2021-04-07

FPGA-Accelerated Digital Signal Processing for UAV Traffic Control Radar

Kacen Paul Moody
Brigham Young University

FPGA-Accelerated Digital Signal Processing

for UAV Traffic Control Radar

Kacen Paul Moody

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

ABSTRACT

FPGA-Accelerated Digital Signal Processing
for UAV Traffic Control Radar

Kacen Paul Moody
Department of Electrical and Computer Engineering, BYU
Master of Science

As an extension of previous work done by Luke Newmeyer in his master's thesis [1], this report presents an improved signal processing chain for efficient, real-time processing of radar data for small-scale UAV traffic control systems.

The HDL design described is for a 16-channel, 2-dimensional phased array feed processing chain and includes mean subtraction, windowing, FIR filtering, decimation, spectral estimation via FFT, cross-correlation, and averaging, as well as a significant amount of control and configuration logic. The design runs near the the max allowable memory bus frequency at 300MHz, and using AXI DMA engines can achieve throughput of 38.3 Gb/s ($\approx$ 0.25% below theoretical 38.4 Gb/s), transferring 2MB of correlation data in about 440µs. This allows for a pulse repetition frequency of nearly 2kHz, in contrast to 454Hz from the previous design.

The design targets the Avnet UltraZed-EV MPSoC board, which boots custom PetaLinux images. API code and post-processing algorithms run in this environment to interface with the FPGA control registers and further process frames of data. Primary configuration options include variable sample rate, window coefficients, FIR filter coefficients, chirp length, pulse repetition interval, decimation factor, number of averaged frames, error monitoring, three DMA sampling points, and DMA ring buffer transfers.

The result is a dynamic, high-speed, small-scale design which can process 16 parallel channels of data in real time for 3-dimensional detection of local UAV traffic at a range of 1000 m.

Keywords: local air traffic information systems, efficient computing, FPGA, digital signal processing, hardware acceleration, Xilinx MPSoC, unmanned air vehicles, heterogeneous computing, phased array radar

ACKNOWLEDGMENTS

I owe many thanks to my parents, Paul and Sharla Moody, for their continued encouragement and support, from the first model rockets in the garage through the completion of this advanced degree. Their support, both active and otherwise, has given me the drive to see it through.

I also thank David Buck for his fabulous mentoring during my years on his research team at BYU. It was David who started me in the right direction when I began, who has expanded my understanding of advanced material in tangible ways, and who established a positive work environment which has cultivated unity and motivation for everyone on the team. From an academic and career perspective, I owe David the best of all my work.

Finally, I thank a supportive and loving Father in Heaven who has facilitated every good thing in my life and who sent His Son, Jesus Christ, that my growth and learning might be possible. To These I owe more than I can repay, and I commend to Them this thesis and all of the work to follow it.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Unmanned aerial vehicles (UAVs) have been a growing topic of research in recent years because of the unique benefits they provide over larger manned aircraft [2], such as lower cost and power, and increased safety for operators. A growing challenge, however, is that as more small UAV devices enter into operation, air traffic control systems must detect a larger number of targets and effectively prevent collisions. To approach this problem, companies like Echodyne have developed small-scale metamaterial radar systems for autonomous devices [3], and researchers have explored radar for high-density air traffic control where malicious UAV devices need to be detected and avoided (DAA) [4].[1] Non-radar based systems have also been explored using passive machine vision to manage UAV air traffic [7].

In all of these systems one of the more difficult questions is where and how the signal processing is to be done. To be effective, traffic management must be performed in real time and produce sufficient range resolution and SNR to be accurate and consistent. These latter factors require relatively large data packets [8], necessitating efficient, high-speed computation. For large aircraft with stationary ground control systems, the size of the computation unit can often be fairly large, making use of multiple integrated GPUs and CPUs, but this is becoming increasingly infeasible for small UAVs in remote locations where size and power consumption quickly become expensive and unrealistic. With these unique limitations, research in FPGA DSP chains has grown dramatically as a computation alternative to large-scale processing, offering a smaller form factor with lower power consumption than their CPU and GPU counterparts [9] [10].

Over the past 35 years or so, FPGA technology has grown in capacity and speed according to Moore's law, while cost and power usage have declined [11]. This progress has led to FPGAs' introduction into a variety of markets where high IO count, high throughput, and configurable control logic are needed. In spite of the fact that configurable systems like FPGAs, which typically

[1]DAA algorithms—also known as Sense and Avoid (SAA) algorithms—have been a growing field of research as UAV traffic safety has become essential to UAV systems [5], [6].

run in the 100-500MHz range, fall far short of the clock speeds of modern CPUs and GPUs, they can often make up for that shortcoming through parallel operations, which can quickly scale to exceed the amount of processing power that a serial-instruction device can accomplish. This is then further compounded by the higher IO count available in FPGAs which improves off-chip data throughput.

Another key benefit of FPGA fabric is its configurability, which provides an advantage over GPU processing as atypical computation configurations and sizes can be customized to perform efficiently in the fabric. Additionally, because the designer can engineer a limitless variety of digital circuits according to their needs, they can dedicate logic to custom processes which occur at impressively predictable time intervals (in contrast to software threads scheduled on a CPU or GPU). This regularity is essential in applications such as Doppler processing of radar data which depends on a regular pulse repetition interval (PRI) to produce a spectral estimate with low phase noise [8].

Thus, DSP processing applications using FPGA fabric have become more commonplace as processing needs have grown. Examples of this include automotive driver assistance systems [12], human sensing [13], and UAV radar detection [14]. With these new computation-heavy applications arising, many FPGAs are now manufactured with dedicated DSP resources which offer low latency computation at a higher potential maximum frequency to improve computational performance, such as the those produced by Xilinx [15]. Newer FPGAs also have larger fabric memories to improve buffering and interim storage during processing. Furthermore, taking the isolated FPGA fabric further, many recent chips have been design as SoC or MPSoC heterogeneous devices which couple multi-processor units with FPGA fabric on the same die, such as the Zynq UltraScale+ MPSoC [16]. Thus, utilizing the flexibility of FPGAs and the operability of CPUs, many parallel channels of computation can be performed synchronously in fabric and then sent to CPU memory directly via fabric DMA engines. This pairing provides a convenient way for fast preprocessing to be done in fabric and then dynamic postprocessing to be done in software.

1.1 Objective and Contributions

This thesis seeks to expound on the work of Luke Newmeyer whose thesis presented an FPGA hardware design for the DSP chain of a UAV radar detection system [1]. Newmeyer's design

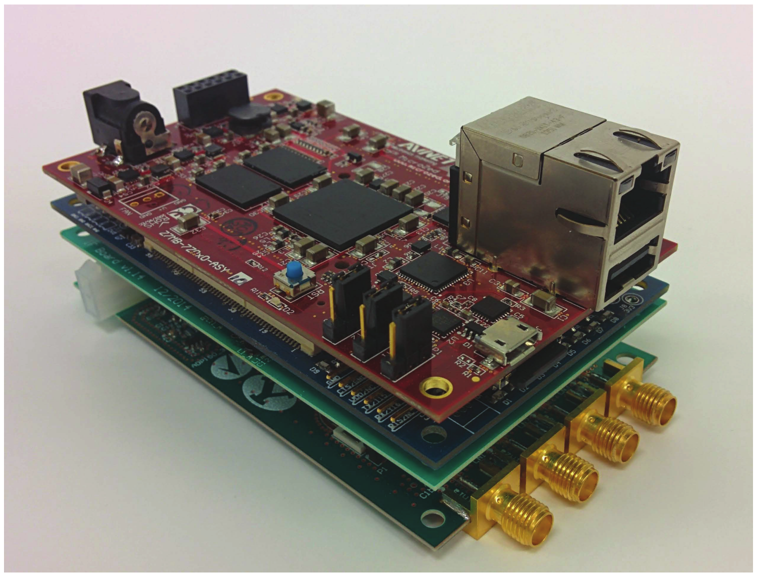


Figure 1.1: First generation integrated system with FPGA DSP architecture by Luke Newmeyer

offered a 4-channel processing chain for 1-dimensional radar, acting as a proof of concept for small-scale, low-power UAV traffic management. The final integrated system using Newmeyer's firmware is shown in Figure 1.1. Our task now is to use this design as a foundation and design a second generation system which can perform target estimation on a 2-dimensional radar feed using 16 parallel input channels, while maintaining a small form factor and reduced power usage. This project is intended to contribute to the research that has been mentioned previously as a solution to local air traffic and information systems (LATIS) for UAV devices. The project as a whole can be broken down into three primary parts: radar transmission and reception hardware, sampling and preprocessing, and postprocessing for target detection and avoidance. Specifically, this thesis addresses the second part—the sampling and preprocessing subsystem—and includes the HDL architecture for a digital signal processing chain built in FPGA fabric, as well as information about the PetaLinux kernel and software API which are used to interface with the DSP chain's configuration features. The integrated system which contains all of this is shown in Figure 1.2. Note that all data shown in Chapter 2 was captured using implemented bitstreams in the FPGA fabric of this system.

Many of the individual blocks in this chain will be tried-and-true modules which are either found in Xilinx' IP catalog or are made custom using traditional algorithms. These include mean subtraction, windowing, FIR filtering, decimation, spectral estimation with an FFT, cross-channel

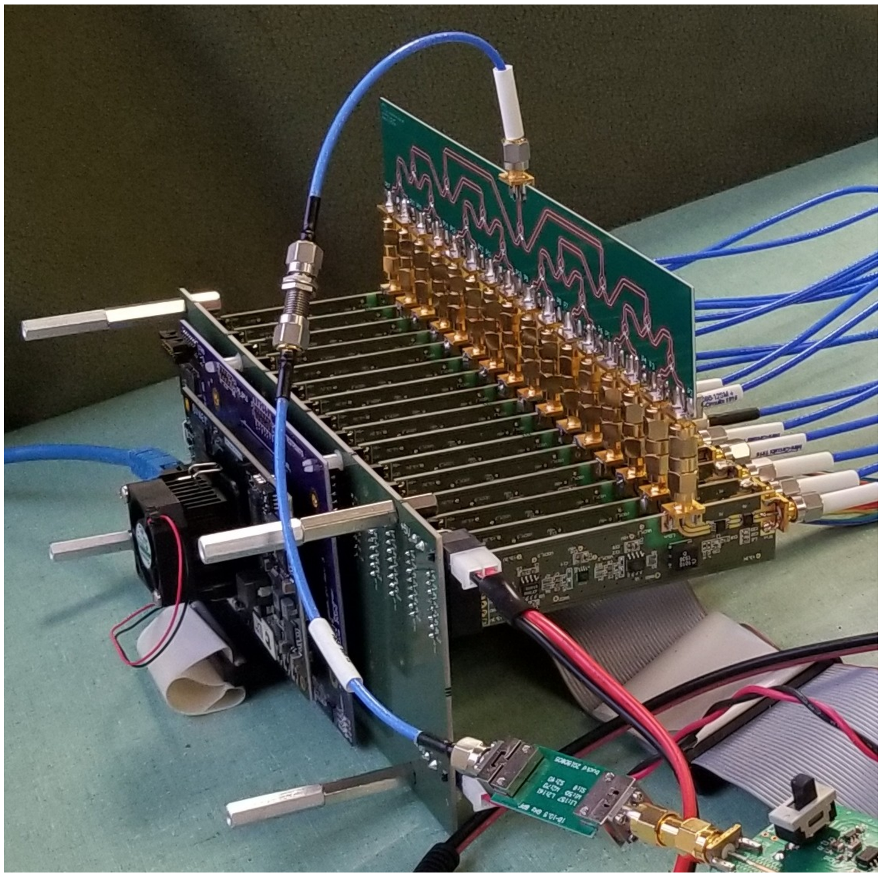


Figure 1.2: Second generation integrated system using the DSP architecture presented in this thesis

correlation, and averaging of correlation frames. Additional control logic will also be developed to provide maximum configurability of various DSP options as discussed in Chapter 2.

In conjunction with fabric processing, user-space code running in a PetaLinux kernel environment will provide a variety of API functions to configure and interact with a register-space controller (called the LATIS radar firmware, or LRF, Controller) in the fabric. From this API, post-processing algorithms will be able to access the results produced by the fabric to compute target estimation.

It is worth noting that while the fabric DSP chain is its main focus, this thesis briefly introduces the PetaLinux image configuration process and user API functions to provide context and explain how the fabric is meant to be integrated with everything else.

Upon its completion, this thesis makes the following contributions:

1. A complete 16-channel HDL DSP chain preprocessing architecture—which is small, inexpensive, and power efficient—to make local air traffic management feasible for small-scale UAV operations

2. Variable-frequency data sampling combined with DSP control options (optional windowing and mean subtraction, variable averaging frames and decimation, run-time configurable windowing and FIR coefficients, etc.), to provide good SNR and balance precise range resolution with maximum target range of airborne UAV devices

3. Advanced fabric timing analysis—ranging from register pipelining to custom Vivado synthesis and implementation strategies—for high-utilization, high-congestion HDL DSP designs, to make possible a fast, low-latency DSP chain with sufficient throughput to process 16 parallel channels of radar data with full correlation and averaging in real time at nearly 2kHz frame rate

4. Predictable chirp and data-frame timing using fabric counters (including fabric-controlled DMA transfers) to enable accurate, real-time Doppler processing of airborne UAVs

1.2 Overview

The following chapters cover the topics that we have described here in the context of their contribution to UAV LATIS research. The first of these chapters (Chapter 2) discusses each IP developed for the design and gives an overview of its configuration options. The second chapter (Chapter 3) provides an in-depth analysis of the measures taken to meet timing at 300MHz given the design's high utilization. This topic is given its own chapter because of the amount of information collected on the subject and because it makes the design possible with all of the desired features. The last two chapters (Chapters 4 and 5) give a brief overview of the process for configuring the PetaLinux image and how the software API was developed to interface with logic in the fabric.

The appendices provide additional information and associated documents for specific topics. Appendix A provides the XDC constraints file developed for the fabric's implemented design in Vivado, Appendix B contains information about every signal available to the user from the LRF

Controller fabric registers, and Appendix C contains the project's PetaLinux device tree and a PetaLinux tutorial created in conjunction with debugging done over the course of the project.

CHAPTER 2. DIGITAL PREPROCESSING OF UAV RADAR DATA IN FPGA FABRIC

2.1 Introduction

As we said in the introduction, this thesis describes only part of the larger UAV traffic management system—specifically, the digital preprocessing performed on fast-time radar data. Earlier in the system, a radar chirp is transmitted and its echos are detected by each antenna of a 16-antenna array. These signals are mixed with the original transmission signal, and from the mixing products, filtering passes the frequencies which correspond to the difference between original and echoed chirps. These frequencies, which usually reside in the kHz and MHz range, correspond to UAV target range in a linear fashion (assuming a linear chirp), and therefore contain the information that we want to process [8]. Having been mixed down to a manageable range, these frequencies can be directly sampled by ADCs and processed immediately in the FPGA.

Range computation from these frequencies is fairly straightforward as it relies firmly on spectral estimation magnitude, but with a system containing multiple input antennas, angle of arrival can be estimated with respect to zenith and azimuth, forming a hemispherical view whose base plane is the plane on which the antennas lie. This computation depends on cross-channel spectral correlation, from which we can estimate arrival angle using the correlation products. In other words, spectral estimation provides a complex output for each range bin where the magnitude across channels will be similar for a given target (corresponding to range) but the phase will be slightly different. This difference in phase is effectively subtracted in the process of correlation, providing a consistent angle difference across the correlation matrix which can be used to estimate angle of arrival.

The angle of arrival estimation itself is done through beamforming in software postprocessing. This process spatially filters correlation data to best receive signals from a desired target (UAVs in this case) while attenuating returns in other directions [17]. This allows the UAV traffic management system to "latch on" to targets and monitor them, estimating where targets are and collecting information about them. Following target estimation, potential targets are used in DAA algorithms for UAV traffic management and collision avoidance.

While beamforming and target estimation algorithms are run in software where different algorithms can be developed and tested with minimal recompiling effort, spectral estimation and correlation are foundational for most postprocessing algorithms and can all be done quickly in the FPGA fabric which excels at parallel computations which are predictable and unlikely to require modification. With analog radar on the front-end, and target estimation algorithms on the software back-end, the preprocessing DSP chain in the FPGA fabric is the integral link between them. In this chapter we describe how the fabric computation works and how it contributes to our goal of a small and efficient UAV traffic management system.

2.2 Design Overview

Our primary objective is to design an architecture which can quickly process 16 parallel channels of incoming data and perform at least spectral estimation and cross-correlation for radar target estimation. Beyond this, we also desire additional signal conditioning operations such as filtering and averaging, as well as enough configurability to allow us to choose what operations to perform and how to process incoming samples.

Thus, the DSP processing chain in this design is made up of several functional blocks: serial-to-parallel input formatting, mean subtraction, windowing, FIR filtering, downsampling, FFT spectral estimation, cross-correlation, and averaging. For maximum configuration, these blocks are connected to and controlled by state machine controllers and glue-logic units such as the the LRF Controller, AXI DMA engines, DMA engine controllers, AXI4-Stream interface switches, clock domain crossing buffers, and IO conditioning circuits. Together, the functional blocks and their connecting logic create an integrated chain which is self-managed and consistent. A diagram of the DSP chain functions is shown in Figure 2.1, and the Vivado IPI block diagram is shown in Figure 2.2.

This discussion is not an exhaustive look at how all of the included IP can be used, nor is the purpose to explore the various status and control options of each. Rather, we look at how each piece fits into the whole to accomplish the UAV traffic management goals of the project. All sections describe the IP's architecture and purpose, and some include data plots and performance

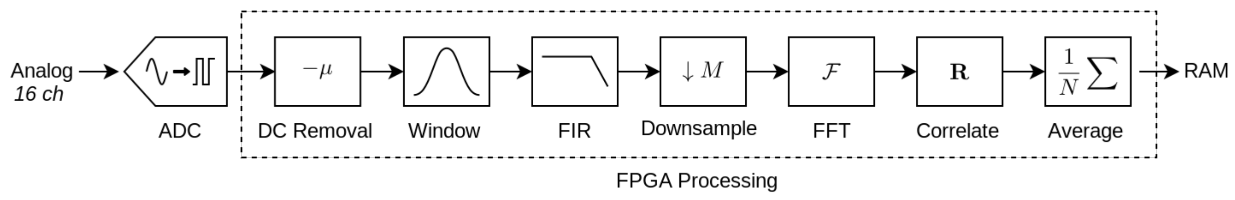


Figure 2.1: Block diagram of essential DSP chain operations; courtesy of David Buck

results in the context of their intended use-case. For convenience, each IP which contains a state machine has a high-level state machine diagram to show essential transitions (not exhaustive); all diagrams begin in the INIT state (except mean subtraction which begins in WAIT) upon reset and startup.

For complete information about features and configurability, see the IP product guides, in particular the LRF Controller product guide which explains the pinout definition for all control-status interfaces (CSI) of the functional blocks and memory-mapped space, and which is included in Appendix B. Furthermore, all code and project files created for this design are found in the project's git repository.

2.3 Latency and Initiation Interval

Two of the most important specifications of digital design are its latency, which describes how many cycles are taken between input and output, and its initiation interval, which describes how many cycles after beginning one computation that another can begin. In this section we make note of the latency of each IP individually and then discuss the packet latency of the design as a whole. We define packet latency to mean the latency from the beginning of the first sample of a packet to the end of the last sample being sent to memory, while IP latency refers to the cycle delay between when a single sample is present on the input of an IP and when it is seen on the output. We then discuss the packet initiation interval which defines the lower limit for radar PRI.

2.3.1 IP Latency

While every IP in the DSP data path has some constant latency associated with it, some IP have latency which depends on wait states that may be asserted by downstream IP. This scenario

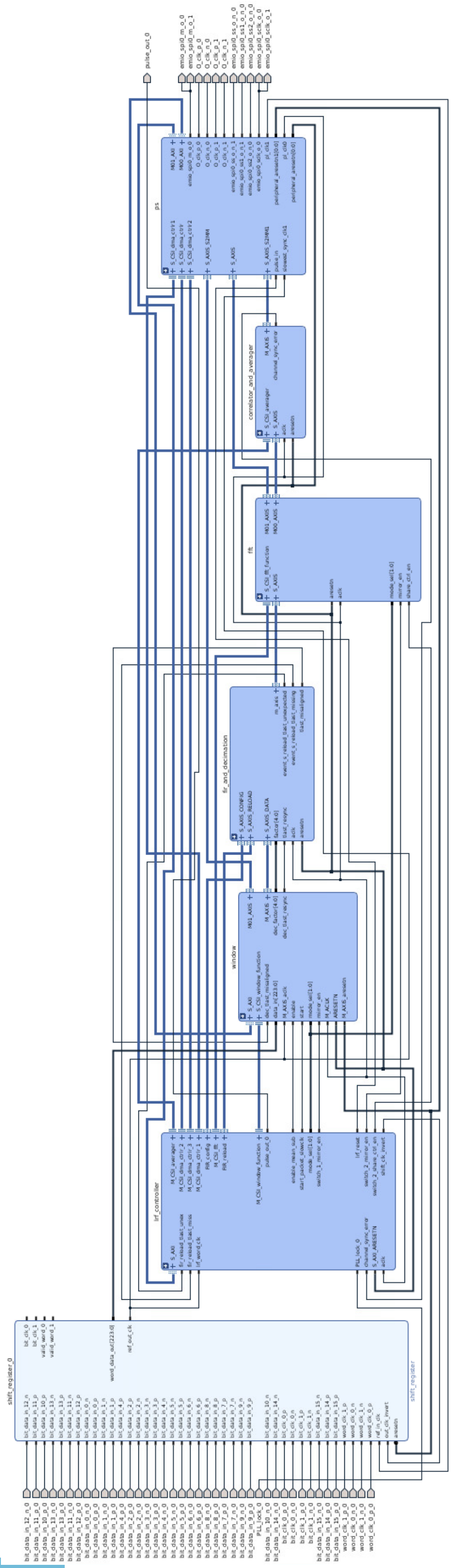


Figure 2.2: Vivado IPI block diagram

is unlikely except when interfacing with the DMAs which may pause a transfer if memory access is congested. Table 2.1 describes the latency of each IP assuming that no wait states are asserted. It also assumes that a packet is being processed so that data is being processed on every cycle. For general reference, all individual IP except the downsizer (see below) have an initiation interval of 1, i.e. new data is accepted at the input on every cycle once it has begun.

2.3.2 Packet Latency

While Table 2.1 is interesting, and may be helpful for using and observing the functionality of each IP, it really doesn't tell us how long the chain takes as a whole. Because processing occurs in packet format, the true latency we care about is how long it takes for a fully sampled packet to make it from one end to the other. Thus, we need to know the packet latency, which depends on features like clock domain crossings (which is complicated by the changing sampling frequency) and variable downsampling rate and window sizes. Packet latency which is subject to change because of clock frequency and packet size requires more than just a count of clock cycles from beginning to end, and may be more easily described using time rather than cycles for

Table 2.1: Latency for data path IP (functional IP)

IP	Latency	Notes
Shift register	2	—
Mean subtraction	1	—
Windowing	4	—
Clock Converter	?	Latency is very unclear for this IP [18], though 8 sync stages are used
FIR Compiler	30	Note that this IP assumes an input to output valid data ratio which is not 1 [19]
Downsampling	0	Data has latency 0 but downsampling implies latency of the downsampling factor between outputs
FFT	14489	Counted from start of first word to end of last [20]
Correlation	12	Blocking configuration is 12, nonblocking is 9
Downsizer	2	Latency is always 2 but the initiation interval depends on input and output size; in our case, II is 2
Averager	4101	Latency of input to output is technically 5 for computation, but averaging of packets implies an entire packet must be introduced before output is available, making latency 4101 (for one packet of 4096)

the sake of calculating PRI. So, we will continue this discussion by counting cycles given certain configurations and then calculate the total packet time in seconds.

The simplest way to explain is to provide application examples which outline the packet latency in extreme cases. In the following examples, we assume the second clock (the AXI clock) is running at 300MHz, and point out that the packet size is constrained by the FFT size of 4096:

1. Consider a sample frequency of 40MHz (the specification max) running with a downsampling rate of 1 and therefore a packet size of 4096 samples, which will be processed through the full chain and sent to memory via the averager DMA. The startup latency taken to pass from the FPGA input pads, through the shift register, mean subtraction, windowing, FIR Compiler, and downsampling, and then into the FFT function takes the latency of the individual IP plus the size of the packet, or $(2+1+4+8+30+0)+4096 = 4141$ total cycles (we assume clock converter latency is the same as sync stages). Running at 40MHz on the ADC clock, this takes about 103.5μs. This amount of time is the total time taken on the ADC clock, because the next IP is the FFT which requires the whole packet to be present before processing.

   Because the total ADC latency includes the time it takes to enter the FFT, we can subtract 4096 from the FFT latency which includes the cycles taken for the whole packet to enter and exit the IP, starting us with 10393 AXI cycles. This next part is more challenging because of the downsizer and FFT half frame rule, but after adding and subtracting the FFT packet exit time, downsizer doubling time, and averager done flag which is asserted a few cycles after the last word is accepted, we get the following latency equation for the AXI clock: $(10393-4096)+4096+12+5 = 10410$ cycles. In this equation, we subtract 4096 from the FFT latency which is the packet exit latency (leaving us with just the FFT processing time), then we add 4096 to this difference because the downsizer takes 2 cycles for each FFT output and we only use 2048 outputs. Even though the FFT must still flush the latter 2048 samples, we only use half of the packet size because the averager asserts its done flag half way through the FFT dumping its packet (at which point the averager has been filled with the first 2048 samples). Finally, the 12 and 5 correspond to the latency of the averager and correlator IP themselves. Because there is some state machine latency, control flow, and pipeline registering between IP in the design, we can add, say, 10 cycles to this total, giving us a final result of 10420 AXI clock cycles, or about 34.7µs. [1]

Thus, adding the ADC and AXI clock domains, we obtain a final latency (from first input to first output) of $(4141/40E6 + 10420/300E6) \times 300E6 = 41478$ AXI clock cycles or 138.3µs. In the fabric, we verified this by adding a counter which reports the time between the start of processing (when the LRF Controller asserts the start flag) and the end of it (when the averager sends its done pulse). The Controller reported a total latency of 40818 AXI clock cycles which is fairly close to our estimate. Reality proved to be about 660 cycles or 1.6% faster than our estimate. This could be due to inaccurate estimates by Xilinx IP or early processing of data in IP like the FFT before the full packet is present.

Now, the data is still in the FPGA RAMs at this point, and the DMA transfer hasn't taken place. As we discuss later on, a DMA transfer has been observed to take 131407 AXI cycles, or about 438µs, which is almost 3 times longer than the actual processing chain in this example. Thus, the entire chain, including processing, takes 576µs. Luckily, the DMA transfer can overlap with the processing as long as the new packet being processed doesn't begin to try to fill the averager and as long as the FFT has flushed its latter 2048 samples before new samples are introduced. Because the FFT is always immediately dumped and the DMA transfer takes quite a bit longer than processing, the latter condition will always be met, even if the DMA transfer time is the lower limit. But, the averager must be emptied before it sees a new input, meaning that starting a new packet is be possible every $131407 + 4101 = 135508$ cycles, or 452µs, which is the final result for our packet initiation interval. Notice that the real bottleneck in this problem is the DMA, and that our calculation of the full latency really only went to show that it took less time than the DMA transfer. New packets can only really be started as quickly as the DMA can send a packet of data to memory.

2. Next, consider a sample frequency of 10MHz (the specification min) running with a down-sampling rate of 31 and therefore a packet size of $4096 \times 31 = 126976$ samples. Following the same logic as above, we obtain a total latency into the FFT of 127021 cy-

[1] Notice that packet latency is essentially just the sum of the IP latencies plus the size of the packet, but that this is complicated by the fact that the packet size changes along the way. In these computations we must be sure to properly handle packet size changes each time they occur.

cles or 12.7ms. This vastly outdoes the processing time of the rest of the chain including the DMA transfer, which takes exactly the same latency as in the previous example. Thus, as soon as packet size is large enough with a low enough sampling frequency, it is the sampling that is the bottleneck and not the DMA engine. Assuming the rest of the chain takes the same amount of time to process, we are left with a total of approximately $(127021/10E6) + ((41478 + 131407)/300E6) = 13.28$ms for a single packet, or 12.84ms if the DMA transfer all overlaps the slow processing (which it always will in this case).

These two examples assume averaging is only 1 packet. If multiple packets are averaged, the processing time is multiplied by the number of averaged packets, which is then added once to the DMA transfer time after a group of packets is averaged.

Because the combination of downsampling factor and sample frequency change the length and time of a packet, changing them will change the packet latency. In order to properly configure the hardware, care should be taken when setting the LRF Controller PRI counter and other configurations which affect packet size. Not all combinations will produce valid or expected behavior, and may cause DMA engines to hang or data to be corrupted. The latency guidelines discussed here should provide some idea of what to consider during configuration.

2.4 Power Consumption

Power metrics for this design are as of yet incomplete, though the current Vivado power estimate shows it using around 10W of power. The breakdown for this estimate is shown in Figure 2.3. According to the tools, this estimate has a confidence rating of “low” because switching probability for the IO wasn’t explicitly declared. However, Vivado assigns a default static probability of 0.5 to signals and a toggle rate of 12.5%. This means that the tools assume signals have an equal likelihood of being a 1 or a 0 (which is a fair assumption for ADC data) and that they will be switching about once every 8 cycles [21]. This latter assumption will likely be faster than the upper bits of each ADC will be changing and slower than the lower 4 or 5 bits will be changing (due to noise). On average it may not be far off from reality, but it is hard to know without detailed observation of actual radar data.

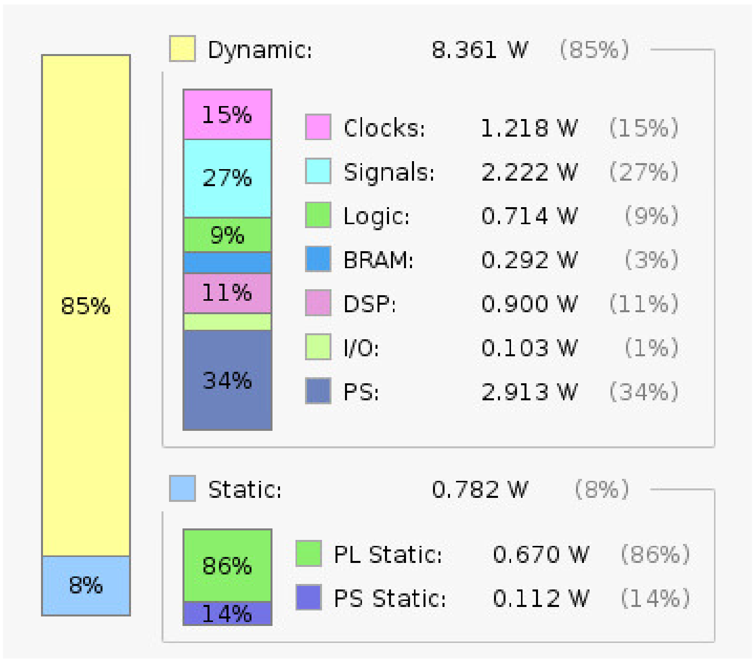


Figure 2.3: Vivado power estimate breakdown

Without accurate knowledge of the inputs, this estimate may be imprecise but it does given an idea of the relative power consumption of each resource group in the fabric, including the PS.

Power consumption for the fabric is assumed to be generally better than what a CPU could accomplish if it were to achieve the same amount of computation and throughput in the same amount of time. This is not necessarily a bad assumption [9] [10], though taking such metrics has not yet been performed and is left to future research as described in the conclusion in Chapter 6.

2.5 Serial to Parallel Shift Register

The first functional block in the DSP chain is a custom serial-to-parallel shift register which takes the serial bits of the ADCs and formats them in parallel words for processing. Because the IO constraints and timing challenges of this IP are unique, this section offers a lengthy discussion of problems encountered and methods used to achieve timing closure within the desired sampling frequency range for this IP.

In the following paragraphs, “frame clock” refers to the clock which delineates groups of 14 DDR bits, while “data clock” refers to the clock which registers the serial DDR bits themselves.

The word “frame” is synonymous with “word” used in this and other IP descriptions, and refers to a group of 14 bits which corresponds to a single sample of the ADCs.

The following list outlines constraints that the shift register must meet, either as the consequence of radar design objectives, or because of other custom hardware such as the ADC chips with which digital logic in the fabric must interface correctly.

1. Digital data formatted as defined by Analog Devices’ AD9257 ADC chip [22]
   (a) 14 bit DDR words transmitted serially
   (b) Differential data, frame clock, and data clock lines
2. Data streams which come from two ADC chips (8 channels per chip) whose clock and data lines may not be synchronous
3. An ADC sample rate range of 10–40 MSPS
   (a) 70–280MHz data clock frequency (due to the 14 bit DDR words arriving at the sample frequency range)
   (b) Effective serial bit rate of 140–560MHz (as implied by DDR data at the data clock frequency range)
4. Input signals which include 16 differential data inputs, 2 differential data clock lines, and 2 differential frame clock lines (for a total of 40 input wires)

2.5.1 HDL Design for Variable Clock Timing

In order to achieve accurate shift register performance on DDR data, some Xilinx FPGA chips provide dedicated DDR register primitives that convert data of varying lengths from serial to parallel. While there are various ways to configure these primitives, because the available output widths for most configurations would complicate achieving 14 bit words, the design choice was made to use the simplest primitive instantiation—the IDDRE1—which samples one rising edge bit, then the next falling edge bit, and produces both bits in parallel on the following rising edge of the data clock [23].

This primitive was replicated for every input channel of data, and each channel was registered 7 times to produce 14 bit parallel output words. After the last shift is performed, the words are registered once to hold the completed, parallelized words constant for a full cycle of the frame clock, allowing for safe sampling of the parallel words at the output of the shift register using a time-advanced version of the frame clock originating from the MPSoC block—the MPSoC clock which drives the ADCs is also used directly to sample the output of the shift register, making the shift register output frame clock a time-advanced version of the ADC frame clock.

We note here that an alternative solution using the data clock to register bits on both rising and falling edges, and using ADC frame clock to directly sample data out of the shift registers, may be a simpler solution but presents significant timing challenges. The issue is that the rising edge of the frame clock as produced by the ADC is aligned exactly with the first bit of a new frame (while the data clock can be dynamically phase delayed relative to both of these [22]). This means that in order for the frame clock to meet timing with the data clock registers while sampling at 40MHz, it would have to meet an ideal hold timing margin of between 0 (assuming a data clock phase of 0°) and 1.79ns (assuming a data clock phase of 360°) which is *very* small for the UtraZed-EV MPSoC. This is also assuming that all PCB lines are perfectly phase matched, that all IO buffers provide exactly the same delay, and that the distance from the frame clock buffer to each data register is the same. This latter point is actually a poor assumption because clock signals generally pass through different input buffers than data signals do. Such a small margin can quickly lead to metastability and bit errors when these variations occur.

Alternatively, different timing challenges arrive when the data is registered in IDDRE1 registers but both clocks (data and frame) are buffered and used as clocks to register bits and words. On one hand, the bits being registered on the data clock are shifted according to the phase of the data clock, thus throwing both data and clock slightly out of phase with the frame clock. On the other hand, the IDDRE1 primitive adds a full data clock cycle of latency (2 bits-worth) to the data path which the frame clock does not experience, putting the frame clock even more out of phase with the data. To avoid these phase issues, rather than directly clocking completed words out of the data registers with the ADC frame clocks, the frame clocks were treated as regular data signals and passed through IDDRE1 registers which were driven using the data clock, allowing them to be sampled (and thus precisely synchronized) with the data. This method is not new, and was motivated in part by work done by Andreas Olofsson shown in his public repository [24]. These new sampled versions of the frame clock were used as a “valid” signal to clock out completed parallel words on the data clock, where they are sampled on the time-advanced frame clock mentioned previously. Because the time-advanced clock comes from the same source as the ADC frame clock, both clocks are guaranteed to have exactly the same frequency but with some unknown phase difference. To avoid the possibility of poor timing due to this unknown phase, an optional clock inversion circuit was placed in the shift register so that either polarity can be used. One of the two is always guaranteed to sample with proper timing.

2.5.2 Vivado Implementation Constraints

To properly monitor timing and ensure that the hardware behavior matched simulation behavior, input timing constraints were added to the Vivado XDC file and adjusted in an iterative fashion using the Vivado timing reports. Using these constraints we were able to ensure either that the shift register met timing or that any timing errors we observed matched expected outcomes and could be safely ignored (using false paths, clock groups, etc. [25] [26]). The input constraints take a form similar to the following:

```
set_input_delay -clock [get_clocks virt_bit_clk] -min 0.12 [get_ports {
    bit_data_in_* word_clk_*}];
set_input_delay -clock [get_clocks virt_bit_clk] -max 0.25 [get_ports {
    bit_data_in_* word_clk_*}];
```

Notice that both lines constrain the rising edge of the word clocks (frame clocks) and data signals, but that the first line constrains the “-min” delay while the second constrains “-max” delay. These min and max tags describe the earliest and latest that the constrained signal is known to arrive following a clock edge.

