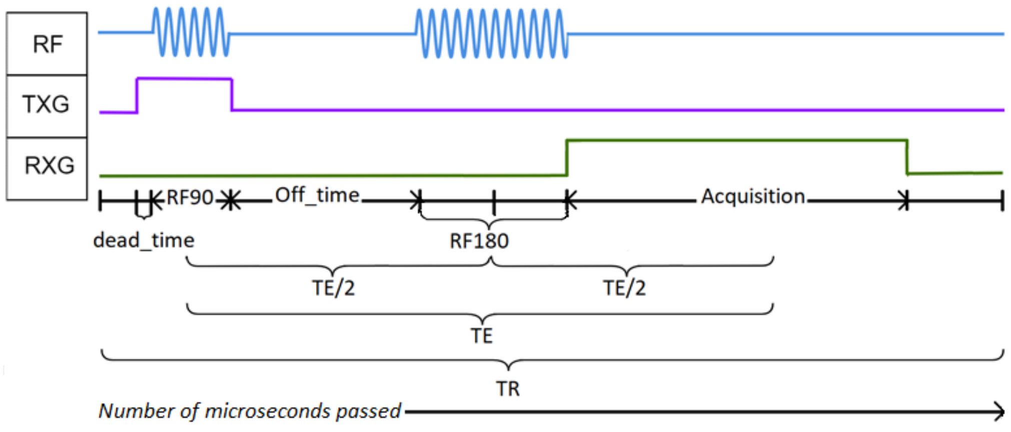
RF90 (in µs)	–	time used to tip magnetic spins by 90°
Off_time (in µs)	–	time between RF90 end and RF180 beginning when RF output is a flatline
RF180 (in µs)	–	time used to tip magnetic spins by 180°
TR (in µs)	–	time after which all pulse events repeat
Acquisition (in µs)	–	time slot after RF180 end when MR echo is expected to form
Repetitions	–	number of times the SE pulse sequence repeats
Frequency (in MHz)	–	Larmor frequency of RF pulses

These events are used to set RF pulses for NMR applications. Each RF pulse generated here is directly tied to the NMR phenomenon, which is governed by the equations for Larmor frequency f_{Larmor} and tip angle α given below.

$$f_{Larmor} = \bar{\gamma} B_0 \qquad (1)$$
$$\alpha = \gamma B_1' t \qquad (2)$$

where $\bar{\gamma}$ = gyromagnetic ratio of sample in Hz T^{-1} H
 γ = gyromagnetic ratio of sample in rad s^{-1} T^{-1}
 B_1'= 0.5 × B field of the solenoid/RF coil
 B_0 = B field of the magnet

Below is a timing diagram demonstrating one repetition of the SE pulse sequence used in mr_experiment_SE, as a function of microseconds.



Timing Diagram for Pulse States - Spin Echo (SE)

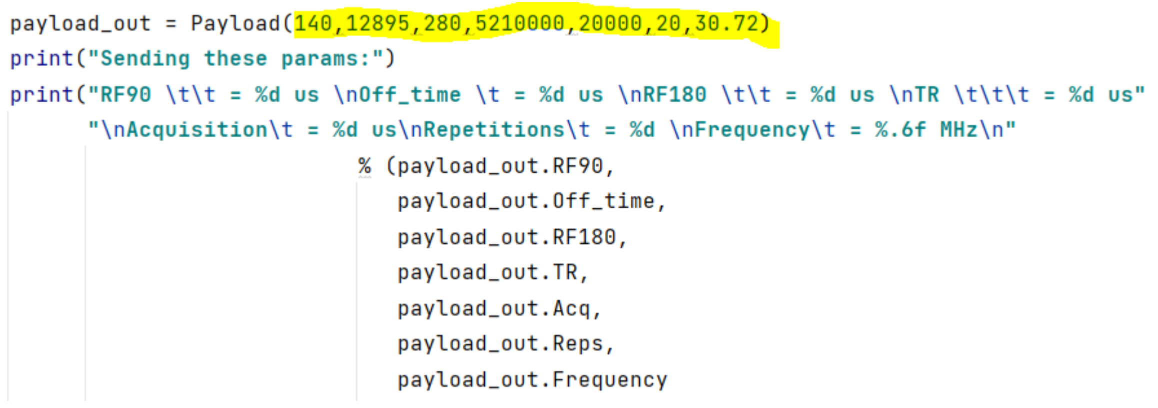
The value dead_time is hard coded within the digital circuit to be 100 µs. It indicates the time period before RF90 begins when TXG pulse is activated. It is useful for gating purposes.
Also note the presence of TE – the echo time. In SE, this is the distance between the center of RF90 and center of the MR echo. This parameter is not an input to the console – it merely indicates the location where the center of MR echo will show up. It is used to make sure the Acquisition time input is large enough to allow digitizing the echo; if it includes the echo center, the digitizer is capturing the right RF data from the receive side.

