ARL-TR-9471 ● JUNE 2022

DEVCOM ARMY RESEARCH LABORATORY

Comparison of Custom Digital Synthesizers for Advanced Radar Waveforms in Radio Frequency Systems-on-Chip (RFSoCs)

by William Diehl and Edward Viveiros

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-9471 ● JUNE 2022

Comparison of Custom Digital Synthesizers for Advanced Radar Waveforms in Radio Frequency Systems-on-Chip (RFSoCs)

William Diehl and Edward Viveiros
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
June 2022	Technical Report	1 July 2020–1 December 2021

4. TITLE AND SUBTITLE
Comparison of Custom Digital Synthesizers for Advanced Radar Waveforms in Radio Frequency Systems-on-Chip (RFSoCs)

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
William Diehl and Edward Viveiros

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
DEVCOM Army Research Laboratory
ATTN: FCDD-RLS-EW
Adelphi, MD 20783-1138

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-9471

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
ORCID IDs: William Diehl, 0000-0002-6293-5018; Edward Viveiros, 0000-0002-5116-1010

14. ABSTRACT
Radio Frequency Systems-on-Chip (RFSoCs) are gaining prominence for their ability to accelerate the process of software-defined radio (SDR) development because computational power, storage, and high-speed communications are combined with integrated high-speed data converters. Since RFSoCs bring together diverse competencies such as software, hardware, and RF engineering, exploration of new digital RF paradigms is required to maximize their potential. Implementation of radars using SDRs is a growing trend and can be efficiently realized in RFSoCs. Specifically, advanced radar waveforms using linear frequency modulation (LFM) are becoming industry standard because they improve target range resolution. In this research, we design and implement two types of custom digital synthesizers of radar waveforms: a linear frequency modulation generator (LFMGEN) and a variable direct digital synthesizer (VARDDS). Both synthesizers are capable of generating up to 1-GHz instantaneous bandwidth of LFM radar waveforms and are demonstrated in the Xilinx RFSoC using external RF measurement equipment. Each synthesizer is shown to have advantages and disadvantages; in particular, the LFMGEN can arbitrarily define and start new LFM radar pulse trains from software control in only 64 ns but requires twice the power-per-bandwidth of the VARDDS.

15. SUBJECT TERMS
Electromagnetic Spectrum Sciences, software-defined radio, SDR, System-on-Chip, SoC, Direct Digital Synthesis, DDS, linear frequency modulation, LFM

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			William Diehl
Unclassified	Unclassified	Unclassified	UU	34	19b. TELEPHONE NUMBER (Include area code) (301) 394-0931

Contents

List of Figures

List of Tables

1. Introduction

The Radio Frequency System-on-Chip, or RFSoC, is an emerging paradigm in RF engineering. Specifically, it combines the flexibility of embedded processing power with a tightly coupled RF analog-to-digital converter (ADC) and a digital-to-analog converter (DAC) on a single chip. This greatly reduces the design complexity of implementing RF transceivers and generally reduces size, weight, and power requirements. RFSoCs come in many forms and with varying degrees of sophistication; some are designed for initial laboratory demonstrations and prototypes, and some come in reduced form-factor and are designed for deployable or production applications. In general, the RFSoC has greatly accelerated the market for software-defined radios (SDRs), which are RF-capable computational devices that can be reprogrammed and reconfigured at various layers of abstraction, from factory, to laboratory, to production floor, and even end-user.

Implementation of radars using SDRs, either stand-alone or as part of multifunctional RF, is a growing trend and can be efficiently realized in RFSoC. In particular, advanced radar waveforms using linear frequency modulation (LFM) or "FM chirps," are becoming an industry standard for radar pulses since their returns can be reliably processed through digital signal processing to achieve highly accurate range resolution.[1]

Waveform design, in particular digitally synthesized radar waveforms, is an active area of research. A design capable of transmitting multiple complex waveforms with high frequencies, and capable of switching between multiple waveforms with very low latencies, should take advantage of the higher speeds and processing power available by implementing waveforms entirely in the programmable logic (PL) section of the RFSoC (i.e., hardware). It is therefore beneficial to explore a wide range of architectural choices of waveform digital synthesis.

In this research, we design and implement two custom digital synthesizers: a linear frequency modulation generator (LFMGEN) and a variable direct digital synthesizer (VARDDS). Synthesizers are designed in Very High Speed Integrated Circuit Hardware Description Language (VHDL) using register transfer level (RTL) methodology and encapsulated in Xilinx Advanced Extensible Interface (AXI)-standard wrappers to facilitate their use in Vivado Intellectual Property (IP) Integrator. Each design is paired with a light hardware abstraction layer and set of software drivers to include in a Xilinx Vitis C applications project. The synthesizers are integrated into a test platform based on the Xilinx Generation (Gen) 1 RFSoC ZCU111 Evaluation Platform, where resulting RF waveforms can be generated and analyzed in a laboratory environment. Synthesizers are implemented (post place

and route results) for a variety of maximum instantaneous bandwidth (IBW) and are compared in terms of their relative advantages and disadvantages for waveform generation, load, and swap latencies, resource usage, timing constraints, and power consumption. Contributions of this work are as follows:

- First-known comparison of multiple custom digital radar waveform synthesizers on Xilinx RFSoC.
- Thorough presentation and comparison of digital synthesizers, each optimized for various criteria including performance (latency and bandwidth), resources (Block Random Access Memory [RAM], Field Programmable Gate Array [FPGA] look-up table [LUT], and power), and flexibility (LFM vs. arbitrary in-phase and quadrature [IQ] waveform generation). Each digital synthesizer is independently capable of reproducing waveforms up to 1 GHz of IBW with 16-bit precision.
- Design philosophy and strategy for developing radar waveform design prototypes or production form-factor based on Xilinx RFSoC.
- Analysis of complex RF-digital and RFSoC-specific design considerations, including sampling rate, block design timing closure, and clock management.

2. Background

2.1 RFSoC

These demonstrations are based on the Xilinx Gen 1 RFSoC, device nomenclature XZCU28DR, as hosted on the ZCU111 Evaluation Board.[2] Notable features of the board include integrated RF-DAC and RF-ADC functionality, FPGA PL quad-core ARM Cortex-A53 processing system based on the Zynq UltraScale+Multi processor system-on-chip, and DDR4 RAM. RF-DAC and RF-ADC are encapsulated inside the RF Digital Converter (RFDC).[3] The designer instances the RFDC as an IP in Vivado IP Integrator, designs signal processing applications (synthesizers, filters, transforms, etc.) around the RFDC, and optionally accesses all the signal-related IP through software drivers. In the Gen 1 RFSoC, ADCs have 12 bits of precision and DACs have 14 bits. However, all digital data entering or leaving the RFDC is mapped to 16-bit signed integer format.

The maximum analog frequency that can be represented from discrete samples depends on the sampling frequency, often depicted in Giga samples per second (GSps). According to the Shannon–Nyquist theorem, this maximum theoretical frequency is one-half of the sampling rate; thus, a DAC with a sampling rate of

2.0 GSps can generate a maximum frequency of 1.0 GHz without the onset of aliasing. In practice, the use of multiple Nyquist zones can be used to transmit aliases or replicas of images at higher frequencies at the cost, however, of required filtering and loss of signal strength.

Digital data within the PL, and encapsulating the RFDC, is optimally represented in complex form as IQ, where each sample has one 16-bit I value and one 16-bit Q value. The IQ quantities are considered 90° out of phase to one another and therefore preserve physical-world coherent relationships. The net effect on our designs is that one digital sample requires 32 bits.

The IBW represents a range across minimum and maximum frequency components that can be detected. Radars due to their historically pulsed nature typically require a higher IBW than communication signals, although modern multifunction RF and complex waveforms are rewriting this assumption. Our IBW is determined by number of samples per second that can be processed by the RFDC, which is equal to AXI streaming clock frequency times the number of samples in the AXI stream in the PL.