Additional constraints for the falling edges, as well as for clock grouping, false paths, and other timing attributes, are described in the XDC file which is included in Appendix A. Also note that additional placement and routing constraints are discussed in Chapter 3—when integrated into the full design, congestion often prevented the shift register from meeting timing so additional constraints were created to force adequate placement of the shift register.

2.5.3 Shift Register Performance on Variable Clock Frequencies

To verify performance of the shift register, the ADCs were configured to produce a PN9 sequence at various sample frequencies. This sequence was formatted in the shift register and sent to memory via DMA engine where it was then checked for correctness in a userspace program. Using the design considerations and timing constraints discussed in this section, the implemented shift register produced accurate output across all ADC channels from 10 to 40 MHz (and following some implementations was seen to succeed at ranges from 8 to 50 MHz) using a data clock phase offset of 60°. In all cases, including many iterations of the fully implemented design, the shift register meets our specifications and produces correct output within the desired sample frequency range.

Balancing SNR with range resolution requires careful selection of packet size (including chirp length), sample frequency, and downsampling rate. For this reason a run-time range of 10–40MHz is desired, as it allows UAV radar systems to change such configurations to best meet the needs of specific traffic profiles. Because a sampling frequency can only capture spectral information up to the Nyquist frequency, decreasing the sampling frequency decreases the bandwidth of measurable frequencies while higher frequencies increases that bandwidth. As the frequencies in the signal we are sampling correspond to target ranges, the lower sampling frequencies will shorten the maximum detectable range of a UAV target while a higher sampling frequency will increase the maximum range. This comes with a tradeoff, however, as the fabric FFT size is fixed and higher maximum range comes at the cost of lower range resolution (as a larger frequency range is divided among the same number of available bins). This balance between range and resolution is discussed throughout this chapter in the context of each IP. As the first point of contact with data, this IP provides a foundation which guarantees that the full desired range of sampling frequencies is possible to provide all of the corresponding ranges and resolutions.

2.6 Mean Subtraction

Due to the chirp's transient response in the analog hardware, there is low frequency disturbance in the data for a short time following the start of the chirp. This disturbance results in DC and other low frequency phase noise which can be amplified by windowing and leak power into neighboring low frequency bins, reducing target visibility in those bins. Because this low-frequency response is fairly constant from chirp to chirp, we placed a mean subtraction IP between the window and shift register. This IP estimates the mean on every packet of data and subtracts that mean from the following packet. Thus, the mean subtraction will give a good approximation for even a changing mean assuming that that mean changes relatively slowly over time. Experimentation has shown that this is a fairly good assumption.

2.6.1 DSP Resource Inference for Multiply-Accumulate

Vivado supports two ways that specific resources can be requested in HDL: instantiation and inference. In instantiation, the user uses a template provided by Xilinx to connect and configure a desired primitive. This can be labor intensive because of the many configuration options, and can make projects hard to maintain. Some primitives are wrapped in HDL modules to provide access to specific features, such as the Xilinx multiply IP which wraps a DSP48 primitive to provide simpler access to a variety of multiply-related functionality. This allows the user to add the multiply IP from the Vivado IP catalog to their project and then instantiate it as a component rather than a primitive. This is even more difficult to maintain because template overhead is coupled with IP configuration options which must be configured from the user interface or via TCL commands, not through the HDL itself. The second way to use a specific primitive is through inference, where the user can follow certain design practices that the synthesis tool recognizes as a corresponding to specific primitive. For example, the '*' operator in Verilog and VHDL is interpreted as a multiply, and Vivado can use this to infer a DSP48 primitive. Then, registering this multiply can enable a pipeline register internal to the DSP48. Inference makes the code more concise and clean but can be hard to understand because the intent of the code may not be apparent without understanding the inference technique. Additionally, inference requires certain syntax and careful observation of resource parameters to produce the desired results. Because HDL can be implemented in different ways, poor coding practice can lead to suboptimal circuits which use the wrong resource (e.g. CLB logic to perform multiplies) or which use resources external to a primitive that could be included internally.

While most IP in this project were developed using instantiations of Xilinx IP for consistency (such as AXI4-Stream FIFOs and FFTs) the mean subtraction IP was designed with inference

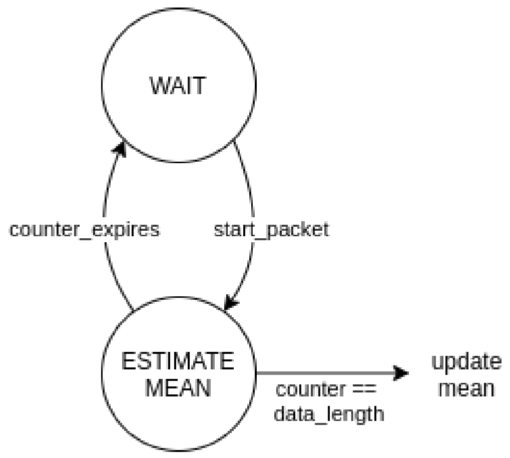


Figure 2.4: Mean subtraction state machine

based on a multiply-accumulate (MAC) circuit in a single DSP48 resource. Internally, the DSP is registered at three places using pipeline registers and connects a reset signal to the final register in the pipeline (which follows the accumulate operation) so that the accumulation can be reset for every new mean that is calculated. To accomplish division after accumulation, 12-bit shifting is done to correspond to the base packet size of 4096, and a 32-word lookup table is used to perform a fractional multiply depending on the decimation factor.

This core is replicated once for each channel and wrapped with a small, two-state state machine shown in Figure 2.4 to count incoming words and update registers which store the mean for each channel. After a new mean is computed, it is subtracted from every input word in a free-running fashion so that the control-flow of the data stream is minimal and doesn't require any handshaking. This makes it easy to enable and disable the mean-subtraction feature at any time during processing. For better timing, the subtraction operation is registered once, adding one cycle of latency to the data path (the MAC registers which estimate the mean aren't included because they aren't part of the data path).

For maximum correctness, the IP was designed so that the start signal which begins the window function (counting the first word at it's input) also corresponds to the first word inside the MAC's DSP48 which is averaged for mean estimation.

2.6.2 Mean Subtraction Performance on Oversampled FFT Data

Figure 2.5 compares raw ADC samples of input noise (interpreted as 16-bit signed integers) with and without the mean subtraction enabled. In the left plots which show time data, enabling mean subtraction visibly recenters the data at zero. The plots on the right show an oversampled FFT with zero padding to extend the 4096 packet size up to 8192 total samples. This oversampling shows the spectral leakage effects of the rectangular window and how they are resolved by mean subtraction.

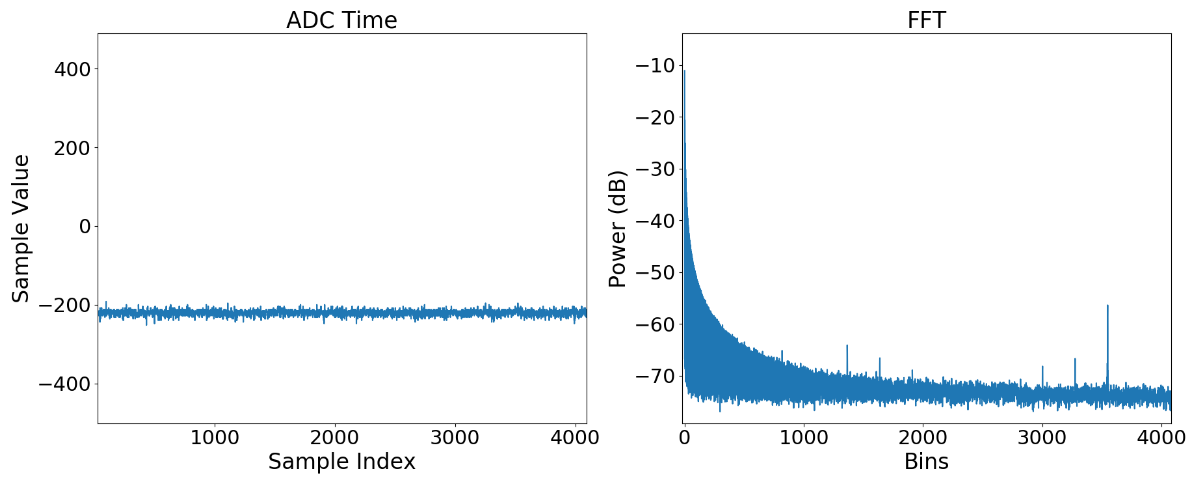
In application, the critically sampled fabric FFT shows slight improvement in the DC bin with mean subtraction, but the FFT's mathematical assumption of periodicity naturally helps to ignore spectral leakage caused by windowing, and so doesn't produce the significant spectral leakage shown in the plots here. However, as ADC data is made available to the user in the event that algorithms are to be run on time-domain samples, the mean subtraction can provide spectral improvement.

2.7 Windowing

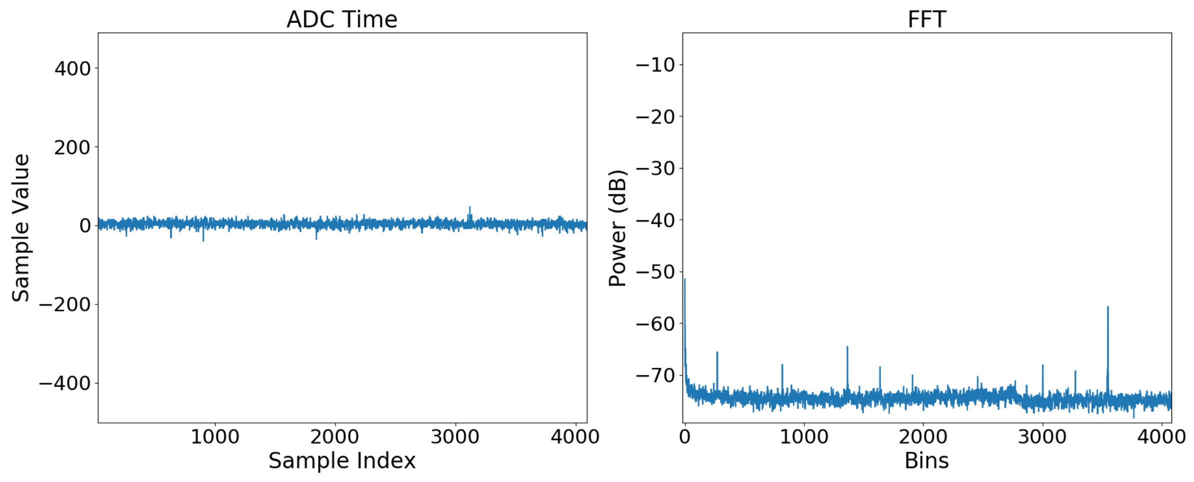
Windowing for this design was done using a custom IP which both packetizes incoming data and multiplies it with window coefficients. The purpose of this block is to provide a consistent multiple-of-4096 packet size for each chirp based on a downsampling value that the user provides. Thus, the downsampling IP later on can reduce the packet to the proper 4096 samples required by the FFT. The state machine for this IP is shown in Figure 2.6.

The WAIT state ensures that proper counter values are obtained before the IP proceeds to allow transfers, and the IP resides by default in the IDLE state until the LRF Controller requests the start of a packet. While a packet is not being windowed, anything at the input is dropped. When the packet is requested, the window reads in downsampling-factor-times-4096 samples, multiplying them one at a time by 16-bit window coefficients from a reconfigurable BRAM block, after which they are made available at the output via non-blocking AXI4-Streaming protocol (slave-asserted wait-states are not supported for this IP) [27].

To reduce BRAM resources by half, while making it possible to achieve a downsampling factor of 31, this IP always assumes a symmetric window. To create a full packet, it will increment



(a) Raw ADC data without mean subtraction



(b) Raw ADC data with mean subtraction

Figure 2.5: Comparison of raw ADC data with and without mean subtraction; left plots show raw ADC data with rect window, right plots show oversampled FFT computed in software

the BRAM address and read half the packet length for the first half of the packet, and then decrement back down to create the second half. Because the Xilinx BRAM Controller requires 32-bit memories, coefficients are stored in pairs and the window function reads out two words at a time. This doesn't affect operation, but does require unique accommodations in the state machine.

Other features for this IP include sign shifting to recenter offset binary data at zero (or accomplish the reverse), as well as a max-data counter which allows the IP to zero the output after a certain point in a packet. Because the window function is configurable at runtime, the user

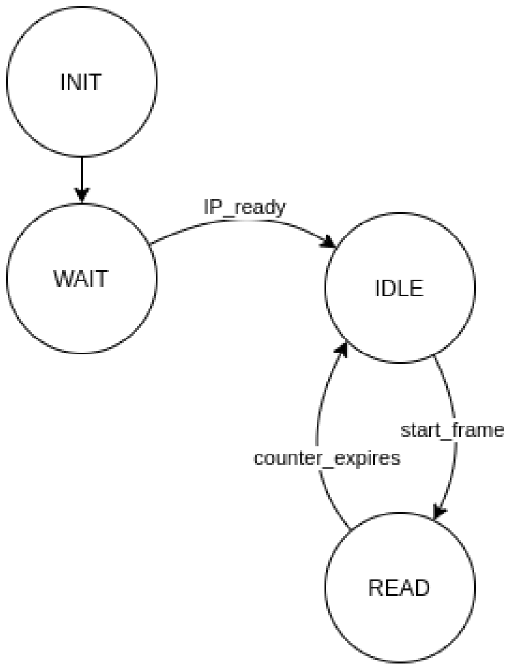
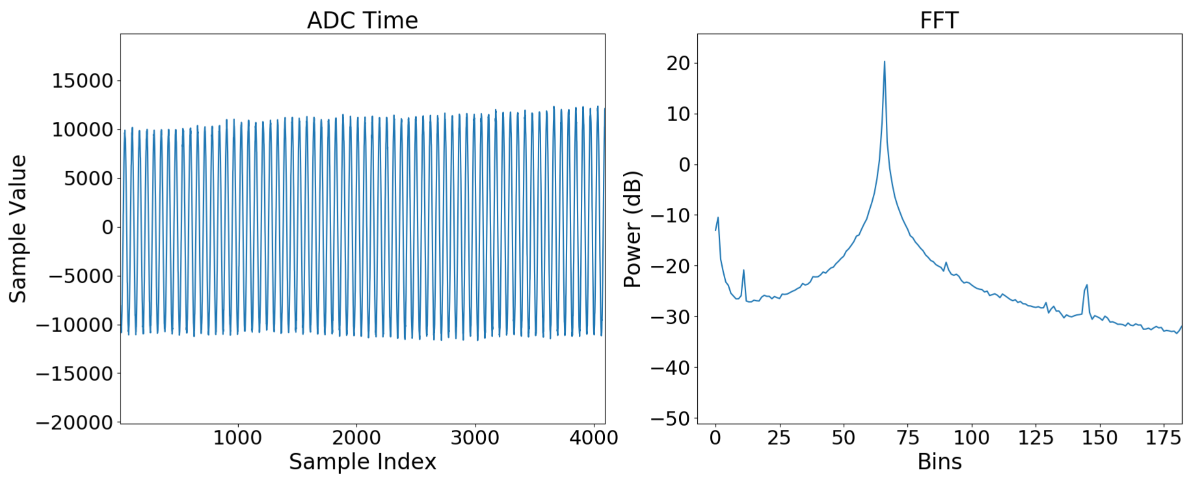


Figure 2.6: Windowing state machine

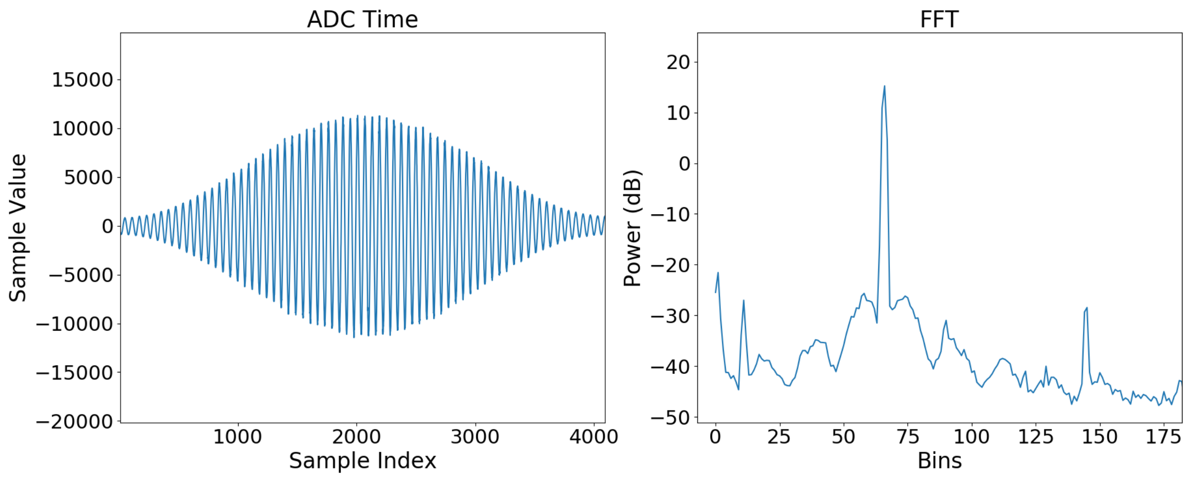
can swap out coefficients depending on the spectral response which is most helpful to UAV target detection. This can happen at any time without consequence to the processing chain because the coefficient BRAMs are configured to have write priority and are managed by the independently mapped BRAM Controller (see Chapter 5 for more information on updating window coefficients).

2.7.1 Windowing Performance on UAV Target Data

Figure 2.7 shows an example of the effects of hardware windowing given two different windows: rectangular and Hamming. The input data for these images is radar data from a chirp loopback test which passes the chirp signal through several hundred meters of optic cable to mimic a bright UAV radar target about 800 meters away. This corresponds approximately to bin 66 in the FFT as shown in the plots on the right. Notice how the Hamming window greatly reduces spectral leakage compared to the rectangular window at the cost of lower overall power (the images show about 5dB loss) and distinct spectral sidelobes.



(a) Raw ADC data with rectangular window



(b) Raw ADC data with Hamming window

Figure 2.7: Comparison of windowed ADC data using rectangular and Hamming windows; left plots show windowed ADC data, right plots show software FFT

2.8 FIR Compiler

FIR filtering for this design is done using the Xilinx FIR Compiler configured for 16 parallel paths, with an input frequency of 40MHz and an output frequency of 300MHz. This distinction in frequencies reduces resources by allowing for fewer DSPs which can be reused and more fully utilized on the faster clock as described in the documentation [19]. The FIR coefficients are created and formatted externally and then sent to the Compiler via a memory-mapped-to-streaming register in the LRF Controller. This configuration creates a relatively small hardware design which can be configured with new coefficients at run time.

The intent of this filter is to act as a precursor to the downsampling block described in the next section so that antialiasing filters can be applied if needed before downsampling is performed.

As this IP is made by Xilinx, its architecture and other features will not be discussed here, however, like the window coefficients, these coefficients are runtime configurable so that multiple filters can be stored and applied as needed from user-space code. Refer to the FIR Compiler product guide [19] for more information about the IP itself, and refer to Chapter 5 for more information about updating FIR coefficients.

2.9 Downsampling

The downsampling function (also called the decimator) is a zero-latency IP which uses the AXI4-Streaming protocol to simply indicate that every Mth word is valid to create a downsampling factor of M. The state machine is shown in Figure 2.8. Additional features include resyncing if the downsampler becomes misaligned with a packet.

The purpose of this IP is to allow us to achieve a wider variety of sample frequencies that would otherwise not be possible. The ADC chips used in the design are only rated for a minimum of 10MHz, so in order to run at the lower sample frequency of 2MHz used in the previous generation design, downsampling is a necessary step.

All of the IP up to this point function together to support a range of sample rates with equivalent SNR: the sample rate can be set between 10MHz and 40MHz, the chirp can be lengthened so that all samples are taken of chirp mixing products, the window function can count up to 31 multiples of 4096 samples, and the FIR compiler can filter all samples in a packet of any size that it receives. Thus one possible way to sample at an effective rate of 2MHz, is to take 20480 samples at 10MHz and then window, filter with an antialiasing filter, and downsample by 5 to produce the 4096 samples required by the FFT. Because the FFT packet size is maintained across input packet sizes, SNR should be relatively stable for whichever configuration profile is used (independent of range and resolution).

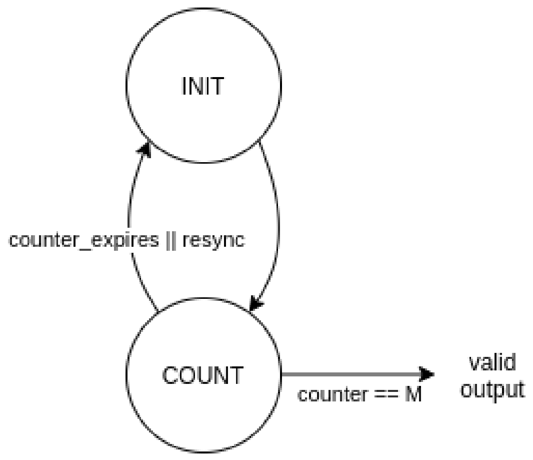
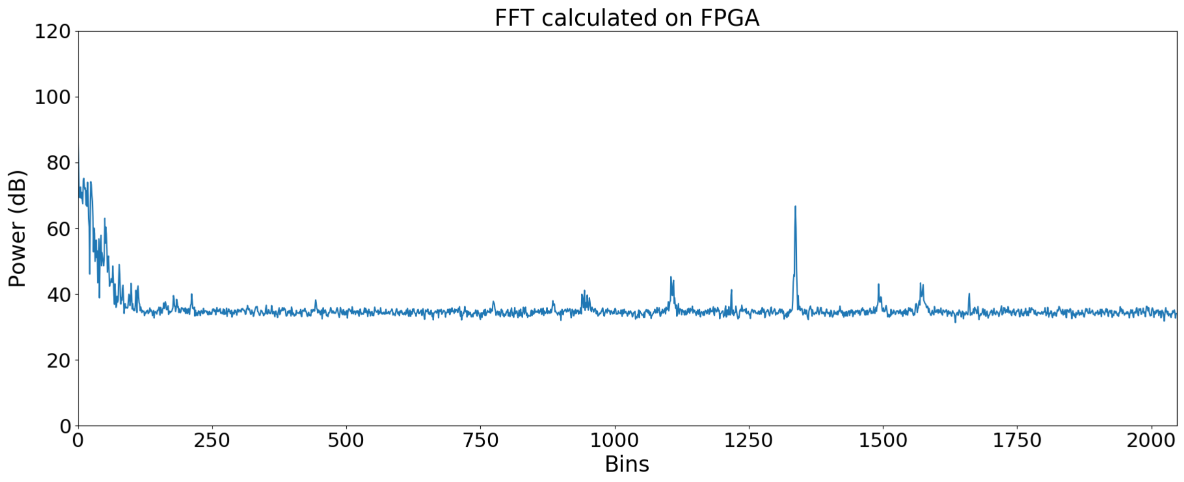


Figure 2.8: Downsampling state machine

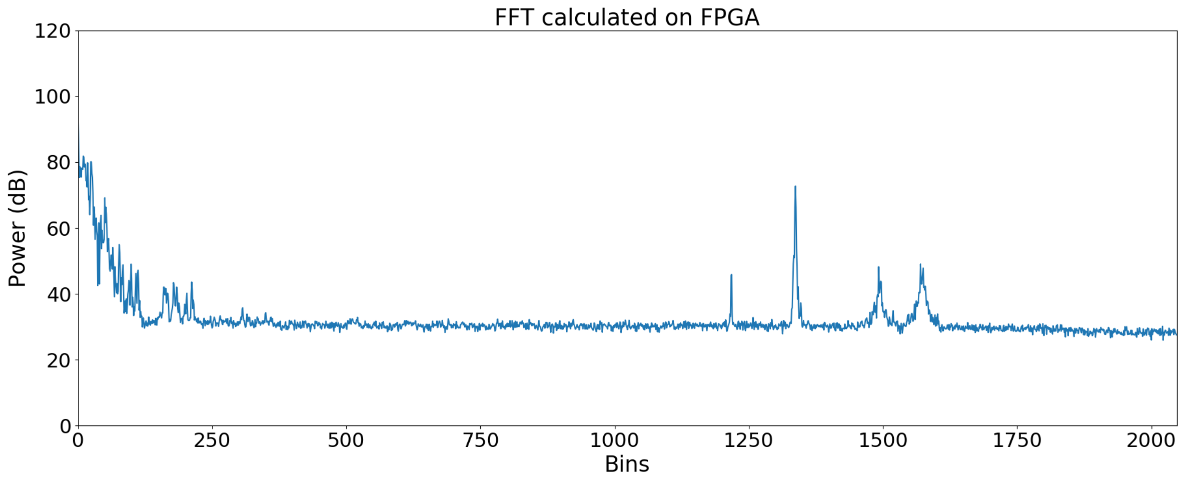
2.9.1 FIR Filtering and Downsampling Performance on UAV Target Data

Figure 2.9 shows the importance of using an antialiasing lowpass FIR filter before downsampling. The plots show loopback data which is first sampled at 40MHz and then passed through the window function containing a Hamming window. In Figure 2.9a the data is filtered using a simple delta to effectively pass the input through to the output. Figure 2.9b shows the same signal which is instead filtered with a $\pi/20$ lowpass filter. Following the FIR filter, both sets of data are decimated by a factor of 20 (for an effective sample rate of 2MHz) and then processed in the fabric FFT engine. The FFT processes 4096 bins, of which the log value (in dB of the FFT's conjugate square) of the first half is shown in the plots.

Because we are using a downsampling rate of 20 in this test, using the $\pi/20$ antialiasing filter suppresses all of the spectrum beyond the Nyquist rate, effectively producing a π filter around the spectral content that we desire. In Figure 2.9a, notice that the noise floor is close to 35dB, and there are several prominent peaks to the left of the target signal which is near bin 1300. When the FIR filter is applied, the noise floor is lowered across the spectrum and several of the peaks disappear. As we have discussed, the lower sampling frequency of 2MHz increases range resolution for a lower maximum range of UAV targets, which is why the target shown in bin 66 in Figure 2.7 is now shifted to a higher bin.



(a) Loopback data fabric FFT with delta FIR filter and M = 20 downsampling



(b) Loopback data fabric FFT with antialiasing FIR filter and M = 20 downsampling

Figure 2.9: Comparison of fabric FFT data with and without antialiasing FIR filter before downsampling

2.10 FFT

To perform spectral estimation on the data after windowing, filtering, and downsampling, a custom IP was created which wraps the Xilinx FFT core [20]. Using the Xilinx core simplified the design, while creating a wrapper allowed for additional monitoring and packaging features, as well as more parallel channels than the Xilinx FFT allows. The wrapper state machine is shown in Figure 2.10.

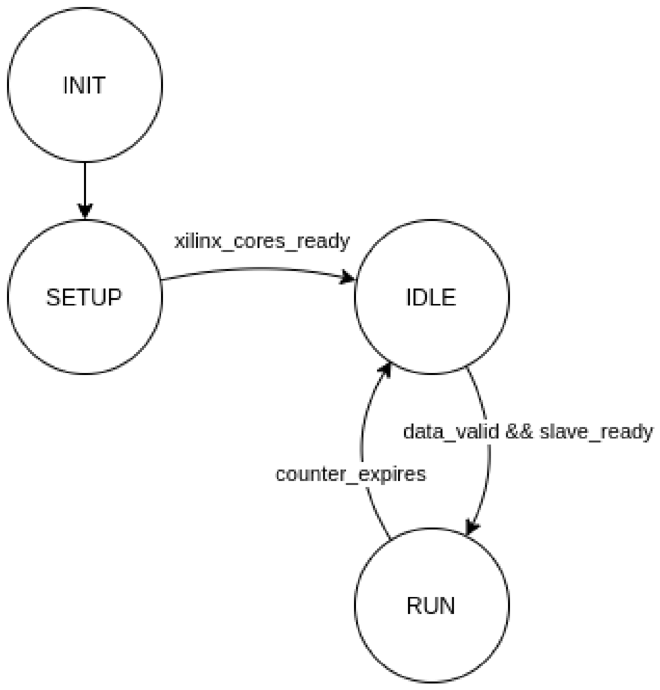


Figure 2.10: FFT wrapper state machine

The FFT cores for this IP were configured with radix-4 butterfly multiplies and a transform length of 4096. The IP's configurable scaling schedule was included as a user configuration option via the LRF Controller. Furthermore, the IP was configured as non-realtime, which means blocking is supported on both sides of the IP. This allows for downsampling of the FIR data on the input and interleaving of the averager packets on the output without packet corruption.

Additional wrapper features include half or full FFT outputs per AXI4-Streaming protocol. If half is selected, the first half of the packet is transmitted and the master tvalid and tlast signals are terminated halfway through the packet. Because our input data is all real, the two halves of the FFT are conjugates, meaning that we lose no information by only using the first half for further processing. This then allows for reduced resources following the FFT as described in the averager section.

Earlier, we mentioned the tradeoff between range and resolution. This is largely because the FFT has a fixed size of 4096 samples which means that increasing the sampling frequency to get a higher maximum range comes at the cost of greater distance packed into each bin, which is to say, reduced range resolution. A variable FFT point size configuration is possible but increases complexity and resources usage; as more than 4096 samples would use too many resources, and we wouldn't want less than 4096, we chose the fixed implementation.

2.11 Cross-Correlation

To perform correlation across frequency bins, we created a correlation IP which generates the upper triangle of the correlation matrix by multiplying each channel with the complex conjugate of all other channels. Because the FFT produces 16 complex channels, correlation produces a total of 256 channels, of which 136 are real and 120 are imaginary. Only the upper triangle is required because the matrix is Hermitian symmetric, and because the diagonal of the matrix is the autocorrelation of the channels, the imaginary part is zero allowing us to compact the data further ad only produce the real half of the diagonal (reducing the imaginary channels from 136 to 120) along with the full complex output of every other channel.

Because of it's relative simplicity, this IP needed no state machine. Additionally, rather than interleaving the data into a smaller systolic pipeline array, the UltraZed-EV board has enough DSP resources for a fully parallel correlator which simplifies the design even further, allowing for 136 parallel complex multipliers with very short latency. Thus, on every cycle a new FFT bin can be introduced and processed, and each correlation word can be produced one after the other at the output.

For the sake of interleaved averager packets discussed later on, the IP was built with blocking multipliers. The blocking configuration is essential for downstream backpressure applied by the downsizer which sends interleaved channels to the averager and only accepts a new word on every other cycle.

2.12 Averaging

The last functional stage of the design, averaging, makes use of the FPGA's UltraRAM blocks to create several parallel multiply-accumulate channels. Each URAM block has a width of 72 bits, and a depth of 4096 [28], allowing us to pack each one with two channels of 36 bits (which is to say an adjacent pair taken from the 136 real and 120 imaginary channels). Additionally, due to failed routing with so many parallel data paths in the design, the number of URAMs used for

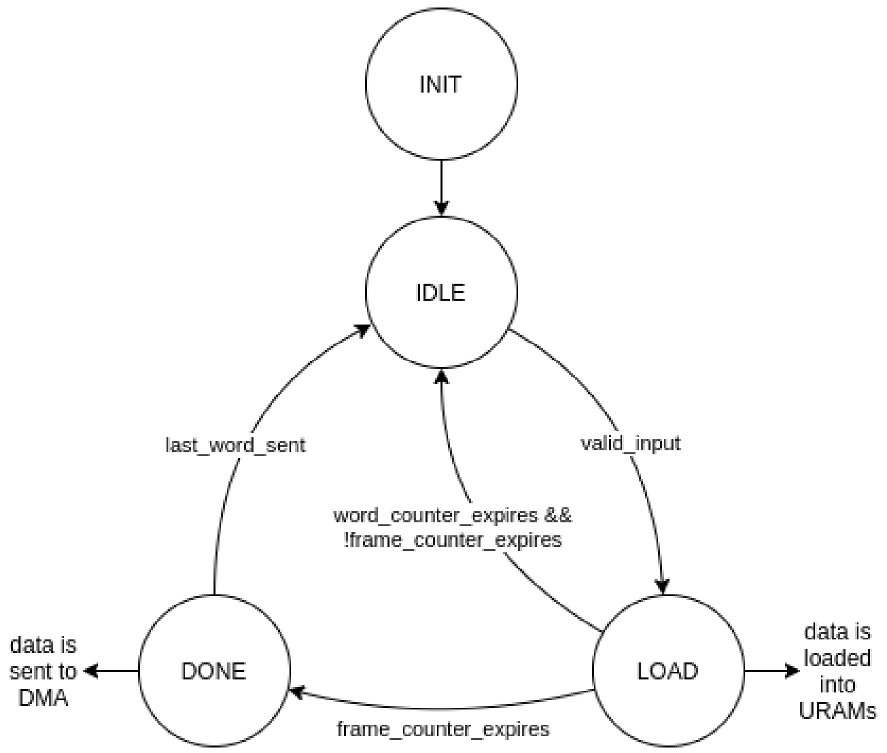


Figure 2.11: Averager state machine

a fully parallel averager was divided in half and the data was interleaved to use the full depth of the URAMs. Referring back to the FFT section, our fully real data input means that we only use the first 2048 samples of the FFT. For the averager that means that each correlator output word downsized into 2 serial words can be interleaved to fit a packet perfectly into the natural 4096 depth of the URAM blocks.

To perform division during averaging, simple bit shifting is performed on every input, and the averager is restricted to only allow averaging for power of 2 input packets between 1 and 64 inclusive. When data is sent to the DMA engine at the output, each 36-bit sample is truncated by 4 bits to form a 32-bit value.

The state machine diagram for the averager controller is shown in Figure 2.11.

2.12.1 Averager Performance on UAV Target Data

Figure 2.12 shows the effects of packet averaging using the same data setup as was used to capture Figure 2.9 in the downsampling section. In these plots, however, rather than raw fabric

FFT data, the plotted data is from the output of the averager which has also passed through the correlator. The plot shows cell [1,1] of the correlation matrix which corresponds to the first channel of FFT data being multiplied by its own complex conjugate, producing the real valued magnitude of each bin for the first channel. Thus, these plots looks very similar to the raw FFT output for the first channel shown in Figure 2.9.

Figure 2.12a shows a noise floor which is at about 20dB, while Figure 2.12b shows how averaging 16 correlation packets reduces the noise floor by several dB, increasing the overall SNR. Averaging improves the spectral estimation by reducing the impact of noise which is uncorrelated with itself from packet to packet, while correlated signal power persists.

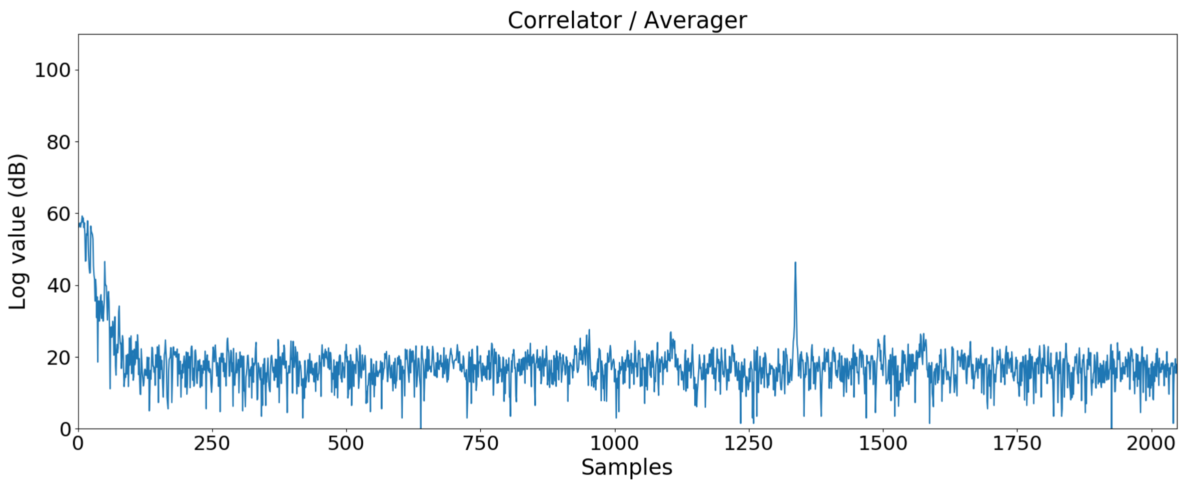
2.13 LATIS Radar Firmware (LRF) Controller

In order for the design to have the most consistent timing possible, a controller IP is necessary which starts chirps and triggers memory transfers at regular intervals. The LRF Controller fulfills this purpose with memory-mapped registers which allow the user to control a variety of configurations at runtime, while at the same time abstracting away direct control over when processing for a packet begins and ends. This section describes the timing features of this Controller and gives an outline of its configurability features.