To implement the above timing diagram, the provided digital circuit used a counter called 'pulse era' to keep track of time passed. According to the range pulse era lies in, a state value is assigned which corresponds to RF, TXG and RXG outputs being activated or deactivated according to the table below.

Pulse Era	State Value	RF	TXG	RXG
0 ~ dead_time	0	OFF	1	0
dead_time ~ dead_time + RF90	1	ON	1	0
dead_time + RF90 ~ dead_time + RF90 + Off_time	2	OFF	0	0
dead_time + RF90 + Off_time ~ dead_time + RF90 + Off_time + RF180	4	ON	0	0
dead_time + RF90 + Off_time + RF180 ~ dead_time + RF90 + Off_time + RF180 + Acq	3	OFF	0	1
dead_time + RF90 + Off_time + RF180 + Acq ~ TR	2	OFF	0	0

MR Pulse State Generation in Verilog

The pulse parameters are set in the client program (client_program.py) in the portion highlighted below.



```
payload_out = Payload(140,12895,280,5210000,20000,20,30.72)
print("Sending these params:")
print("RF90 \t\t = %d us \nOff_time \t = %d us \nRF180 \t\t = %d us \nTR \t\t\t = %d us"
      "\nAcquisition\t = %d us\nRepetitions\t = %d \nFrequency\t = %.6f MHz\n"
                          % (payload_out.RF90,
                             payload_out.Off_time,
                             payload_out.RF180,
                             payload_out.TR,
                             payload_out.Acq,
                             payload_out.Reps,
                             payload_out.Frequency
```

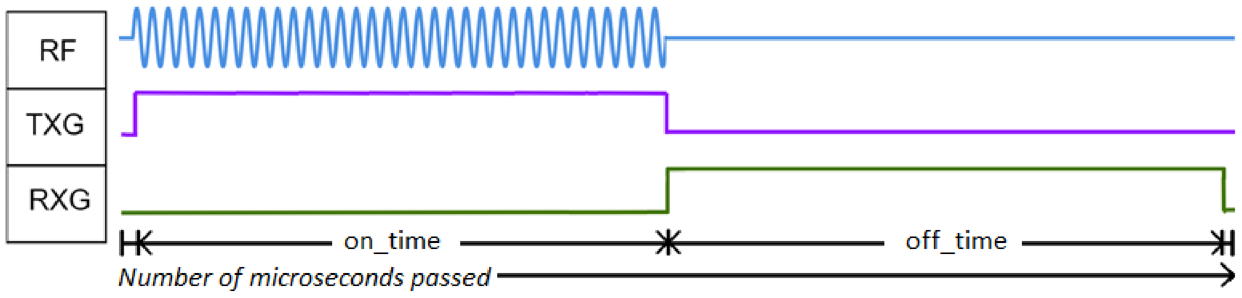
After running the server code server_program.c followed by client code client_program.py as outlined in the manual in Appendix E, RF and GPIO (i.e., TXG and RXG) outputs can be used to drive MR experiments.