2.2 Radar

Radar stands for "radio detection and ranging" and consists of an RF transmitter, receiver, and signal processing. The transmitter sends out an RF pulse, which reflects off a target and travels back to the radar where it is received. A processor determines the range to the target and its displacement in angle or elevation according to a number of methods. It is desirable to get as much energy on target, which can be achieved by long pulse lengths. However, long pulses create a large distance between leading and lagging edge of the return pulse, which makes precise range resolution of the target difficult. Modern radar processing resolves this problem through modulations on pulse including LFM. In an LFM chirp, the frequency of a sinusoidal waveform linearly increases as a function of time, provided as $s(t) = sin(2\pi f(t)t)$, and where $f(t)$ increases linearly with time t. A transmitter that produces a steadily rising frequency can be produced through analog or digital means; we consider direct digital synthesis—generation of all sinusoidal components at their correct frequency—in the PL in this research.

2.3 Related Work

Direct Digital Synthesis (DDS) has long been explored as an alternative to analog devices for generation of LFM waveforms. However, previous constructions rely on partial use of analog waveform generator components, lack wideband performance, or lack reconfigurability. For example, in Li et al.,[4] authors explore

DDS and frequency multiplication to realize an L-band LFM synthesizer with 500-MHz BW. However, their construction relies on the Analog Devices AD9858 direct digital synthesizer as a waveform generator. In Pallavi et al.,[5] the authors employ a fully LUT-based digital construction using MATLAB System Generator (SysGEN) and achieve 200-MHz BW—but for only one pulse width (3 μs). Further, in Chekka and Aggala,[6] the authors implement an LUT-based DDS approach in a Zynq FPGA and Xilinx SysGEN but achieve a maximum BW of only 2 MHz. An observation of previous research is that the capability to build fully digital synthesizers in FPGA for meaningful high-bandwidth RF waveforms has existed only recently.

Design of radar waveforms on RFSoC is a new phenomenon; earlier research on this topic is scarce. An early example is Fagan et al.,[7] where the authors used a prototype RFSoC as an SDR and waveform generator along with real-time adaptive beamforming in an S-band phased array radar. Their signal processing elements were programmed in C, simulated in MATLAB, and synthesized from Vivado High Level Synthesis instead of manual RTL. Their prototype used the earliest Xilinx RFSoC prototype available and consumed 112K LUTs, 124K registers, and 776 RAMB18 (Block RAM [BRAM]) for an adaptive beamformer based on 16 DACs. However, this prototype does not document investigation of waveform generators, particularly LFM. In Ispir and Yildirim,[8] the authors implemented a real-time digital signal generator for a noise radar using a pseudo random-number generator. The generator produces 250 MSps and consumes 26K LUTs, 46K registers, and 20 BRAMs, which illustrates the resource-intensive nature of digital signal generation. Finally, in Norheim-Naess and Finden,[9] the authors compare several versions of digital passive radars including one version on the RFSoC. While specific waveforms are not explored, the authors indicate that RFSoC can provide size, weight, and power advantages where many reception channels are required.

As RFSoC is a new development, there are no previous comparisons of different methods of digital synthesis of LFM waveforms on RFSoC.

3. Custom Digital Synthesis Designs

3.1 Linear Frequency Modulation Generator

We implement a Custom VHDL IP module using RTL design methodology as a DDS for LFM radars called LFMGEN. The LFM radar pulse train can be generated with variable BW (or "frequency excursion"), pulse width (PW), and pulse repetition rate (PRI). Parameters describing the variables mentioned previously are passed to the hardware module via AXI protocol[10] from an accompanying software

driver and can be arbitrarily defined at runtime. Updates from software to hardware are asynchronous, as they can be provided at any time from software and take effect at the completion of the current PRI window.

LFMGEN implements IQ elements of the sinusoid as LUTs, inferred in synthesis as Read Only Memory (ROM), with a fixed number of samples ($Nsamples$). Our implemented LFMGEN version uses 16,384 samples but this can be decreased or increased. Individual samples from IQ tables are accessed according to an index (e.g., 0 to 16,383). LUTs are never updated at runtime; rather, the rate at which the index updates is constantly changed to reflect the pulses' increasing frequency. In Fig. 1, δ represents the change in index, which is increased from δ (a) to 2δ (b). The increase in δ is directly proportional to the increase in frequency.

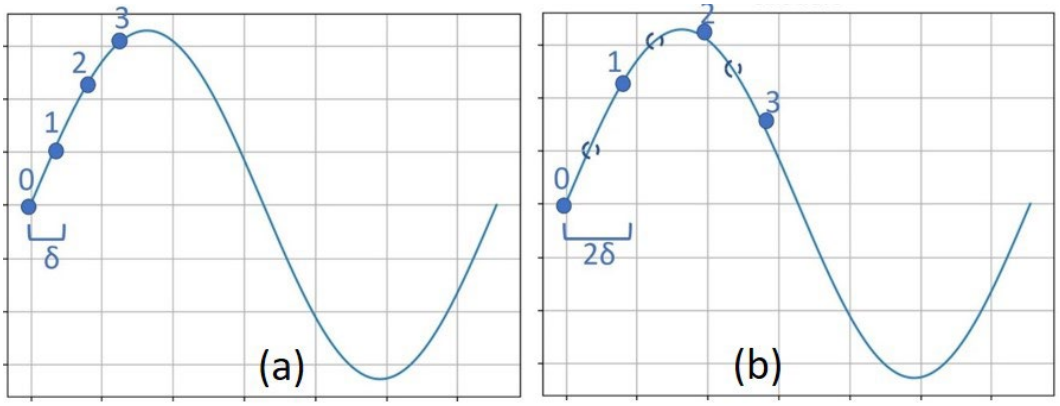

Fig. 1 Indices on sine wave increasing at increments δ (a) or 2δ (b)

The PW is defined as the length of time that the pulse is active. The PRI is defined as the length of time between the start of one pulse to the start of the next pulse. To set the parameters from LFMGEN, four arguments are passed from software to the hardware module: PW cycles (in clock cycles), wave index increment (wii), fractional increment (fi), and Delay cycles (in clock cycles). These four parameters are computed according to Algorithm 1 (Alg 1), described in the following. The method of integer and fractional updates is motivated by similar methods used to control phased-locked loops (PLLs) and universal asynchronous receiver–transmitters (UARTs).[11,12]

Algorithm 1 Compute LFM generator parameters

Inputs: $fclock\ (MHz), Nsamples, BWtarget\ (MHz), PW\ (ns), PRI\ (ns)$
Outputs: $PW\ cycles, wii, fi, Delay\ cycles$

$Start$
1. $\tau clock\ =\ 1/fclock$
2. $fmax\ =\ BWtarget$
3. $\tau chirp\ =\ 1/BWtarget$
4. $T\ =\ \tau chirp/\ \tau clock$
5. $rii\ =\ Nsamples/T$
6. $PWcycles\ =\ PW\ /\ \tau clock$
7. $rwii\ =\ PWcycles\ /\ rii$
8. $wii\ =\ \lfloor rwii \rfloor$
9. $fi\ =\ \lceil 1/(rwii - wii) \rceil$
10. $PRIcycles\ =\ PRI\ /\ \tau clock$
11. $Delaycycles\ =\ PRIcycles - Pwcycles$

End

In this research, we generate two generic LFM pulse trains. A sample pulse train is described as follows: Pulses are generated with $PW\ =\ 10$ μs, $PRI\ =\ 100$ μs, and $BW\ =\ 2.0$ MHz. In the software driver, this pulse train is rendered as 4-tuple array vectors, {2464, 18, 2, 22176}, where parameters are computed using Alg 1 and where $fclock\ =$ 246.4 MHz and $Nsamples$ = 16,384. Loading and starting a new LFM pulse train requires transferring each element of the 4-tuple vector to a memory-mapped register across an AXI-Lite interface. Assuming 4 clock cycles per write access at LFMGEN clock speed (246.4 MHz), this can take as little as 64.9 ns; the new LFM waveform is automatically started on completion of the write of the last element and takes effect at the completion of the current PRI.