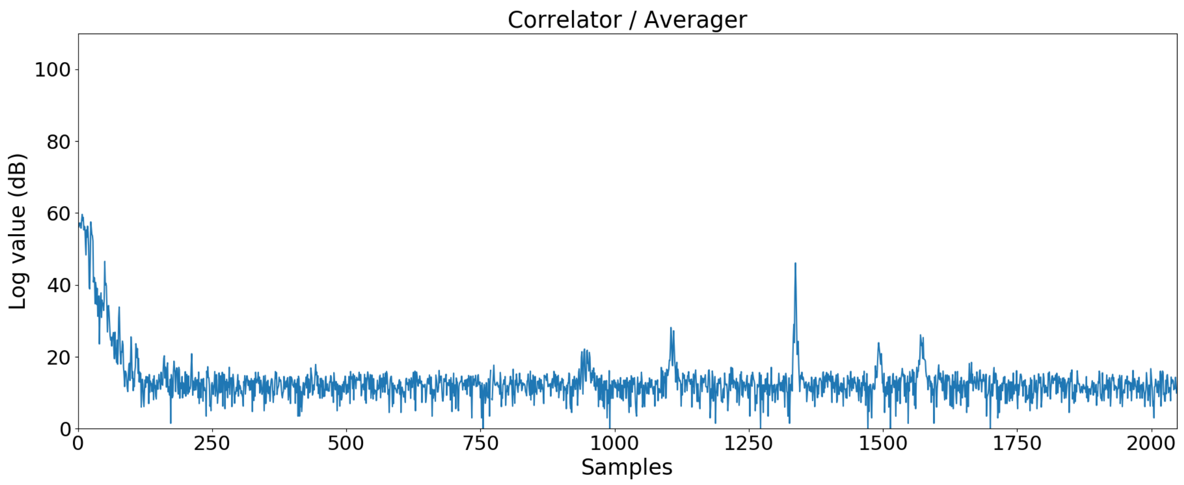
2.13.1 Chirp and Packet Capture Timing

One essential part of the LRF Controller is its ability to begin chirps which are evenly spaced in time. This is one area where FPGAs excel because they can run processes continuously with no interruption. In this design, we want the Controller to be started by the user and then trigger chirps and data capturing in a free-running fashion. This was accomplished with the state machine in Figure 2.13.

During normal operation, the state machine will stay in the RUNNING state where a free-running counter (the PRI counter) defined by the user is continuously incrementing and looping to zero. At counter == 0, the chirp signal is sent, when the counter reaches a user-defined delay value, a packet is started and processing begins, and when the counter reaches a user-defined max value, the counter is reset and starts again at 0. This counter only controls the beginning of a packet or



(a) Loopback data averager output with 1 averaged packet (no averaging)



(b) Loopback data averager output with 16 averaged packets

Figure 2.12: Comparison of averager data with and without averaging enabled

chirp, after which hardware runs autonomously until completion, so it is up to the user to make sure that the maximum and delay values will allow for complete processing of data as described in the latency section of this chapter.

Note that the RUNNING state is only exited if the user turns off the state machine by de-asserting the user_run signal, or if the user changes a significant configuration which requires the current packet to complete and all IP to return to the IDLE state. If not interrupted by the user, packet and chirp timing will proceed consistently as defined by the counters and clock frequency.

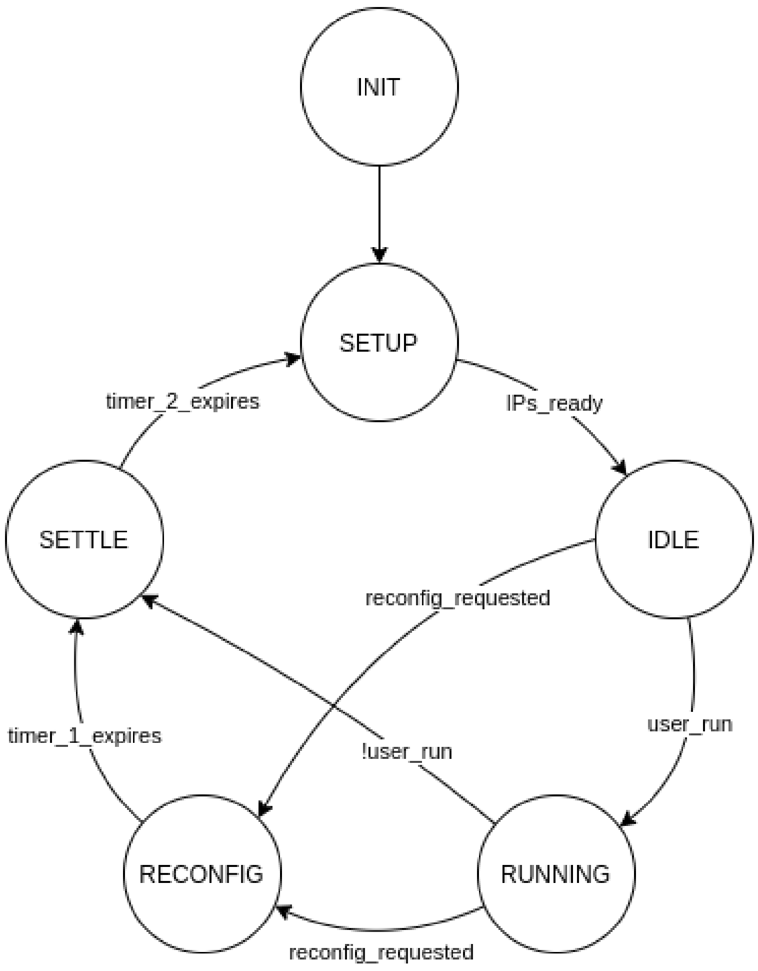


Figure 2.13: LRF Controller state machine

2.13.2 Memory-Mapped Register Space: Control and Status

To provide the user with maximum control over the DSP chain and the ability to monitor it while it is running, a 32-register AXI4-Lite register space was used as a template for the LRF Controller. These registers provide control buses to all IP which require some degree of run-time configurability, and accept status buses which communicate to the user current status and error reports. More information about how to communicate with the register space, and about the control/status interface (CSI), is found in Chapter 5. For a description of all status and control signals available to the user, see the LRF Controller pinout definitions in Appendix B.

2.14 AXI DMA Engines

Three Xilinx DMA engines are integrated into the data path so that data can be observed following the window, the FFT, and the averager. Each DMA is configured in direct register mode with only the write channel enabled, and each is connected directly into the MPSoC processor without passing through an interconnect IP. Though Vivado will naturally try to memory-map the DMA engines into the processor, all DMA port mappings were manually excluded because the DMA naturally has access to all RAM anyway and the processor doesn't need to be aware of where the DMA memory buses are mapped to.

In the LRF Controller, three modes exist: ADC mode, FFT mode, and full DSP mode. The user can set the Controller to any of these three to enable DMA transfers from one of the three access points. Additionally, as the DMA engines can be manually configured to send data to different memory locations, a bit-accuracy mode is included which allows the user to request a single packet. In the LRF Controller, this request will begin transfers in all three DMA engines, saving data from each location as it reaches them during processing.

In the PetaLinux device tree discussed briefly in Chapter 4, each DMA is given a block of reserved memory which is guaranteed to be unused by the kernel, and which corresponds to the default destination addresses for each DMA engine per the DMA controllers. This memory is an ideal place for ring buffers which can use the memory freely and store packets for long periods without worry of data being lost. Ring buffers are discussed in the DMA controller section, and for more information about the DMA engines themselves see the AXI DMA product guide [29].

2.14.1 DMA Transfer Throughput

Logic was added in the LRF Controller which counts the number of cycles between the beginning and end of the averager DMA transfer (running in full DSP mode). In user space this counter value can be read to verify the total time a transfer takes and calculate throughput. One such observation produced a value of 131407 AXI cycles (running at 300MHz). Converting this to time, we get 438µs, and then knowing that the averager sends a total of 2MB of data, we can compute a throughput of 38.3Gb/s. Furthermore, the theoretical throughput can be calculated by assuming that one 128-bit DMA word is transferred on every cycle, resulting in an ideal throughput of 38.4Gb/s. This puts our actual throughput only 0.25% below theoretical which indicates very few wait states asserted by the DMA engine during a transfer. With this lower limit, as long as data is being processed as continuously as possible, this DSP chain can achieve a PRF of almost 2kHz as described in the latency section of this chapter.

2.15 AXI DMA Engine Controller

In order to make the DSP chain as tightly controlled and independent as possible, the fabric DMA engines need to complete transfers immediately when there is data that needs to be sent. To accomplish this, we created a DMA controller that communicates with the AXI4-Lite registers in the AXI DMA engines (configured with S2MM ports in direct register mode) and handles various runtime changes made to packets. The state machine in Figure 2.14 demonstrates the functionality of this controller. For detailed information about the AXI4-Lite register space for the DMAs, see the IP product guide [29].

The controller has a five primary goals. The first is to simply initialize the DMA engine and then wait for a transfer, which it will begin by writing to the DMA length register when the start signal is received. The second is to properly handle the interrupt signal when it is received by writing an IRQ reset bit to the status register when the IRQ output of the DMA (which is an input to this controller IP) is asserted. The third is to enter the DISABLED state when DMA transfers are not desired. This feature is used when a mode enables transfers from certain DMAs while prohibiting transfers from others. This state allows the controller to ignore any start-transfer signals received and prevent the DMA engine from beginning a transfer. The fourth goal is to allow the user to manually rewrite the DMA destination address, which is done in the state machine by returning to the WRITE_DESTINATION_ADDR state from the IDLE state when the new_destination control signal is asserted. The fifth and final goal is to establish a ring buffer configuration that sends every packet to some multiple of a user-defined offset address, forming a contiguous set of packet "links" starting at the destination address. This is to be done in real time and without user intervention once the parameters have been set and the ring buffer started.

While the state machine shown in the figure was intentionally simplified to only show major transitions, this simplicity fails to show transition priority out of the IDLE state, which is to first disable the DMA if that is requested, second, to wait for the start of a transfer, or third,

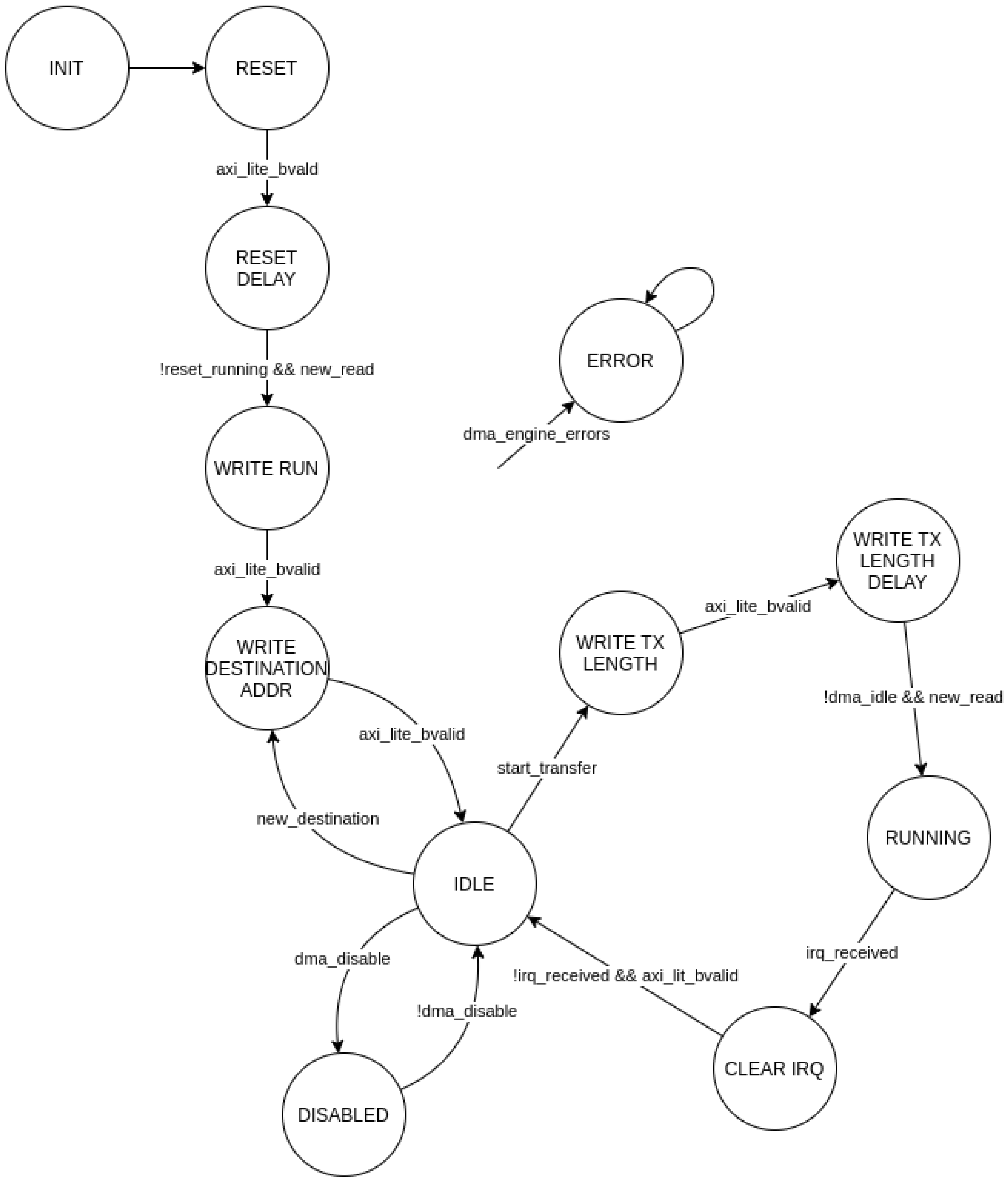


Figure 2.14: DMA controller state machine

to write a new destination address. Also note that from any state, if an error is received from the DMA engine, the controller will enter a permanent ERROR state, which can only be overcome by performing a complete reset so that the DMA engine can be returned to a valid state as the controller reinitializes.

The primary benefit of this IP, in contrast with the previous generation of radar FPGA architecture, is that the data transfers are completely independent of user intervention. If the user fails to collect the data from memory before the next transfer takes place, the data will be lost. This means that proper monitoring of the interrupt registers (or DMA engine interrupt signals) is the only way to guarantee data is not missed, and that proper configuration of the LRF Controller is essential to ensure that the data can be moved in time for new packets to be processed without packet corruption.

These automatic transfers are one of the most important elements of this IP and a great achievement in this design. This controller is the key to packet regularity and consistent pulse repetition interval (PRI) for accurate UAV Doppler processing.

2.16 AXIS Downsizer

As we have mentioned in a few of the previous sections, a downsizer IP was designed to divide the correlation output in half and serialize the two halves one after the other for better usage of the averager memories. This IP observes the AXI4-Stream protocol's standard handshake signals [27] and can be configured to accept an input word of a user-defined size on one cycle and divide it evenly into serial words at the output on the subsequent N cycles.

In our DSP chain, the ratio of input to output words is simply 1:2, so every word taken on the input is clocked out serially on two cycles, with the part associated with the LSB exiting first.

2.17 AXIS Switch

In order for the DSP chain to support shared control between DMA engines and downstream IP, a switch IP was designed which allows for both explicit switching between DMA and downstream control, and automatic shared control between both streams. Explicit switching is necessary in most modes when the DMA is on and the downstream IP is off, or the downstream is on and the DMA is off. Shared control is essential when all DMAs are enabled and data must be sent to the DMA engine and downstream simultaneously.

This IP is designed as a large, zero-latency mux with some minor additional control logic to support shared control and clean mode switching, and uses the AXI4-Stream protocol with standard handshake signals [27].

2.18 Clock Domain Crossing Buffers

As described in the shift register section, the incoming data from the ADCs is arriving at somewhere between 10MHz and 40MHz, while the rest of the DSP chain is running at 300MHz. To synchronize the data path, Xilinx provides the AXI4-Stream Clock Converter IP [18] which we placed after the window function and before the FIR Compiler so that the Compiler could benefit from the low-to-high clock rate ratio. To synchronize control logic, buffer circuits were added throughout the design where signals crossed clock domains. The LRF Controller, for example, which runs on the AXI clock, generates the start signal for the window function running on the ADC clock—buffering this signal requires a brief handshake to guarantee that the start signal has been registered on the slow clock before it is deasserted by the LRF Controller.

This sort of custom synchronizer circuitry ranges from status monitoring in the window function, to starting a packet of data capturing, to conditioning IO before use in the fabric. All of this logic follows good design principles for minimizing metastability (by using multiple synchronization registers) and ensuring that data is carried correctly across domains (using handshakes). Furthermore, false path and clock group constraints were added in the XDC to signal to Vivado that such crossings are managed by the user and should not be considered as critical paths.

CHAPTER 3. RESOURCE AND TIMING SOLUTIONS FOR HIGH-UTILIZATION DSP DESIGN

3.1 Introduction

In order to implement the desired size of processing and control logic present in the design, a large quantity of FPGA resources is required. After reviewing various system on a chip (SoC) options, we selected UltraZed-EV MPSoC produced by Xilinx and Avnet. This device combines a quad-core ARM Cortex-A53 MPCore processor with large UltraScale+ FPGA fabric space, including 38Mb of BRAM and URAM memories, 1728 DSP slices and about 500k logic cells and flip-flops, with a max attainable memory interface frequency of 333MHz and plenty of high-speed IO. Luke Newmeyer's first generation processing chain, which targeted the MicroZed 7010 SOM, contained a fraction of these resources, and would not allow us to build the desired 16 parallel channels. In Newmeyer's words, "future work, will likely require sensing in 3D. This would require the development of a 2D phased array radar. The computation required for a 2D phased array is significantly more complex and intensive than that of a linear phased array. An increase in the number of radar channels results in a quadratic increase in computation. A future radar system will most likely have a 4 by 4 array of elements totaling to 16 channels. Significantly more processing hardware would be required to implement" [1]. The UltraZed-EV MPSoC supplies this additional processing hardware and allows us to achieve 3D sensing with a 2D phased array radar.

Naturally, with a design so large, and with a hope to run it near the upper frequency limit of 333MHz, it is important to follow good design practices and carefully set up timing constraints in order to meet timing. This chapter discusses solutions for ensuring that timing is met in spite of high-utilization and congestion in our design, and ensuring that it will have the best chances for meeting timing if changes are made later on.

To underline why this is necessary, we draw from Chapter 2 which explains the core features of the DSP chain, including spectral estimation and cross-correlation, and various signal conditioning operations for better target estimation. With an endless supply of potential operations, there is a delicate balance between adding more features and the ability of the design to support a feature's increase in resources and routing tracks. While more features may be helpful, they come at a cost in fabric with limited resources.

In our current design, which uses a significant amount of many resources and whose congestion level is consistently 6 (ranging from 5 to 7), routing is most often "difficult" according to the Vivado Design Methodology Guide [30]. This difficulty results in increased net delay from poor implementation quality (poor QoR), which can in turn decrease the maximum allowable frequency at which the design can be run. In order to meet a target pulse repetition frequency (PRF) of 2kHz, we require a low-latency design running at as high a frequency as possible.

In other words, we want the design to have lots of computation logic and we want it to be fast, putting the design in a tight spot for meeting all requirements. As the result of the timing analysis discussed in this chapter, we were able to accomplish both, running at 90% of the max allowable memory bus frequency (at 300MHz) with significant digital processing on impressively wide data bus widths, managing good quality of target estimation with a fast, real-time chirp repetition frequency. This is good news for UAV traffic management, improving Doppler estimation, UAV tracking, and quality of DAA algorithm postprocessing results.

Many strategies found here were collected from online solutions in Xilinx forums and answer records, as well as from Xilinx documentation (primarily [30] [31] [32], though others are listed throughout the chapter).

3.2 Resource Usage

Because the correlation matrix scales according to the square of the number of channels, the number of DSP slices and memory blocks increases dramatically when transitioning from a 4-channel radar to 16 channels. Furthermore, this second generation design gains mean subtraction, windowing, FIR filtering, and decimation, plus two more DMA engines and extensive control logic throughout the design, which greatly increases its resources in contrast with the first generation design. To visualize this comparison, Table 3.1 shows the approximate resource usage of both generations side by side, including the percent increase. Resources are said to be approximate because different implementation strategies produce different resource usage, and we often

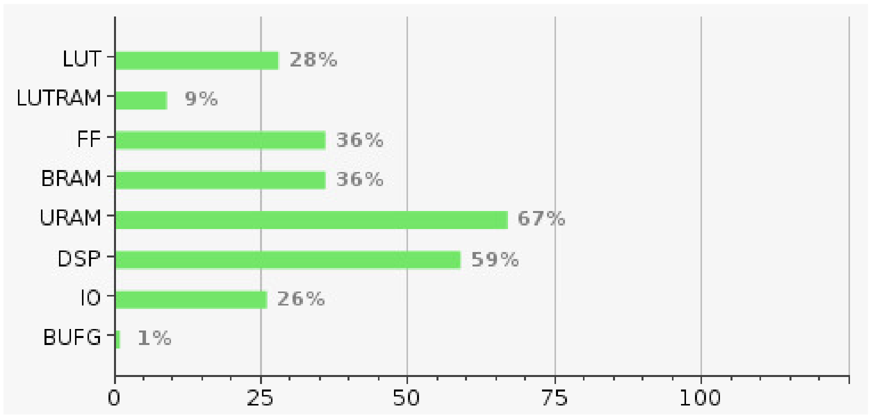


Figure 3.1: Percent utilization of resources on the UltraZed-EV FPGA

ran multiple strategies and used the result with the best timing. Figure 3.1 depicts the percent utilization of the second generation design relative to the available resources on the UltraZed-EV FPGA fabric.

As both the table and figure show, a new resource is available in this design called URAM (UltraRAM) [28]. These RAM are dedicated FPGA memories that are less configurable than traditional BRAM, but are larger per block and can contain all of the averager data for the 16 channel design. This allows the design to maintain large, static memory storage with URAM blocks and smaller, more dynamic memory in other IPs with BRAM blocks.

Table 3.1: Resource comparison between first and second generation FPGA designs

Resources	Gen 1	Gen 2	Increase
LUTs	6335	63830	908 %
LUTRAM	475	9355	1869 %
Flip Flops	7741	164319	2023 %
BRAMs	56.5	112.5	100 %
URAMs	0	64	—
DSP Slices	66	1019	1444 %
IO	37	54	46 %

3.3 Timing Closure

To provide some context, FPGA clocks frequencies normally fall between 100–500MHz, and some IP, such as the AXI DMA engine provided by Xilinx, aren't designed to operate above a given frequency (333MHz in the DMA case). With this in mind, trying to run our design at 300MHz required careful planning to achieve timing closure. This planning involves reducing resources, pipeline registers to reduce delay between computation elements, reduced signal fanout to reduce total delay of signals driving the fanout, improved logic placement to offer better routing and reduce net delay, reduced congestion to improve routing and reduce net delay, and other optimizations such as register retiming. Several of these approaches are discussed here in the context of this design.

3.3.1 Resource Reduction and Sharing

The first and most obvious solution to improve design timing is to use fewer resources. Just as with software, hardware can be designed in many ways and proper attention can lead to smaller and faster designs that accomplish the same objective as an equivalent larger and slower design.

As timing became tighter with the addition of more processing IP, we began to ask the question of what resources could be reused, shared, or removed altogether, which led to optimizations such as the following:

1. By raising the AXI processing clock frequency, the FIR Compiler can make use of the slow-to-fast clock frequency ratio to reduce resources. We exploit this feature by placing the clock converter before the FIR Compiler and guaranteeing that the slow clock frequency will have a max of 40MHz and that the fast clock will be fixed at 300MHz, reducing DSP resources by a factor of 7 (the floor of 300/40).

2. In the averager section of Chapter 2, we discussed how correlation data was serialized to twice the depth at half the width. This allowed us to reduce the number of routing resources required at the output of the averager by half while also using fewer total RAMs at no cost to throughput. It was this change that allowed us to convert from 16 bit averager output words to 32 and still meet timing with 256 total channels.

3. Careful planning of the mean subtraction block allowed us to infer a single DSP48 primitive for each channel's mean estimation. This IP came last in the design, and caused minimal damage to timing as it runs on the slower clock and uses few resources and logic. "Careful planning" in this case meant precise observation of all bit widths to guarantee mean estimation would fit within a single DSP48, and an iterative implementation approach to ensure that inference led to exactly the right resource usage and configuration, including internal pipeline registers.

While some IP can be optimized for minimal resource usage at little cost, others are hard to reduce such as the FFT and correlator, where reducing resources costs a significant increase of latency or complexity. Thus the final design balances some IP which are optimized for resources where routing and timing are most critical, while others are optimized for latency where an increase in resources is possible without significant damage to timing.

3.3.2 Pipeline Registers

One of the most important tenants of good HDL design is the use of evenly (and at times frequently) spaced registers which break up long chains of logic and allow for a faster clock frequency at the cost of higher latency. Because registers define the beginning and end of a timing path, the delay between two registers decides how quickly clock edges can reach them and maintain valid data. The more logic there is between the registers, the longer the propagation time, and the slower the maximum clock frequency. Adding pipeline registers between logic reduces this delay, and therefore increases the maximum possible frequency.

As all data-path IP in the design were built to be size-configurable, (i.e. a variable number of parallel channels, data width, data length, etc.) the full DSP chain was first implemented with just two channels to ensure proper functionality and make sure that timing would be met for a small design. When this was completed, the design parameters were updated to generate the full 16 channels, and because it was difficult to predict how IP resources were going to be placed and routed in the context of a larger design, implementation immediately revealed many timing issues, huge negative slack between registers, and failed routing.

Upon observation, the Vivado timing report showed that many failures were caused by large delays between registers, such as those used for multiplexers on data lines (in the averager), as well as shift and arithmetic operations on data lines (in the window function). In many cases, control signals caused a variety of timing issues because one consequence of more parallel channels is an increase in data bus width, putting more strain on control signals which must reach hundreds or thousands of registers. If these high-fanout control signals must first traverse logic before reaching destination registers, timing may be very difficult.

To attack these timing issues, each IP with failing paths was revisited and registers were added between long chains of logic, as well as at the inputs and outputs of various IP. This improved timing significantly, eliminating failed endpoints and critical paths altogether in many cases.

Adding pipeline registers is fairly straightforward when no handshake or control logic signals are present as free-running registers can be added to the data path. When control logic is present, such as when state machines control the loading and unloading of data from memories, or when multiple parallel arithmetic paths are being performed for optional configurations, registers must be added very carefully to ensure that the control system still moves data with respect to the correct latency.

Later in the design process, as timing began to fail between IP, AXI4-Stream and AXI4 (memory-mapped) register blocks were added between IP to isolate timing issues of connected IP. While some of these registers were used for testing and then removed, others remain, such as those on the output of the DMA engines which buffer the S2MM data paths before they enter the MPSoC memory ports.

3.3.3 Fanout Reduction

Following the addition of pipeline registers, another optimization we performed was to limit the fanout of control signals. Fanout is defined as the number of destinations that a single source must reach. The problem with high-fanout nets is that a single signal is used to drive endpoints in many different places, sometimes spanning multiple clock regions in the fabric, or at least multiple CLBs. This often means that timing can't be met for all endpoints because it is impossible for them to be sufficiently close to the driving source.

The best example of this is in the averager, where a MUX was used to select between seven different shifted versions of the 10240-bit wide input bus using a three bit select signal.[1] This is shown in the following code block (widths have been removed for better visibility):

```
with avg_frames select
    s_tdata_z1_div(...) <=
    std_logic_vector(shift_right(signed(s_tdata_z1(...),1))   when "001",
    std_logic_vector(shift_right(signed(s_tdata_z1(...),2))   when "010",
    std_logic_vector(shift_right(signed(s_tdata_z1(...),3))   when "011",
    std_logic_vector(shift_right(signed(s_tdata_z1(...),4))   when "100",
    std_logic_vector(shift_right(signed(s_tdata_z1(...),5))   when "101",
    std_logic_vector(shift_right(signed(s_tdata_z1(...),6))   when "110",
    s_tdata_z1(...)                                           when others;
```

In this case, the three bits of the avg_frames signal select between different versions of the input s_tdata_z1 bus which is shifted right on a channel-by-channel basis.

Even when the avg_frames, s_tdata_z1_div, and s_tdata_z1 signals are all registered, each bit of avg_frames still has to spread to thousands of LUTs and 2-bit MUXs across the fabric and settle at each data register before the next clock edge. Thus, there is no good central location to place the avg_frames registers which will make it easy to reach every data register and still meet timing.

Obviously pipelining can't help to move the sources closer to their destinations, but if there are many copies of the source registers, they can all be places closer to a subset of the destination registers and improve timing for each group. In Vivado, adding the MAX_FANOUT attribute to the avg_frames register restricts the number of endpoints that a single source can drive, telling the tools that if more than the maximum number are needed, it should replicate the source so that the signal can be placed closer to the endpoints. This is shown in the following lines of code.

```
attribute MAX_FANOUT : integer;
signal avg_frames : std_logic_vector(2 downto 0);
attribute MAX_FANOUT of avg_frames : signal is 64;
```

With this attribute, the avg_frames register bits are only allowed to drive 64 endpoints. The avg_frames registers are replicated as many times as needed to meet this requirement and can all

[1]This width corresponds to the data path before packets were interleaved and the averager width was cut in half.

be places closer to the 64 endpoints they communicate with. Thus, timing can be improved with minimal additional complexity and latency.

One thing to note is that the avg_frames registers themselves are all driven by some other logic in the design, and thus replicating these registers will increase the fanout of the signal driving them. This new fanout can be restricted as well if needed, forming a sort of register tree which grows out in increments. Such an approach increases resources in a logarithmic fashion, but does not increase latency.

When synthesis was run with the MAX_FANOUT attribute included, Vivado performed replication as described in the following log report:

```
INFO: [Synth 8-4618] Found max_fanout attribute set to 64 on net
    avg_frames[0]. Fanout reduced from 11270 to 64 by creating 176 replicas
INFO: [Synth 8-4618] Found max_fanout attribute set to 64 on net
    avg_frames[1]. Fanout reduced from 11270 to 64 by creating 176 replicas
INFO: [Synth 8-4618] Found max_fanout attribute set to 64 on net
    avg_frames[2]. Fanout reduced from 5894 to 64 by creating 92 replicas
```

One important note about the MAX_FANOUT attribute is that it has limitations when the signal crosses hierarchy boundaries [33]. If the attribute is applied to a signal which exits it's module and fans out to drive loads in another module, the attribute will likely not be applied because its application is only observed within the scope of its own module. Some techniques of hierarchy flattening may help this, but often require redesigning the HDL or additional effort in the Vivado tools.

3.3.4 Control Logic Reduction

Control signals, such as set, reset, and enable signals, are often timing critical—both in the sense that they are likely to form part of a critical path and that their timing is essential for proper behavior. Synthesis and implementation tools will prioritize control signal routing and try to give them the best routing tracks, which in turn causes other paths, such as data or state machine logic, to be given lower routing priority. Such prioritizing makes sense but is often unnecessary, and control signals can often be consolidated or removed to improve timing and reduce congestion in critical areas.

As we have mentioned in the previous sections, this design contains many wide, parallel data paths propagating through the FPGA fabric. Initially, all data pipeline registers were connected with the same control logic as other signals, frequently resetting them with the rest of the registers in the IP. Such resetting added thousands of high-priority routing tracks to the design which in turn contributed to increased congestion and poor critical paths. During the development phase of most IP, it seemed as though resetting was important for proper cleaning of the data path, but we realized that most of our data path registers are cleared out naturally after a few cycles, making the reset basically redundant if the first few cycles of data are ignored. Because our radar application doesn't care if the first few startup packets are meaningless, leaving register resets disconnected causes no damage during operation. Removing the reset signal from these pipeline registers allowed register reset ports to be routed directly to ground via whatever routing tracks were lowest priority, freeing up the critical tracks for use by other critical paths.

Making this adjustment didn't resolve all timing issues, but it was noted that several tens of picoseconds could be gained during implementation. Reducing control logic should at least produce slight improvement because of what we have discussed here, though large improvement is unlikely in designs like ours where the original reset tree was still quite small compared to the size of the data buses. Additionally, as routing tracks are still needed to tie resets to ground, the overall routing resources might only improve slightly, and the primary benefit comes from a change in routing priority.

3.3.5 Placement Constraints

For IP which can't be modified, such as those offered by Xilinx, and IP which still have difficulty meeting timing in spite of good design practices, Vivado provides a constraint called a pblock, which allows the designer to easily group cells that they want to be placed near to one another. The following averager pblock constraints illustrate this:

```
create_pblock pblock_avg;
add_cells_to_pblock [get_pblocks pblock_avg] [get_cells -quiet [list
   design_1_i/axis_averager_0]];
```

These lines create the pblock called pblock_avg and then add to it the design cell called axis_averager_0, a cell which includes the complete averager IP and its internal hierarchy. Optionally, pblocks can be constrained to one or more clock regions in the FPGA fabric, but this is less effective for large cells and can be difficult to implement successfully due to more restricted routing, though it can be helpful when working with partial reconfiguration and small cells for which precise association with a clock is needed. Pblocks with no clock region constraints provide softer suggestions to the implementation tools so that the tools can physically group them as best as possible. With or without clock region constraints, pblocks can help reduce net delay on critical paths. In our design, only the shift register was given a pblock constraint limited to certain clock regions because the net delay from natural routing was large enough to prevent the design from meeting timing regardless of input delay constraints. Forcing the shift register to be in the same clock regions as the IO pins reduced the net delay by half and brought the design within range of valid timing closure.

Pblock constraints without clock region restrictions were created for the averager and correlator (as they use resources which will likely be placed all across the fabric by default), as well as for the DMA engines which have long unregistered logic chains. Other IP were also given pblocks of their own or included in preexisting pblocks when critical paths could not be otherwise reduced.

3.3.6 Routing Congestion and Net Delay Improvement

Congestion in our design has been one of the largest obstacles for timing. The Vivado timing rating in most of our placed designs is between 5 and 7, which can quickly impact successful routing and timing closure (both of which did fail in many cases). The most common ratings, 5 and 6, signify that there exist 32x32 or 64x64 areas (respectively) of tiles which have over 100% of their routing resources used [34]. When routing tracks in critical regions are being completely utilized, some nets are forced to take longer, sub-optimal paths which worsens net delay at best, or prevents routing altogether at worst. Figure 3.2 shows the congestion density map for one of our implemented designs, red and orange indicating the highest congestion.