2. Executing RF Pulse Generator – from base_project

The timed events are set by the following RF pulse parameters:

on_time	–	number of microseconds for which RF pulse and TXG signal is generated
off_time	–	number of microseconds for which only RXG signal is generated
Frequency	–	frequency of RF pulse
Repetitions	–	number of times the RF sequence repeats

These are NOT tied to the MR phenomenon – they are meant to demonstrate a simple RF pulse generator. Below is a timing diagram demonstrating one repetition of the RF pulse sequence used in base_project, as a function of microseconds.



Timing Diagram for Pulse States – RF pulses

To implement the above timing diagram, the digital circuit created in base_project (after following instructions from the provided user manual in Appendix E) utilizes a counter called 'pulse era' to keep track of number of clock cycles passed. According to the range pulse era lies in, a state value is assigned which corresponds to RF, TXG and RXG outputs being activated or deactivated according to the table below.

Pulse Era	State Value	RF	TXG	RXG
0 ~ off_time	0	OFF	0	1
off_time ~ off_time + on_time	1	ON	1	0

RF Pulse State Generation in Verilog

While these are the parameters visible on the user side, the first three need to be converted into values usable by the digital circuit. This conversion is done by the server program first_server_program.c in the manual. Basically, it translates the parameters on_time, off_time and frequency into on_cycles, off_cycles, and phase_inc and phase_off. These values are described below.

on_cycles	–	number of clock cycles for which RF pulse and TXG signal is generated
off_cycles	–	number of clock cycles for which only RXG signal is generated
phase_inc	–	determines frequency of RF pulse
phase_off	–	determines initial phase of RF pulse

In simple terms, phase increment and phase offset are used to control the frequency and initial/starting phase of the RF pulse(s) to be generated. Our digital circuit uses a DDS (Direct Digital Synthesizer) to generate sine pulses, hence it is crucial that its phase increment and phase offset inputs are computed correctly before sending in as inputs.

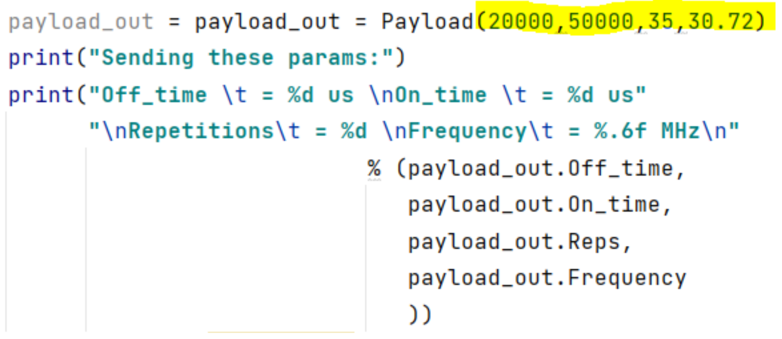
Phase increment decides how fast the wave amplitude changes with each clock cycle, while phase offset determines the starting angle value for which the sine wave is generated. They are governed by the formulas given below.

$$phase\ increment, p_{inc} = \frac{f_{out} \times 2^{B_{\theta}(n)}}{f_{in}} \tag{1}$$

$$phase\ offset, p_{off} = \frac{2^{B_{\theta}(n)} \times \alpha}{360} \tag{2}$$

Note that first_server_program.c in base_project sets phase_offset (i.e., p_off) as 0. While the digital circuit accepts this as a processed value input, this value has not been incorporated as a Python parameter nor any conversion has been done by first_server_project.c. An upgrade to base_project may be to add an input called ‘Angle Offset’ in first_client_program.py, convert to the required value of phase_offset in first_server_project.c, and send this converted value to the digital circuit via AXI GPIO instead of 0.