The derivation of the 4-tuple vector corresponding to the first pulse train using Alg 1 is as follows: Inputs: $fclock$ = 246.4 MHz, $Nsamples$ = 16,384, and $BWtarge_t$ = 2.0 MHz. Desired pulse width of each chirp PW = 10.0 μs (= 10,000 ns), and desired PRI = 100.0 μs = 100,000 ns. We compute an AXI stream clock period $\tau clock\ =\ 1/fclock\ =$ 4.058 ns; maximum chirp frequency $fmax\ =\ BWtarget\ =$ 2.0 MHz; and period of maximum chirp frequency (assuming chirp starts at 0 MHz) $\tau chirp$: $1/BWtarget\ =$ 500 ns. Next, we compute clock cycles required to render one period at $fmax$: $\tau chirp/\tau clock\ =$ $500\ ns/4.0584\ ns\ =\ T\ =$ 123.2 cycles. Therefore, the index to a LUT-based sinewave with $Nsamples$ must complete a period every 123.2 clock cycles at $fmax$. To traverse $Nsamples\ =$ 16,384 samples at $fmax$, the LUT index must be increasing by 1 every 16,384/123.3 = 132.9 clock cycles, which is the rate of increasing index (rii). Since PW = 10,000 ns, there are $10{,}000\ ns/4.0584\ =$ $\lfloor 2464.27\ \rfloor =$ 2,464 clock cycles per period (*PW cycles*).

Starting with an index increment 0 at $t = 0$, δ must increase by 1 every $2{,}464.27/132.9 = 18.54$ clock cycles, which is the raw wave index increment ($rwii$). Since index increment cannot increase a fractional number of clock cycles, it will be increased by 1 every $\lfloor 18.54 \rfloor = 18$ clock cycles for i iterations, called the *wave index increment* (wii), where $i = \lceil 1/(18.54 - 18) \rceil = \lceil 1/0.54 \rceil = \lceil 1.85 \rceil = 2$, called *the fractional increment* (fi). On every fi[th] increment, index increment will be delayed by 1 clock cycle. For this example, the first index increment occurs after 18 clock cycles; the second index increment occurs after 19 cycles; the third increment occurs after 18 cycles; the fourth increment occurs after 19 cycles, and so on. The average effect is that index increment occurs every $(18 + 19)/2 = 18.5$ clock cycles. After 2,464 clock cycles ($= 1\ PW$), the index is incrementing 133.2 every clock cycle. Therefore, the period is $16{,}384/133.2 = 120.65 * 4.0584\ ns = 499.2$ ns. Therefore, at the conclusion of one PW, $fmax$ (actual) $= 1/499.2\ ns = 2.003$ MHz. This is tolerant within $(2.003 - 2.000)/2.000$, or 0.2% of target BW. The last step is to compute the Delay cycles. This is computed as PRI cycles = $100{,}000/4.0584 = 24{,}640\ cycles - 2{,}464$ (PW cycles) $= 22{,}176$ *Delay cycles*.

An arbitrary number of LFMGEN instances can be used in a single block design. Each LFMGEN instance outputs to the transmit stream of $32 * n$ bits, where $n = 1\ to\ 8$, and where n represents the number of IQ samples interleaved in the transmit stream according to Xilinx RFDC requirements.[3] Interrupts are asserted at the start and end of each LFM pulse. The interrupts can be used to signal the application or are provided as external oscilloscope trigger signals if properly configured to external ports, such as the PMOD ports on the Xilinx ZCU111 RFSoC.

An example of rapid switching between short ($PW = 10$ µs, $PRI = 100$ µs) and long ($PW = 20$ µs, $PRI = 190$ µs) LFMGEN pulse sequences under simulated software control (via AXI-Lite register commands), performed in Vivado Simulator and processed by a Python script, is shown in Fig. 2.

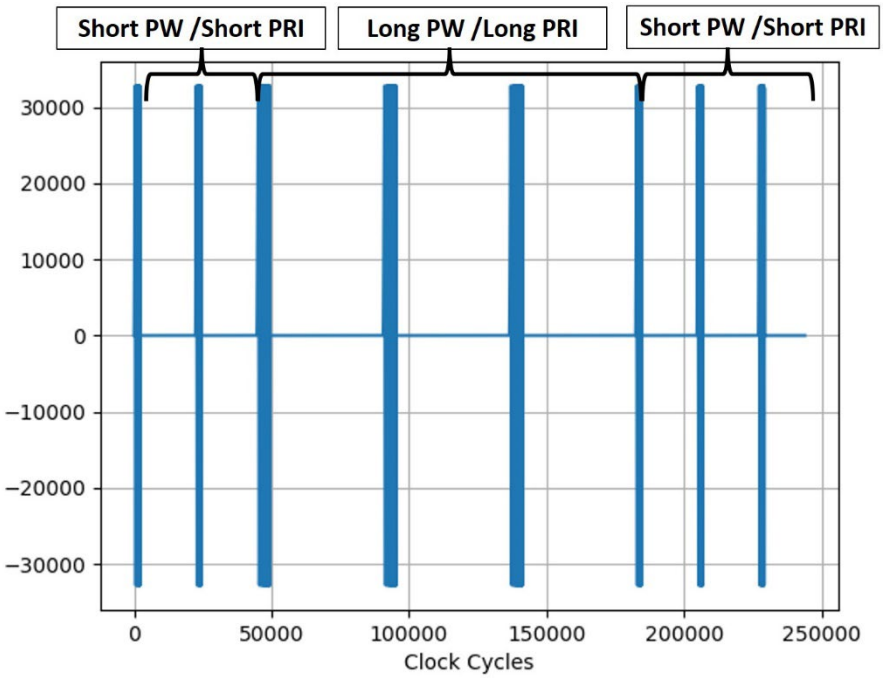

Fig. 2 Transition from LFM short PW/PRI to long PW/long PRI and back under simulated software control in Vivado simulator

3.2 Variable Direct Digital Synthesizer

The second custom digital synthesizer in this research is the VARDDS, which generates a waveform based on an arbitrary IQ input file. The VARDDS is a custom IP module written in VHDL using RTL methodology and is instantiated in the PL portion of a Xilinx block design. The VARDDS loads one or more waveforms from external memory (e.g., DDR RAM), stores them in buffers consisting of BRAM, and selects the desired buffer for transmission. An overview of the VARDDS is depicted in Fig. 3.

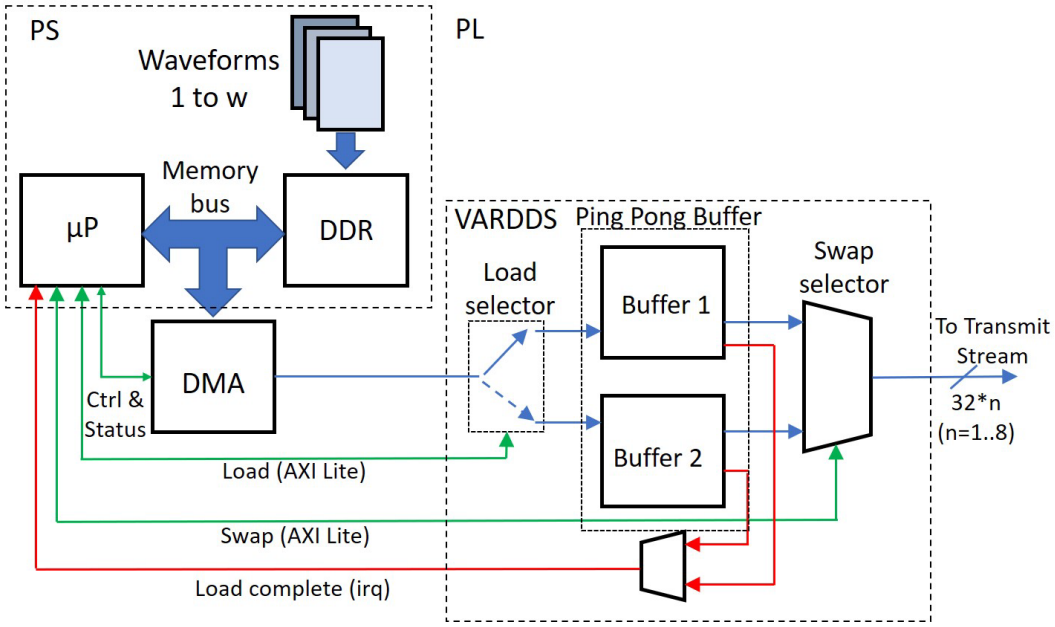

Fig. 3 Variable direct digital synthesizer

A desired IQ waveform is loaded from memory using Direct Memory Access (DMA). Only the DMA read channel is required since the VARDDS is open loop (i.e., no data returns to memory). The DMA is used to load a *ping pong buffer*, where one buffer is loaded while the other is actively transmitting a waveform. An example operational sequence consists of selecting a waveform in software to be loaded, commanding the loading of the VARDDS, then swapping buffers to make the waveform just loaded into the active signal. A second waveform can then be loaded to the offline buffer.