Net delay can be a significant timing issue when a design is large or has high congestion. The problem is that when elements are routed via long tracks and must traverse between clock regions and multiple logic tiles, the delay of propagation between two registered points can take nearly all (say, >80%) of the total path delay. One such example of this is shown in Figure 3.3. Notice that the net passes directly from the output of one register to the input of another with no

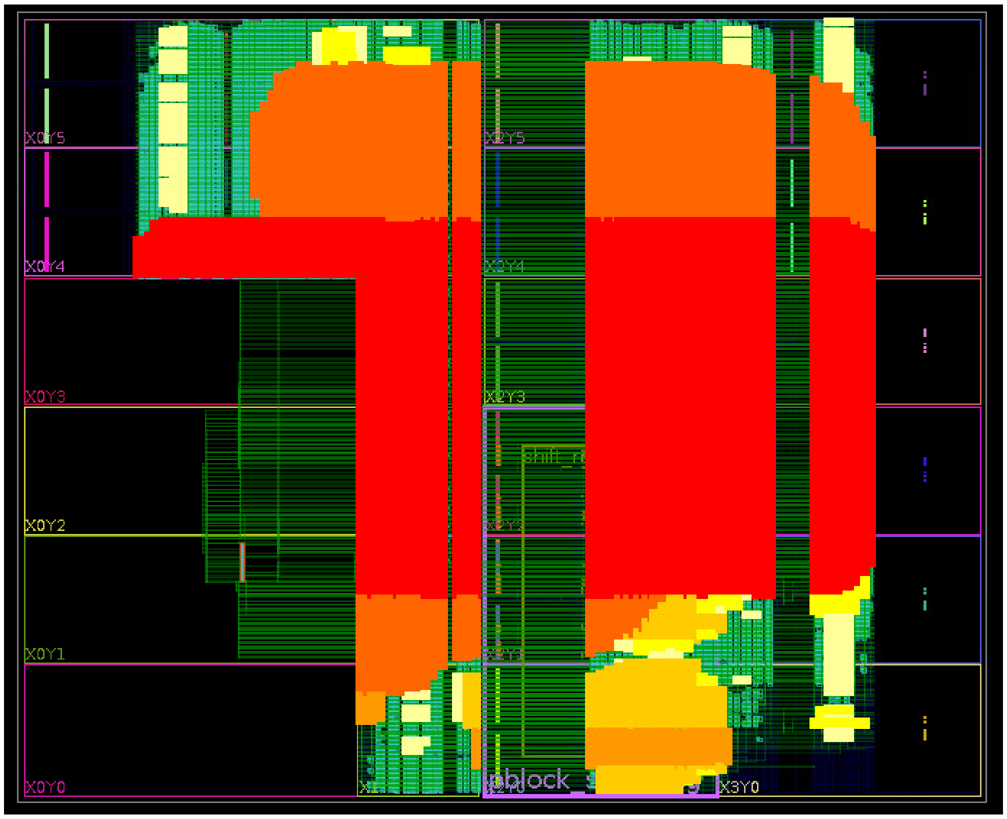
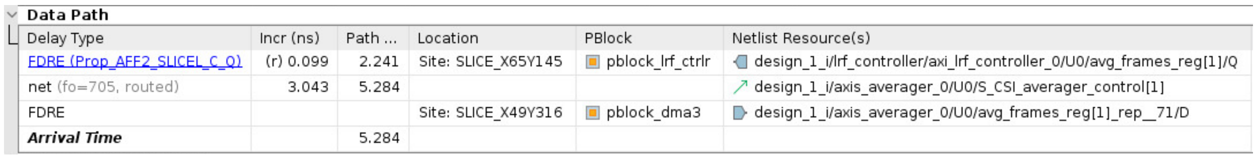


Figure 3.2: Vivado congestion density map overlayed on routing



Data Path

Delay Type	Incr (ns)	Path ...	Location	PBlock	Netlist Resource(s)
FDRE (Prop_AFF2_SLICEL_C_Q)	(r) 0.099	2.241	Site: SLICE_X65Y145	pblock_lrf_ctrlr	design_1_i/lrf_controller/axi_lrf_controller_0/U0/avg_frames_reg[1]/Q
net (fo=705, routed)	3.043	5.284			design_1_i/axis_averager_0/U0/S_CSI_averager_control[1]
FDRE			Site: SLICE_X49Y316	pblock_dma3	design_1_i/axis_averager_0/U0/avg_frames_reg[1]_rep__71/D
Arrival Time		5.284			

Figure 3.3: Long net delay timing report example

additional logic in between, and that the net delay is 3.043 ns. Per the critical path summary, the net alone takes up 96.8% of the total path delay, clock-to-Q of the first register taking up the remainder of the delay. This net delay alone limits the maximum frequency to 330MHz, and that doesn't even allow for additional logic between registers.

Resolving net delay is challenging. Vivado provides some implementation strategies that focus on logic spreading to reduce congestion, and pipeline registers may help so that the net itself can be broken up, but net delay is different from logic delay in that it largely depends on the result of the place and routing algorithms, which can change from run to run as the design changes. Using pblocks as discussed in the previous section may help to reduce net delay by preventing nets from crossing clock regions or at least by placing cells closer together, though the benefit of this may be lost for larger cells. Additionally, Xilinx provides some specific tool-based guidance to improve congestion and net delay [31]. Ultimately, we experimented with a variety of Vivado implementation strategies which proved to be the best solution to resolve net delay and achieve timing closure when all other design improvements were exhausted.

3.3.7 Fixed Placement and Routing

One unique challenge with FPGA design is that placement and routing of individual IP during their development can be much different than when they are integrated into a large project. During the design process, timing can be analyzed, IO constrained, clock definitions added, and good HDL practices used, but there is always some degree of unknown variability in how integrated designs will be organized in the fabric, especially as the designs grow larger. This means that while placement, routing, and timing analysis for individual IP may be insightful, timing-critical paths may benefit from manual placement and routing constraints if timing cannot be met when a design is implemented. Adding these constraints can help to guarantee timing results of constrained cells by not allowing the tools to move them around during implementation. Vivado allows such constraints, and generates them itself during implementation, but creating them manually can be very difficult because each constraint requires knowledge of which HDL cell is to be tied to which precise resource of the fabric. This is infeasible in most designs, as even small circuits can use hundreds of resources, so using Vivado to generate a set of good constraints that can be copied and used elsewhere is a better place to start.

This strategy was used in the shift register of this design (discussed in Chapter 2) which has very tight timing margins that consistently failed timing when implementation was run. When the shift register was implemented alone, placement and routing were obviously satisfactory (registers and logic were placed close IO ports) and produced good timing margins, but when integrated into the complete design, register placement was significantly spread out across the fabric. Because we wanted to maintain the shift register's standalone timing when it was ported into the complete design, we exported and saved the constraints generated during place and route of the standalone project to a separate XDC file. This XDC file was added to the constraints of the integrated project so that when implementation was run, every element in the shift register would be given the same good placement that it had been assigned when it was not competing with other IP.

Figure 3.4a shows the implemented timing results of the standalone shift register design while Figures 3.4c and 3.4b show the difference between runs of the large design, one without manual place and route and one with it. In all three images, keep in mind that the inter-clock path between virt_bit_clk and bit_clk is actually an intra-clock path because both of these clocks are timed together. Functionally, all intra-clock paths are internal to the fabric, while the inter-clock virt_bit_clk-to-bit_clk path demonstrates the transition from data outside to inside of the fabric at the IO ports.

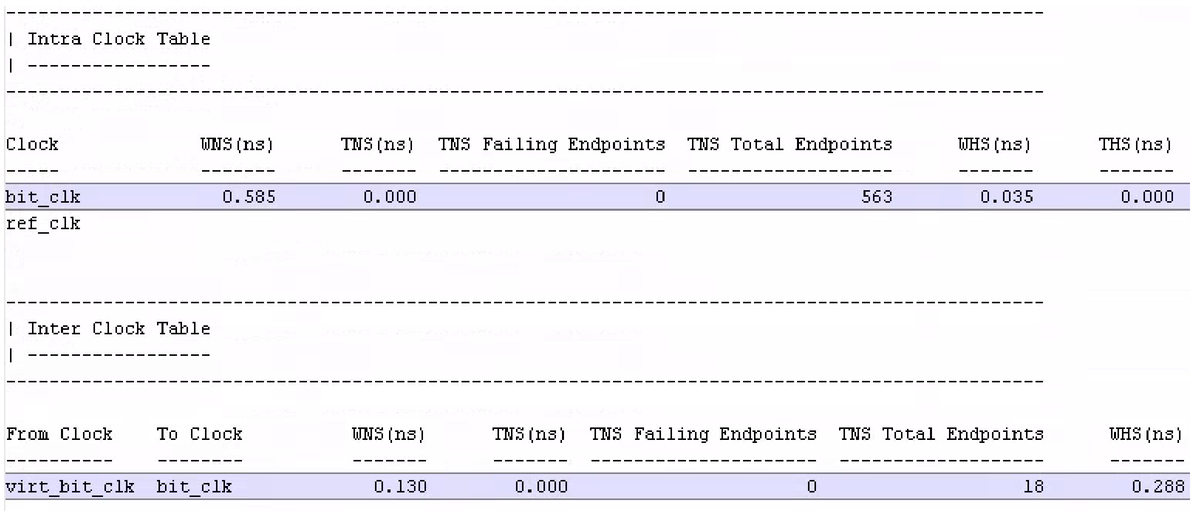
Notice how all intra-clock setup timing (the WNS column) in Figure 3.4c is much better with manual place and route than in Figure 3.4b without them, but that the inter-clock setup timing is slightly worse. Also, inter-clock hold timing (the WHS column) is significantly better with manual place and route, while intra-clock hold timing is nearly the same. Then, notice how the inter-clock path timing of the standalone design in 3.4a is very close to that of the inter-clock path of manual place and route of the full design in 3.4c. To summarize all of these comparisons, using manual place and route constraints improved timing of the shift register by forcing cells and nets to be located where they were in a more optimal design, and also enabled the IO timing in the full design to match the optimal design, (even though setup timing ended up being slightly worse).

At any rate, some variation between the implementation of individual IP and implementation of the integrated design is inevitable. However, using a manual place and route approach for the shift register has consistently produced the best results out of many implementation strategies, meeting timing by the best margins, especially with regard to hold time at the IO.

For more information about manual placement and routing techniques, see the Vivado implementation user guide [35].

3.3.8 Synthesis and Implementation Strategies

One final method to achieve timing improvements is to modify the synthesis and implementation strategies in Vivado. While the available options are endless, the following lists show a few of the changes we made to synthesis and implementation strategies. Note that these strategy modifications are not at all exhaustive, they are just options that were tried to improve timing.



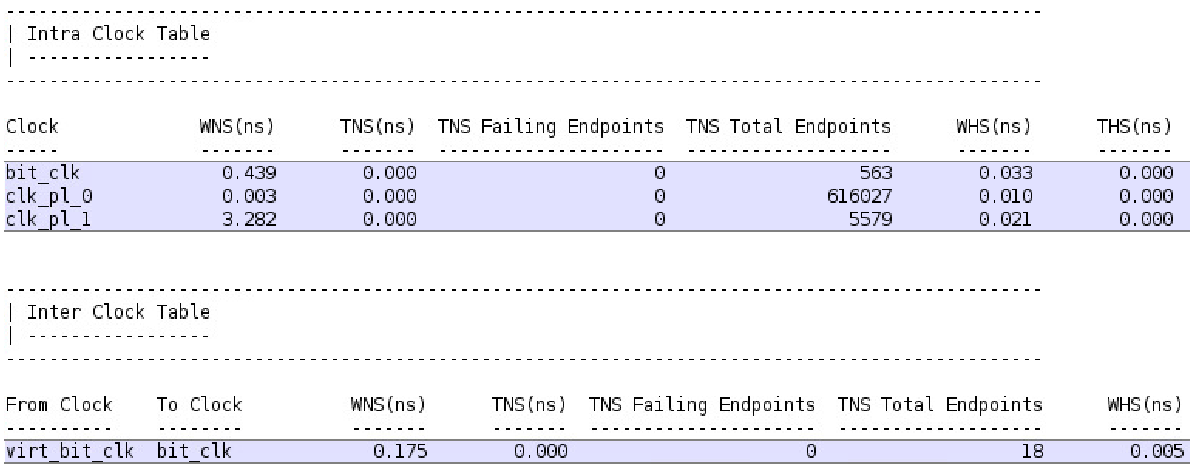
```
------------------------------------------------------------------------------------------------
| Intra Clock Table
| -----------------
------------------------------------------------------------------------------------------------

Clock             WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)
-----             -------      -------  ---------------------  -------------------      -------      -------
bit_clk             0.585        0.000                      0                  563        0.035        0.000
ref_clk


------------------------------------------------------------------------------------------------
| Inter Clock Table
| -----------------
------------------------------------------------------------------------------------------------

From Clock    To Clock          WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)
----------    --------          -------      -------  ---------------------  -------------------      -------
virt_bit_clk  bit_clk             0.130        0.000                      0                   18        0.288
```

(a) Shift register timing report—standalone shift register



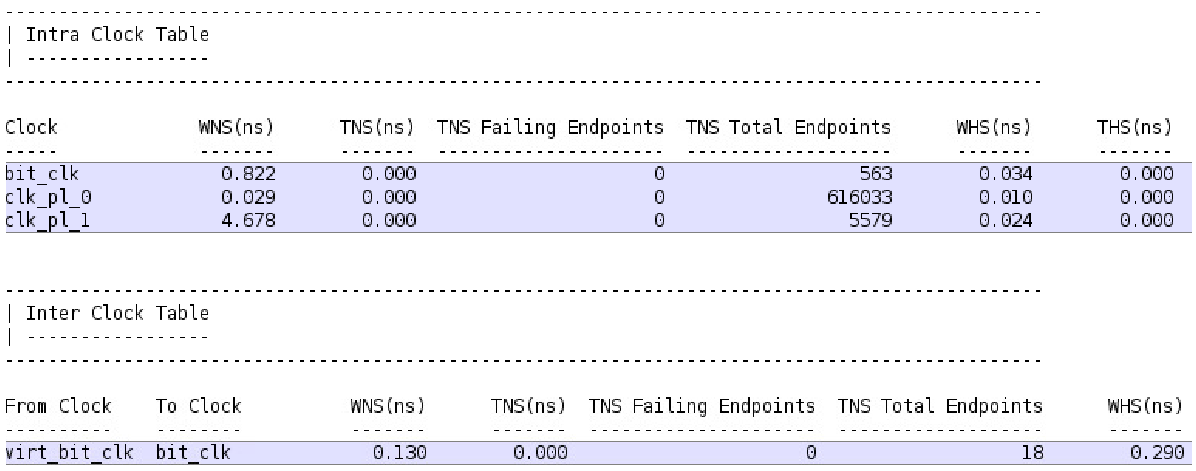
```
------------------------------------------------------------------------------------------------
| Intra Clock Table
| -----------------
------------------------------------------------------------------------------------------------

Clock             WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)
-----             -------      -------  ---------------------  -------------------      -------      -------
bit_clk             0.439        0.000                      0                  563        0.033        0.000
clk_pl_0            0.003        0.000                      0               616027        0.010        0.000
clk_pl_1            3.282        0.000                      0                 5579        0.021        0.000


------------------------------------------------------------------------------------------------
| Inter Clock Table
| -----------------
------------------------------------------------------------------------------------------------

From Clock    To Clock          WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)
----------    --------          -------      -------  ---------------------  -------------------      -------
virt_bit_clk  bit_clk             0.175        0.000                      0                   18        0.005
```

(b) Shift register timing report—full design, no manual place and route



```
------------------------------------------------------------------------------------------------
| Intra Clock Table
| -----------------
------------------------------------------------------------------------------------------------

Clock             WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)
-----             -------      -------  ---------------------  -------------------      -------      -------
bit_clk             0.822        0.000                      0                  563        0.034        0.000
clk_pl_0            0.029        0.000                      0               616033        0.010        0.000
clk_pl_1            4.678        0.000                      0                 5579        0.024        0.000


------------------------------------------------------------------------------------------------
| Inter Clock Table
| -----------------
------------------------------------------------------------------------------------------------

From Clock    To Clock          WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)
----------    --------          -------      -------  ---------------------  -------------------      -------
virt_bit_clk  bit_clk             0.130        0.000                      0                   18        0.290
```

(c) Shift register timing report—full design with manual place and route

Figure 3.4: Comparison of shift register timing reports with and without manual place and route constraints

In general, optimizations attempt to improve timing by spreading resource placement, balancing registers, replicating and grouping logic, and rerouting nets.

It is difficult to know what each synthesis or implementation option really does because the Xilinx documentation is sparse and algorithms are private, but our practice with the tools has shown some noticeable alteration in the final results in accordance with what the optimizations selected by the user proclaim to do. Some of this is obvious, like no LUT combining and limiting the fanout, as both of these increase the final number of logic and register resources used, while other optimizations are more subtle and difficult to verify, especially when they are treated as suggestions and might be ignored by the tools. The same goes for implementation attributes, which more directly affect timing results.

In general, it is best practice to use very small designs to test the behavior of a specific attribute and make sure that the results are as expected, though we found relatively good success by taking a more "educated brute force" approach and running 7 or 8 different implementation runs at the same time and observing which ones produce the best results.

Each item in the following lists corresponds with a specific optimization tag added to the synthesis or implementation TCL commands e.g. "-directive" and "-retiming". These tags focus the tool's efforts on optimizing for specific outcomes as described by their attribute written in italics.

Synthesis [36]

In HDL design, the synthesis process validates the hardware and prepares a netlist for implementation. During this step, resources are gathered and timing is estimated in reference to the board being targeted, and the logic in the design is effectually built out as a complete RTL block diagram. Because all of the resources are reviewed and the netlist is fully compiled in this step, Vivado allows the user to request the methodology and basic limitations for what the synthesis tools will look for and allow.

In our search for better timing, we tried many different synthesis options, of which the following were found to be helpful most consistently:

1. -directive *AlternateRoutability*—This directive is recommended in various forums and Xilinx answer posts as a way to help with high congestion and improve timing [34]. The Xilinx documentation about this strategy simply says, "set of algorithms to improve route-ability." This directive was used for the top level synthesis strategy as well as the synthesis strategy for many IP which had difficulty meeting timing in the implemented design.

2. -retiming <*on/off*>—This flag seeks to improve timing by balancing existing registers across chains of logic, and was added to synthesis for various Xilinx IP which have large logic trees with the hope that register balancing might improve critical paths.

3. -fanout_limit <*value*>—This integer value sets a fanout limit for the entire IP or project, for the same purpose as the MAX_FANOUT attribute but on a more global scale. In contrast with the attribute, this acts more as a suggestion to the tools, and may be ignored (such as for control signals). We applied this optimization to the synthesis runs of various individual IP (particularly those by Xilinx) when nets with a large fanout failed timing.

We found that the best global synthesis strategy disabled LUT combining and used the AlternateRoutability directive. For the synthesis of many individual IP, the fanout_limit was reduced from 10000 (the default) to around 32 or 64 when an IP's high-fanout nets failed timing, and the retiming attribute was often enabled. Occasionally, the shreg_min_size attribute was increased, as some sources say that increasing it slightly may help to better package registers, though this was more difficult to verify.

Implementation [35]

The implementation step of HDL development takes the resources and netlist provided by synthesis and goes through the process of optimizing logic, placing resources in the fabric, and then routing those resources together.

Implementation strategies, like synthesis strategies, were found to improve timing in some cases and make it worse in others, but in all cases it was equally difficult to know exactly why changes helped or did not. After trying many various directive combinations, we settled on a few that worked consistently well, from which this list was compiled.

1. Place Design

    (a) -directive *AltSpreadLogic_high*—This directive is the strongest directive of logic spreading available, which can improve congestion by moving logic further apart.

    (b) -directive *ExtraNetDelay_low*—This directive runs placement with more pessimistic delays associated with high fanout and long distance nets in an attempt to reduce net delay of critical paths.

    (c) -directive *ExtraNetDelay_high*—This functions just as the previous directive but is more pessimistic with delays.

    (d) -directive *ExtraPostPlacementOpt*—In Xilinx documentation, this directive simply states, "higher placer effort in post-placement optimization," which sounds nice so we tried it in some implementations.

    (e) -directive *ExtraTimingOpt*—This directive runs placement with a more timing-driven focus.

2. Post-Place Phys Opt Design

    (a) -directive *AlternateFlowWithRetiming*—This directive is more aggressive with register balancing, replication, and optimization.

    (b) -directive *AggressiveExplore*—This is the default directive for some pre-defined implementation strategies. It runs a variety of optimizations (see documentation [35]).

    (c) -directive *AlternateReplication*—This directive is more aggressive with cell replication on critical paths.

    (d) -directive *AggressiveFanoutOpt*—This directive is more aggressive with fanout reduction on critical paths.

3. Route Design

    (a) -directive *MoreGlobalIterations*—This directive runs more timing iterations to improve timing and uses more detailed analysis during all of routing.

    (b) -directive *HigherDelayCost*—This directive emphasizes delay optimization at the cost of fewer iterations.

(c) -directive *NoTimingRelaxation*—This directive doesn't allow the router to relax timing constraints to meet timing, thus forcing it to try harder to meet timing at the original constraints.

(d) -directive *Explore*—This directive tells the tools to try different critical path placement after an initial routing.

Variation Between Vivado Implementations

In later implementation runs we have noticed that Vivado doesn't always produce bitstreams that behave the same way. From this, we have learned that poor design strategy may lead the tools to make poor assumptions about logic and produce invalid retiming and logic reduction as described here.

The correlation block in our DSP architecture creates a massive fanout for the tready handshake signal if it is left unregistered. The bus width after correlation is nearly 10000 registers wide and 10 deep, and includes about 400 DSP slices. Originally, we replicated the AXIS complex multiplier for each multiply, resulting in 128 replications of the AXI4-Stream protocol as well, which created excessive control logic that needed to be reduced to provide the single interface control handshake. To handle this, we tried using "and" and "or" reduction of all tvalid signals produced by each complex multiplier, but this produced too much additional logic to consistently meet timing and was difficult to pipeline. To improve timing, we decided to use the handshake for a single multiplier (corresponding to the LSB of the data bus) to connect with the module interface, and then connect all of the tvalids together to create an error signal which was asserted if the handshakes were ever asynchronous (though by design they shouldn't be). It was after this change that we began to notice different Vivado implementation strategies producing different results, and small, unrelated changes to design logic began to produce bitstreams with unpredictable functionality. The curious thing was that running the same design through various implementation strategies began to produce different bitstream behavior: sometimes bitstreams would report synchronization issues in the correlator but they would still function correctly, other bitstreams would hang when trying to produce averager data and would produce averager and correlator errors, others still would seem to work, with or without errors, but would produce data that was incorrect.

This kind of unpredictability in a design, especially dependent upon synthesis tool output, is unacceptable. To resolve such issues, the best approach is to code HDL that can confidently withstand tool optimizations. This means it is unwise to use control signals which are not uniform across all logic they apply to, or to include modules with unnecessary logic that you assume will be left alone or modified by the tools in a particular way. Our reworking of the correlator involved using the raw complex multiplier template provided in Vivado which ended up significantly reducing LUTs and flip flops, as well as a few DSPs. This multiplier was replicated in the same fashion as the original AXIS multipliers, but now single handshake signals were connected to all registers of the design, and a skid buffer was added so that the tready signal could be registered for isolation and replicated to reduce fanout. In current runs, timing has continued to be tight, but implementation results have been consistent and without error assertion in bitstream logic.

In short, it is wise to code with as little ambiguity as possible to leave minimal room for the tools to assume the designer's intentions.

CHAPTER 4. INTEGRATED PLATFORM MANAGEMENT WITH PETALINUX

4.1 Introduction

In order to best integrate fabric preprocessing and target estimation postprocessing into the complete UAV traffic management system, we use the PetaLinux kernel [37] as a central management unit to tie everything together into a single user space. This way we can support the hardware configuration options of the HDL design from software and benefit from various operating system utilities such as the rootfs for local storage of postprocessing executables and dependencies, the FPGA manager for reloading bitstreams on the fly, and straightforward access of peripheral devices such as Ethernet and USB. The Ethernet connection specifically is used to send data packets and completed target estimates to platforms running DAA algorithms, as well as for remote access to processes running on the board.

Beyond the conveniences it provides, using the PetaLinux kernel isn't inherently essential to our UAV LATIS research, nor does it enable any significantly new functionality in such a system. However, as it has come to be an integral link between FPGA fabric and software, it is included here as a knowledge contribution for those who work with similar systems and may benefit from a proven starting point. For this reason, the information included is rather brief, though an extensive tutorial explaining the steps to configure and build kernel images is included in Appendix C. This appendix also provides the full device tree used to build the images.

Each section in this chapter touches on a key element of building and booting the kernel images in the same fashion that we use in our design. This includes such things as booting from the SD card, configuring the image, and building the device tree.

4.2 Booting from an SD Card

A coworker on this project, Mick Gardner, designed a carrier card for the UltraZed-EV board which contains ports for USB, Ethernet, and an SD card which are not present natively on the UltraZed. Without such a carrier card, only the local RAMs and flash chips could be used for software storage and development, and very minimal IO interfacing would be available, none of which is sufficient for our purposes.

With the added SD card port, the MPSoC block in Vivado can be configured to support an SD card driver and boot an image from SD memory, providing large, local, long-term storage. To boot PetaLinux from this memory, we followed the following steps:

1. Change the UltraZed's boot mode switches to on-off-on-off. These switches configure a hard-wired boot mode that the board will use to search for and load a kernel image.

2. Configure the PetaLinux project (in the main configuration menu) to boot from the SD device, and then change the boot device to mmcblk1p2 (default is 0p2).

3. Update the SD device node in the device tree to disable write protect and be sure that the node itself is enabled.

4. Build the image and package the associated BOOT.BIN.

5. Properly format the SD card into boot (FAT32) and rootfs (EXT4) partitions according to the PetaLinux documentation [37], and copy the images and rootfs to their proper partitions.

6. To ensure proper booting, listen on the USB port of the UltraZed to watch the boot messages when the board is powered on.

A properly booted kernel on the SD card will appear the same as the default flash images, but will allow files to be read and written from SD memory which persists when the board is powered off.

4.3 Including HDL Devices in the Device Tree

Because the PetaLinux tools are developed by Xilinx, one major benefit is that they can automatically import a Vivado hardware description file (HDF or XSA) and load fabric devices into autogenerated device trees. This is especially helpful (and important) when the fabric contains memory-mapped devices that communicate with the processor, as the processor needs to know that devices exist and what their addresses are. To import an HDF or XSA into PetaLinux, we followed the following steps:

1. In Vivado, export the HDF or XSA file (including the bitstream if desired) after successfully implementing a design.

2. Create a new PetaLinux project using the zynqMP template and then run the configuration command pointing to the hardware file as shown in the following two lines:

```
$ petalinux-create --type project --template zynqMP --name
   my_petalinux
$ petalinux-config --get-hw-description=<path-to-hdf-directory>
```

3. Configure and build the images. Notice that the pl.dtsi device tree file which is autogenerated will contain information about the memory-mapped devices from the hardware design.

If custom drivers have been designed, or changes must be made to existing devices, they can be included in the system-user.dtsi file which is created by the tools for user changes. In this file we added and changed several nodes according to the project's needs; this is described in the next section.

4.4 Device Tree Modifications and Peripheral Devices

As the project has developed, new peripheral devices have been added which require modifications to the device tree. One such example is the use of processor SPI engines, of which our project uses three: two to program the ADC chips, and one to program the chirp PLL. Including these in the image build requires the following steps:

1. Include and constrain the appropriate SPI signals in the processor block of the Vivado project. In our case, this meant all three SPI engines of the SPI0 device, the IO of which were connected to the FPGA ports that are routed to the correct devices on the carrier card.

2. Add three SPI nodes to the device tree. See the device tree in Appendix C for what these nodes look like. Important in this step is to ensure that the num-cs attribute was three, and that the is-decoded attribute is not present.

3. In the kernel configuration menu enable user-space control of the SPI engines by enabling the SPIDEV attribute. This can be done by searching for “spidev” and then enabling it as “built-in”.

4. After building the images and booting, the SPI devices can be accessed from where they are mounted in the /dev/ folder.

In general, the syntax for device nodes can be difficult to get just right, but if syntax is done correctly, adding and changing nodes is fairly straightforward. Preexisting nodes can be modified by referring to them from the system-user.dtsi file using the ‘&’ symbol, and new nodes can be added by giving them a unique name and populating the required fields (names which are already used will cause new the nodes to overwrite the existing ones). The only wholly new node that we added to the device tree is described in the next section.

4.5 Reserved Memory

Of the few configurations discussed here, this is the most optional, but is still recommended because it guarantees memory for the DMA engines which will not be inadvertently overwritten by kernel processes. This is done by simply adding a reserved memory block in the root node (the node denoted by “/{};”) of the device tree as shown in the following lines (which are the same as those in Appendix C):

```
reserved-memory {
    #address-cells = <2>;
    #size-cells = <2>;
    ranges;

    dma_mem_0: dma_mem@0x0F000000 {
        reg = <0x0 0x0F000000 0x0 0x01000000>; /* 16Mib ADC */
    };
    dma_mem_1: dma_mem@0x10000000 {
        reg = <0x0 0x10000000 0x0 0x01000000>; /* 16Mib FFT */
    };
    dma_mem_2: dma_mem@0x11000000 {
        reg = <0x0 0x11000000 0x0 0x08000000>; /* 128Mib AVG */
    };
};
```

In short, this node defines three separate memory blocks which are to be completely excluded from the kernel's available memory. Each block is given a base address and size which the kernel will not map as part of system RAM, but which the DMAs can use freely and the user can still read from.

CHAPTER 5. USER SPACE API FOR EFFICIENT FABRIC COMMUNICATION AND POSTPROCESSING

5.1 Introduction

Chapter 2 discusses work done on the FPGA fabric DSP chain, including its place in the end-to-end target estimation process for UAV LATIS. In order for this FPGA preprocessing to be useful, the user needs to know when and how to access packets of data that are sent to memory via the DMA engines, thus enabling further postprocessing in an efficient, real-time manner. Having made mention of PetaLinux in Chapter 4, the final missing link is how user space code which is running within the PetaLinux environment interacts with the FPGA fabric register space.

The fabric registers contained in the LRF Controller are based on the AXI4-Lite interface [38], which is memory-mapped into the processor address space and kernel device tree, allowing us to communicate with it by reading from and writing to that memory via user space code. When communication occurs, the kernel recognizes that the requested address range is reserved for the PL fabric and routes each request to the registers in the LRF Controller via the full power domain (FPD) memory bus (which is the M_AXI_HPM0_FPD in our case). Thus, fabric memory devices act like any other device in the device tree and can have custom driver code if desired.

As mentioned previously, the LRF Controller contains a variety of features including DSP configuration options, DMA management options, error monitoring reports, and status registers. All of this is available through the LRF Controller registers, of which there are 32, allowing us to develop simple API functions that the user can use to run target estimation postprocessing in software while simultaneously running preprocessing in the FPGA. This concurrent operation is essential for real-time radar, and requires that communication with the preprocessing registers and packet memory be fast and predictable. The postprocessing algorithms are limited by the time taken for preprocessing, which may repeat 500µs as described in the latency section of Chapter 2; thus, every cycle spent on fabric communication subtracts from the time allotted to the postprocessing algorithms. Thankfully, basic memory reads and writes (which make up the majority of fabric communication) are fairly fast and predictable, minimizing general communication and DMA interrupt monitoring times.

This chapter surveys a few of the many API functions to give examples of how the LRF Controller register space is managed from the software user's perspective. The first section explains the nature of the PS-to-PL interface, including the types of features offered, while the second section explains briefly how the API functions themselves are built to achieve maximum efficiency.

5.2 Fabric Register Space: Control/Status Interfaces

As was discussed briefly in Chapter 2, various IP are connected to the LRF Controller via CSI ports, or Control/Status Interface ports. This interface is a custom interface made in Vivado containing two required buses—control and status—and an optional third bus called the address bus (which is used primarily for the DMA controllers). Intuitively, the control bus is meant to send configuration commands from the LRF Controller to a given IP, while the status bus carries information back to the Controller. The address bus is configured with the same directionality as the control bus, as an output from the Controller. All three buses have 32 bits so as to correspond to the AXI4-Lite address space.

All IP which have extensive configuration or status options are given a CSI. Such IP include windowing, FFT, averaging, and all three DMA controllers. Other IP with fewer configuration and status signals, such as switches and decimation, have IO pins which are not part of a formal interface. Regardless of their interface, most control and status signals are connected into the LRF Controller register space so that the user can observe and interact with them via the user API in software.

Software API access can be divided into four basic categories: functional configuration options (such as enabling or disabling the window), process configuration options (PRI counters, decimation factor, averaging packets, etc.), updatable coefficients (FIR and windowing), and status reports (general status and error reporting). The following subsections describe each of these in more detail. For information about how each pin available to user space works, refer to Appendix B.

5.2.1 Functional Configuration Options

Functional configuration options enable and disable specific fabric functions. Some of these features include the run/stop bit of the LRF Controller which starts and stops automatic chirps and packet processing; the enable/disable bit of the window function (also called bypass mode) which will either window incoming data or pass it through unchanged; and the enable/disable bit corresponding to the mean subtraction IP, which turns the mean subtraction feature on and off.

These three are all user-controlled functional features, but there are also various derived functions, such as enabling a specific DMA controller to process data transfers. Except for in bit-accuracy mode, the DMA controllers are enabled and disabled depending on the mode selected by the user. If the user selects ADC mode, for example, the ADC DMA controller is enabled and the other two are disabled.

Some features are both user-controller and derived like the FFT half-frame mode. This option is always enabled in full DSP mode (only allowing the first 2048 words of a packet to be sent through to correlation), but is open for user-configuration when in FFT mode so that the user can observe the entire packet of FFT data if desired. In a sense, the run/stop bit in the LRF Controller also falls into this category as the Controller will only allow a single packet to run at a time when the Controller is in bit-accuracy mode, automatically deasserting the run bit after a single packet if the user asserts it.

Functional configuration features are the most basic control options of the LRF Controller and they are all configured using a single bit in the LRF Controller register space.

5.2.2 Process Configuration Options

Process configuration options differ from functional options in the sense that they describe *how* the functional features will work. Some of these options include the number of packets to be averaged in the averager, the mode of the LRF Controller, and the decimation factor.

For the most part, process configuration options can be configured at will, but they have lower priority than functional configuration options. Thus, some DMA process options may not take effect if requested while the DMA controller is disabled. Furthermore, process options have configuration limitations which must be observed for proper functionality. The max value of the

PRI counter in the LRF Controller, for example, cannot be set below a certain value to help protect against start signals which are too frequent to produce proper behavior. This max value must also be higher than the delay counter or the LRF Controller will not run. Other configuration signals, like the number of averaged packets, are designed to be used as an exponent or multiplier value, so sending the value ‘2’ to the averager will average four packets (2^2). This option uses three bits but can only support values from 0 to 6 (1 to 64 averaged packets), defaulting values outside of that range to 0. Similarly, the decimation factor in the window function acts as a multiplier, computing packet size as a multiple of 4096.

These special considerations make process options more challenging to configure correctly, and can easily result in undefined behavior if used incorrectly. For this reason, the documentation in Appendix B is provided. Furthermore, to correctly configure the PRI counter max value, tips from the latency section of Chapter 2 should be observed.

5.2.3 Updatable Coefficients

For maximum flexibility, the FIR Compiler and window function support runtime configurable coefficients. This means that the user can precompile sets of coefficients in preparation for a variety of different postprocessing algorithms and target estimation strategies.

The window function coefficients are stored in a BRAM block which is memory-mapped into kernel memory via the Xilinx BRAM Controller [39]. The coefficients are all 16-bit signed and the BRAM is configured as 32-bit memory space, such that the window function reads out a single 32-bit word every two clock cycles and performs an operation on every cycle. For size, we also assume that the window is symmetric and the state machine reads up to a given point in memory and then reads back down to make a full window. This way the user can send half of a window to the BRAM memory and obtain a full window operation for a variety of different packet sizes. Because the BRAM Controller is mapped separately from LRF Controller’s register space, it can be sent coefficients at will at any stage of operation (as long as the RAM is ready).

The FIR Compiler behaves differently from the window function, as it is a Xilinx IP [19] designed for minimal resource usage and simple interfacing. The coefficients are all sent into the Compiler via an AXI4-Stream interface, meaning that coefficients must be sent in the proper order and are not manually assigned to addresses. While this can be unintuitive, it simplifies the design and allows us to create a single 32-bit dedicated register in the LRF Controller which has an AXI4-Stream protocol layer to communicate with the FIR Compiler and transact coefficients. From user space, coefficients are transmitted by writing to the same LRF Controller register over and over until all coefficients have been sent. Sending these coefficients can technically be done at any time, however the FIR Compiler has been observed to go into an error state if the coefficients are sent or updated at the wrong time or in the wrong way.

5.2.4 Status Reports

The final variety of API signals is the broadest and most common—status signals. To make debugging easier, as well as for monitoring any run-time issues that may occur, extensive status signals are provided from most IP to the LRF Controller, and then from the LRF Controller to user space. The LRF Controller itself also generates a variety of unique status signals regarding the timing of packets and DMA transfers, and also offers some control signals that can reset status error reports. For clarity, the following list gives a brief survey (not exhaustive) of error and status reports provided to the user.

1. State machine states—Most IP with a state machine report which state they are in to the LRF Controller. This allows the Controller to know when packets can be started and when DMA transfers can be performed.

2. Channel synchronization errors—Because there are so many channels running in parallel, it can be helpful to ensure that all channel transactions are being performed synchronously, especially as the handshake for channels becomes independent at times (in the FFT, for example). The channel synchronization signal monitors transaction flags and reports their synchronicity.

3. DMA interrupts—One LRF Controller register for each DMA engine is dedicated as an interrupt. This register is set to all 1s within a few cycles of the DMA interrupt signal reporting a completed transfer. Using this method to monitor transfers has proven to be faster and more reliable than other interrupt techniques.

4. Counters—Various counters report the progress status of certain IP processes. The window function reports the internal word counter for each word that is windowed. The LRF Controller manages an interrupt counter for the averager DMA engine to report the number of AXI clock cycles it takes to perform the DMA transfer. And the DMA controllers all provide counters corresponding to the current link in the ring buffer that a transfer is being written to.

5. TLast unexpected or missing—As the DSP chain relies on 4096 samples being processed through to the end, guaranteeing the movement of complete packets is important for proper behavior. IP such as the FFT, FIR Compiler, and downsampler provide error signals to indicate that a packet was misaligned according to the TLast flag.