These conversions by first_server_program.c ensure that the correct values are passed on to the digital circuit for RF and GPIO output generation.
Overall, pulse parameters are set in the client program (client_program.py) in the portion highlighted below.



```
payload_out = payload_out = Payload(20000,50000,35,30.72)
print("Sending these params:")
print("Off_time \t = %d us \nOn_time \t = %d us"
      "\nRepetitions\t = %d \nFrequency\t = %.6f MHz\n"
                                  % (payload_out.Off_time,
                                     payload_out.On_time,
                                     payload_out.Reps,
                                     payload_out.Frequency
                                     ))
```

After running the server code first_server_program.c followed by client code first_client_program.py as outlined in the manual in Appendix E, RF and GPIO (i.e., TXG and RXG) outputs can be viewed on an oscilloscope.

3. Simulating RF Pulse Generator – from base_project

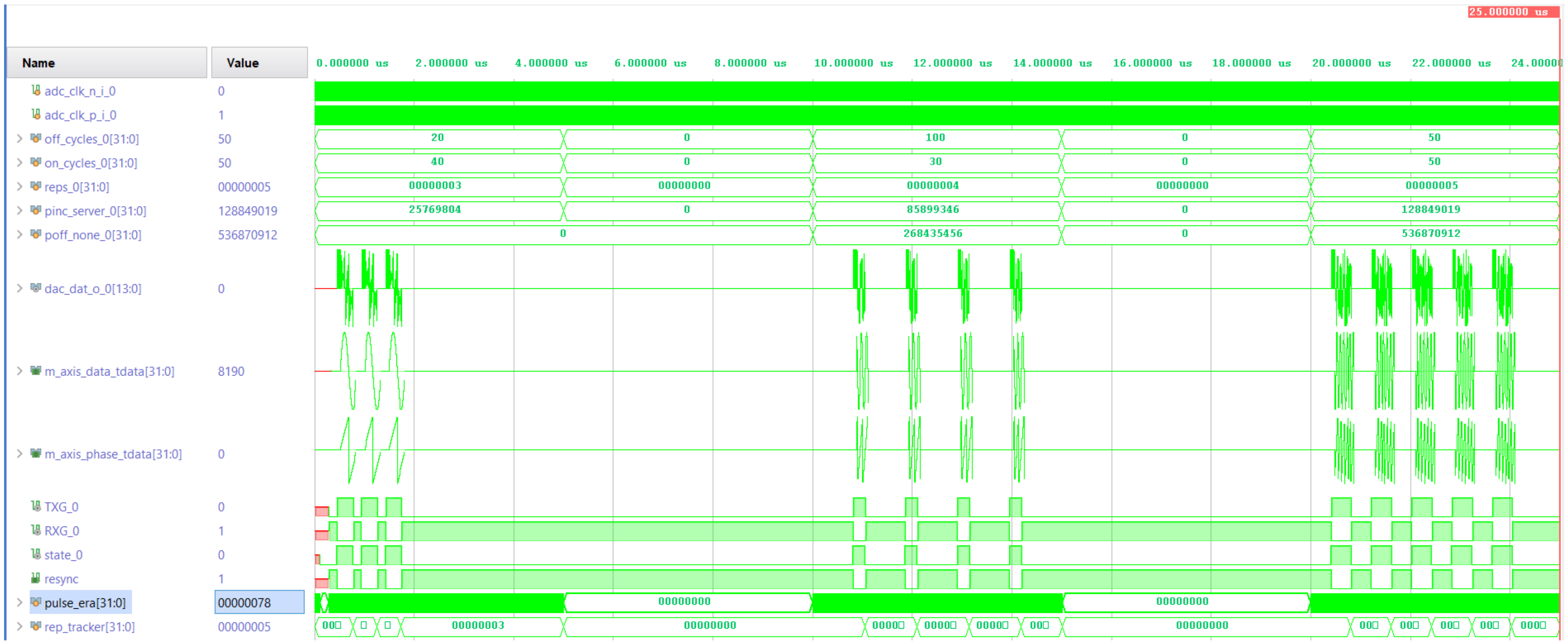
As mentioned in chapter 4 of the manual, a portion of the digital circuit (the hierarchy hier_0) is taken and simulated against a set of input values using the testbench program provided – first_TB.v. It sends out 5 sets of input values to the sectioned-out digital circuit, i.e., hier_0. This is done by sending one input set, waiting for a duration of 5 microseconds, and then sending out the next input set. All values used are listed in the table below. Note that values are sent in directly to the digital circuit without any conversions. The RF pulse parameters corresponding to these values are listed in brackets.