The VARDDS is commanded through software running on one Xilinx RFSoC ARM Core A53 processor. VARDDS software drivers written in C map to the command functions of the VARDDS, including *load*, *swap*, and *start*. The VARDDS additionally signals completion of a load event using an interrupt and an associated interrupt service routine; the interrupt must be properly initialized according to the programmer's desires.

The waveforms can consist of many different software and data file types, including .bin, .dat, .txt, or .h files appended to the software program, or can be imported via external interface. However, individual IQ samples are 32 bits and must adhere to the following format: $< Signed\ I > [31..16]\ ||\ < Signed\ Q > [15..0]$; in other words, 16-bit I and 16-bit Q per sample. Note the Xilinx RFDC automatically maps the signed 16-bit IQ to signed 14-bit IQ for its associated DACs.[3] In this application, waveforms are stored in C header files and instantiated in arrays of unsigned integer of length M, where $M = 2^k$, which represents the number of

samples in the waveform. In this research, k is fixed at 14 for 16,384 samples per waveform, though this parameter can be generically updated at synthesis time. We demonstrate the capability of VARDDS to emulate LFM waveforms by generating a pulsed waveform with 2.0-MHz BW using an offline Python script. However, VARDDS is not limited to LFM; any arbitrary IQ waveform can be loaded and output by the RFDC. We show this by generating a nonlinear frequency modulation (NLFM) on pulse with the same frequency excursion but where frequency $f(t)$ increases as $\sim t^3$ across the same pulse width.

VARDDS outputs an AXI streaming signal that is sent directly to the RFDC or through intermediate modules in the transmit stream. By default, VARDDS outputs 32 bits (1 I and 1 Q sample) per clock cycle; however, this can be increased to up to 256 bits (8 I and 8 Q samples) per clock cycle to increase BW. At a target AXI stream clock frequency of 250 MHz, a transmit bandwidth of up to 1 GHz can be achieved using the maximum of 256 streaming bits per VARDDS instance.

VARDDS latencies can be characterized by software and hardware components. Software latencies are dependent on program flow. In our test application, software commands for load, swap, and start are controlled from a nonblocking operator command line interface (CLI), for example, using the "toggle" command to swap buffers and start load on the offline buffer. The major load latency is the DMA. While DMA latencies are difficult to compute precisely, a DMA load latency for a 16,384-sample waveform is computed as 156.6 μs based on Xilinx DMA user's guides.[13] The VARDDS itself requires 66.4 μs to buffer the incoming waveform (assuming our actual AXI streaming frequency of 246.4 MHz). However, these latencies are overlapping, in that samples are buffered as soon as they are available in the DMA memory-map-to-stream (MM2S) interface. However, swap latency is very low; upon receipt of an AXI Lite command from software, the online and offline buffers swap in only one clock cycle. Assuming four clock cycles for proper receipt and acknowledgement of an AXI Lite swap command, waveforms can be swapped with as little as 16.2 ns latency.

Multiple VARDDS can be instantiated in a single design and interact with a single DMA through another Custom IP called the VARDDS Selector (VARDDSSEL), shown in Fig. 4. The VARDDSSEL is addressed through software to select a VARDDS for interaction prior to attempting to load a specific VARDDS. Once VARDDSSEL has been aligned to the required VARDDS, it enters a "locked" state and signals the processor via an interrupt.

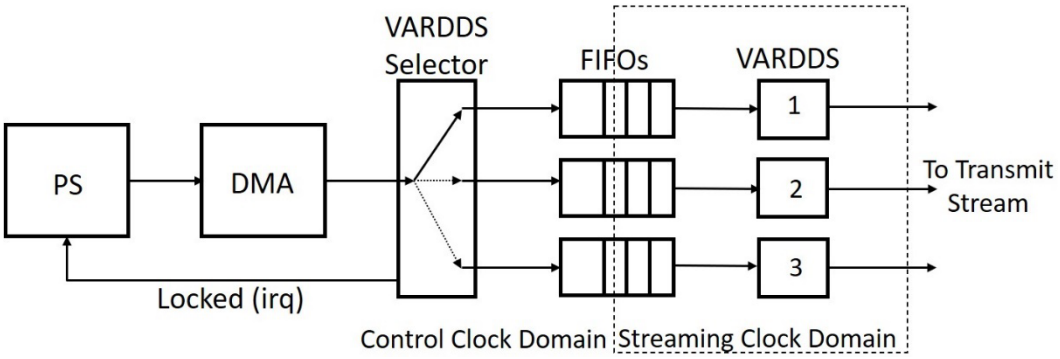

Fig. 4 Instantiation of multiple VARDDS using VARDDSSEL

3.3 Design Philosophy and Strategy

We implement and demonstrate our custom digital synthesizers on the Xilinx ZCU111 RFSoC evaluation board. The design is a bare metal instantiation (i.e., no operating system) and runs on a single ARM Cortex A53 core at a processor frequency of 1.3 GHz. It is loaded and initiated using the First Stage Boot Loader (FSBL) generated by Xilinx Vitis 2020.1 and can be run through a Joint Test Action Group (JTAG) interface connected to a PC via the Micro USB cable or automatically booted through the ZCU111's Micro SD card. In either case, a serial connection at 115,200 baud should be initiated between the ZCU111 and host PC to receive operator commands through the CLI.

We use a layered design philosophy (Fig. 5) including hardware, driver, and transmit layers. The driver and transmit layers are software and coded in ANSI-standard C; the hardware consists of Xilinx IP coded in VHDL and Verilog, and Custom IP coded in VHDL. Custom IPs in this research are constructed in manually developed RTL design.

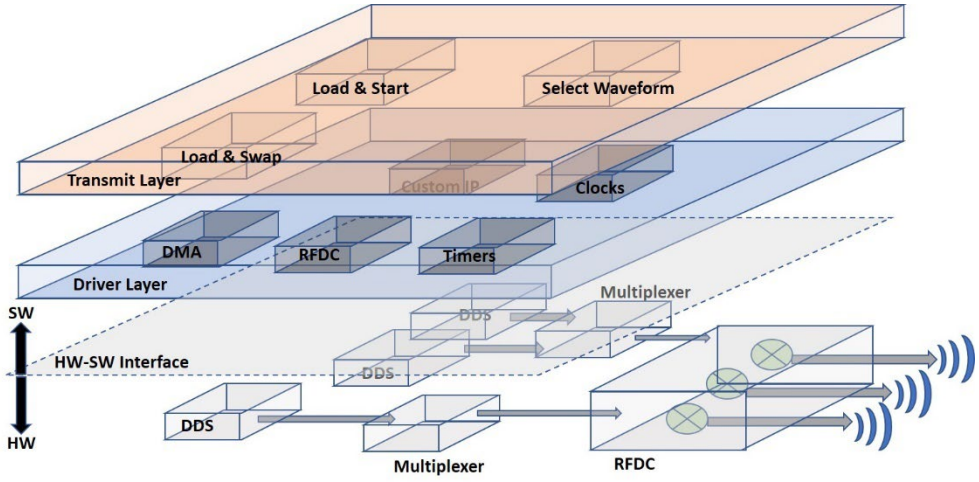

Fig. 5 Layered design philosophy

A design capable of transmitting multiple complex waveforms with high frequencies, and capable of switching between multiple waveforms with very low latencies, should take advantage of the higher speeds and processing power available by implementing waveforms entirely in the PL section of the RFSoC (i.e., hardware). We showcase this advantage by creating DDSs for all waveforms in this research, as described previously. Since multiple waveforms can theoretically use the same RFDC transmission chains, the hardware layer can include multiplexers to facilitate rapid switching.