6. FFT arithmetic overflow etc.—Some IPs provide unique error signals depending on their operation, such as the FFT overflow flag which reports whether or not the FFT arithmetic has overflowed given the user's input scaling schedule. Other unique status examples include the BRAM busy signal from the window function, write collision from the averager, and error signals from the Xilinx DMA engines (see the documentation [29]), and the PLL lost-lock signal.

The only control feature associated with status signals is that some error signals can be reset. A global error reset signal clears all clearable errors in the LRF Controller so that the user can monitor which new errors occur in certain circumstances. Various errors signal, such as those regarding the TLast flag, can also be cleared directly by writing a zero to the asserted error bit.

5.3 Communicating with Fabric Registers

Having discussed what the fabric LRF Controller register space offers to the user, now we will look at how the software makes use of the various features. First, a note on the order of operations for configuration, then a discussion of how communication takes place.

5.3.1 Order of Operations

In a word, this section is a disclaimer for configuring the LRF Controller. While there are many valid ways to configure IP in the DSP chain, and many valid orders in which commands can be sent, not all configurations and orders will produce correct behavior. Because of the complexity of the design, it is difficult to perform exhaustive testing of every input in every order, and occasionally during testing we have seen errors which require a reset of the fabric logic or a complete reloading of the bitstream. Unfortunately, these errors are not always easy to repeat or debug, and resolving them is complicated by the unknown timing between when commands are sent from code and when they are actually written to fabric register space. The following list provides a few scenarios that have been seen to cause issues, and tips to avoid them:

1. Take care when reloading FIR coefficients. Hanging has occurred in the FIR Compiler where new coefficients are not accepted, usually requiring the bitstream to be reloaded to fix the Compiler. Because the FIR Compiler receives coefficients from a register in the LRF Controller, and because we don't know the exact architecture for how the FIR Coefficients are accepted and managed, it is difficult to understand exactly what causes the error. The vast majority of times that FIR errors have occurred have been when the coefficients were sent in code immediately following other register access. Providing some delay between other commands and setting the coefficients should help to avoid this problem.

2. Originally, changing LRF modes on the fly caused a host of issues because packets would still be running and DMAs wouldn't be able to finish transfers, but extensive efforts have gone into resolving this by waiting in hardware for packets to finish before making changes. Likewise, a variety of control logic checks were applied to prevent invalid configurations from causing issues while packets are being processed, such as when changing the decimation factor. However, care should be taken when changing any configurations and note that anticipating all possible configuration orders is difficult.

3. Should persistent errors occur, try resetting the fabric using the reset bit in the LRF Controller register 0. This is a soft reset which resets all control logic in the design and has proven effective in various error situations. If errors persist, reload the bitstream. If errors still persist, then try power cycling the board.

5.3.2 Communication Technique

Actually communicating with the register space is easy. In the Vivado address editor, each memory-mapped slave, such as the LRF Controller, is given a base address. This address is passed into PetaLinux via the HDF or XSA file to be loaded as a node in the device tree. From this base address, because each 32-bit register is technically 4, 8-bit addresses, writing to each register is the same as writing 32-bit words to addresses offset by multiples of 4 from the base address (0x0, 0x4, 0x8, 0xC, etc.).

Because the control and status registers are mostly compact groupings of several individual features, accessing specific features requires a mask. The majority of the user API functions are simply calls which read a register, mask out the corresponding bits for the requested feature, and then return those bits. If changes are made, the mask is used to change only the desired bits while leaving others alone and then writing the new value back into the register. As these operations appear frequently in the API, set and get functions were created to perform the writes and reads of select bits given a mask as follows:

```c
// Retrieve only particular bits of a register
uint32_t regGetBits(uint32_t* virtual_address, int offset, uint32_t mask)
{
  // Get current reg value
  uint32_t currentReg = regGet(virtual_address, offset);

  // Get trailing zero count of mask to know how much to shift
  int lowerBitOffset = getTrailingZeros(mask);

  // Extract value
  uint32_t value = (currentReg & mask) >> lowerBitOffset;

  return value;
}
```

```c
// Update only particular bits of a register
void regSetBits(uint32_t* virtual_address, int offset, uint32_t mask,
    uint32_t value)
{
  // Get current reg value, excluding the bits we want to change
  uint32_t currentReg = regGet(virtual_address, offset);
  uint32_t keepBits = currentReg & (~mask);

  // Get trailing zero count of mask to know how much to shift
  int lowerBitOffset = getTrailingZeros(mask);

  // Move the value to the proper location and re-mask in case of user
    mistake
  uint32_t bitsToSet = (value << lowerBitOffset) & mask;

  // Build new register value and send back to reg
  uint32_t newReg =  keepBits | bitsToSet;
  regSet(virtual_address, offset, newReg);
}
```

Both functions require the memory-mapped base address (the virtual address in this case), the particular register offset, the mask corresponding to the bits being read or written to, and in the case of a write, the value to be written.

These functions are the lowest level of API, directly writing to the LRF registers via the virtual address mapping. The second tier of functions, usually denoted by the prefix "lrf_module_" operate on user inputs and call one of these two base functions to perform an operation. The function to start or stop the LRF Controller, for example, accepts an input flag and sets the LRF run/stop bit accordingly:

```c
// Set the RUN bit
void lrf_module_run(bool doRun)
{
  uint32_t runValue = doRun ? 1 : 0;

  regSetBits(getLRFVirtAddr(), LRF_CTRLR_R0_LRF_CONTROL, LRF_RUN_MASK,
    runValue);
}
```

Above these "lrf_module" functions, more involved functions actually request and accept user input, check bounds, and then call lower level functions to perform actions. These functions can be fairly complicated, such as those which wait for DMA interrupts and then send data over the network, or relatively simple, such as setting the decimation factor shown here:

```c
// Set decimation factor from user input
void setDecimationFactor()
{
  printf("\n\n");
  printf("What decimation factor? (1-%d)\n", LRF_MAX_DEC_FACTOR);

  // Wait for user input
  uint32_t dec_factor = ui_readIntegerFromPrompt("Enter value: ", 10);

  // Check bounds and write dec factor if valid
  if(dec_factor < 1 || dec_factor > LRF_MAX_DEC_FACTOR)
    printf("Invalid decimation factor\n");
  else
  {
    lrf_module_win_setDecimationFactor(dec_factor);

    // Print new register value to verify
    printf("Window register updated: 0x%.8x\n", regGet(getLRFVirtAddr(),
   LRF_CTRLR_R6_WIN_CONTROL));
  }
}
```

To provide some scope for integration, these three levels of user API functions provide the foundation for postprocessing algorithms. The user can first configure the fabric and initiate packet preprocessing, and then run software postprocessing on data pulled from the reserved memory blocks where the DMAs transfer their data. Estimated targets can then be used locally or sent over the network for object detection and avoidance in UAV traffic systems.

CHAPTER 6. CONCLUSION

As we described in the introduction, local air traffic information systems are not currently suitable for low-altitude, small-scale UAV devices. They are expensive and large and poorly suited for the rapidly developing market of short-range airborne devices. Such devices are growing in number each year but traffic systems which ensure safe travel have not yet been implemented to match this growth.

Is it possible to ensure air safety for these devices? Yes. Can we adapt current LATIS platforms to small-scale operation? Yes. This thesis has sought to do so and the results are very promising.

Herein we have described the HDL processing chain for 2-dimensional UAV radar using 16 phased array input channels. We discussed the HDL design itself, including performance and contribution to UAV traffic systems, as well as resources created in conjunction with the processing system which describe how to interface with it. Specifically, Chapter 2 described each component of the design in detail with some instruction on how each is used and the effect it has on data processing. Chapter 3 discusses resource and timing constraints of the design and various strategies used to optimize and improve timing of the design. Chapter 4 contains some basic information regarding the PetaLinux platform developed for this project, including how it is used in conjunction with the design, and Chapter 5 describes how software running in the PetaLinux environment can interface with the FPGA fabric for effective postprocessing. Various appendices support each of these chapters, including XDC and device tree documents, a full PetaLinux tutorial, and the latest pinout definition for the LRF Controller.

We hope that this material may be of benefit for future research in UAV traffic management applications and offer this HDL development as a working proof of concept for low-cost, efficient digital signal processing for UAV traffic control radar. In recent field tests with airborne UAV devices the integrated design as produced promising results in target detection algorithms which are currently under development.

6.1 Contributions

This thesis and the UAV LATIS project it was completed for provide a new approach to UAV traffic management systems. The FPGA hardware in this design makes the following contributions:

1. Inexpensive
   (a) UltraZed-EV MPSoC provides a complete processing system with low cost
2. Optimized for small-scale LATIS
   (a) Efficient DSP chain processes 16 channels of radar data in real time to create 3-dimensional target estimates for local air traffic
   (b) Precise PRF as high as 2kHz to balance range and velocity estimates for local air traffic
   (c) Range resolution optimized for a variety of short-range UAV profiles
3. User-driven research
   (a) Extensive analysis of timing and resource improvements for designs with high congestion and utilization
   (b) PetaLinux information including a basic but complete PetaLinux tutorial describing its use in our design
   (c) Processing chain functions optimized for maximum user control and easy data management for postprocessing algorithms

6.2 Future Work

While much improvement has been made over the course of this project, the design is not yet complete. Anticipated future work is described in the following list:

1. Ensuring a robust fabric design against all configurations and implementation runs (including predictably met timing)
2. Form factor and packaging improvement
3. Power optimization
4. Extensive integrated outdoor testing with UAVs
5. Beamforming and target estimation (including Doppler processing) which matches real-time performance of fabric processing for large datasets
6. Exploration of alternate FPGA processing architectures for more efficient computation
7. Full DSP chain bit-accuracy testing and complete verification suite

REFERENCES

[1] L. O. Newmeyer, “Efficient FPGA SoC processing design for a small UAV radar,” 2018. ii, 2, 40

[2] C. Schwartz, T. Bryant, J. Cosgrove, G. Morse, and J. Noonan, “A radar for unmanned air vehicles,” *The Lincoln Laboratory Journal*, vol. 3, no. 1, pp. 119–143, 1990. 1

[3] Echodyne, “Radar for autonomous machines.” [Online]. Available: https://www.echodyne.com/autonomy/ 1

[4] A. Guerra, D. Dardari, and P. M. Djuric, “Dynamic radar networks of UAVs: A tutorial overview and tracking performance comparison with terrestrial radar networks,” *IEEE Vehicular Technology Magazine*, vol. 15, no. 2, pp. 113–120, 2020. 1

[5] M. Skowron, W. Chmielowiec, K. Glowacka, M. Krupa, and A. Srebro, “Sense and avoid for small unmanned aircraft systems: Research on methods and best practices,” *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, vol. 233, no. 16, pp. 6044–6062, 2019. 1

[6] M. Abramson, M. Refai, and C. Santiago, “The generic resolution advisor and conflict evaluator (GRACE) for detect-and-avoid (daa) systems,” in *17th AIAA Aviation Technology, Integration, and Operations Conference*, 2017, p. 4485. 1

[7] S. Siewert, M. Andalibi, S. Bruder, I. Gentilini, A. Dandupally, S. Gavvala, O. Prabhu, J. Buchholz, and D. Burklund, “Drone net, a passive instrument network driven by machine vision and machine learning to automate UAS traffic management,” *AUVSI Xpotential*, 2018. 1

[8] M. William L. and S. James A., *Principles of Modern Radar : Radar Applications, Volume 3.*, ser. Principles of Modern Radar. SciTech Publishing, 2014, no. Vol. III, Radar applications. 1, 2, 7

[9] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, “Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels,” in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. IEEE, 2019, pp. 1–8. 1, 15

[10] B. Betkaoui, D. B. Thomas, and W. Luk, “Comparing performance and energy efficiency of fpgas and gpus for high productivity computing,” in *2010 International Conference on Field-Programmable Technology*. IEEE, 2010, pp. 94–101. 1, 15

[11] S. M. S. Trimberger, “Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology: This paper reflects on how Moore’s Law has driven the design of FPGAs through

three epochs: the age of invention, the age of expansion, and the age of accumulation," *IEEE Solid-State Circuits Magazine*, vol. 10, no. 2, pp. 16–29, 2018. 1

[12] J. Saad, A. Baghdadi, and F. Bodereau, "FPGA-based radar signal processing for automotive driver assistance system," in *2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, 2009, pp. 196–199. 2

[13] Y. Wang, Q. Liu, and A. E. Fathy, "CW and pulse–doppler radar processing based on FPGA for human sensing applications," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 51, no. 5, pp. 3097–3107, 2013. 2

[14] J. Drozdowicz, M. Wielgo, P. Samczynski, K. Kulpa, J. Krzonkalla, M. Mordzonek, M. Bryl, and Z. Jakielaszek, "35 GHz FMCW drone detection system," in *2016 17th International Radar Symposium (IRS)*, 2016, pp. 1–4. 2

[15] "UltraScale architecture DSP slice user guide, UG579," September 2020. 2

[16] "Zynq UltraScale+ MPSoC data sheet: Overview, DS891," October 2019. 2

[17] B. D. Van Veen and K. M. Buckley, "Beamforming: A versatile approach to spatial filtering," *IEEE assp magazine*, vol. 5, no. 2, pp. 4–24, 1988. 7

[18] "AXI4-Stream infrastructure IP suite v3.0, PG085," December 2018. 11, 39

[19] "FIR Compiler v7.2, PG149," January 2021. 11, 25, 26, 67

[20] "Fast Fourier transform v9.1, PG109," January 2021. 11, 28

[21] "Vivado design suite user guide: Power analysis and optimization, UG907," May 2019. 14

[22] D. S. AD9257, "Octal, 14-bit, 40/65 msps, serial lvds, 1.8 v analog-to-digital converter." 16, 17

[23] "UltraScale architecture libraries guide, UG974," p. 354, October 2019. 16

[24] A. Oloffson, git repository, https://github.com/aolofsson/oh/blob/master/src/elink/hdl/erx_io.v. 18

[25] Xilinx, "Setting false path exceptions," Online video, https://www.xilinx.com/video/hardware/setting-false-path-exceptions.html. 18

[26] "Vivado design suite user guide—using constraints, UG903," December 2019. 18, 80

[27] A. AMBA, "AXI4-stream protocol specification," *Volume IHI A*, vol. 51, 4. 22, 38, 39

[28] "UltraRAM: Breakthrough embedded memory integration on UltraScale+ devices, WP477," June 2016. 30, 42

[29] "AXI DMA v7.1, PG021," pp. 30,37, June 2019. 35, 36, 69

[30] "UltraFast design methodology guide for the Vivado design suite, UG949," August 2020. 41

[31] “UltraFast design methodology timing closure quick reference guide, UG1292,” October 2019. 41, 51

[32] “Vivado design suite user guide: Design analysis and closure techniques UG906,” June 2018. 41

[33] Xilinx, Answer Record #62162, https://www.xilinx.com/support/answers/62162.html. 47

[34] ——, Answer Record #66314, https://www.xilinx.com/support/answers/66314.html. 49, 55

[35] “Vivado design suite user guide: Implementation, UG904,” pp. 73, 74, December 2019. 52, 55, 56

[36] “Vivado design suite user guide: Synthesis, UG901,” pp. 11, 12, January 2020. 54

[37] “PetaLinux tools documentation reference guide, UG1144,” May 2019. 59, 60

[38] A. ARM, “AMBA AXI and ACE protocol specification,” 2011. 64

[39] “AXI Block RAM (BRAM) Controller v4.1, PG078,” May 2019. 67

APPENDIX A. VIVADO XDC CONSTRAINTS

The constraints file provided here was used to supply constraints for the complete implemented design. These constraints have many notes to accompany them in the pages below, though additional information about Vivado XDC constraints in general can be found in Xilinx documentation [26]. Though the manual place and route constraints for the shift register were also part of the design, they were all automatically generated by Vivado and thus are not included here. To view these constraints, see the project's git repository.

```
##########################################################################################
##################################### CLOCK CONSTRAINTS ######################################
##########################################################################################
##### CREATE CLOCKS #####
# Create clocks for timing the shift register inputs---bit clock is 315MHz with data on the 630MHz edges
# word_clk is not generated as a clock, because it is never actually used as a clock (it is always just sampled by the bit_clk)
# The equivalent word clock is about 45MHz
# Note 1: this will give lots of warnings about creating one clock from two ports, which isn't physically possible, but it works
# just fine for timing and simulation so it doesn't matter. The point is to time the nets created by both input clocks the same way.
# Note 2: there are two clocks here---Vivado times all clocks together by default, so if they have the same attributes, they are
# effectively the same clock. We use two for the bit clock to time the incoming clock differently but together with the clock
# after it enters the FGA
# This clock represents the clock internal to the FPGA and is timed with the virt_bit_clk
create_clock -period 3.174 -name bit_clk -waveform {0.000 1.587} [get_ports {bit_clk_0_p_0 bit_clk_1_p_0}];
# This clock is what everything will be timed against, and represents the clock of the ADC itself
create_clock -period 3.174 -name virt_bit_clk -waveform {0.000 1.587};

##### GROUP CLOCKS #####
# Group clocks as asynchronous to skip timing analysis in both directions---see UG903
# Each group in this command is said to be asynchronous with the rest:
#   1. clk_pl_0 is asynch with everything else because the other clocks may change at runtime
#   2. clk_pl_1 is related to the bit clock and word clock in frequency, but we don't know the phase
#      relationship between them because the adc pll may produce some phase inconsistency
#   3. bit_clk can change at run time (we can change the phase), and is therefore also asynchronous to the other clocks
#
# NOTE0: The assumption with grouped clocks is that we are properly handling the synchronization and don't want the tools to
#        give us errors for clocks that we know are going to give us errors, but that we are actually handling properly
# NOTE1: This constraint may throw synthesis critical warnings on the pl clocks because internal IP constraints are not
#        processed together until implementation; see "report_compile_order -constraints" tcl command report
# NOTE2: This is apparently a "dangerous" command, which I would agree with, because it eliminates timing reports for places
#        where the clocks exchange data. However, the command supports different modes, and the asynchronous mode implies that the
#        clocks do interact physically, but that there is no known relationship between them. This is the case between the clk_pl_0
#        and clk_pl_1 because the latter will change at runtime. Dangerous, yes, but so is the nature of the design.
set_clock_groups -asynchronous -group {clk_pl_0} -group {clk_pl_1} -group {bit_clk virt_bit_clk};

##### BIT CLOCK FALSE PATHS #####
# Ignore timing between the rising and falling edges of the bit clocks
# We can ignore these paths because the reset signal in the iddre1 primitive (which is synchronized to rising edge of the bit clock
# but must reset both rising and falling edge registers) has asynchronous de-assertion, so we should be able to safely ignore timing
# errors.
# Falling to rising
set_false_path -fall_from [get_clocks bit_clk] -rise_to [get_clocks bit_clk];
# Rising to falling
set_false_path -rise_from [get_clocks bit_clk] -fall_to [get_clocks bit_clk];

##### INPUT DELAY AND TIMING #####
# These constraints are confusing but they appear to work with the following theory:
#   1. Because we have two clocks which are times together, by default data launched on the edge of one is clocked in on the next
#      edge of the other. But, these are the same clock, so we tell the tools that the data is clocked into the FPGA and then is to
#      be registered on the same edge. Thus the "set_multicycle_path 0" attribute for setup checks (which is 1 by default).
#   2. By nature, the hold checks work on the 0 edge of the clock, which is to say the N-1 edge where N corresponds to the edge
#      which clocks the data. This is because the hold checks measure whether or not the signal is held long enough before changing,
#      which is evalued on the launch edge of the data. So, to correspond correctly with the new setup check multicycle path of 0,
#      we must also make hold 1 less than that or -1.
#   3. For the setup checks, we ignore the opposite edge paths because as far as setup is concerned, data is clocked from each edge
#      to its corresponding same edge. This is because, even though the data is changing twice as often, we are still telling the
#      tools that the data is being clocked in on is launch edge.
#   4. For the hold checks, we ignore the same edge paths because the data actually is changing twice as often as each edge thinks,
#      and we are telling the tools that the data is to be analyzed as though it will be changing after each edge. (As far as I
#      understand it, anyway.)
#   5. Note 1: If the multicycle paths are different, the design still may meet timing, but the values will not be correct. For example,
#      using a hold multicycle path of 0 instead of -1 shows that hold timing is met by over 3 nano seconds. This doesn't make sense
#      because the date is changing twice as often as that. Instead, using the -1 value produces a timng margin of less than 1ns, which
#      is more what we would expect from a tight timing path.
#   6. Note 2: These notes are all just from my own understanding and interpretation of documentatin and forums. The way that Vivado
#      handles timing can be confusing and there is likely much more to the values used, and perhaps better ways to constrain things.
#      If any of this is incorrect, then it because information is sparce and timing analysis within the tools is nebulous.
# Resolve setup issues on false paths
set_multicycle_path 0 -from [get_clocks virt_bit_clk] -to [get_clocks bit_clk];
set_false_path -setup -rise_from [get_clocks virt_bit_clk] -fall_to [get_clocks bit_clk];
set_false_path -setup -fall_from [get_clocks virt_bit_clk] -rise_to [get_clocks bit_clk];
# Resolve hold issues on false paths
set_multicycle_path -1 -hold -from [get_clocks virt_bit_clk] -to [get_clocks bit_clk];
set_false_path -hold -rise_from [get_clocks virt_bit_clk] -rise_to [get_clocks bit_clk];
set_false_path -hold -fall_from [get_clocks virt_bit_clk] -fall_to [get_clocks bit_clk];

# These input delays were tuned in a different project with only the shift register which gave the IP more ideal placement than this
# project as it had no competition with other IP placement.
# Note: these min and max values imply that the data arrives just after the corresponding clock edge, however this is relatively
# arbitrary because we can adjust the clock phase relative to the data and technically clock bits on the opposite edge if desired.
# In reality, if the timing for the shift register is met, it is saying that with clock and data propagation, the data will be able to
# be clocked into the FPGA on the same edge that is "launching" it into the FPGA. This is why the phase for the ADC is so tight, because
# the timing margin suggests that for a clock phase of less than 0 or very close to 0 (relative to the data edge), timing will barely
# be met at 45MHz. The lesson is that if a larger phase range is desired, a delay block will have to be used because the natural FPGA
# net and logic delays will only allow a certain timing margin to be met as defined.
# Alternatively, having established that the data to clock edge relationship is arbitrary as far as timing is concerned, delaying the
# clockby 360 degrees would also technically produce the same timing results as produced by the timing tools, but a different edge would
# be clocking the data.
set_input_delay -clock [get_clocks virt_bit_clk] -min 0.12 [get_ports {bit_data_in_* word_clk_*}];
set_input_delay -clock [get_clocks virt_bit_clk] -max 0.25 [get_ports {bit_data_in_* word_clk_*}];
# falling edge input delay
set_input_delay -clock [get_clocks virt_bit_clk] -clock_fall -min -add_delay 0.12 [get_ports {bit_data_in_*}];
set_input_delay -clock [get_clocks virt_bit_clk] -clock_fall -max -add_delay 0.25 [get_ports {bit_data_in_*}];

##### CLOCK GROUPS #####
# Reduce skew between incoming data clocks---see UG912
set_property CLOCK_DELAY_GROUP shift_reg_group [get_nets -of_objects [get_ports {bit_clk_0_p_0 bit_clk_1_p_0}]];
# Reduce skew between outgoing adc clocks
set_property CLOCK_DELAY_GROUP adc_clock_group [get_nets -of_objects [get_ports {O_clk_p_0 O_clk_p_1}]];

##### TIMING NOTES #####
# The following notes describe cross clocking domains as errors arise. They may not reflect current errors because constraints will
# be changed as needed.
#   A. Intra-Clock paths are the critical timing issues because they mean that the delay is to long between registers on the same clock
#       0. Some failures with clk_pl_0 have been noted in the FIR compiler and the DMA engines; any failures with this clock must be resolved
#       1. There appear to be hold time errors on the bit_clock falling edge with the IDDRE1. Upon observation, the phase can be adjusted
#          in the ADCs to produce fully valid frames, but this may be an issue later. The interesting thing is that the violations are only
#          on the hold time of the falling edge (from what has been observed). This may be researched more later but for now will be ignored,
#          particularly because the input delays have been user-defined.
#   B. Inter-Clock paths require synchronization as clock domains are crossed, such as in the following noted timing error cases
#       1. Clock path "bit_clk to clk_pl_1" involves the domain crossing from the bit clock in the shift register (which clocks out new
#          words based on the frame clock) to the window function where data is clocked in on the pl_1 clock, which is the same frequency
#          as the word clock but with unknown phase. This crossing may not meet timing because the phase is unknown.
#             a. The issue is mitigated by allowing for an optional clock inversion in the shift register to clock on a different edge if
#                phase alignment is poor. Because the frequency of the word and pl_1 clocks are the same, this is a low severity issue.
#       2. Clock path "clk_pl_1 to bit_clk" involves the reset synchronization used for the IDDRE blocks. This path may fail timing
#          for the same reason as shown in the reverse direction for these clocks: the pl_1 clock and word clock don't have a known phase.
#             a. This issue is mitigated by synchronizing the reset signal with two registers on which the async_reg attribute has been set.
#       3. Clock path "word_clk to bit_clk" involves the domain crossing between the word clocks and the bit clocks in the shift register.
#          This crossing is really always going to be an issue because the phase can be adjusted and depends on the location of registers
#          and logic in the shift register. This fails timing because the word clock is sampled on the bit clock, but they technically are
#          both produced at the same time.
#             a. This issue is mitigated by the ability we have to adjust the phase of the bit clock relative to the word clock
#       4. The set_clock_groups removes timing warnings for clocks not grouped together. This means that some paths will not be reported
#          depending on how the clocks are currently being constrained.
#       5. The **async_default** errors are kind of odd, because the registers are constrained to have asynchronous resets, but then they
#         still seem to try to report timing errors. Looking at the timing report, the issue is caused by the fact that there are both
#         rising and falling edge registers in the IDDRE1 blocks, which means that synchronizing the reset to the rising edge, makes it
#         very difficult to meet timing on the falling edge register. In this design, the reset is synchronized on the positive edge, so
#         the falling edge warnings are valid, but there is not much that can be done to mitigate the timing errors. Timing will be VERY
#         hard to meet with a frequency requirement of 14*40 = 560MHz from rising to falling edge. In the IDDRE1 documentation, it says
#         that the reset pin is asyncronous with release synchronous to the clock, so either way, the reset functionality should be fine.

##########################################################################################
############################ ADC board pinout (FMC connector) ##################################
##########################################################################################
##### ADC DATA #####
# CHANNEL 1A
set_property -dict {PACKAGE_PIN AA16 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_0_p_0}];  # HP_DP_05_P
# CHANNEL 1B
set_property -dict {PACKAGE_PIN AC17 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_1_p_0}];  # HP_DP_04_P
# CHANNEL 1C
set_property -dict {PACKAGE_PIN AC16 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_2_p_0}];  # HP_DP_09_P
## CHANNEL 1D
set_property -dict {PACKAGE_PIN AG16 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_3_p_0}];  # HP_DP_08_P
## CHANNEL 1E
set_property -dict {PACKAGE_PIN AG13 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_4_p_0}];  # HP_DP_17_P
## CHANNEL 1F
set_property -dict {PACKAGE_PIN AK13 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_5_p_0}];  # HP_DP_16_P
## CHANNEL 1G
set_property -dict {PACKAGE_PIN AE14 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_6_p_0}];  # HP_DP_21_P
## CHANNEL 1H
set_property -dict {PACKAGE_PIN AC14 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_7_p_0}];  # HP_DP_20_P
## CHANNEL 2A
set_property -dict {PACKAGE_PIN AJ10 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_8_p_0}];  # HP_DP_25_P
## CHANNEL 2B
set_property -dict {PACKAGE_PIN AF10 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_9_p_0}];  # HP_DP_24_P
## CHANNEL 2C
set_property -dict {PACKAGE_PIN AK7 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_10_p_0}];  # HP_DP_29_P
## CHANNEL 2D
set_property -dict {PACKAGE_PIN AJ5 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_11_p_0}];  # HP_DP_28_P
## CHANNEL 2E
set_property -dict {PACKAGE_PIN AJ4 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_12_p_0}];  # HP_DP_37_P
## CHANNEL 2F
set_property -dict {PACKAGE_PIN AJ11 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_13_p_0}];  # HP_DP_36_P
## CHANNEL 2G
set_property -dict {PACKAGE_PIN AG11 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports {bit_data_in_14_p_0}];  # HP_DP_41_P
```

```
## CHANNEL 2H
set_property -dict {PACKAGE_PIN AH9 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports
{bit_data_in_15_p_0}];   # HP_DP_40_P

##### ADC FCO #####
# ADC0
set_property -dict {PACKAGE_PIN AD17 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports
word_clk_0_p_0];  # HP_DP_13_GC_P
# ADC1
set_property -dict {PACKAGE_PIN AH6 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports
word_clk_1_p_0];  # HP_DP_33_GC_P

##### ADC DCO #####
# ADC0
set_property -dict {PACKAGE_PIN AF16 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports
bit_clk_0_p_0];  # HP_DP_12_GC_P
# ADC1
set_property -dict {PACKAGE_PIN AG6 IOSTANDARD LVDS DIFF_TERM_ADV TERM_100} [get_ports
bit_clk_1_p_0];  # HP_DP_32_GC_P

##### SPI #####
set_property -dict {PACKAGE_PIN AH4 IOSTANDARD LVCMOS18} [get_ports emio_spi0_sclk_o_0];   #
HP_SE_05
set_property -dict {PACKAGE_PIN AG9 IOSTANDARD LVCMOS18} [get_ports emio_spi0_m_o_0];   # HP_SE_06
set_property -dict {PACKAGE_PIN AG19 IOSTANDARD LVCMOS18} [get_ports emio_spi0_ss_o_n_0];   #
HP_SE_01
set_property -dict {PACKAGE_PIN AC13 IOSTANDARD LVCMOS18} [get_ports emio_spi0_ss1_o_n_0];   #
HP_SE_02

##### PS to ADC Clocks #####
set_property -dict {PACKAGE_PIN AA14 IOSTANDARD LVDS} [get_ports O_clk_p_0];  # HP_DP_22_P
set_property -dict {PACKAGE_PIN AH12 IOSTANDARD LVDS} [get_ports O_clk_p_1];  # HP_DP_42_P

##### PLL #####
# Chirp
set_property -dict {PACKAGE_PIN C14 IOSTANDARD LVCMOS18} [get_ports pulse_out_0];  # HD_SE_14_P
# SPI
set_property -dict {PACKAGE_PIN G13 IOSTANDARD LVCMOS18} [get_ports emio_spi0_ss2_o_n_0];   #
HD_SE_18_GC_P
set_property -dict {PACKAGE_PIN G14 IOSTANDARD LVCMOS18} [get_ports emio_spi0_m_o_1];   #
HD_SE_15_N
set_property -dict {PACKAGE_PIN H14 IOSTANDARD LVCMOS18} [get_ports emio_spi0_sclk_o_1];   #
HD_SE_15_P
# PLL Lock
set_property -dict {PACKAGE_PIN F13 IOSTANDARD LVCMOS18} [get_ports PLL_lock_0];  # HD_SE_18_GC_N

##########################################################################################
#################################### PBLOCK Constraints #######################################
##########################################################################################
##### Shift Reg #####
create_pblock pblock_shift_reg;
add_cells_to_pblock [get_pblocks pblock_shift_reg] [get_cells -quiet [list design_1_i/shift_register_0]];
resize_pblock [get_pblocks pblock_shift_reg] -add {CLOCKREGION_X2Y0:CLOCKREGION_X2Y2};

###### LRF Controller #####
create_pblock pblock_lrf_ctrlr;
add_cells_to_pblock [get_pblocks pblock_lrf_ctrlr] [get_cells -quiet [list
design_1_i/lrf_controller/axi_lrf_controller_0]];

##### DMA 2 #####
create_pblock pblock_dma2;
add_cells_to_pblock [get_pblocks pblock_dma2] [get_cells -quiet [list
design_1_i/ps/dma_engines/dma_block_2/axi_dma_0]];
add_cells_to_pblock [get_pblocks pblock_dma2] [get_cells -quiet [list
design_1_i/ps/dma_engines/dma_block_2/axis_dwidth_converter_0]];

##### DMA 3 and Averager #####
create_pblock pblock_dma3;
add_cells_to_pblock [get_pblocks pblock_dma3] [get_cells -quiet [list
design_1_i/ps/dma_engines/dma_block_3/axi_dma_0]];
add_cells_to_pblock [get_pblocks pblock_dma3] [get_cells -quiet [list
design_1_i/ps/dma_engines/dma_block_3/axis_dwidth_converter_0]];
add_cells_to_pblock [get_pblocks pblock_dma3] [get_cells -quiet [list
design_1_i/correlator_and_averager/axis_packet_averager_0]];

##### Correlator #####
create_pblock pblock_cor;
add_cells_to_pblock [get_pblocks pblock_cor] [get_cells -quiet [list
design_1_i/correlator_and_averager/axis_correlator_0]];

##### SmartConnect #####
create_pblock pblock_smc;
add_cells_to_pblock [get_pblocks pblock_smc] [get_cells -quiet [list design_1_i/ps/axi_smc_1]];
```