Sequence	#1	#2	#3	#4	#5
Duration (in µs)	5	5	5	5	5
off_cycles	20 (0.160 µs)	0 (0.000 µs)	100 (0.800 µs)	0 (0.000 µs)	50 (0.400 µs)
on_cycles	40 (0.320 µs)	0 (0.000 µs)	30 (0.240 µs)	0 (0.000 µs)	50 (0.400 µs)
phase_inc	25769804 (3 MHz)	0 (0 MHz)	85899346 (10 MHz)	0 (0 MHz)	128849019 (15 MHz)
phase_off	0 (0°)	0 (0°)	268435456 (90°)	0 (0°)	536870912 (180°)
Repetitions	3	0	4	0	5

Values used in XSim simulation

Observe sequences #2 and #4 – these effectively send out zeros to the digital circuit and wait for 5 microseconds. The result of this is shown below – rep_tracker is reset after accepting a new set of inputs, causing the corresponding sequence to run for the requested number of repetitions. Hence, it is important to perform a 'blank out' operation on the inputs every time a new sequence of RF and GPIO pulses must be generated.

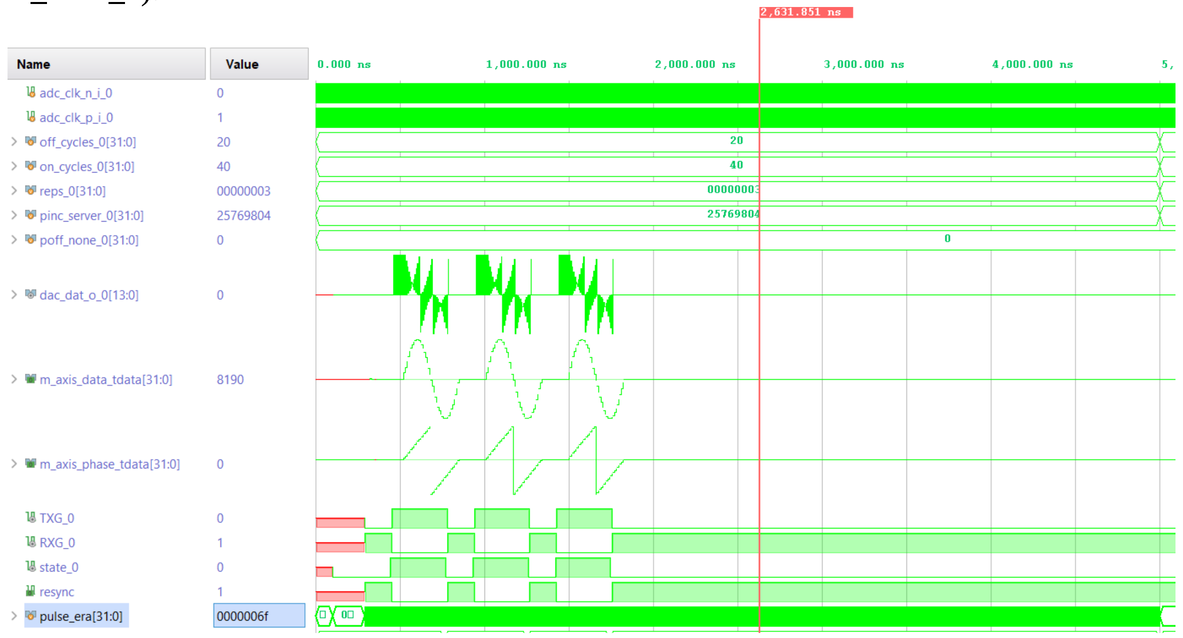
Upon a successful run, XSim generates the simulation results as shown on the next page.