The driver layer contains Xilinx or ZCU111-specific high-level drivers to control complex IPs in the hardware layer such as Scatter-Gather Direct Memory Access (SG-DMA), UART, RFDC, ZCU111 onboard clocks and PLLs, AXI timers, interrupt controller, and drivers for several custom IPs. These drivers are adopted primarily from publicly available Xilinx drivers.[14]

The transmit layer contains the next-higher level of C functions to load, start, swap, stop, and select waveforms for transmission. The user can manually operate all waveform functions (in manual mode) through direct manipulation of the transmit layer functions as well as run board-diagnostic functions and conduct resets.

The application (under user or automated control) is controlled through a presentation layer. The current presentation layer is a textual CLI accessed via a serial terminal application from the host PC via a UART channel (e.g., MobaXterm or TeraTerm). A GUI will be implemented in future upgrades.

Sampling rate and clock frequency planning on the Xilinx RFSoC, and in general in RF designs, is challenging. RFDC sampling frequencies (which determine maximum RF to be transmitted or received) and streaming frequency of hardware IP in PL (which determine IBW and latency), are related. Ideally, one chooses a DAC sampling frequency as close to maximum as possible (6.554 GSps on Xilinx Gen 1 RFSoC) and then sources individual DACs with as many samples per second as possible for high IBW. However, AXI streaming sources are limited to 256 bits (eight IQ samples) per DAC, and designing RF IP modules that close timing at frequencies considerably higher than 250 MHz in UltraScale+ FPGA PL fabric is difficult. Further, AXI streaming clocks must be derived from RFDC reference clocks, which are related to both sampling frequencies and external PLL reference sources (if used).

There are eight DACs in the Gen 1 RFSoC, each capable of driving an output RF signal. Quad-DACs are grouped into two tiles (Tile 228 and 229), and each tile can have only one sampling frequency. This suggests dividing tiles between “low” and “high” sampling rates. For example, the low-frequency tile can output radar waveforms close to baseband (perhaps, for diagnostic purposes) in the first Nyquist

zone, and the high-frequency tile can transmit waveforms in real-world bands (e.g., S-Band [2–4 GHz] using second Nyquist zone if needed. Further, using the XM500 balanced–unbalanced (Balun) board included with the ZCU111, DAC04 and DAC05 on Tile 229 are routed to a high-frequency Balun that attenuates baseband but minimizes losses in S-Band. So, we choose DAC00 and DAC01 on Tile 228 for low band and DAC04 and DAC05 on Tile 229 for high-band RF outputs. We choose 4.0 GSps as a target sampling rate for high band and 2.0 GSps for low band.

The ZCU111 contains one Texas Instruments (TI) LMX 2594 PLL used to source both DAC tiles. Since we require two different sampling frequencies, we must select PLLs internal to each tile and source them using a single LMX 2594 PLL reference. While LMX 2594 PLL frequencies can be customized using the TI TICS PRO application, formulas used to design LMX 2594 and RFDC PLL frequencies are not compatible (parameters for integer and fractional divisor N, voltage-controlled oscillator [VCO] frequency, etc., are heterogeneous). We use a custom Python script (shown in the Appendix) to choose sets of parameters that are close to target range and apply to both PLLs. We adopt an example solution with 3.9424 GSps (high-band sampling rate), 1.9712 GSps (low-band sampling rate), 537.6-MHz LMX 2594 PLL reference frequency, and 246.4-MHz AXI streaming frequency.

A simplification of the resulting block design, prepared using Vivado IP Integrator, is shown in Fig. 6. There are three clock domains in the PL: 246.4-MHz AXI streaming domain sourcing DAC Tile 228, 246.4-MHz AXI streaming domain sourcing DAC Tile 229 (which is not guaranteed to be phase-synchronous with Tile 228's domain), and the 100.0-MHz PL fabric domain. Independent clock Xilinx first-in-first-out modules are used to facilitate clock domain crossings.

One instance of each custom digital synthesizer is included per band: VARDDS 0 and LFMGEN 0 route to DAC00 and DAC01, respectively, and VARDDS 1 and LFMGEN 1 route to DAC04 and DAC05, respectively. Note that a minimum of 64 bits of streaming input per clock cycle is required to achieve the high-band sampling rate; therefore, two different sizes of VARDDS and LFMGEN instances are used, with sizes fixed by VHDL generics at synthesis time.

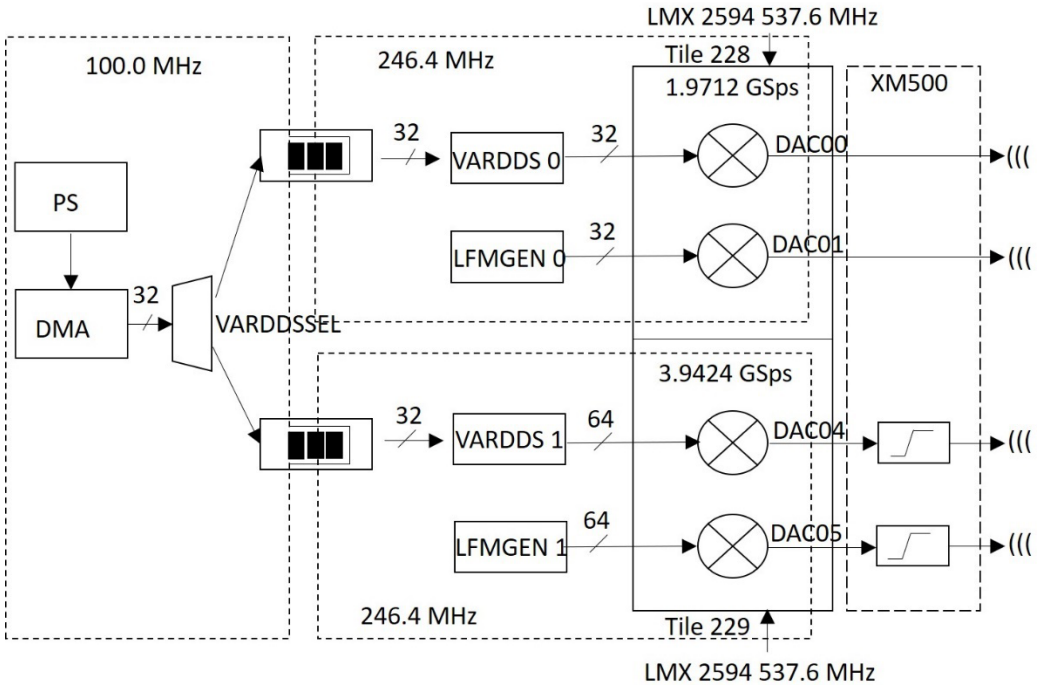

Fig. 6 Simplified block design

4. Demonstration

A Vitis application project consisting of a bare metal implementation of the design in Fig. 6 was demonstrated in a laboratory environment. RF time-domain results are shown on a Keysight DSOX6004A Digital Storage Oscilloscope (20 GSps maximum), and frequency domain results are shown on a Keysight Field Fox Microwave Analyzer N9916A. A representative sample of results are shown in the following figures.

The output of LFMGEN Waveform 1 (with 2.0 MHz of BW) from DAC01 (first Nyquist zone) is shown in Fig. 7. The output is upconverted using the RFDC oscillator in fine adjustment mode to a center frequency of 6 MHz to show the familiar two-sided LFM spectrum.[1] The output of VARDDS 1 Waveform 1 (LFM with 2.0 MHz of BW) from DAC04 (second Nyquist zone) is shown in Fig. 8. The RFDC oscillator is adjusted to 3.2 GHz to represent a typical S-Band radar.