APPENDIX B. LRF CONTROLLER PINOUT DEFINITION

The following pages are the full product guide for the LRF Controller. These pages describe the pinout definition for all LRF Controller registers, which includes all CSI ports in the design, as well as a variety of other configurable features. These notes describe what each pin is for, where it is found, which pins can be set by the user and/or hardware, and some guidelines for how each should be used.

```
---------------------------------------------------------------------------------------------------------
--------------------------------------IP DESCRIPTION --------------------------------------
---------------------------------------------------------------------------------------------------------
-- This IP is a controller for the second generation of LATIS Radar Firmware. It contains extensive control and system monitoring logic, and is designed to produce very consistently timed frames in the system it is controlling. Each of various processing IP is controlled from this central location and status signals from each of them is returned for observation. DMA transfers are handled in FPGA fabric and are triggered by this module to provide consistent and automatic high speed data transfer that can be observed, but not blocked, by the user.

-- In addition to the registers described below, this controller has FIR config and reload ports which can be connected directly into the FIR compiler corresponding ports. The FIR reload register will stream coefficients as they are written, and the appropriate bit in the LRF main control register will send the config command.

-- Non-CSI ports include:
    -- 1. Shift register clock invert which controls the polarity of the shift register clock edge for timing
    -- 2. FIR tlast error signals
    -- 3. An external channel sync error input which will trigger the same internal error signal when asserted if used, (this was made with the correlator in mind which doesn't have a CSI interface)
    -- 4. The start chirp signal which is pulsed every time the LRF controller state machine reaches 0
    -- 5. A LRF word clock input for cross clock timing of the start of each processing frame (should be connected to the shift register ref clock)
    -- 6. Various switch configuration signals, including enable signals and a mode signal, as well as a share control signal for DMA 2 in bit accuracy mode
    -- 7. A reset output which can be tied into the processor system resets aux port for resetting the whole design
    -- 8. FIR config and reload AXIS ports which can be connected directly into the FIR compiler's corresponding ports. The FIR reload register will stream coefficients as they are written, and the appropriate bit in the LRF main control register will send the config command to load new coefficients.
    -- 9. The begin packet option is the same signal as is sent through the CSI interface to the window, and is made external for the mean subtraction.
    -- 10. The enable mean subtraction port enables the mean subtraction IP. If left low, mean subtraction registers the input directly to output with latency of 1.
-- Generic options include:
    -- 1. Include external reset port
    -- 2. Interleaved averaging mode, which configures the FFT to allow blocking in full DSP mode
    -- 3. Include external channel sync port

-------------------------------------------------------------------------------------------------------------------------------
---------------------- LRF CONTROLLER CSI REGISTER PINOUT DEFINITIONS ----------------------
-------------------------------------------------------------------------------------------------------------------------------
-- The following comments describe the purpose of all pins in all registers in the LRF controller.
-- These registers make up the status and control signals for each functional custom IP in the Latis Radar Firmware project.
-----------------------------
-- Hierarchy Explanation:
-- Register_Group
    -- reg_ID_#: reg_name
        -- pin-#      pin name; pin-is-modifiable-by-user,pin-is-modifiable-by-controller;  pin explanation
-----------------------------
-- LRF
    -- 0: lrf module control register
        -- 0          lrf reset; 1,1; this pin is connected to an optional external pin to be sent to the proc_sys_reset block (toggled for 1 cycle)
        -- 1          run; 1,1; turn on or off the lrf module (this is reset after each frame in bit accuracy mode)
        -- 2-3        lrf modes; 1,0; 00-adc mode (only adc data output is enabled), 01-fft mode (only fft output is enabled), 10-full dsp mode (only final output is enabled), 11-bit accuracy mode (all dmas are enabled, note that non-blocking IP in this mode can be disasterous)
        -- 4          clear all errors; 1,1; set a 1 to this bit to clear bits error registers in status register (toggled for 1 cycle)
        -- 5          fir config; 1,1; reload fir coefficients after they have been sent (held high until accepted)
        -- 6          shift register clock invert; 1,0; invert the output clock from the shift register
        -- 7          start manual frame; 1,1; begin a single frame manually (goes to running state)
        -- 8          enable mean subtraction; 1,0; enable the mean subtraction IP before the window
        -- 9-31       '0'
    -- 1: lrf_module_status_reg
        -- 0          initializing; 0,1; lrf module (or internal IPs) are resetting and/or are not yet set up and ready to run (in setup_st)
```

```
            -- 1          idle; 0,1; lrf module is in idle state
            -- 2          running; 0,1; the lrf module is enabled and running
            -- 3-4        '0'
            -- 5          dsp chain complete error; 0,1; the averager dma didn't finish sending data before the averager tried to send the next frame, this can be cleared by user with bit 4 of control register
            -- 6          dma 1 tx error; 1,1; dma 1 sent more than 1 transfer during a frame, this can be cleared by user with bit 4 of control register or directly
            -- 7          dma 2 tx error; 1,1; dma 2 sent more than 1 transfer during a frame, this can be cleared by user with bit 4 of control register or directly
            -- 8          dma 3 tx error; 1,1; dma 3 sent more than 1 transfer during a frame, this can be cleared by user with bit 4 of control register or directly
            -- 9          lrf sync error; 0,1; timing or synchronization error between adc channels (when configured in such a way tha multiple channels are present), this can be cleared by user with bit 4 of control register
            -- 10         fir coeff error; 0,1; fir coefficient that was sent to controller was not accepted in time by fir compiler, this can be cleared by user with bit 4 of control register
            -- 11         fir reload tlast unexpected; 1,1; coefficient tlast was received early, this can be cleared by user with bit 4 of control register or directly
            -- 12         fir reload tlast missing; 1,1; coefficient tlast was not received on time, this can be cleared by user with bit 4 of control register or directly
            -- 13         fft overflow; 1,1; fft scaling has overflowed, this can be cleared by user with bit 4 of control register or directly
            -- 14         fft tlast unexpected; 1,1; tlast in fft arrived was received early, this can be cleared by user with bit 4 of control register or directly
            -- 15         fft tlast missing; 1,1; tlast in fft arrived was received late, this can be cleared by user with bit 4 of control register or directly
            -- 16         decimation tlast misaligned; 1,1; tlast in decimator is not aligned with the decimation counter, this can be cleared by user with bit 4 of control register or directly
            -- 17         averager write collision; 1,1; averager was sent data before it was ready (either according to SM or internal fifo), this can be cleared by user with bit 4 of control register or directly
            -- 18         pll lock error; 1,1; indicates whether or not the pll is locked (high means lock was deasserted), this can be cleared by user with bit 4 of control register or directly
            -- 19-31      '0'
        -- 2: lrf_chirp_counter_value_reg
            -- 0-31       lrf chirp counter value; 1,0; full number of clock cycles for a chirp, meaning that the counter will start the chirp on '0', count to the max value minus 1, roll over, and start another chirp on '0' with no latency between complete counts
        -- 3: lrf_chirp_counter_delay_value_reg
            -- 0-31       lrf chirp counter delay value; 1,0; number of clock cycles to delay between chirp and beginning of processing, meaning that '0' begins the frame on the same cycle that begins the chirp
    -- Empty Register
        -- 4-5: extra register
            -- 0-31       '0'
    -- Window
        -- 6: M_CSI_window_function_control
            -- 0          '0'
            -- 1          begin frame; 0,1; signals the start of a frame (aligns with the first word)
            -- 2          bypass mode; 1,0; enable bypass mode
            -- 3          enable luke shift; 1,0; enable flipping the sign bit functionality for offset data
            -- 4-9        '0'
            -- 10-26      data counter max value; 1,0; (data stream is zeros after this value)
            -- 27-31      dec factor; 1,0; (multiplied by frame size to get words per blocked frame)
        -- 7: M_CSI_window_function_status
            -- 0          idle; 0,1; window is in an idle state, no data is being sent, IP is ready
            -- 1          running; 0,1; bram is being read and output data is valid (while tvalid is high)
            -- 2          rstb busy; 0,1; bram is busy and should not be read
            -- 3          bypass mode; 0,1; setting this bit high will send the adc data through a buffer register and to the output without windowing it
            -- 4          sleep mode; 0,1; bram is asleep due to bypass mode (may include more use later)
            -- 5          waiting; 0,1; window function is in a wait state, either bram or the window counter value have not been set correctly
            -- 6          decimation tlast misaligned; 0,1; decimation counter is not aligned with the tlast signal
            -- 7-14       '0'
            -- 15-31      word counter; 0,1; counter of values as they are multiplied (counted at output)
    -- FFT
        -- 8: M_CSI_fft_control
            -- 0          '0'
            -- 1        config tvalid; 1,1; controller has valid configuration data for fft (this value is reset to 0 after configure transaction completes); only configures enabled ffts
            -- 2        master tready enable; 0,1; enable the master tready signal to allow the downstream to pause stream
            -- 3        half fft; 1,1; only produce half of the fft at the output (tvalid goes low, tlast is early)
            -- 4-19     '0'
            -- 20-31    scale schedule; 1,0; scale schedule for the ffts
        -- 9: M_CSI_fft_status
            -- 0        idle; 0,1; fft is in idle state, ready
            -- 1        config tready; 0,1; fft is ready for reconfigure
            -- 2        '0'
            -- 3        init; 0,1; fft is initializing, waiting to exit init state or waiting for ffts to become ready for data
            -- 4        channel sync error; 0,1; the functioning fft channels are not synchronized
            -- 5        event frame started; 0,1; a frame has begun to processm (start for dma)
            -- 6        fft running; 0,1; tied to m_tvalid of the ffts indicating a frame is complete and being transferred
            -- 7        event fft overflow; 0,1; indicates that the scaling in the fft blocks has overflowed
            -- 8        event tlast unexpected; 0,1; indicates that tlast entered the fft early
            -- 9        event tlast missing; 0,1; indicates that tlast entered the fft late
            -- 10-14    '0'
            -- 15-31    word counter; counter of output values
    -- Empty Registers
        -- 10-11: extra register
            -- 0-31     '0'
    -- Averager
        -- 12: M_CSI_averager_control
            -- 0        '0'
            -- 1-3      avg frames; 1,0; exponent number of frames to be averaged (2^avg_frames are averaged)
            -- 4-31     '0'
        -- 13: M_CSI_averager_status
            -- 0        idle; 0,1; IP is ready
            -- 1        done; 0,1; IP has completed averaging (pulse at start of done state)
            -- 2        load; 0,1; IP is loading data into fifos
            -- 3        send; 0,1; IP has finished processing and is sending data to down stream (high during whole state where data is being taken out)
            -- 4        channel sync error; 0,1; averaging fifos aren't syncronized at output (m_tvalid is not syncronized across fifos)
            -- 5        write collision; 0,1; a write was attempted on the fifos when they weren't ready to receive data
            -- 6-17     '0'
            -- 18-31    counter; 0,1; counter of output data as it is being sent
    -- FIR_reload
            -- 14: FIR_Coefficients
            -- 0-15     fir coefficients; 1,0; user sends coefficients to this address, which are then sent to the fir compiler (sending sequentially to this register, will cause each coefficient to be sent out sequentially)
            -- 16-30    fir coefficient ID; 1,0; user sends a coefficient id with the coefficient for tx verification
            -- 31       fir coefficient tlast; 1,0; the user must associate this with the last coefficient sent to the fir compiler (the last coefficient expected by the compiler)
    -- DMA 1 Controller
        -- 15: M_CSI_dma_ctrlr_1_control
            -- 0        disable dma; 0,1; tell the controller to put the dma in a disable state
            -- 1        start dma; 0,1; tell the controller arm the dma to send data
            -- 2        ring buffer; 1,0; run the controller as a ring buffer
            -- 3        new destination; 1,1; tell the controller to load a new destination address into the dma
            -- 4-9      max link index; 1,0; maximum index that the ring buffer will count to (0 to 63 possible, for max ring buffer of size 64)
            -- 10-31    link size; 1,0; size of each link in the ring buffer in bytes
        -- 16: M_CSI_dma_ctrlr_1_status
            --- DMA register space status signals (see PG021) ---
            -- 0        dma halted; 0,1
            -- 1        dma idle; 0,1
            -- 2        dma reserved (0); 0,1
            -- 3        dma SG included (should be 0); 0,1
            -- 4        dma internal error; 0,1
            -- 5        dma slave error; 0,1
            -- 6        dma decode error; 0,1
            -- 7        dma reserved (0); 0,1
            -- 8-10     dma SG errors (not used; should be 0); 0,1
            -- 11       dma reserved (0); 0,1
            -- 12        dma interrupt on complete; 0,1
            -- 13        dma interrupt on delay (only for scatter gather; not used); 0,1
            -- 14        dma interrupt on error (present but never cleared by this controller); 0,1
            -- 15        dma reserved (0); 0,1
            --- User status signals ---
            -- 16-19     '0'
            -- 20-25     current link; 0,1; indicates current link of the ring buffer being written to and incremented at the end of each frame, this is reset to 0 when the new destination flag is set high
            -- 26        dma disable; 0,1; controller is disabled
            -- 27        dma resetting; 0,1; controller is in reset and dma is in the process of a graceful reset
            -- 28        dma idle; 0,1; controller is in idle state (waiting for start signal)
            -- 29        dma tx running; 0,1; controller is running a transfer
            -- 30        dma tx complete; 0,1; controller has completed a transfer
            -- 31        dma clearing irq; 0,1; controller is clearing the irq flag
        -- 17: M_CSI_dma_ctrlr_1_dest_address
            -- 0-31      destination address; 1,0; optional alternate destination address if desired (when 0, default is used)
    -- DMA 2 Controller
        -- 18-20: See DMA 1 Controller
    -- DMA 3 Controller
        -- 21-23: See DMA 1 Controller
    -- DMA 3 Counter
        -- 24: DMA 3 TX Counter
            -- 0-31      counter; 0,1; counter which begins incrementing from zero when the third dma begins a transfer and holds its value when the dma completes the transfer
    -- DMA Interrupt Registers
        -- 25: DMA 1 Interrupt Register
            -- 0-31      interrupt register; 1,1; set to all 1s when interrupt occurs, reset by user to all zeros
        -- 26: DMA 2 Interrupt Register
            -- 0-31      interrupt register; 1,1; set to all 1s when interrupt occurs, reset by user to all zeros
        -- 27: DMA 3 Interrupt Register
            -- 0-31      interrupt register; 1,1; set to all 1s when interrupt occurs, reset by user to all zeros
    -- Processing Counter
        -- 28: DSP Processing Counter
            -- 0-31      counter difference; 0,1; reports the difference between the counter delay value and the counter itself when the DSP dma is started (avg done flag is asserted)
    -- Empty Registers
        -- 29-31: extra registers
            -- 0-31      '0'
```

APPENDIX C.   PETALINUX HELPS

As described in Chapter 4, the following pages contain the custom device tree used to build this project's PetaLinux kernel images, as well as the complete PDF tutorial and troubleshooting information developed as a starting point for building the images.

C.1 PetaLinux Device Tree

The user device tree (system-user.dtsi) is initially autogenerated by the PetaLinux tools as an empty file where the user can add their own device nodes and overrides. A few of these nodes, such as gem3, i2c1, qspi, and dwc3_0 were copied from the board support package (BSP) system-user.dtsi file provided by Avnet. Other nodes were either created new for our purposes or were modified from the original BSP device tree.

```
/include/ "system-conf.dtsi"
/ {
    /* everything in this chosen node except for "uio_pdrv...uio" is stuff
    that normally petalinux will add to the boot args, but this node overwrites
    all of those so we have to add them again here as well */
    chosen {
        bootargs = "earlycon console=ttyPS0,115200 clk_ignore_unused root=/dev/mmcblk1p2 rw rootwait
        isolcpus=3";
        stdout-path = "serial0:115200n8";
    };

    reserved-memory {
        #address-cells = <2>;
        #size-cells = <2>;
        ranges;

        dma_mem_0: dma_mem@0x0F000000 {
            reg = <0x0 0x0F000000 0x0 0x01000000>; /* 16Mib ADC */
        };
        dma_mem_1: dma_mem@0x10000000 {
            reg = <0x0 0x10000000 0x0 0x01000000>; /* 16Mib FFT */
        };
        dma_mem_2: dma_mem@0x11000000 {
            reg = <0x0 0x11000000 0x0 0x08000000>; /* 128Mib AVG */
        };
    };
};

&gem3 {
    status = "okay";
    local-mac-address = [00 0a 35 00 02 90];
    phy-mode = "rgmii-id";
    phy-handle = <&phy0>;
    phy0: phy@0 {
        reg = <0x0>;
        ti,rx-internal-delay = <0x5>;
        ti,tx-internal-delay = <0x5>;
        ti,fifo-depth = <0x1>;
    };
};

&i2c1 {
    status = "okay";
    clock-frequency = <400000>;

    i2cswitch@70 { /* U7 on UZ3EG SOM, U8 on UZ7EV SOM */
        compatible = "nxp,pca9542";
        #address-cells = <1>;
        #size-cells = <0>;
        reg = <0x70>;
        i2c@0 { /* i2c mw 70 0 1 */
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <0>;
            /* Ethernet MAC ID EEPROM */
            mac_eeprom@51 { /* U5 on UZ3EG IOCC and U7 on the UZ7EV EVCC */
                compatible = "at,24c08";
                reg = <0x51>;
            };
            /* CLOCK2 CONFIG EEPROM */
            clock_eeprom@52 { /* U5 on the UZ7EV EVCC */
                compatible = "at,24c08";
```

```
                    reg = <0x52>;
                };
            };
        };
};

&qspi {
        #address-cells = <1>;
        #size-cells = <0>;
        status = "okay";
        is-dual = <1>; /* Set for dual-parallel QSPI config */
        num-cs = <2>;
        xlnx,fb-clk = <0x1>;
        flash0: flash@0 {
          /* The Flash described below doesn't match our board ("micron,n25qu256a"), but is needed */
          /* so the Flash MTD partitions are correctly identified in /proc/mtd */
            compatible = "micron,m25p80"; /* 32MB */
            #address-cells = <1>;
            #size-cells = <1>;
            reg = <0x0>;
            spi-tx-bus-width = <1>;
            spi-rx-bus-width = <4>; /* FIXME also DUAL configuration possible */
            spi-max-frequency = <108000000>; /* Set to 108000000 Based on DC1 spec */
        };
};

/* SD0 eMMC, 8-bit wide data bus */
&sdhci0 {
        status = "okay";
        bus-width = <8>;
        max-frequency = <50000000>;
        disable-wp;
};

/* SD1 with level shifter */
&sdhci1 {
        status = "okay";
        max-frequency = <50000000>;
        no-1-8-v; /* for 1.0 silicon */
        disable-wp;
};

/* ULPI SMSC USB3320 */
&usb0 {
        status = "okay";
};

&dwc3_0 {
        status = "okay";
        dr_mode = "host";
        phy-names = "usb3-phy";
       snps,usb3_lpm_capable;
       phys = <&lane2 4 0 2 52000000>;
};

&spi0 {
        num-cs = <3>;
        status = "okay";
        spidev@0x00 {
            compatible = "spidev";
            spi-max-frequency = <50000000>;
            reg = <0>;
        };
    spidev@0x01 {
        compatible = "spidev";
        spi-max-frequency = <50000000>;
        reg = <1>;
        };
    spidev@0x02 {
        compatible = "spidev";
        spi-max-frequency = <50000000>;
        reg = <2>;
        };
    };

```

C.2 PetaLinux Tutorial

This tutorial was started as part of classwork and then expanded as part of research to explain how to begin and see PetaLinux image development through to completion. It contains a variety of debugging tips and explains how to configure various features that we included in our design.

The following tutorial pages are attached as a PDF. Depending on the format of the attachment, the links may or may not work. However, the original tutorial can be found in the project's git repository.

UltraZed-7EV PetaLinux Tutorial

Kacen Moody
BYU Electrical and Computer Engineering

September 3, 2020

Purpose

The goal of this tutorial is to build a PetaLinux 2019.2 image for the UltraZed-EV board (specifically, the UltraScale+ MPSoC using the Xilinx XCZU7EV-FBVB900 FPGA), including some advanced image configuration features. Expected outcomes are the following:

- Create a working PetaLinux image and its corresponding boot image, to be booted from an SD card, and accessed via UART (through a micro USB port) and SSH
- Learn to navigate the PetaLinux tools, including various configuration features

Requirements

- Installed PetaLinux 2019.2 tools (page 9, UG1144)
- Installed Vivado 2019.2 (or PetaLinux 2019.2 compatible version) with UltraZed-7EV board files
- Correctly formatted SD card (page 65, UG1144)
- UltraZed-EV board with a carrier card that includes an SD reader, Ethernet, and micro USB
- Basic understanding of Linux command line

Contents

1 Building Simple PetaLinux Images

1.1 Set Up the Environment

First, **the PetaLinux environment must be set up**. This is done by running a script in the command line and must be done every time a new terminal window is opened to use the tools. For a bash shell, the command is as follows: (page 13, UG1144):

```
$ source <path-to-installed-PetaLinux>/settings.sh
```

1.2 Create a Project

Next, we can **create and initialize a project using the petalinux-create command** (page 20, UG1144). The tools will create a root directory and populate it with various configuration and build files, so this command should be executed in the directory where you want the project's root folder to reside.

```
$ petalinux-create --type project --template zynqMP --name my_petalinux
```

Template options are zynqMP, zynq, or microblaze, and as the UltraZed-EV is an MPSoC, the zynqMP is appropriate in this case.

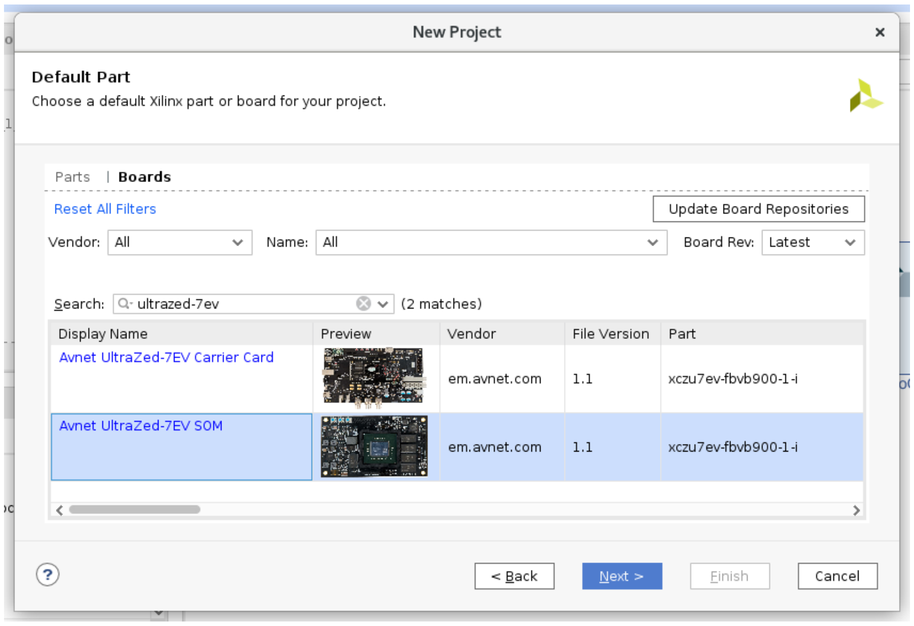
Alternatively, a project can be created using a board support package (BSP) produced by Avnet, but that will not be discussed here. Refer to UG1144, page 16.

1.3 Importing Hardware and Configuring the Image

With the project initialized, go into the project's root directory in the terminal. The next step is to configure the image in accordance with the hardware present on the UltraZed-EV. Implied here is that while the template we used in the previous step will set up the framework for the general zynqMP architecture, the PetaLinux tools must have an idea about how the processor system is to be configured on whichever MPSoC the project is targeting. To do this we must **build a Vivado project and export either the HDF or XSA file**, which will then be imported into the PetaLinux tools. Below, we will describe the steps for building a very slimmed down project containing just the ZynqMP processor. If you have already built a Vivado project, skip ahead to number 8.

1. Start Vivado 2019.2

2. Under "Quick Start" **click on "Create Project" and set up the project using the defaults** without any additional source files (it doesn't matter what the name or location of the project are). When you arrive at the "Default Part" page, **select either the "UltraZed-7EV SOM" or "UltraZed-7EV Carrier Card" board** as shown in Figure 1, choosing whichever option corresponds with what you are targeting (visit the "Using a Custom Carrier Card" section for more information on which board to use). This way, the UltraZed board files can populate the project constraints using existing board peripherals if necessary. If these boards are not present in the list, you will likely need to install the UltraZed board files into Vivado before proceeding.


New Project

Default Part

Choose a default Xilinx part or board for your project.

Parts | **Boards**

Reset All Filters — Update Board Repositories

Vendor: All — Name: All — Board Rev: Latest

Search: ultrazed-7ev (2 matches)

Display Name	Preview	Vendor	File Version	Part
Avnet UltraZed-7EV Carrier Card		em.avnet.com	1.1	xczu7ev-fbvb900-1-i
Avnet UltraZed-7EV SOM		em.avnet.com	1.1	xczu7ev-fbvb900-1-i

Figure 1

3. After finishing the setup and waiting for Vivado to initialize the project, **click on "Create Block Design"** under "IP Integrator" on the left.

4. In the center of the new window that opens, **click on the plus sign to add an IP, and select "Zynq Ultrascale+ MPSoC."**

5. When the IP has been added, **click "Run Block Automation"** in the green strip above it and then click "OK." This will reduce the number of IO pins visible on the block but will populate it with typical features as shown in Figure 2.

   Notice that UART and SD are both enabled. On the block in the block diagram, there will only be the pl_resetn0 and pl_clk0 ports remaining as shown in Figure 3, which we will leave unconnected.

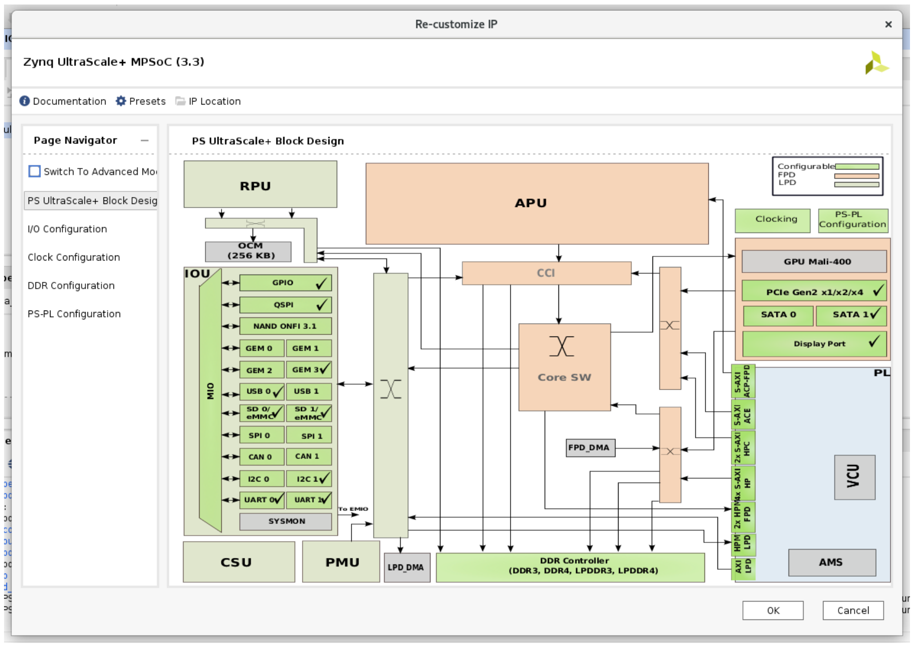
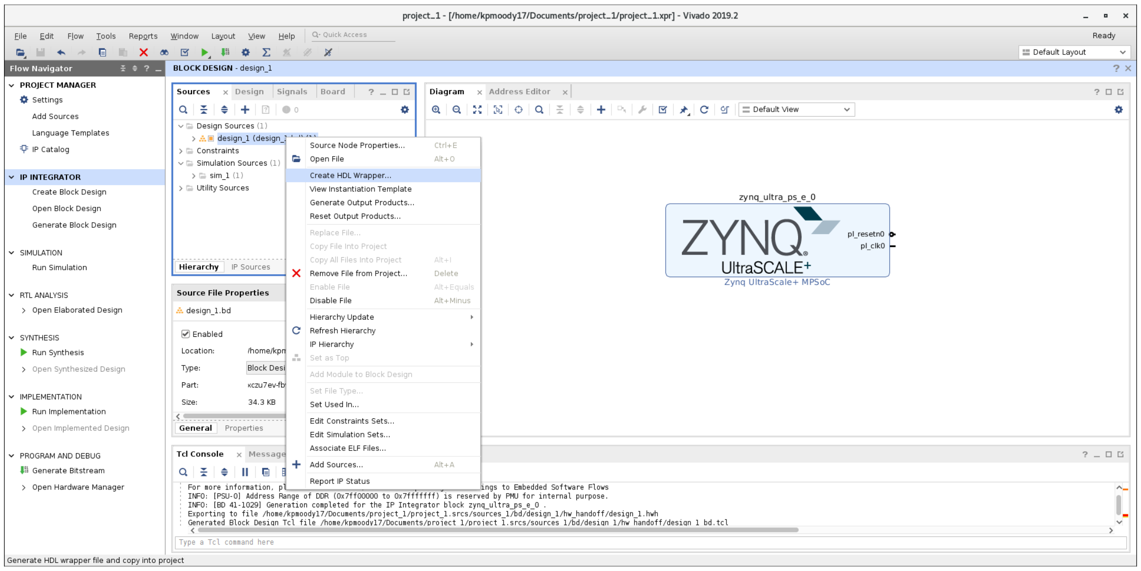


Figure 2

6. Next, in order to build the project, Vivado needs the block diagram to have a top level wrapper module, so in the upper left pane, **click on "Sources" and then right click on "design_1"** next to the little orange symbols. From the menu, **click "Create HDL Wrapper..."** and then click "OK." This is shown in Figure 3.



Source File Properties: design_1.bd — Enabled; Location: /home/kpm; Type: Block Desi; Part: xczu7ev-fb; Size: 34.3 KB

Tcl Console:

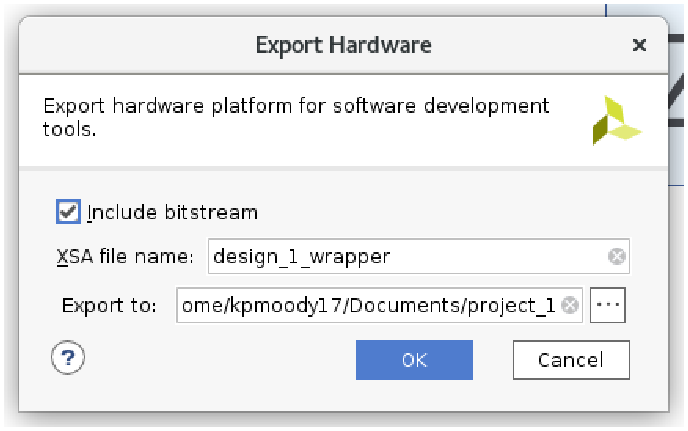
```
For more information, pl                                  ings to Embedded Software Flows
INFO: [PSU-0] Address Range of DDR (0x7ff00000 to 0x7fffffff) is reserved by PMU for internal purpose.
INFO: [BD 41-1029] Generation completed for the IP Integrator block zynq_ultra_ps_e_0 .
Exporting to file /home/kpmoody17/Documents/project_1/project_1.srcs/sources_1/bd/design_1/hw_handoff/design_1.hwh
Generated Block Design Tcl file /home/kpmoody17/Documents/project_1/project_1.srcs/sources_1/bd/design_1/hw_handoff/design_1_bd.tcl
```

Figure 3

7. This is all for the processor so **click "Generate Bitstream"** below "Program and Debug" in the left pane. Click "Yes" if asked to save the project and then choose to launch runs first as well. It may take a while for the project to compile.

8. After the bitstream has been compiled, click on "File" in the upper left of the GUI and select

"Export Hardware..." from the "Export" option toward the bottom of the drop down menu. In the window that comes up, check the "Include bitstream" box as shown in Figure 4 and click "OK." This step will **create the HDF or XSA file** that we want. (Vivado 2019.2 will create an XSA by default, while earlier versions of Vivado create an HDF.) Take note of where the output will be saved.



Export hardware platform for software development tools.

[✓] Include bitstream

XSA file name: design_1_wrapper

Export to: ome/kpmoody17/Documents/project_1

Figure 4

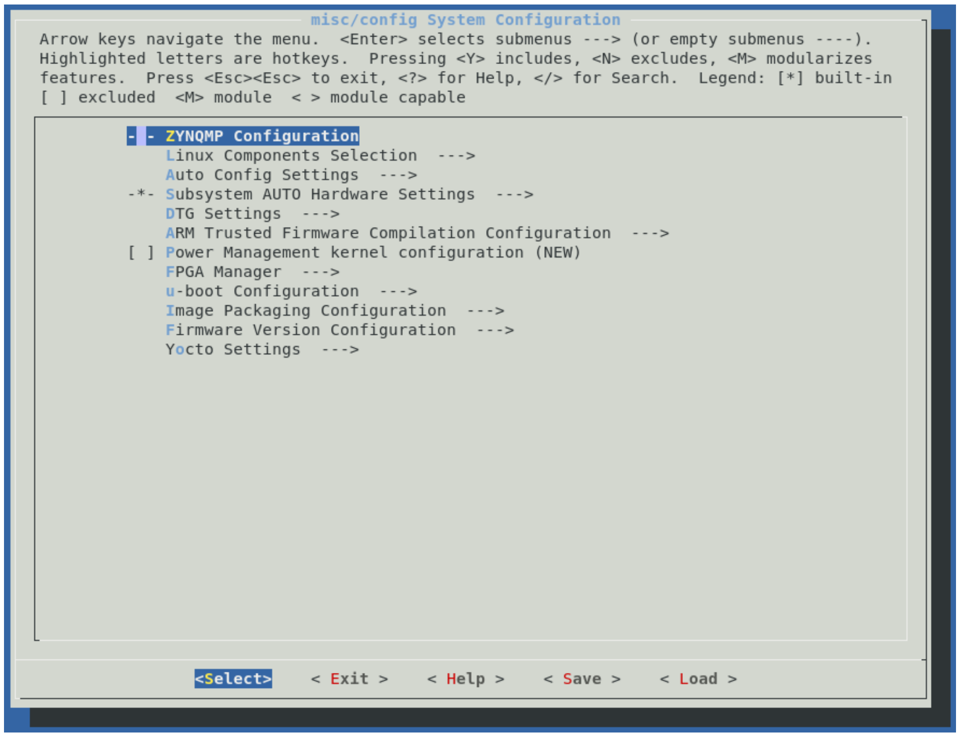
9. In the file system, navigate to the save location of the exported hardware. By default, the file will be called design_1_wrapper.xsa (or .hdf). **Copy this file into the root directory of the PetaLinux project we have already created.**

10. We can now **import the exported hardware file into the project** using the petalinux-config command in the terminal as follows (page 23, UG1144):

```
$ petalinux-config --get-hw-description=.
```

This command will start the configuration process by extracting information from the hardware file.

Notice that the command searches within a directory (specified in this case by the period after the equals sign) and does not look for a specific file name. Don't include the name of the hardware file in this command, just give the path of the directory where the hardware file is located. After importing the file, the command line GUI shown in Figure 5 will open.



```
misc/config System Configuration
Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in
[ ] excluded  <M> module  < > module capable

         -*- ZYNQMP Configuration
             Linux Components Selection  --->
             Auto Config Settings  --->
         -*- Subsystem AUTO Hardware Settings  --->
             DTG Settings  --->
             ARM Trusted Firmware Compilation Configuration  --->
         [ ] Power Management kernel configuration (NEW)
             FPGA Manager  --->
             u-boot Configuration  --->
             Image Packaging Configuration  --->
             Firmware Version Configuration  --->
             Yocto Settings  --->

              <Select>    < Exit >    < Help >    < Save >    < Load >
```

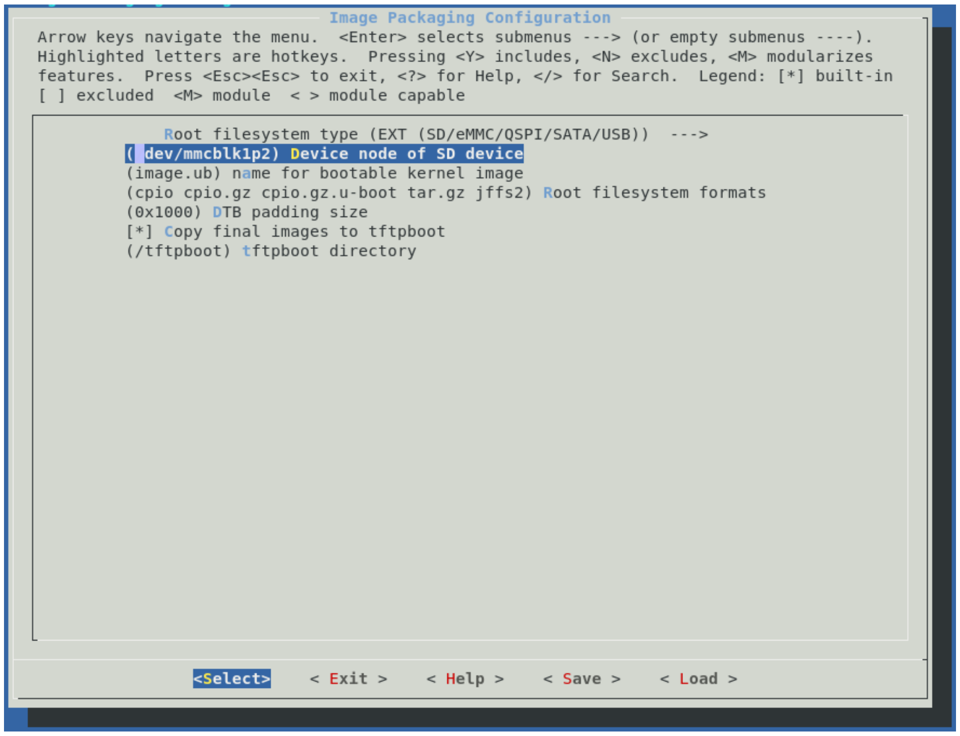
Figure 5

11. This GUI will allow us to control various build options, but we want the tools to handle everything for us so the only thing we will do is change the boot location to be the SD card. To do this, select the "Image Packaging Configuration" option and then select "Root filesystem type." This will bring up a smaller window with the options, from which we will **select the "EXT" option before "other" which includes "SD"** (among other things) as shown after selection on the first line in Figure 6.

12. After the type has been selected, a new menu item called "Device Node of SD device" will have appeared below it. **Select this option and change mmcblk0p2 to mmcblk1p2** as shown on the second line in Figure 6.

    Additional Information: The change in the previous step will tell the boot sequence to search for the ext file system type on the SD card (in our case) when setting up the filesystem.