Name	Value
adc_clk_n_i_0	0
adc_clk_p_i_0	1
off_cycles_0[31:0]	50
on_cycles_0[31:0]	50
reps_0[31:0]	00000005
pinc_server_0[31:0]	128849019
poff_none_0[31:0]	536870912
dac_dat_o_0[13:0]	0
m_axis_data_tdata[31:0]	8190
m_axis_phase_tdata[31:0]	0
TXG_0	0
RXG_0	1
state_0	0
resync	1
pulse_era[31:0]	00000078
rep_tracker[31:0]	00000005

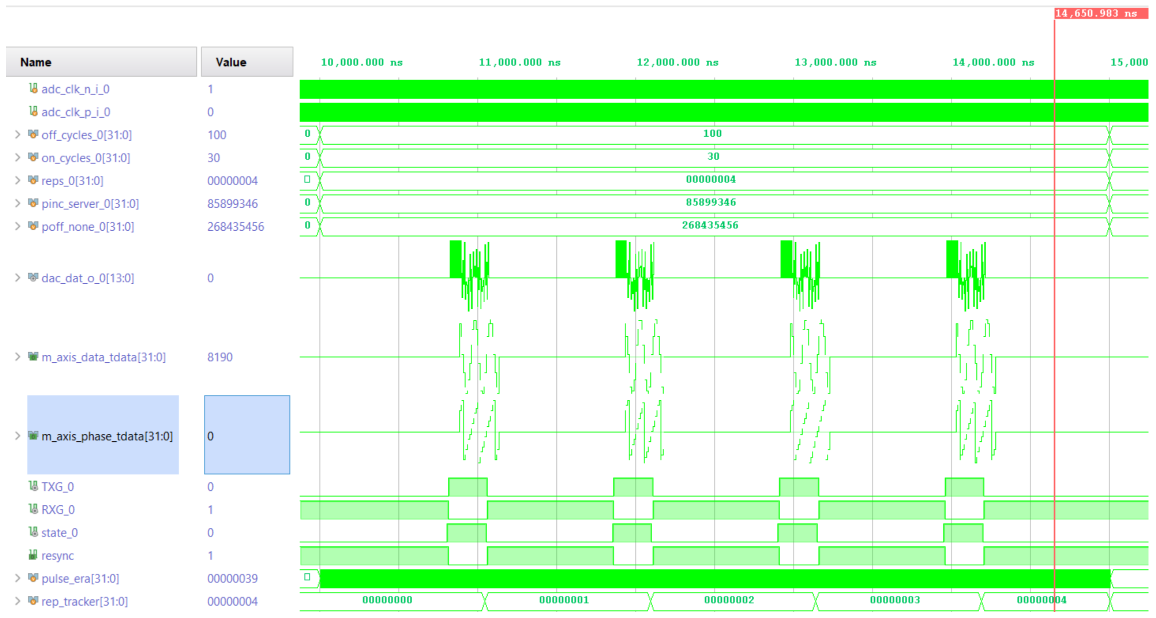
Simulation results for testbench code in base_project

Zooming in to sequences 1, 3 and 5 shows that the DAC output (dac_dat_o_0), DDS outputs (m_axis_data_tdata for amplitude, m_axis_phase_tdata for phase) and TXG and RXG are indeed being generated correctly in accordance to the set input values of off cycles (off_cycles_0), on cycles (on_cycles_0), repetitions (reps_0), phase increment (pinc_server_0) and phase offset (poff_none_0).