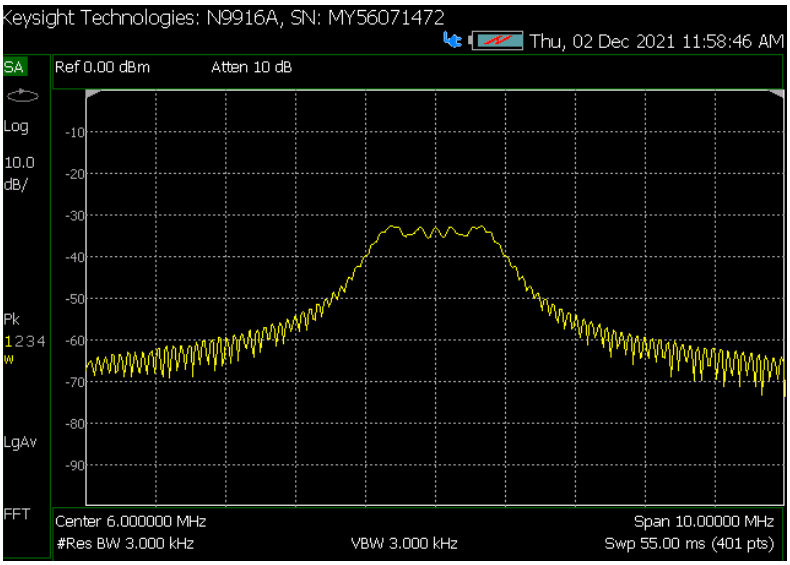

Fig. 7 LFMGEN Waveform 1 (with 2.0 MHz of BW) from DAC01

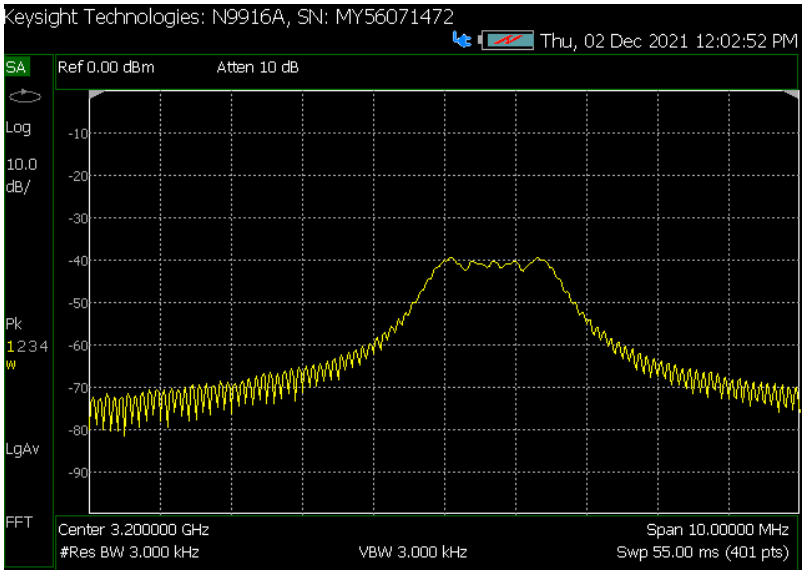

Fig. 8 VARDDS Waveform 1 (LFM with 2.0 MHz of BW) from DAC04

In Fig. 9, a baseband representation of LFMGEN (DAC01) Waveform 1 with 10-µs PW and 100-µs PRI, and 2.0 MHz of BW, is shown in the oscilloscope at 50 µs per division, while a baseband representation of LFMGEN (DAC01) with 9.2-µs PW is shown in Fig. 10 at 2 µs per division.

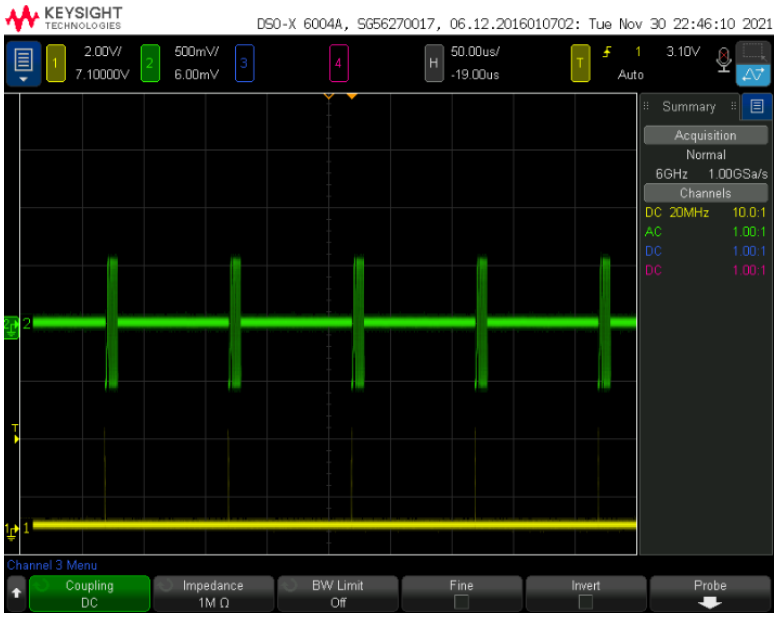

Fig. 9 LFMGEN (DAC01) Waveform 1, with 10-µs PW and 100-µs PRI, and 2.0 MHz of BW

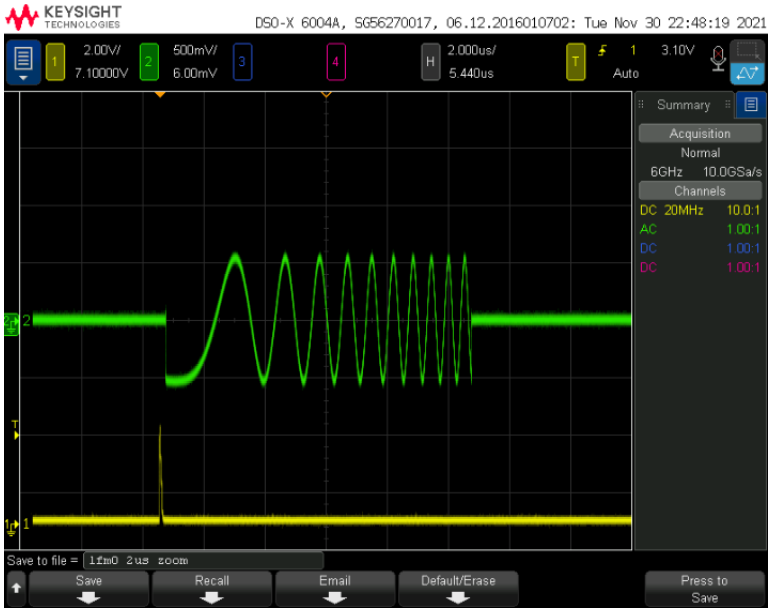

Fig. 10 LFMGEN (DAC01) with PW slightly reduced to 9.2 µs

An LFM waveform from VARDDS 0 (DAC00) at baseband is shown at 10 µs per division in Fig. 11, while the NLFM waveform from the same VARDDS instance is shown in Fig. 12.

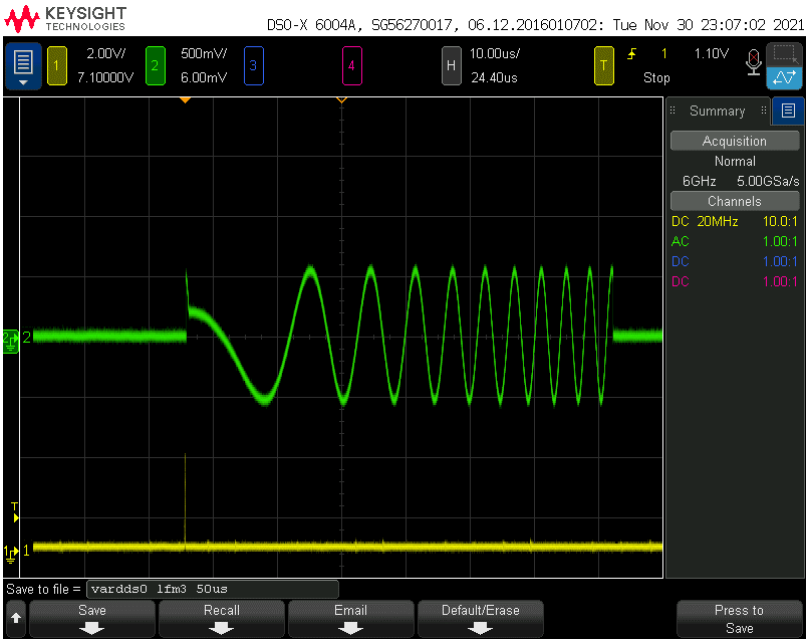

Fig. 11 LFM waveform from VARDDS 0 (DAC00)

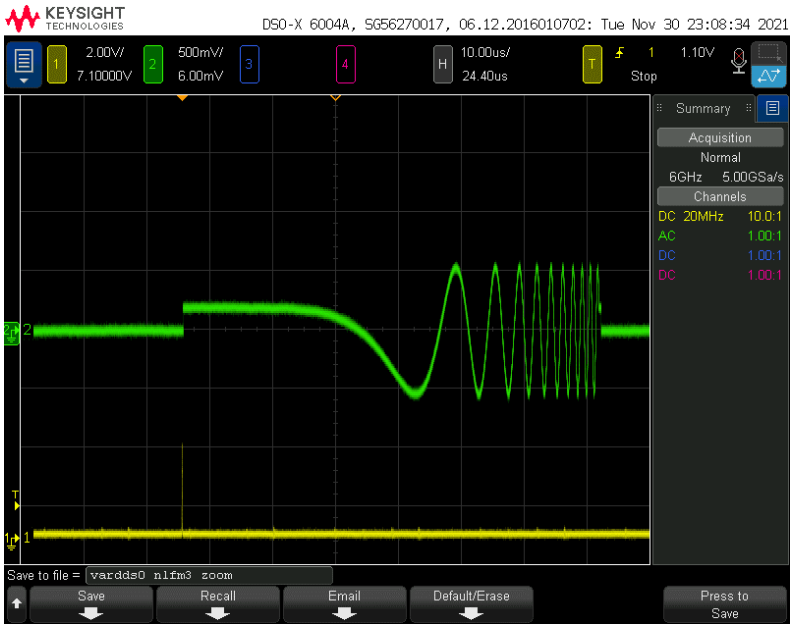

Fig. 12 NLFM waveform from the same VARDDS instance

5. Analysis

The design illustrated in Fig. 6 is implemented in the Xilinx Gen 1 RFSoC, device nomenclature XZCU28DR, as hosted on the ZCU111 Evaluation Board, using Vivado 2020.1. To facilitate comparisons between VARDDS and LFMGEN, place

and route implementations for output bus widths of 32, 64, 128, and 256 bits (corresponding to 1, 2, 4, or 8 IQ samples per clock cycle, respectively) are completed. All use a fixed clock frequency of 250 MHz; worst negative slack (WNS) for each implementation is recorded. All implementations are encapsulated in a "wrapper" where the number of external pins and input/output buffers is identical for all implementations. Results are shown in Table 1.

Table 1 Summary of implementation statistics for VARDDS and LFMGEN in XZCU28DR RFSoC at 250-MHz clock frequency

IQ samples	**1**	**2**	**4**	**8**
Bus width	**32**	**64**	**128**	**256**
VARDDS				
Worst negative slack (ns)	1.183	0.889	0.5	0.358
LUT	154	233	394	746
RAMB36 (BRAM)	28	56	112	224
Static power (W)	1.085	1.086	1.087	1.09
Dynamic power (W)	0.066	0.127	0.225	0.421
Total power (W)	1.151	1.213	1.312	1.511
Bandwidth (BW) (MHz)	125	250	500	1000
Power/BW(μW/MHz)	528	508	450	421
LFMGEN				
Worst negative slack (ns)	0.364	0.317	0.31	0.09
LUT	5,820	7,171	22,570	46,164
RAMB36 (BRAM)	0	0	0	0
Static power (W)	1.086	1.086	1.089	1.095
Dynamic power (W)	0.119	0.171	0.498	1.135
Total power (W)	1.205	1.257	1.587	2.23
Bandwidth (BW) (MHz)	125	250	500	1000
Power/BW(μW/MHz)	952	684	996	1135

Results are summarized in Fig. 13. In Fig. 13a, WNS decreases exponentially for VARDDS and linearly for LFMGEN with increasing size, and LFMGEN comes closer to running out of timing slack at largest size (this could possibly be improved with additional pipelining at the cost of latency and greater size).

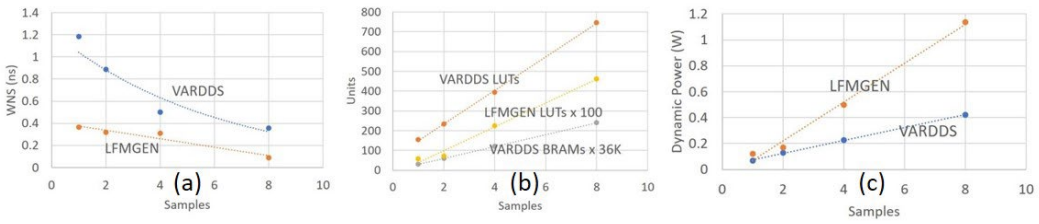

Fig. 13 Comparisons of VARDDS and LFMGEN

In Fig. 13b, the fundamental difference in architectural approaches between VARDDS and LFMGEN is that VARDDS is BRAM-centric while LFMGEN is LUT-centric. Both scale linearly with increased size: VARDDS uses 224 36K BRAMs at 256 bits (i.e., 7.8 MB of PL-based RAM), while LFMGEN uses no RAM but 46,164 LUTs ROM at largest form-factor. The BRAM usage of the VARDDS is a more significant toll against total PL resources than the LFMGEN: 224 RAMB36 (equivalents) is 22% of total RAMB36s on the XZCU28DR, while 46,164 LUTs is only 10.8% of total chip LUTs. In either case, generating a full 1-GHz IBW (e.g., 250-MHz clock frequency ∗ eight samples per clock, at Nyquist rate) uses a significant amount of FPGA resources.

Static power consumption of both modules is nearly identical; it is not considered here. In Fig. 13c, dynamic power of each synthesizer increases linearly with size. However, we note that LFMGEN, with its LUT-based approach, consumes an average of 942 μW/MHz (dynamic power per bandwidth), while the VARDDS, with its BRAM-based approach, uses an average of 477 μW/MHz. Therefore, LFMGEN requires 1.97 as much power per bandwidth as VARDDS.

If one specifically requires a large possible number of LFM pulse trains, not known at synthesis time, and where performance (e.g., low latency) is a factor, the LFMGEN is a better choice. However, if the user is content to wait hundreds of microseconds to load a new waveform, needs to represent arbitrary waveforms beyond LFM, or is more concerned about power, then VARDDS might be more suitable. A mixture of both capabilities probably provides the most flexibility for SDR-based variable radar implementations.

6. Conclusions

We designed, implemented, and demonstrated two types of custom digital synthesizers for advanced radar waveforms, an LFMGEN and a VARDDS. Both synthesizers are capable of generating up to 1-GHz IBW of LFM radar IQ waveforms with 16-bit precision. Demonstrations on the Xilinx Gen 1 RFSoC using RF laboratory equipment show time- and frequency-domain representations of several baseband and S-band LFM pulse-train presentations. Each synthesizer is shown to have advantages and disadvantages. Specifically, the LFMGEN can arbitrarily define and start new LFM radar pulse trains—parameterized for pulse width, pulse repetition interval, and bandwidth—from software control in only 64 ns but requires twice the power per bandwidth of the VARDDS. On the other hand, the VARDDS DMA-based memory-to-hardware waveform load time is much longer, but VARDDS can switch preloaded waveforms in only 16 ns and is

more versatile since it can transmit any IQ waveform commensurate with the allotted bandwidth.

Complex radar implementation on RFSoC is an immature research area; further exploration in many directions is possible. The four-parameter LFMGEN interface was shown through simulation and RF measurements to provide very accurate PW and PRI (within one-half clock period or ~2-ns accuracy) and highly accurate BW (<0.2% error in the evaluated example). However, this is ultimately a digital approximation of an analog process; further regression testing over a large number of test cases should be performed to quantify LFMGEN's performance. Additionally, no attempt was made to suppress sidelobes or compare LFM to NLFM sidelobe performance[1]; this is an area of future research. Other future topics could include reducing the power per bandwidth of the LFMGEN exploration on the upper limits of clock frequency for digital streaming IP as well as optimal allocation of frequency modulation techniques between software and hardware.

7. References

1. Blunt SD, Mokole EL. Overview of radar waveform diversity. IEEE Aerospace and Electronic Systems Magazine. 2016 Nov;31(11):2–42. doi: 10.1109/MAES.2016.160071.

2. Xilinx. ZCU111 evaluation board user guide UG1271 (v1.2); 2018 Oct 2.

3. Xilinx. Zynq UltraScale+ RFSoC RF data converter v2.3 LogiCORE IP product guide, Vivado design suite, PG269 (v2.3); 2020 June 3.