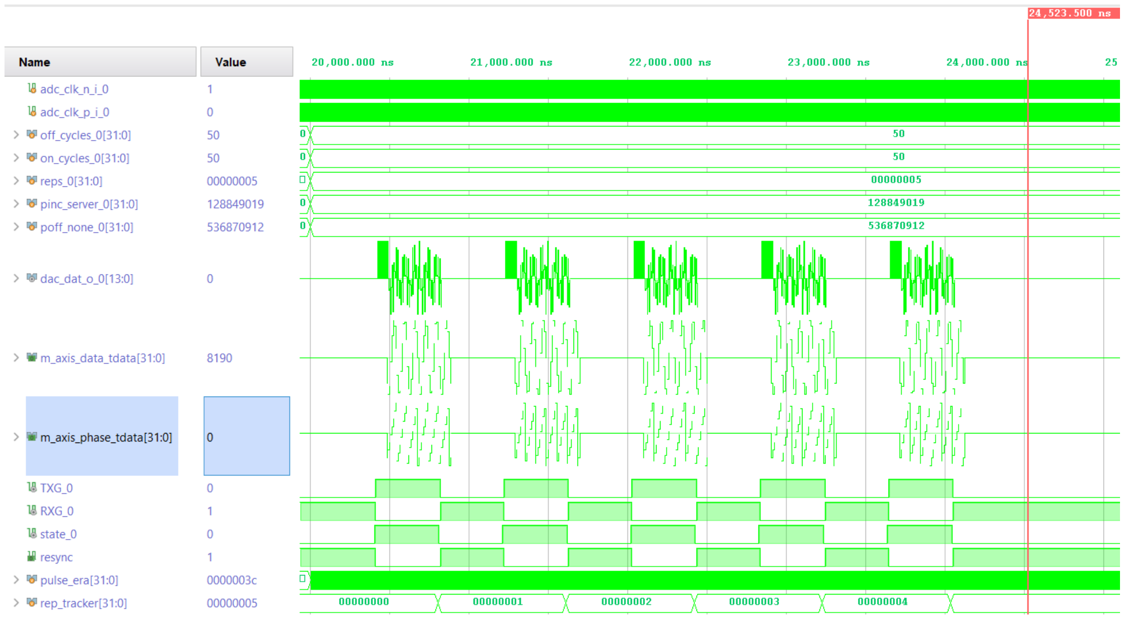
Name	Value
adc_clk_n_i_0	0
adc_clk_p_i_0	1
off_cycles_0[31:0]	20
on_cycles_0[31:0]	40
reps_0[31:0]	00000003
pinc_server_0[31:0]	25769804
poff_none_0[31:0]	0
dac_dat_o_0[13:0]	0
m_axis_data_tdata[31:0]	8190
m_axis_phase_tdata[31:0]	0
TXG_0	0
RXG_0	1
state_0	0
resync	1
pulse_era[31:0]	0000006f

Zoomed in – Simulation results for sequence #1



Name	Value
adc_clk_n_i_0	1
adc_clk_p_i_0	0
off_cycles_0[31:0]	100
on_cycles_0[31:0]	30
reps_0[31:0]	00000004
pinc_server_0[31:0]	85899346
poff_none_0[31:0]	268435456
dac_dat_o_0[13:0]	0
m_axis_data_tdata[31:0]	8190
m_axis_phase_tdata[31:0]	0
TXG_0	0
RXG_0	1
state_0	0
resync	1
pulse_era[31:0]	00000039
rep_tracker[31:0]	00000004

Zoomed in – Simulation results for sequence #3



Zoomed in – Simulation results for sequence #5

NOTE: Blankout is initiated after the sleep() function in server_program.c (used in mr_experiment_SE) and first_server_program.c (used in base_project).
The input to sleep() is the total time (in seconds) required to complete all RF/MR pulse sequences. It is computed as follows:
For mr_experiment_SE, it is equal to TR*Repetitions/1,000,000.
For base_project, it is equal to (on_time + off_time)*Repetitions/1,000,000.
A limitation in both C programs is that sleep() works correctly if the total time is at least one second long. If it is less than a second, the sleep() function may not execute and force the digital circuit to blank out its RF and GPIO outputs even before all repetitions have been completed.
Hence, due to the nature of blank out operations performed by server_program.c and first_server_program.c, it is important to enter pulse parameters such that they are at least 1 second long.