4. Li Q, Yang D, Mu XH, Huo QL. Design of the L-band wideband LFM signal generator based on DDS and frequency multiplication. Proceedings of the International Conference on Microwave and Millimeter Wave Technology (ICMMT); 2012. p. 1–4. doi: 10.1109/ICMMT.2012.6230321.

5. Pallavi N, Anjaneyulu P, Reddy PB, Mahendra V, Karthik R. Design and implementation of linear frequency modulated waveform using DDS and FPGA. Proceedings of the International Conference of Electronics, Communication and Aerospace Technology; 2017, p. 237–241. doi: 0.1109/ICECA.2017.8212806.

6. Chekka AB, Aggala NJ. High frequency chirp signal generator using multi DDS approach on FPGA. Proceedings of the International Conference on Trends in Electronics and Informatics; 2021. p. 137–142. doi: 10.1109 /ICOEI51242.2021.9453012.

7. Fagan R, Robey FC, Miller L. Phased array radar cost reduction through the use of commercial RF systems on a chip. Proceedings of the IEEE Radar Conference (RadarConf18); 2018, p. 0935–0939. doi: 10.1109/RADAR .2018.8378686.

8. Ispir M, Yildirim A. Real-time signal generator for noise radar. IEEE Aerospace and Electronic Systems Magazine. 2020 Sep;35(9):42–49. doi: 10.1109/MAES.2020.2997415.

9. Norheim-Naess I, Finden E. DVB-T passive radar experimental comparisons of a custom made passive radar receiver, RFSoC and a Software Defined Radio. 21st International Radar Symposium (IRS), 2021. p. 1-10. doi: 10.23919/IRS51887.2021.9466177.

10. Xilinx. AXI reference guide UG761 (v13.1); 2011 Mar 7.

11. Texas Instruments. LMX2594 15-GHz wideband PLLatinum RF synthesizer with phase synchronization and JESD204B support (SNAS696); 2017 Mar.

12. Texas Instruments. MSP432P4xx SimpleLink microcontrollers technical reference manual, SLAU356H; 2015 Mar; revised 2017 Dec.

13. Xilinx. LogiCORE IP AXI DMA v7.1 2 (PG02)1; 2019 June 24.

14. Xilinx. Drivers [accessed 2021 Oct 6]. Embeddedsw/XilinxProcessorIPLib /drivers at master Xilinx/embeddedsw.

Appendix. Clock Coordinator Python Script

Python script to choose sets of parameters close to target sampling frequency range and which apply to TI LMX 2594 and Xilinx RFDC internal PLLs.

```
# clocks.py
# LMX 2594 and RFDC PLL coordination tool
# Outputs all of the possible PLL parameters simultaneously meeting TI
LMX 2594 and Xilinx RFDC PLL constraints
# Desired combinations are printed out, and designer can manually choose
settings to fit RFDC design constraints
# Selected combinations should be manually verified in Xilinx Vivado RFDC
IP Customization Wizard, and TI TICS Pro design GUI for LMX 2594 PLL
# Assumptions
# Assumes only Integer N-counter for LMX 2594 PLL (Integer Mode)
fpd = 245.76  # assume LMK 04208 input 122.8 MHz doubled to 245.76 MHz
fblow = 13 # FbDIV range in RFDC PLL
fbhigh = 160 # FbDIV range in RFDC PLL
Nlow = 28 # N divider range for TI PLL
Nhigh = 50 # N divider range for TI PLL (Note: A lower N divider can
reduce phase noise)
rfdcpllosclow = 8500.0 # RFDC PLL Oscillator Parameter
rfdcplloschigh = 13200.0 # RFDC PLL Oscillator Parameter
tipllosclow = 7500.0 # TI PLL Oscillator Parameter
tiplloschigh = 15000.0 # TI PLL Oscillator Parameter
rhigh = 4 # RFDC Pre Divider Parameter (Note: Setting R to max of 1 can
improve phase noise)
# Desired sampling frequency target range (in MHz)
fsmax = 4000.0
fs = 3899.75
fsmin = 3900.0
#LMX 2594 reference clock input range (MHz)
# Note: Higher Input reference frequency can improve phase noise
tirefmin = 400.0
tirefmax = 600.0
tiref = 0.0
# PLL Integer Divider and Output Divider parameters
D = [2.0,4.0,6.0,8.0,12.0,16.0,24.0,32.0,48.0,64.0,72.0,96.0,128.0,192.0,
256.0,384.0,512.0,768.0]
M = [2.0, 3.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0, 22.0,
24.0, 26.0, 28.0, 30.0, 32.0, 34.0, 36.0, 38.0, 40.0, 42.0, 44.0, 46.0,
48.0, 50.0, 52.0, 54.0, 56.0,58.0, 60.0, 62.0, 64.0]
# Returns 1 if selected oscillator frequency is within target VCO range,
otherwise returns 0
def plloscvalid(fpdin, Nin, plllow, pllhigh):
    oscfreq = fpdin * Nin
    if ((oscfreq >= plllow) and (oscfreq <= pllhigh)):
        return 1
    else:
        return 0

# Returns 1 if target parameters represent a valid coordinated PLL
solution; otherwise returns 0
def validcalc(N, M, tiref, fs):
    valid = 1
    if (plloscvalid(fpd, N, tipllosclow, tiplloschigh) == 0):
        valid = 0
    if (plloscvalid(fs, M, rfdcpllosclow, rfdcplloschigh) == 0):
        valid = 0
    if ((fs<fsmin) or (fs>fsmax)):
        valid = 0
    if ((tiref<tirefmin) or (tiref>tirefmax)):
        valid = 0
    return valid
```

```
# Iterates through all possible combinations and displays only valid
combinations
for a in range(0, len(D)):
    for b in range (Nlow, Nhigh+1):
        for c in range (fblow, fbhigh+1):
            for d in range (1, rhigh+1):
                for e in range (0, len(M)):
                    tiref = fpd * b/D[a]
                    fs = (tiref/d)*(c/M[e])
                    valid = validcalc(b, M[e], tiref, fs)
                    if (valid == 1):
                        pstr = "tiref=" + str(tiref) +  " fs=" + str(fs)
+ " N=" + str(b) + " D=" + str(D[a]) + " R=" + str(d) + " FbDiv=" +
str(c) + " M=" + str(M[e])
                        print(pstr)
```

List of Symbols, Abbreviations, and Acronyms

ADC	analog-to-digital
Alg 1	Algorithm 1
ANSI	American National Standards Institute
ARM	Advanced Reduced Instruction Set Computer Machine
AXI	Advanced Extensible Interface
Balun	balanced–unbalanced
BRAM	Block Random Access Memory
BW	bandwidth
CLI	command line interface
DAC	digital-to-analog
DDS	Direct Digital Synthesis/Synthesizer
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
FSBL	First Stage Boot Loader
Gen	Generation
GSps	Giga samples per second
GUI	graphical user interface
IBW	instantaneous bandwidth
IP	Intellectual Property
IQ	in-phase and quadrature
JTAG	Joint Test Action Group
LFM	linear frequency modulation
LFMGEN	linear frequency modulation generator
LUT	look-up table
MHz	megahertz
MM2S	memory map to stream

NLFM	nonlinear frequency modulation
PC	personal computer
PL	programmable logic
PLL	phased-locked loop
PRI	pulse repetition interval
PW	pulse width
RAM	Random Access Memory
RF	radio frequency
RF-ADC	radio frequency analog-to-digital
RF-DAC	radio frequency digital-to-analog
RFDC	Radio Frequency Digital Converter
RFSoC	Radio Frequency System-on-Chip
RTL	register transfer level
SDR	software-defined radio
SG-DMA	Scatter-Gather Direct Memory Access
SysGEN	System Generator
TI	Texas Instruments
UART	universal asynchronous receiver–transmitter
USB	Universal Serial Bus
VARDDS	variable direct digital synthesizer
VARDDSSEL	VARDDS Selector
VCO	voltage-controlled oscillator
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WNS	worst negative slack

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLD DCI TECH LIB
2 (PDF)	DEVCOM ARL FCDD RLS EW W DIEHL E VIVEIROS