    This step tells the boot sequence to search on the second partition of the SD card for the filesystem (hence the "p2"), and we change the 0 to a 1 because on boot up, every memory driver is given an ID based how the hardware is set up. According to the Avnet documentation for the UltraZed-EV board, the eMMC flash device is given the device name SD0 which corresponds to mmcblk0, while the SD card is called SD1 and is called mmcblk1. Refer to page 14 of the UltraZed documentation (this link may require login and download of the pdf) for more information.


```
                    Image Packaging Configuration
 Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
 Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
 features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in
 [ ] excluded  <M> module  < > module capable

            Root filesystem type (EXT (SD/eMMC/QSPI/SATA/USB))  --->
        (/dev/mmcblk1p2) Device node of SD device
        (image.ub) name for bootable kernel image
        (cpio cpio.gz cpio.gz.u-boot tar.gz jffs2) Root filesystem formats
        (0x1000) DTB padding size
        [*] Copy final images to tftpboot
        (/tftpboot) tftpboot directory

          <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 6

13. After these two items have been changed, **save the configuration and exit the GUI** (leaving the save location as it is). It may take a little while for the configuration to finish, especially the first time.

14. The last necessary step of the configuration process is to modify the device tree entry for the SD card block to disable write protect. To do this, go to project-spec/meta-user/recipes-bsp/device-tree/files/ starting from the project root directory. In this directory, **open the system-user.dtsi**, which at this point should be empty, and **add the following lines so that the contents of the file look as follows**:

```
/include/ "system-conf.dtsi"
/ {
};

/* SD1 with level shifter */
&sdhci1 {
        status = "okay";
        max-frequency = <50000000>;
        no-1-8-v; /* for 1.0 silicon */
        disable-wp;
};
```

Additional Information: This system-user.dtsi file will be parsed by the PetaLinux tools and used to add or overwrite features of modules which reside in other device tree files automatically generated by the tools.

The device node that we have added is taken from the system-user.dtsi file in the out-of-box BSP produced by Avnet, and in this case, because it is outside of the "/ { };" block, and because we use the "&node_name" syntax, what we are adding will be included with the attributes of the same node which already exists in the auto-generated device tree, overwriting attributes of the same name. There are a few things going on here. The first is that we enable the SD driver by setting the status to "okay." Then we disable write protect with "disable-wp," which is necessary because the default configuration prevents drivers from changing memory they can access as we need to be able to do on the SD card. The other two lines essentially just configure the maximum frequency (which per Avnet docs is 52MHz) and configure the driver to have the level shifter which is required by the SD card reader. Without the addition of this node, the boot process will begin but the kernel won't be able to take control of the root file system, causing it to hang.

1.4 Building the PetaLinux Image

1. Now that things have been configured, we can **build the PetaLinux kernel image**. This is done using the petalinux-build command with no additional tags as shown here (page 25, UG1144):

   ```
   $ petalinux-build
   ```

   Depending on the available hardware of the host OS, this build can take around 20 minutes or over an hour. The result will be an image.ub file that will contain the PetaLinux kernel with all of the hardware and software configurations we have given it.

   Additional Information: Note that the petalinux-build command can also be run with the -c flag which allows you to build out single components. If you only want to build, say, the device tree or kernel image files, you can run petalinux-config -c device-tree or petalinux-build -c kernel. There are many component options to build various parts independently which can reduce build time by avoiding building unnecessary parts.

   WARNING: rebuilding an image after making changes may require cleaning the project first. Refer to this section if you are rebuilding a project after making changes.

1.5 Packaging the Boot Image

1. Next, we nee to **compile the boot image** using the petalinux-package command as shown (page 27, UG1144):

   ```
   $ petalinux-package --boot --fpga --u-boot
   ```

   Additional Information: The tags in this command tell the tools what is being packaged and what to include. The boot tag in this case will request that the BOOT.BIN file be produced, while the other two tags tell the tools to include the bootloader (u-boot) and the FPGA bitstream (fpga) within the BOOT.BIN. For the latter two tags, the locations of the files can be specified if they have been created separately, but if not the default location used is wherever the PetaLinux tools created and placed those files originally. We point this out because we can only use the FPGA tag without a file location afterward due to the fact that we included the bitstream in the exported file from Vivado. If the bitstream were not included, using this tag as we have would produce an error. More information about this command (and others) can be found in UG1157 (the link references page 20 where petalinux-package begins).

1.6 Copying Files to the SD Card

1. Now that everything has been generated, we can **copy the image.ub and BOOT.BIN files over to the first partition of the SD card**. These images are found, again starting from the project root directory, in the images/linux/ directory. There are many generated image files but we only want these two.

2. In the second partition, which should have been formatted as ext4, we need to set up the filesystem. To do this, **copy the rootfs.tar.gz zip folder to the second partition and extract it using the following terminal command** (while the terminal path is in the rootfs partition). You will need to put in your user password for the command to execute because it must be run with "sudo" privileges. After the filesystem has been extracted, the rootfs.tar.gz archive can be removed.

   ```
   $ sudo tar -xzf rootfs.tar.gz
   ```

Additional Information: The .tar.gz archive format preserves access permissions so that when the filesystem is zipped and then extracted elsewhere, all of the files and directories have the same permissions that they did originally. This is required for proper booting because the kernel expects certain directories to be set up beforehand with restricted permissions. The extraction command must be run with root (sudo) privileges for all restricted permissions to be set up correctly.

2 Booting Petalinux

In order to boot PetaLinux and access it through the network using SSH a few different steps must be followed.

2.1 Preparing the Workstation

1. **Insert the SD card** into the reader on the UltraZed's carrier card.
2. **Make sure that the boot configuration switches on the board are set to boot from SD** as described in the Avnet documentation, which will most likely be "OFF-ON-OFF-ON," [1:4].
3. **Connect a USB-to-micro-USB cable between the board and your computer.**
4. **Connect the board's Ethernet port to the network your computer is connected to.**
5. Before turning the board on, **run the screen command in your terminal** as follows:

   ```
   $ sudo screen /dev/ttyUSB1 115200
   ```

   This will allow you to observe the boot messages as the board starts up and see if any errors occur. It will also allow you to log in to the board via UART so that you can find out the current IP address that the board is given on the network.

   Note that the command assumes the USB port on your computer is called "USB1," which may not be the case. If not, you may need to run dmesg in the terminal to see where the board was connected, or try different USB# ports as they appear in the /dev/ directory.

   Additional Information: Notice that command uses a baud rate of 115200. This is due to the default configuration set up by the PetaLinux tools.
6. **Power on the board.** In the terminal you should see boot messages begin almost immediately and describe things like the bootloader, PetaLinux build version, and then list the kernel boot messages as the system is loaded.

2.2 Gaining Access to PetaLinux After Boot

1. When PetaLinux is done booting, you will see "my_petalinux login:" followed by the cursor in the terminal (assuming your project is named my_petalinux). **Enter the username and password**, which are both "root". You should then see "root@my_petalinux:~#".
2. You are now logged into PetaLinux. **To find the IP address, use the ifconfig command**, which will produce several lines of information starting with those shown here:

   ```
   root@my_petalinux:~# ifconfig
   eth0      Link encap:Ethernet  HWaddr 00:0F:36:01:23:01
             inet addr:10.2.116.194  Bcast:10.2.116.255  Mask:255.255.255.0
             inet6 addr: fe81::20d:35fe:fe00:2202/64 Scope:Link
   ```

   The part we care about is "inet addr:10.2.116.194" which will allow us to SSH into the board.
3. In a new terminal, **SSH into the UltraZed** using the following command, substituting whatever IP address you saw in the previous step:

   ```
   $ ssh root@10.2.116.194
   ```

   Again, this should connect to the board and allow you to enter the same credentials as before to login.

3 Advanced PetaLinux Configuration

The PetaLinux tools provide extensive user control over the kernel image, its utilities and features. This section goes over some resources and examples to outline how additional changes to the kernel can be made.

3.1 Important Note about Custom Configurations—Cleaning the Project, etc.

Before beginning to change any additional configuration options please note the following:

1. **If you have already built a PetaLinux project, and then you want to make changes to it and recompile, it is wise—and sometimes essential—to clean the project.** The following PetaLinux commands provide a complete clean of the build space (see page 14 of UG1157):

   ```
   $ petalinux−build −x distclean
   $ petalinux−build −c kernel −x finish
   $ petalinux−build −x mrproper
   ```

   These commands perform the following tasks as described in the user guide:

   (a) *petalinux-build -x distclean* removes the shared state cache

   (b) *petalinux-build -x mrproper* cleans the entire build area and removes all build-generated files

   (c) *petalinux-build -c kernel -x finish* isn't in the documentation for some reason, but this is required by the tools before doing mrproper to finish compiling some uncommitted component files. This is the case when a component (such as the kernel in the command above) has been configured using petalinux-config.

   Keep in mind that there are other clean commands than these which do various degrees of cleaning on various parts of the project. See the documentation for information about these other commands. The ones here were chosen because they provide a complete project clean.

   WARNING: if the project is not cleaned, the tools may not apply new changes because it sees that build products already exist, therefore changes to the system-user.dtsi file and even changes to the configuration menus, may go un-applied.

2. When you run these commands, it is then important to **note that some configuration files will be removed while others may not**. In particular, we have noticed that these commands sometimes remove configurations in the petalinux-config -c kernel configuration menu, but that the system-user.dtsi and petalinux-config menu options are left untouched. In general, after cleaning the project, it is wise to double check all of your configurations to make sure that they have been retained. Also note that these commands may remove user-added patch files and changes to bbappend files if they have not yet been applied.

3. **If you make any changes to the FPGA hardware (memory mapped devices or IO), you must start a new PetaLinux project rather than trying to import the new hardware description into a previous project.** For some reason, the new file may often cause build errors that prevent completion of the image. This doesn't mean that all FPGA changes require a new image—for the most part, you only need to build a new image if you change your PS configuration (including changes to memory-mapped elements), and not elements that are only part of the PL.

3.2 PetaLinux Device Trees

A device tree is a text file that defines various memory-mapped peripherals and their configurations. The information contained in the device tree will be parsed and compiled into the kernel image, and during the first stages of the boot process, necessary nodes will be initialized for use by the kernel during execution. The following two sites are a good place to start as you begin to learn more about how they work.

1. Device Tree Tips is a page on the Xilinx Wiki which "is intended to be a collection place for tips and tricks related to device trees."

2. Device Trees for Dummies is a slide presentation which goes over device tree background and syntax for typical nodes. This information is for Linux systems at large, so the PetaLinux device trees may contain some differences, but the idea and general structure is the same.

Because it is the memory-mapped peripherals that the kernel needs to be aware of, FPGA hardware designs which contain memory-mapped IP should include those IP in the device tree. Thankfully, when configuring a project based on the hardware file exported from Vivado, the PetaLinux tools will add nodes for each memory-mapped IP to auto-generated device tree files without the user having to do any of this manually. The following steps show how to observe this:

1. In a PetaLinux project like the one made previously that has been configured but not yet compiled, **notice that the components/plnx_workspace/ directory only has the conf/ folder in it**. This plnx_workspace/ directory will eventually contain the build products for device trees and other things.

2. Now, build the image. After it completes, you will **notice the addition of a device-tree/device-tree/ directory branch in the plnx_workspace/ directory, which is populated with various .dtsi files** among other things.

   Additional Information: When the petalinux-build command is run, the default device trees are auto-generated and parsed together with the system-user.dtsi, which, as we have mentioned before, allows the user to add in their own nodes and device configurations. If you want to build the device tree without building the full kernel image, you can run the petalinux-config command as follows after importing the HDF or XSA into PetaLinux.

   ```
   $ petalinux-config -c device-tree
   ```

   This will produce all of the same files observed in the following steps.

3. **In this device-tree/ directory, open the pl.dtsi (if present) and notice that it is mostly empty**, indicating that our FPGA design doesn't have any memory-mapped peripherals. This makes sense because our design only has the zynqMP processor block in it.

4. Now **say that we have an FPGA design in Vivado that has a memory-mapped IP called "axi_lrf_controller_0,"** and which is mapped to the PS memory base address 0xA0000000. After building the bitstream, exporting the hardware from Vivado and importing it into a PetaLinux project as we did previously, **the petalinux-build command will create the pl.dtsi and populate it with something similar to the following** (along with the header and perhaps a node for PL clocks):

   ```
   / {
       amba_pl: amba_pl@0 {
           #address-cells = <2>;
           #size-cells = <2>;
           compatible = "simple-bus";
           ranges ;
           axi_lrf_controller_0: axi_lrf_controller@a0000000 {
               clock-names = "lrf_word_clk", "S_AXI_ACLK";
               clocks = <&misc_clk_0>, <&zynqmp_clk 71>;
               compatible = "xlnx,axi-lrf-controller-1.0";
               reg = <0x0 0xa0000000 0x0 0x1000>;
           };
       };
   };
   ```

5. This node is essentially saying that the PL exists, and then lists the memory mapped IPs it contains and their configurations within it. **Notice that the axi_lrf_controller_0 is now in the device tree**, and that it has been given the same address that it had in Vivado. You can also see how it shows which clocks are connected to it on lines 8 and 9, and that the block will need 0x1000 range of address space on line 11. This size parameter should also be the same as the size in Vivado.

Additional Information: To find the default SD memory driver in the device tree, open zynqmp.dtsi in the same folder. This will have all driver nodes that are included in the processor by default. Search for “sdhci” and you will see the two memory device drivers with all of their configuration material. You will recall that it is “sdhci1” that we overwrote previously to configure the UltraZed to boot from the SD card.

3.3 FPGA Manager Utility

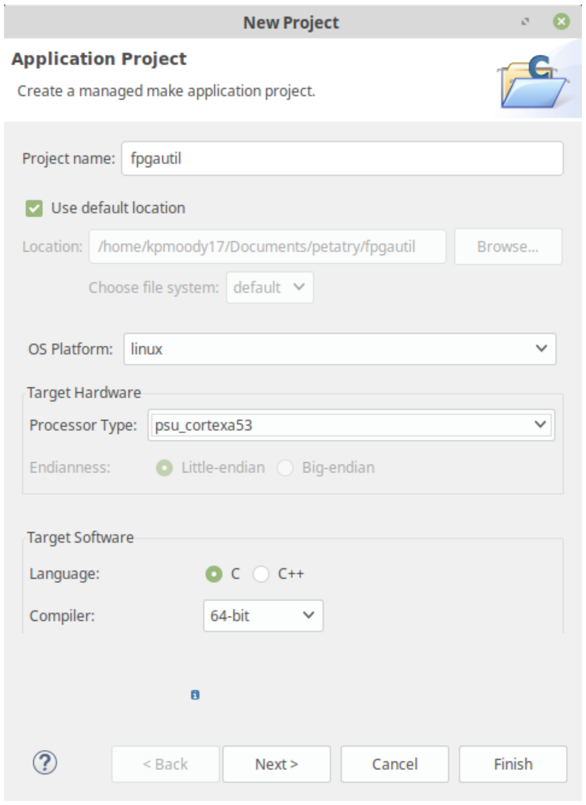
The FPGA Manager is a feature enabled by default in the PetaLinux configuration. One of the purposes of this feature is to allow the user to change the FPGA bitstream while booted into PetaLinux. In order to interface with the Manager properly, Xilinx provides the “fpgautil” utility, which the user can execute to upload a bitstream as described briefly in the following steps:

1. **Visit the** FPGA Programming **section of the** Solution ZynqMP PL Programming **Xilinx Wiki page.**

2. Under the subsection “Exercising FPGA programming using fpgautil”, **click on the link “fpgautil.c” to download the utility source code.**

3. Next, we have to compile this source code to use it on the board, which can be done using Vitis or the Xilinx SDK. To do this in the SDK, first **open the IDE and set up an application project to operate on Linux, targeting the psu_cortexa53 processor, and using C**, as shown in Figure 7.

   Additional Information: The Xilinx SDK version used for these instructions was 2019.1, which is the last version before the SDK was absorbed into Vitis for 2019.2. Even though we are using 2019.2 for the rest of the tools, compiling the executable for this utility only needs to know that the platform is Linux and what kind of processor it will be running on. Getting a quick application project up and running from the SDK is several steps simpler than Vitis, which is why it was chosen here.


New Project

Application Project

Create a managed make application project.

Project name: fpgautil

[✓] Use default location

Location: /home/kpmoody17/Documents/petatry/fpgautil

Choose file system: default

OS Platform: linux

Target Hardware

Processor Type: psu_cortexa53

Endianness: [● Little-endian] Big-endian

Target Software

Language: [● C] C++

Compiler: 64-bit

Figure 7

4. If asked what kind of application project you want to create, the "hello world" template will work fine. When the project is set up, **copy the fpgautil.c file into the src directory and delete the helloworld.c file**.

5. **Save this project and build it.** If auto-build is not turned on, you can click the hammer in the upper left of the GUI or press ctrl+b on the keyboard to build the project.

6. In the console window, you should see "Build Finished" which indicates that the executable is ready in the Debug/ directory. Assuming your application project was called fpgautil as shown in Figure 7, **copy the fpgautil.elf executable from <project-root-directory>/fpgautil/Debug/ to a convenient location in the PetaLinux filesystem.**

   Additional Information: This can be done from the terminal using the scp command as shown here (assuming the board is booted and connected to the network as described previously, and also assuming that you have gone to the SDK project's root directory in the terminal first).

```
$ scp fpgautil/Debug/fpgautil.elf root@10.2.116.194:~/
```

7. **Copy a bitstream to the same location as the fpgautil executable.** (Or elsewhere on the board, as long as you point to it correctly in the next command.)

8. **Program the bitstream onto the board using the executable as shown here:**

```
$ ./fpgautil.elf -b design_1_wrapper.bit
```

   The "-b" in this command indicates that the file following it is the bitstream. If successful, the output should look something like the following:

```
Time taken to load BIN is 190.000000 Milli Seconds
BIN FILE loaded through zynqMP FPGA manager successfully
```

Additional Information: For more information on additional fpgautil options, visit the Xilinx Wiki page mentioned in the first step.

3.4 Unlocking Protected Device Memory (enabling /dev/mem)

From version to version, Xilinx will occasionally change the defaults for the PetaLinux configuration. One such example, when migrating from 2019.1 to 2019.2, is that Xilinx enabled a memory protection module which allows only the kernel to access device memory which is not owned by a device in the device tree. Prior to this version, the user could write programs, open device memory—called /dev/mem—and use Linux commands like "mmap" to get a pointer to any memory address and read from and write to that memory. Because some applications (like DMA) benefit from this flexibility, these next steps show how to turn off the memory protection module. This will also give a glimpse into another corner of the PetaLinux tools.

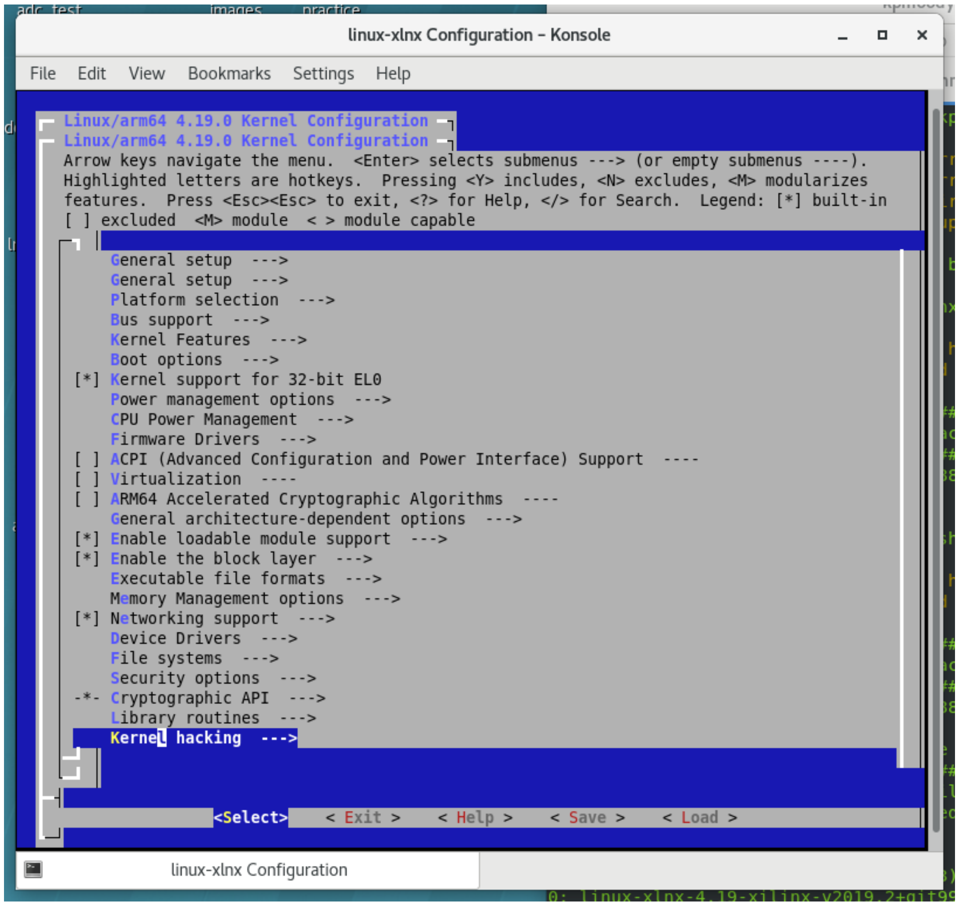
Caveat: Because these steps will remove the restrictions on memory access, there is greater inherent risk of inadvertently messing up memory contents. This is due to the fact that removing the protection module will allow the root user (which is the only default user in PetaLinux) to access kernel memory and restricted memory, such that they can read and write anywhere at will. Furthermore, this will give malicious users the opportunity to cause intentional damage if they manage to log in as root. So, it is recommended, where possible, to create a reserved memory node in the device tree that you can access freely in spite of the memory protection module, rather than disabling that module altogether. However, because this feature was disabled by default in the past, and for the sake of exploring the tools, the steps are shown here.

1. These instructions assume that a PetaLinux project has already been created and that the hardware file has been imported. If the images have been created as well, that's ok, but they will need to be rebuilt again after following the rest of the steps.

2. From in the root directory of the PetaLinux project, in a terminal which already has the PetaLinux tools environment set up, **run the petalinux-config command with the kernel component as shown** to open the kernel configuration GUI shown in Figure 8. Unfortunately this GUI has some formatting issues, but everything is still visible.

```
$ petalinux-config -c kernel
```

3. **Click on "Kernel hacking" option**, which is highlighted at the bottom of Figure 8.



```
Linux/arm64 4.19.0 Kernel Configuration
Linux/arm64 4.19.0 Kernel Configuration
Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in
[ ] excluded  <M> module  < > module capable

        General setup  --->
        General setup  --->
        Platform selection  --->
        Bus support  --->
        Kernel Features  --->
        Boot options  --->
    [*] Kernel support for 32-bit EL0
        Power management options  --->
        CPU Power Management  --->
        Firmware Drivers  --->
    [ ] ACPI (Advanced Configuration and Power Interface) Support  ----
    [ ] Virtualization  ----
    [ ] ARM64 Accelerated Cryptographic Algorithms  ----
        General architecture-dependent options  --->
    [*] Enable loadable module support  --->
    [*] Enable the block layer  --->
        Executable file formats  --->
        Memory Management options  --->
    [*] Networking support  --->
        Device Drivers  --->
        File systems  --->
        Security options  --->
    -*- Cryptographic API  --->
        Library routines  --->
        Kernel hacking  --->

              <Select>    < Exit >    < Help >    < Save >    < Load >
```

linux-xlnx Configuration – Konsole
File Edit View Bookmarks Settings Help
linux-xlnx Configuration

Figure 8

4. In the new menu that appears, scroll down using the arrow keys until you reach the line that says “[*] Filter access to /dev/mem.” The asterisk in the brackets indicates that it is enabled, so **click “n” on the keyboard to disable it**.

5. **Save and exit the kernel configuration menu.**

6. **Proceed as usual to build and package the images.** Now, when you boot up PetaLinux, you will be able to map to all device memory and use it in applications.

Additional Information: One key takeaway from these brief steps is that there are *many* configuration options that the user can control. In Figure 8 you can see configuration options for drivers, the filesystem, booting, power management, cryptography, etc. and within each of these, as we saw with the kernel hacking option, there are several modules which can be enabled or disabled.

3.5 Enabling SPI Communication from PS

In order to enable SPI in PetaLinux so as to communicate with a peripheral, the SPI port must be enabled in Vivado and correctly connected. Note that there are many ways to do this, include a SPI IP in the PL, using the default SPI EMIO ports to communicate with an off-chip SPI peripheral, or building a custom board that has a SPI peripheral that requires different IO pins. Here, we assume there is an off chip SPI peripheral to which we want to send data only as a master without requesting data back. One good tutorial with additional help (though it is for the MicroZed board) can be found here.

1. In the Vivado project, **double click on the PS block** to open up its configuration GUI.

2. In this GUI **click on "I/O Configuration" on the left pane, and navigate to "Low Speed," then "I/O Peripherals," then "SPI."**

3. **Select the first option** as shown in Figure 9, and **make sure that it says "EMIO"** next to it. Note, if multiple SPI slaves are desired, click the little arrow next to "SPI 0" and enable more. This will allow you to put multiple SPI nodes within the SPI0 device as will be shown later on.

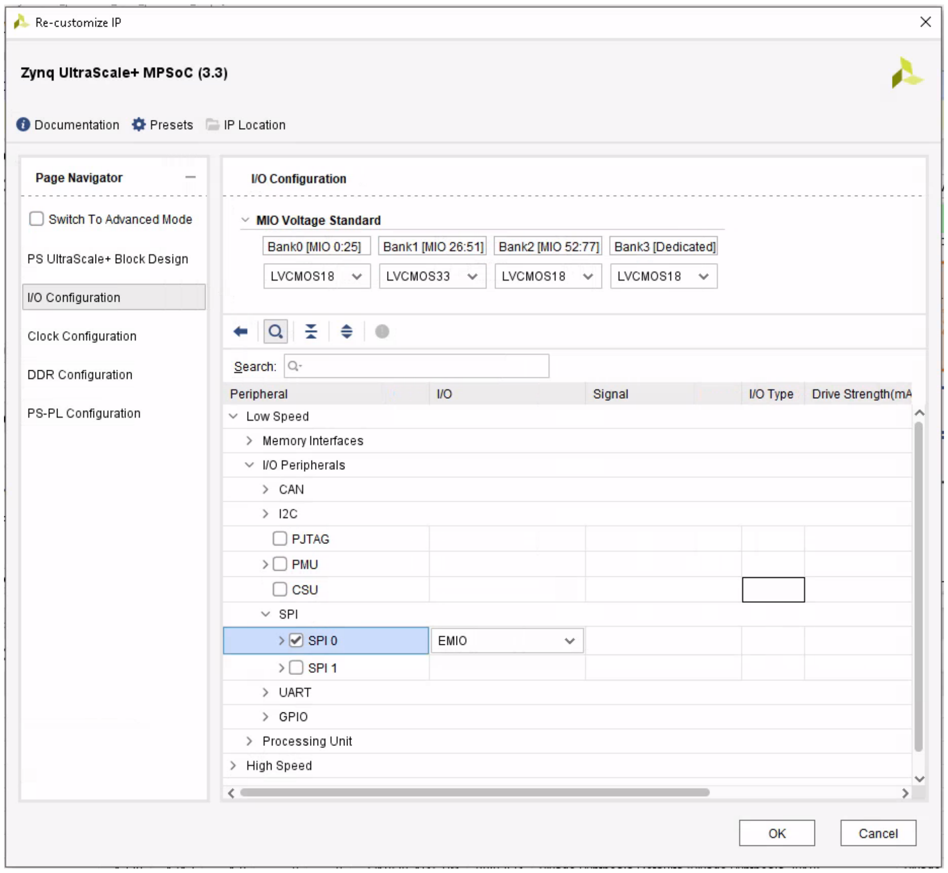


Figure 9

4. Click ok, and then observe that the PS adds a SPI_0 port. Click on the plus sign of the port and **follow Figure 10 to make some signals external**. If multiple SPI nodes were enabled in the PS, you will notice that there will be additional SPI select pins such as emio_spi0_ss1_o_n—these should be made external as well.

   Additional Information: We choose this configuration with the assumption that we want to communicate with a peripheral that only acts as a slave, and therefore will only send out the clock (SCLK), master output (MOSI), and slave select (SS) pins. There are various pins in this port, including inputs from the slave as well as tristate signals, none of which we need to use if only sending data out to a slave.

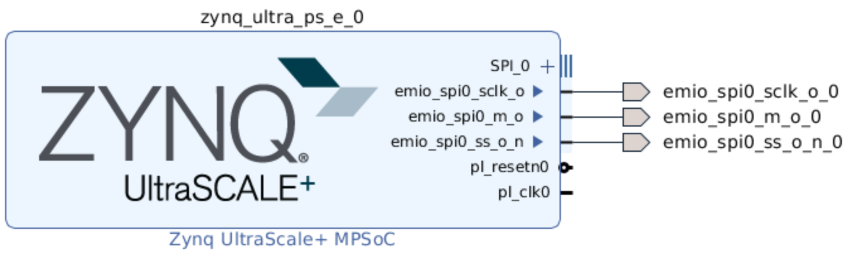
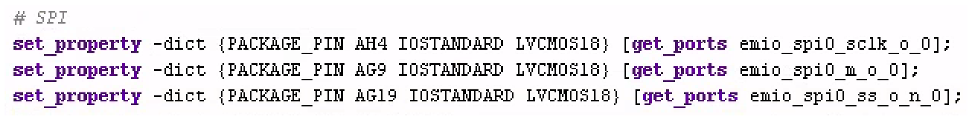


Figure 10

5. Next, the Vivado constraints must be set to declare the type of output for each SPI connection. If your project doesn't have a constraints file, add one by clicking "Add Sources" in the "Project Manager" section of the left pane in Vivado. After doing this, **add the lines shown in Figure 11 to the constraints file**. Again, remember that if there are multiple SPI nodes, the additional SPI select (ss) ports will need to be constrained.



```
# SPI
set_property -dict {PACKAGE_PIN AH4 IOSTANDARD LVCMOS18} [get_ports emio_spi0_sclk_o_0];
set_property -dict {PACKAGE_PIN AG9 IOSTANDARD LVCMOS18} [get_ports emio_spi0_m_o_0];
set_property -dict {PACKAGE_PIN AG19 IOSTANDARD LVCMOS18} [get_ports emio_spi0_ss_o_n_0];
```

Figure 11

Additional Information: It is important to know that the PACKAGE_PIN attribute may change depending on your custom carrier card. The idea is that you select whichever package pins are associated with which ever SPI pin and connect them in this fashion. The IOSTANDARD attribute should also reflect your correct pin type as the voltage output will depend on this.

6. With everything connected and configured, **built the Vivado bitstream and import the xsa/hdf into a new PetaLinux project**.

7. In the PetaLinux project, after importing the hardware file and configuration the necessary elements for booting, **go to the kernel configuration menu** with the following command:

```
$ petalinux-config -c kernel
```

8. Then, **go into "Device Drivers" and then "SPI Support"**.

9. In this menu, **scroll down to "User mode SPI device driver support" and click the spacebar until you see <*>**.

10. After this, be sure to **save the configuration** and exit the config menu.

11. Next, you need to **update the system-user.dtsi device tree** (see this step) with the following node:

```
&spi0 {
    num_cs = <1>;
    status = "okay";

    spidev@0x00 {
        compatible = "spidev";
        spi-max-frequency = <50000000>;
        reg = <0>;
    };
};
```

If multiple SPI slaves are connected, use the following as a template.

```
&spi0 {
        num-cs = <2>;
        status = "okay";

        spidev@0x00 {
                compatible = "spidev";
                spi-max-frequency = <50000000>;
                reg = <0>;
        };
        spidev@0x01 {
                compatible = "spidev";
                spi-max-frequency = <50000000>;
                reg = <1>;
        };
};
```

Additional Information: In the parent node, num-cs is used to tell the kernel how many SPI devices are connected. Each child node is given a different address as shown, to which the reg attribute must also correspond. Note that if multiple SPI parent nodes are used (e.g. SPI 1 in the PS config) they will need to have their own node (e.g. &spi1 {...}).

12. Now, **the project can be built and packaged**. Communication with a device via SPI from code is outside of the scope of this tutorial, though various driver files are available (such as those found here) to help you build a project.

13. To verify that the SPI device is seen by the kernel **check the /dev directory where you should see spidevX.Y nodes** (X is the device number and Y is the child node). Note that the device number may not correspond with the value that you expect. For example, using the steps in this tutorial for two SPI child nodes—0 and 1—within one parent node—spi0—the two spidev devices were spidev1.0 and spidev1.1.

Additional Information: While preparing this part of the tutorial, we discovered another SPI device tree attribute is-decoded-cs, which allows the user to use fewer SPI select ports to communicate with more SPI peripherals by using the ss pins to decode which SPI to talk to. To make this work, additional FPGA decode hardware may need to be present for proper functionality.

3.6 Enabling and Working with Interrupts

Interrupts are used in a large variety of ways and can be monitored by an OS in many different ways. This section is dedicated to help you get started with connecting interrupts to the PS in Vivado and then enabling them in the device tree for use in programs during runtime. Here is a list of links used which may help with troubleshooting. The first few are tutorials or general information and the rest are responses to specific challenges. **Before doing this tutorial, be sure to read the section on** project cleaning as adding interrupts to a project that has already been built will likely not regenerate the necessary files.

- Interrupt driven user space application with the uio driver
- Testing UIO with Interrupt on Zynq Ultrascale
- Linux GIC Driver
- ARM GIC Interrupt Documentation
- Zynq Ultrascale+ Device Technical Reference Manual
- Introduction to Device Trees (In particular the section on interrupts.)
- Device Tree Tips (In particular sections 7 and 8.)

- Petalinux 2017.4 Zynq PL-PS Interrupt Question (In particular see the pdf posted in the marked solution.)
- Petalinux custom kernel boot args
- PL-PS INTERRUPT on Ultrazed on Petalinux

Next, we will go through the steps here for configuring PetaLinux correctly and verifying that interrupts are properly added. Fort this tutorial, we elect to use the Linux GIC driver which can be built into the PetaLinux image and then modify the device tree with a device from the PL that is producing an interrupt we want to monitor.

For now, this tutorial doesn't cover accessing the interrupts from code, but the "Interrupt driven user space application with the uio driver" link in the previous list has some good starter code for this.

1. In your Vivado project, **enable interrupts by double clicking the PS block and going to "PS-PL Configuration," then "General," then "Interrupts," then "PL to PS," and enabling the first option** as shown in Figure 12.

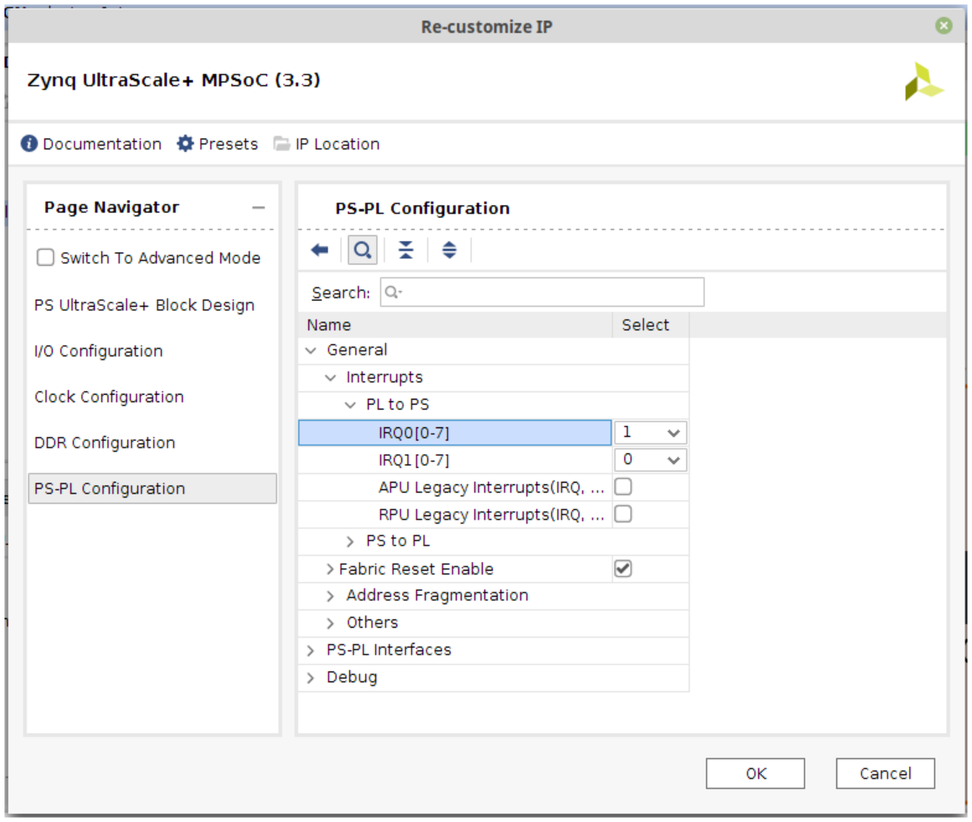


Figure 12

2. Click ok. Next, **connect whichever interrupt-producing IPs you desire to the PS interrupt port using a Concat block** as shown in Figure 13. In this case we just use counters for demonstration. Note however, that if an interrupt is configured as level sensitive, the signal may need to be held asserted for several clock cycles before being de-asserted.

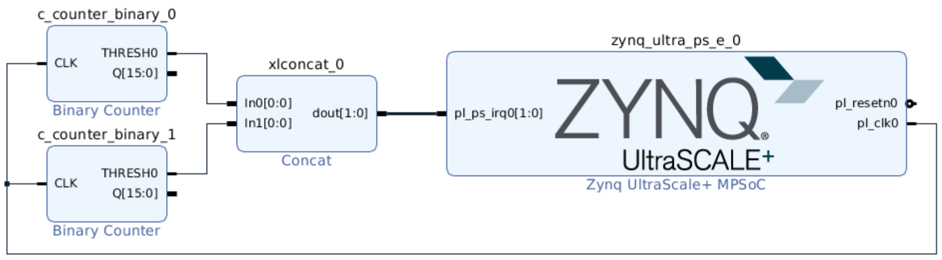


Figure 13

Additional Information: By default, the port is 1 bit wide, but using a Concat block will propagate a larger width into the port, which will update as validation is run. The max width of each PL to PS interrupt port is 8. Also, though not explored as much in this tutorial, some IPs, such as DMA, which are memory-mapped into the PS and which also may have an interrupt connected, may build out the interrupt information into the DMA device tree node for you. So, we are choosing counters to prevent this automatic process from occurring and so that we can go through the steps of adding a new node to the device tree to make the PS aware of the interrupts that aren't coming from a memory-mapped block.

3. After connecting the IP, **build the bitstream and export the hardware file and copying it into a PetaLinux project as usual** (see this step and those following it for more information).

4. Run the usual configuration of PetaLinux to set up the board for booting and then **go to the kernel config menu** with the following command:

```
$ petalinux-config -c kernel
```

5. In this menu, **go to "Device Drivers," then "Userspace I/O drivers"**. In this menu, **go to "Userspace I/O platform driver with generic IRQ handling" and "Userspace platform driver with generic irq and dynamic memory" and click the spacebar until you see <*>.** The default is likely <M> which modularizes the features but we want them built-in.

6. **Save this configuration** and exit the menu.

7. Next, we have to make two separate modifications to the device tree: adding a "uio_pdrv_genirq.of_id" attribute to the boot arguments and creating a node that represents the source of the interrupts. **Open the system-user.dtsi file** (discussed previously) and **create a chosen node in the empty device block with the following lines:**

```
/ {
    chosen {
        bootargs = "earlycon console=ttyPS0,115200 clk_ignore_unused
root=/dev/mmcblk1p2 rw rootwait uio_pdrv_genirq.of_id=generic-uio";
        stdout-path = "serial0:115200n8";
    };
};
```

Additional Information: Notice that this node is within the "/ { };" block which is present in the file by default. All nodes in this block are new and will overwrite blocks if they exist or create them if they do not. The chosen block allows us to update the boot arguments passed to the kernel in the first stages of boot up. While what all of the present commands do is outside of the scope of this tutorial, two things are important to know:

(a) Everything from "earlycon" through "rootwait" as well as the second attribute "stdout-path" are required. These are normally populated automatically by the PetaLinux tools, but because we add this node again here in the user-overwrite file, if we don't include the required bootargs they will be overwritten and left out. (Keep in mind that if you are following this tutorial and using a different board than the one used here, the bootargs will be different and can be found in the petalinux-config menu, or by finding the system-conf.dtsi file generated by the PetaLinux tools during build.)

(b) The main addition with regard to interrupts is the "uio_pdrv_genirq.of_id=generic-uio." This is not present by default and we add it here to tell the kernel where our interrupt driver is to be found. The scope of the User I/O (UIO) driver is larger than we can discuss here, but suffice it to say that it allows the user to communicate with peripherals which are self contained and/or where the peripheral doesn't need a full blown driver to communicate with it. Examples of these are the AXI DMA which can be controlled just by writing to PL registers, and these interrupts where all we really need is to tell the kernel to watch for them. Neither the kernel nor the peripheral require any more resources than what they have to communicate with the other.

8. With this bootargs attribute added, we now need to make a device node for the kernel to associate the interrupts with. In the same system-user.dtsi, **add the following lines** outside of the block we just added the chosen node to:

```
&amba_pl {
    counter_irq: counter_irq {
        compatible = "generic-uio";
        interrupts = < 0 89 4 >;
        interrupt-parent = <&gic>;
    };
};
```

Additional Information: You will notice that this node is a modification of the amba_pl node generated by the PetaLinux tools in pl.dtsi. We are modifying the parent node by adding a new node called counter_irq. You can make the name whatever you like; it will be this name that appears in the processor interrupt list. The three lines work as follows:

(a) *compatible:* This is a form of ID that allows us to associate the device node with a certain protocol or family of devices which are all compatible. **This must be the same as the string on the right of the uio_pdrv_genirq.of_id attribute in the boot arguments.**

(b) *interrupts:* This array of values gives the kernel information about the interrupt. The first value indicates whether it is a SPI, PPI, or SGI device (shared, private, or sofware generated interrupt). Zero indicates SPI which works in our case (and testing with PPI suggests it isn't enabled on our board anyway). The second value is the location of the interrupt pin. In the Zynq Ultrascale+ Device Technical Reference Manual in the list above, we learn that the PL_PS_Group0 ranges from ID 121 to 128. But, in order to properly reference these pins, we must subtract 32, giving us the value of 89 present in the node (if non-SPI is used—'1' in the first position—subtract 16 instead to produce 105 in the second position). These 8 values correspond to the 8 possible pins in the PS interrupt port we enabled. The last value indicates the triggering of the interrupt: 0 = default per kernel, 1 = low-to-high edge, 2 = high-to-low edge (except for SPI), 4 = high level, 8 = low level (except for SPI). More of this information can be found in the ARM GIC Interrupt Documentation listed previously.

(c) *interrupt-parent:* This attribute lets the kernel know which interrupt controller to monitor the interrupts with. The node for this controller can be found by searching "gic:" in the zynqmp.dtsi file generated by the PetaLinux tools.

9. Having done this configuration, **save the file and build and package the project** as described previously.

10. After booting the image, **print the bootargs** to the screen to make sure that they have been updated correctly:

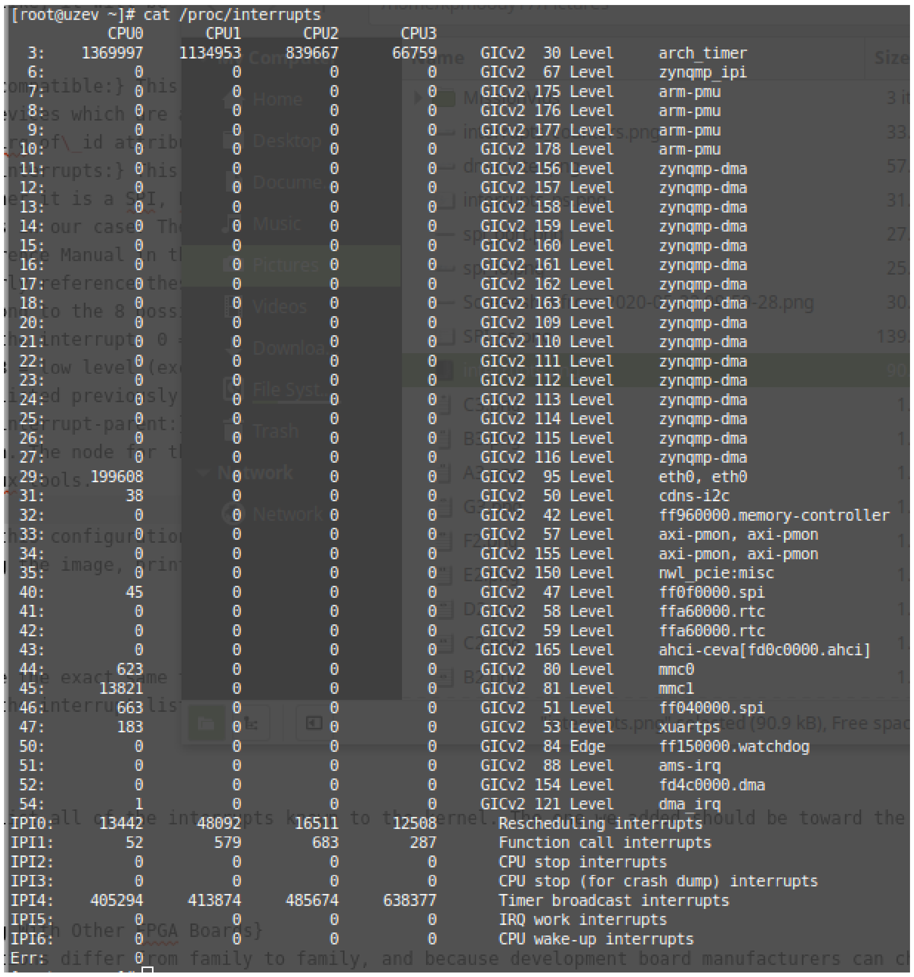
```
$ cat /proc/cmdline
```

This should produce the exact same text we added to the bootargs attribute in the chosen node.

11. Then, **print the interrupt listings** to the screen:

```
$ cat /proc/interrupts
```

This command will list all of the interrupts known to the kernel. The one we added should be toward the bottom of the GICv2 list as shown in listing 54 in Figure 14. The device node name used for for the project in the image was dma_irq, which should be counter_irq for this example.



```
[root@uzev ~]# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  3:    1369997    1134953     839667      66759     GICv2  30 Level     arch_timer
  6:          0          0          0          0     GICv2  67 Level     zynqmp_ipi
  7:          0          0          0          0     GICv2 175 Level     arm-pmu
  8:          0          0          0          0     GICv2 176 Level     arm-pmu
  9:          0          0          0          0     GICv2 177 Level     arm-pmu
 10:          0          0          0          0     GICv2 178 Level     arm-pmu
 11:          0          0          0          0     GICv2 156 Level     zynqmp-dma
 12:          0          0          0          0     GICv2 157 Level     zynqmp-dma
 13:          0          0          0          0     GICv2 158 Level     zynqmp-dma
 14:          0          0          0          0     GICv2 159 Level     zynqmp-dma
 15:          0          0          0          0     GICv2 160 Level     zynqmp-dma
 16:          0          0          0          0     GICv2 161 Level     zynqmp-dma
 17:          0          0          0          0     GICv2 162 Level     zynqmp-dma
 18:          0          0          0          0     GICv2 163 Level     zynqmp-dma
 20:          0          0          0          0     GICv2 109 Level     zynqmp-dma
 21:          0          0          0          0     GICv2 110 Level     zynqmp-dma
 22:          0          0          0          0     GICv2 111 Level     zynqmp-dma
 23:          0          0          0          0     GICv2 112 Level     zynqmp-dma
 24:          0          0          0          0     GICv2 113 Level     zynqmp-dma
 25:          0          0          0          0     GICv2 114 Level     zynqmp-dma
 26:          0          0          0          0     GICv2 115 Level     zynqmp-dma
 27:          0          0          0          0     GICv2 116 Level     zynqmp-dma
 29:     199608          0          0          0     GICv2  95 Level     eth0, eth0
 31:         38          0          0          0     GICv2  50 Level     cdns-i2c
 32:          0          0          0          0     GICv2  42 Level     ff960000.memory-controller
 33:          0          0          0          0     GICv2  57 Level     axi-pmon, axi-pmon
 34:          0          0          0          0     GICv2 155 Level     axi-pmon, axi-pmon
 35:          0          0          0          0     GICv2 150 Level     nwl_pcie:misc
 40:         45          0          0          0     GICv2  47 Level     ff0f0000.spi
 41:          0          0          0          0     GICv2  58 Level     ffa60000.rtc
 42:          0          0          0          0     GICv2  59 Level     ffa60000.rtc
 43:          0          0          0          0     GICv2 165 Level     ahci-ceva[fd0c0000.ahci]
 44:        623          0          0          0     GICv2  80 Level     mmc0
 45:      13821          0          0          0     GICv2  81 Level     mmc1
 46:        663          0          0          0     GICv2  51 Level     ff040000.spi
 47:        183          0          0          0     GICv2  53 Level     xuartps
 50:          0          0          0          0     GICv2  84 Edge      ff150000.watchdog
 51:          0          0          0          0     GICv2  88 Level     ams-irq
 52:          0          0          0          0     GICv2 154 Level     fd4c0000.dma
 54:          1          0          0          0     GICv2 121 Level     dma_irq
IPI0:     13442      48092      16511      12508       Rescheduling interrupts
IPI1:        52        579        683        287       Function call interrupts
IPI2:         0          0          0          0       CPU stop interrupts
IPI3:         0          0          0          0       CPU stop (for crash dump) interrupts
IPI4:    405294     413874     485674     638377       Timer broadcast interrupts
IPI5:         0          0          0          0       IRQ work interrupts
IPI6:         0          0          0          0       CPU wake-up interrupts
Err:          0
```

Figure 14

Notice that various parts of our configuration are visible: the pin number is 121 which is the lowest in the PS range we used, it is level sensitive (high in our case), and it has the correct name. And, you can already see that CPU0 has registered an interrupt.

3.7 Reserved CPUs

Reserving CPUs can be helpful because it prevents the kernel from sending tasks there if not needed. It appears that CPU usage by the kernel can't be completely avoided (interrupts still occur occasionally on the reserved CPU), but it can be reduced and the reserved kernel can be dedicated to run a user-defined task with more consistent performance. This is done with the following two steps:

1. **Add "isolcpus=3" to the bootargs** in the same fashion as the uio_pdrv_genirq.of_id attribute described in this interrupts section. Note: using this argument exactly will reserve the 3rd CPU; CPUs are zero indexed and to reserve multiple CPUs, separate each one with a comma and no spaces.

2. After building and booting the image as described previously, **run the following command to see which CPUs have been properly reserved**.

```
$ cat /sys/devices/system/cpu/isolated
```

3.8 Reserved Memory

Reserving memory can be helpful when an AXI DMA engine is writing data to RAM and you don't want the kernel to inadvertently claim that same memory for other things. Creating reserved memory will tell the kernel not to touch it or only to allow it to be accessed under certain circumstances. First, the below links provide some helpful information used when building the reserved memory device tree nodes:

1. Reserved Memory kernel device tree bindings documentation
2. Reserved Memory Xilinx Wiki
3. Xilinx forum response about node syntax (see response by Xilinx employee Ibaie)

The following steps explain how the device tree was configured for the board in this tutorial and how to verify that it is working. This example is for the purpose of making three contiguous blocks of reserved memory for three separate DMA engines.

1. First, **open the system-user.dtsi file in your PetaLinux project** (discussed previously) and **add the following node into the root node provided** (root node being the "/{}" as done with the chosen node).

```
reserved-memory {
    #address-cells = <2>;
    #size-cells = <2>;
    ranges;

    dma_mem_0: dma_mem@0x0F000000 {
        reg = <0x0 0x0F000000 0x0 0x01000000>; /* 16Mib */
    };
    dma_mem_1: dma_mem@0x10000000 {
        reg = <0x0 0x10000000 0x0 0x01000000>; /* 16Mib */
    };
    dma_mem_2: dma_mem@0x11000000 {
        reg = <0x0 0x11000000 0x0 0x08000000>; /* 128Mib */
    };
};
```

Additional Information: The node "reserved-memory" is known to the kernel and thus must be spelled correctly. Additionally, it must follow certain syntax. If not, the following error or something similar can be present in the boot log:

```
Aug 25 17:35:20 uzev kernel: OF: fdt: Reserved memory: unsupported node
format, ignoring
```

The "#address-cells" attribute says that the reg attribute in each child node will have 2 address values, while the "#size-cells" indicates two size attributes in each child node. Notice how each child node has our desired address, such as 0x0F000000 in dma_mem_0 and the associated size of 0x01000000. Before these, there is the additional address value of 0x0 and associated size of 0x0.

There is little documentation as to why these additional null values are required, but if they are not present, (and if the address and size cells are set to 1) the previous error will occur and memory will not be reserved. Based on the forum response in the list at the start of this section, it is likely something to do with 64 vs 32-bit architectures.

Each of the dma_mem_X names are lables for the child nodes, followed by the actual name of the nodes associated with address spaces.

Note that the ranges attribute is required but empty in this case.

2. After configuring the device tree, **build the project (using the same considerations and techniques discussed throughout this tutorial) and package it**.

3. After booting the board, **run the following command to observe the boot logs**:

   ```
   $ journalctl -b
   ```

   Make sure that there are not reserved memory errors.

4. Next, run the following to observe that reserved memory has been registered as part of boot.

   ```
   $ cat /proc/iomem
   ```

   The output of this command should produce something like the following (first few lines, line numbers added):

   ```
   1   00000000-7feffffff : System RAM
   2      00080000-0104ffff : Kernel code
   3      01050000-0111ffff : reserved
   4      01120000-01213fff : Kernel data
   5      07ff4000-07ffbfff : reserved
   6      0f000000-18ffffff : reserved
   7      6bc00000-7fbfffff : reserved
   8      7fef7000-7fefefff : reserved
   9      7feff000-7fefffff : reserved
   ```

   Notice line 6 which contains the full range of our reserved memory from the device tree. The memory has been reserved and the kernel can access it upon request (with /dev/mem or a driver) but will not use it for kernel memory.

3.9 Creating Custom PetaLinux Drivers

This topic is *much* broader than can be covered here, but one simple version is offered to give an idea of the possibilities. First, here are some helpful links that were used to create the code for this driver (starting with most helpful):

1. Writing a Simple Linux Kernel Module (this was used as a template for this tutorial's module)
2. Stack Overflow forum post: How to create a device node from the init_module code of a Linux kernel module?
3. UG1144: PetaLinux Tools Documentation, page 75 (documentation for doing custom modules in PetaLinux)
4. Stack Overflow forum post (the question shows a helpful code snippet)
5. A simple char device example for linux module (git repo)
6. User space mappable DMA Buffer (git repo)
7. Spidev kernel driver code (and other supported kernel drivers)
8. Xilinx solution for adding modules to a PetaLinux design

9. Character Device Drivers (Linux Wiki)

Writing device drivers is no simple task. While syntactically they aren't different from other C programs, they must be written bug free and interface correctly with the kernel API so as not to hang or break the kernel. Standard libraries are not available, and the kernel API functions must be used for the kernel to recognize the device and load it successfully. With all of this in mind, the driver code is provided at the end of this document in the appendix, but further explanation for how to write it is left to the previous links. Instead, the following points explain how to use PetaLinux to create a module and compile it, and then how to load it into the kernel. Some of these steps assume that certain attributes of the provided code are used in the driver, such as registering the device in the __init function and creating a node in /dev/.

1. In a PetaLinux project that has already been created and set up, run the *petalinux-create* command again to set up a module inside of it:

   ```
   $ petalinux-create -t modules -n rmd --enable
   ```

   This command requests that a module be made called "rmd" and that it be enabled during the build process.

2. After this is complete, go into the project-spec/meta-user/recipes-modules/rmd directory and you will notice that a Makefile, rmd.c source file, and COPYING file have been created for you. **Modify the rmd.c and add your own driver sources to this directory**, and then **update the Makefile** accordingly for proper compiling.

3. After the driver has been created, run the *petalinux-build* command to compile the module.

   ```
   $ petalinux-build -c rmd
   ```

   This command will compile your "rmd" driver component.

4. After compiling, you will **find your compiled rmd.ko file** in the build/tmp/sysroots-components/ plnx_zynqmp/rmd/lib/modules/4.19.0-xilinx-v2019.2/extra/ directory, or by searching for your module in the command line from the project root directory:

   ```
   $ find -name rmd.ko
   ```

   For simple testing and running, you can **copy this file to a directory in a working PetaLinux filesystem**.

5. To load the driver into the kernel, **run the insmod command** in the directory where your module is kept (see also this blog for more explanation about loading the module with simpler drivers):

   ```
   $ insmod rmd.ko
   ```

6. After the kernel is loaded, **run dmesg to observe the kernel messages**. The example driver provided here, because of the messages reported in the __init function, will show something like the following as it loads:

   ```
   [64409.520964] Initializing RMD
   [64409.523847] RMD module loaded with device major number 253755392
   [64409.529896] Device class created
   [64409.533236] Device node created
   ```

   Note that you will also see a message that the module "taints" the kernel, which means that a module has been loaded which is not built into the kernel and/or which is not supported.

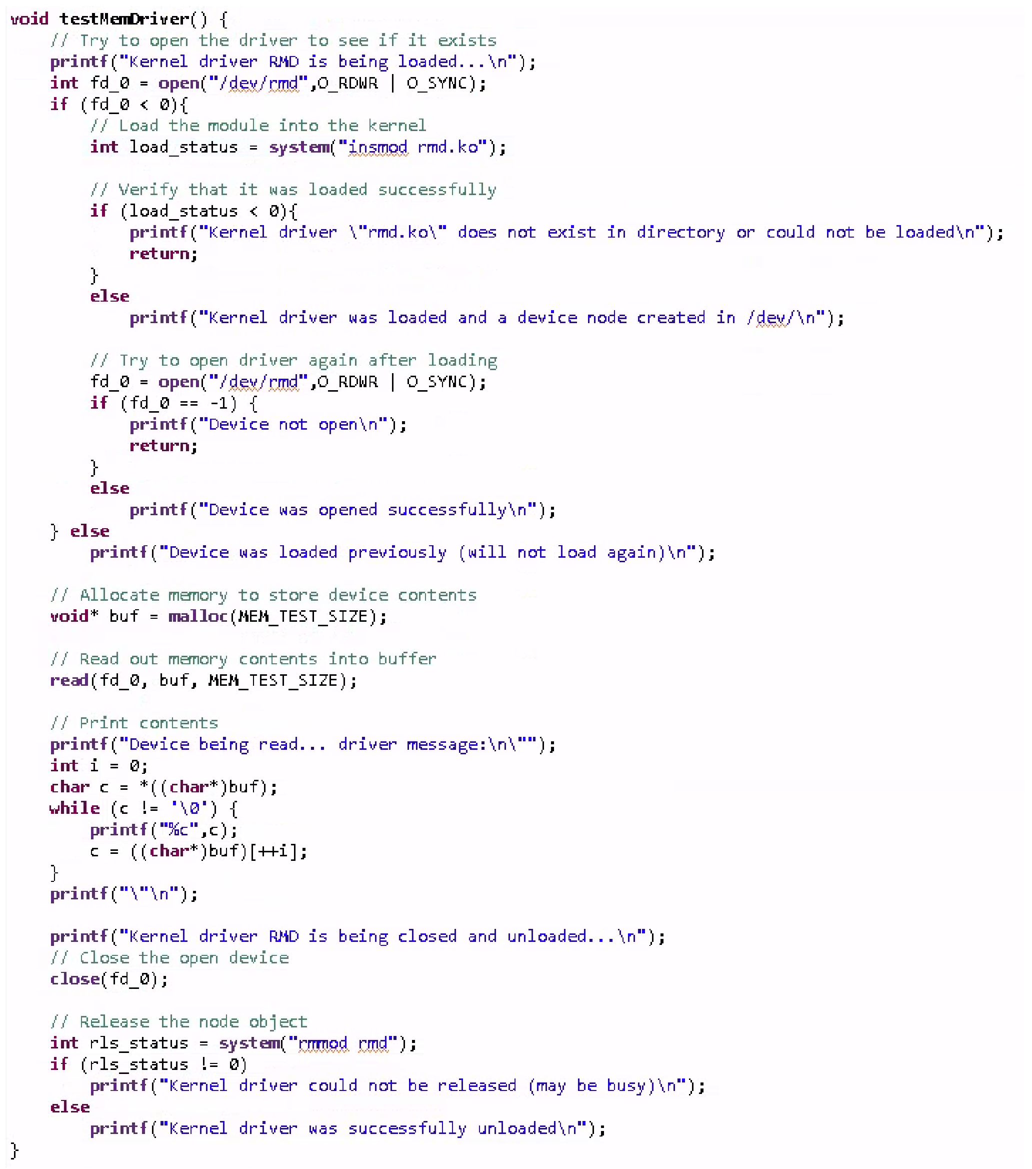
7. Next, **verify that a module node has been created by searching in the /dev/ directory** (this will only happen when the correct driver initialization commands are used as shown in the provided code):

   ```
   $ ls /dev/rmd
   ```

If the module exists, it will be listed. If not, it was either not registered correctly or at all in the driver code.

8. **If the device is present in /dev/, it can be opened, interacted with, and closed from program code** using the “open” and “close” commands as well as others depending on the configuration of the driver.

   Additional Information: The code block shown in Figure 15 was used to test this driver. Note that it both loads the module by sending a system call, and then opens, reads from, and closes it. The result of running the test function is to produce to the console the same text held in the driver message buffer by using the driver’s read function which the kernel associates with the kernel “read” method. Other functions may be used, such as mmap and ioctl provided that the associated function is defined and correctly handled in the driver.



```
void testMemDriver() {
    // Try to open the driver to see if it exists
    printf("Kernel driver RMD is being loaded...\n");
    int fd_0 = open("/dev/rmd",O_RDWR | O_SYNC);
    if (fd_0 < 0){
        // Load the module into the kernel
        int load_status = system("insmod rmd.ko");

        // Verify that it was loaded successfully
        if (load_status < 0){
            printf("Kernel driver \"rmd.ko\" does not exist in directory or could not be loaded\n");
            return;
        }
        else
            printf("Kernel driver was loaded and a device node created in /dev/\n");

        // Try to open driver again after loading
        fd_0 = open("/dev/rmd",O_RDWR | O_SYNC);
        if (fd_0 == -1) {
            printf("Device not open\n");
            return;
        }
        else
            printf("Device was opened successfully\n");
    } else
        printf("Device was loaded previously (will not load again)\n");

    // Allocate memory to store device contents
    void* buf = malloc(MEM_TEST_SIZE);

    // Read out memory contents into buffer
    read(fd_0, buf, MEM_TEST_SIZE);

    // Print contents
    printf("Device being read... driver message:\n\"");
    int i = 0;
    char c = *((char*)buf);
    while (c != '\0') {
        printf("%c",c);
        c = ((char*)buf)[++i];
    }
    printf("\"\n");

    printf("Kernel driver RMD is being closed and unloaded...\n");
    // Close the open device
    close(fd_0);

    // Release the node object
    int rls_status = system("rmmod rmd");
    if (rls_status != 0)
        printf("Kernel driver could not be released (may be busy)\n");
    else
        printf("Kernel driver was successfully unloaded\n");
}
```

Figure 15

3.10 Applying Patches to a PetaLinux Build

Often, PetaLinux releases have bugs or configuration issues that require code patches. Making a patch requires manually creating a differential file between the code to be patched and a new, modified version so that the compiler tools can change existing code without forcing the user to manually change the kernel source code itself (which, in PetaLinux' case is pulled from the kernel repo during the build and often isn't editable). While the process for making a patch isn't discussed here, applying a patch in Petalinux is fairly straightforward as described in the following steps:

1. First, (after going to the project root folder of the project where the patch will be applied, and setting up the terminal) **copy the patch file into the correct source sub-directory in one of the branches of the project-spec/meta-user/ directory** (generally a directory next to a .bbappends file).

   Additional Information: Notice that in this folder contains a few options for you to choose from depending on the type of patch you are applying:

   (a) *recipes-apps* is for modifying applications
   (b) *recipes-bsp* is for modifying device tree and u-boot files
   (c) *recipes-kernel* is for changes to kernel modules or sources

   Bear in mind that these and other directories may or may not exist depending on what stage of build the project is currently at. For example if you want to apply a kernel patch as explained here, you will need to configure the kernel via the kernel configuration menu and then finish building the kernel dependencies. The kernel configuration is opened by running the config command:

   ```
   $ petalinux-config -c kernel
   ```

   To complete dependencies and populate the recipes-kernel directory where a patch can be placed, run the build command:

   ```
   $ petalinux-build -c kernel -x finish
   ```

   Furthermore, notice that each of the branches of this meta-user directory has a slightly different hierarchy depending on the modifiable options it contains. Again assuming we are applying a kernel path as an example, the patch itself should be copied into project-spec/meta-user/recipes-kernel/linux/linux-xlnx/. The next step explains why.

2. After the patch is copied into the project, we need to **modify the associated .bbappends file** so that it will be properly referenced. Open the file and **add a line referencing the patch** as shown:

   (a) The kernel .bbappends is initialized with the following contents:

   ```
   FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
   SRC_URI += "file://devtool-fragment.cfg"
   ```

   (b) We want to add one more line:

   ```
   FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
   SRC_URI += "file://devtool-fragment.cfg"
   SRC_URI += "file://linux-fragment-fix.patch"
   ```

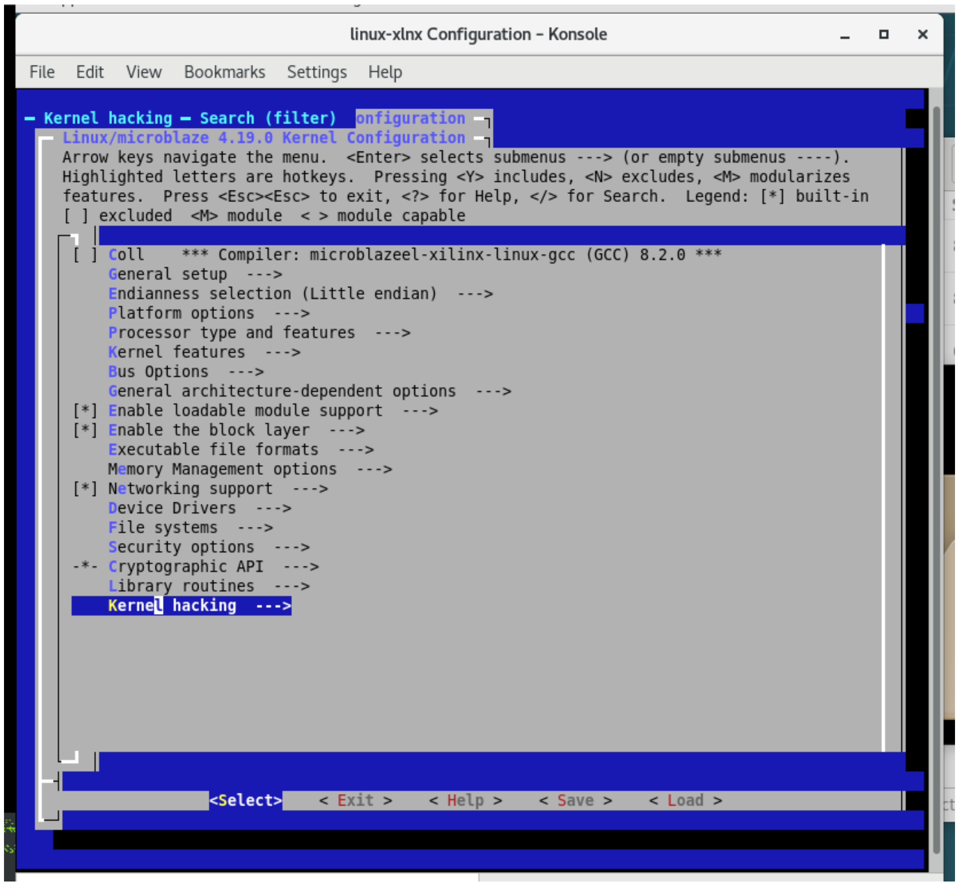
   Additional Information: Using the same kernel patch example, the file should be called something like "linux-xlnx_%.bbappend" and be located, as stated before, one directory up from the patch. These SRC_URI lines by default reference the neighboring directory in which are located the patches. If the patch is somewhere else, it can be referenced using the path to it instead of "file://", but will be different from what is shown here.

3. After adding the patch to the proper place, and adding the reference to the .bbappends file, **build the project as usual for the patch to be applied**. Note that if you clean the project after copying the patch in and changing the .bbappends, the clean process might remove your additions so be sure to check that they are there before running the build command.

4 Working With Other FPGA Boards

Because FPGA specifications differ from family to family, and because development board manufacturers can choose how they want to implement a given FPGA chip, building a PetaLinux image on a board other than the UltraZed will naturally involve some differences from the steps described in this tutorial. This is evident in this tutorial by the fact that we had to change the memory block device in order to boot from the SD card because Avnet elected to organize their memory drivers in the way that they did.

Similarly, considering designs like those that use the MicroBlaze processor, the degree of configurability may be quite a bit different from this tutorial because the processor architecture is different, the size is smaller, and general use-case is unique from the ARM processors on Zynq-7000 and ZynqMP chips. The kernel config window for a MicroBlaze PetaLinux project is shown in Figure 16 to show its contrast from Figure 8.


linux-xlnx Configuration - Konsole
File Edit View Bookmarks Settings Help

```
- Kernel hacking - Search (filter) onfiguration
  Linux/microblaze 4.19.0 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
  features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in
  [ ] excluded  <M> module  < > module capable

    [ ] Coll    *** Compiler: microblazeel-xilinx-linux-gcc (GCC) 8.2.0 ***
        General setup  --->
        Endianness selection (Little endian)  --->
        Platform options  --->
        Processor type and features  --->
        Kernel features  --->
        Bus Options  --->
        General architecture-dependent options  --->
    [*] Enable loadable module support  --->
    [*] Enable the block layer  --->
        Executable file formats  --->
        Memory Management options  --->
    [*] Networking support  --->
        Device Drivers  --->
        File systems  --->
        Security options  --->
    -*- Cryptographic API  --->
        Library routines  --->
        Kernel hacking  --->

                <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 16

If you would like more information about using MicroBlaze with PetaLinux, one tutorial for PetaLinux on a MicroBlaze can be found here.

Even among boards with the same template (such as the UltraZed and Ultra96, for example, which both fall under “zynqMP”), there can be challenging differences because different boards handle hardware differently. While it is impossible to list everything that may be different between boards and templates here, we at least find it a valuable disclaimer to say that this tutorial will differ from others, and that it is likely that setting up PetaLinux on another board will require some adjustment. One such example using the ZCU111 board is described here, including some additional configurations that are necessary.

4.1 PetaLinux on the ZCU111

This section is taken—some text and images exactly and others summarized—from the wiki page written by Mitch Burnett found here under the section “Custom hardware design.” They describe additional configuration steps for a proper build (to get additional peripheral hardware recognized) as well as how to add example designs into the build as well as additional utilities.

1. Using the same PetaLinux tools and setup described previously, **run the following commands**:

   ```
   $ petalinux-create -t project --template zynqMP -n rfdc_linux
   $ cd rfdc_linux
   $ petalinux-config --get-hw-description=<path to HDF/XSA>
   ```

   (a) In “DTG Settings” **change the MACHINE_NAME to zcu111-reva**

   (b) In “Image Packaging Configuration” **change “Root filesystem type” to “SD card”**

   (c) In “Yocto Settings” **change YOCTO_MACHINE_NAME to zcu111-zynqmp**

   (d) **Save and exit** the configuration window

   Additional information: By setting the DTG settings to zcu111-reva this indicates to PetaLinux that there is a default board device tree configuration to load. The most recent PetaLinux UG typically has a current list of all available values. This configuration also changes the default rootfs packaging type from INITRAMFS to SD card. Instead of a RAM disk we are now able to boot from an SD card and any changes will be persistent as they are saved to disk instead of volitale and stored in RAM.

2. In addition to default board configurations this can help indicate to PetaLinux a repository of optional example designs to include in our custom desing. For the RFDC Xilinx has provided some examples that can be compiled in.

   To include these examples in the build **open rfdc_linux/project-spec/meta-user/recipes-core/images/petalinux-image-full.bbappend and add the following lines**:

   ```
   IMAGE_INSTALL_append = " rfdc"
   IMAGE_INSTALL_append = " rfdc-read-write"
   IMAGE_INSTALL_append = " rfdc-selftest"
   ```

3. Next, if the example designs are included, we need to provide a custom bitbake file to compile them with the following steps

   (a) **Make a directory for the bitbake file:**

   ```
   $ mkdir -p project-spec/meta-user/recipes-bsp
   ```

   (b) **Create the file rfdc-selftest_%.bbappend and add this content to the file** (everything after “make all” is part of the make command and should be on the same line):

   ```
   do_compile (){
       make all BOARD_FLAG=-DXPS_BOARD_ZCU111 OUTS=rfdc-selftest
       RFDC_OBJS=xrfdc_selftest_example.o
   }
   ```

4. Next, having configured the hardware, we need to add software libraries. For the ZCU111 the RFDC uses i2c to communicate with the onboard LMK/LMX clocking network that provides on board reference clocks to the data converter. We therefore need to enable the i2c communication driver libraries. **Run the following config command** to open the rootfs configuration window:

   ```
   $ petalinux-config -c rootfs
   ```

(a) In “base” **include the i2c-tools as “built-in”** so that you see the following:

```
[∗] i2c−tools
```

(b) If the rfdc-examples were included as described above then the following user packages (in the “user packages” option from the base menu) will be made available in the rootfs configuration, which you should update as follows:

```
[∗] rfdc
[∗] rfdc−read−write
[∗] rfdc−selftest
```

5. At this point our PetaLinux image is in a minimum viable state that would allow us to finish building the project and deploying the image. However, our Linux image is so vanilla it is almost unusable. So, we also have the option to use the roofts config menu to go shopping and peruse through a list of common utilities and libraries that can be added to our project manually. While still in the rootfs menu, **take a look around and see what additional utilities you may like to add and, when finished, save and exit the configuration window.**

   Additional information: Using the zynqMP template, Yocto build machine and zcu111-reva DTG has given us this barebones setup. But it leaves out many of the standard development libraries and utilities that most of take for granted when working in a Linux development environment. By default there is no sudo, wget, bash, git, make, ...insert-a-common-utility-here. Without manually adding in utilities, many of these standard commands are delivered as a single compiled utility called busybox. So while some of these commands are present they are just accessed in a different way than may be accustomed (man busybox for a better explanation and how to use it).

6. Now, we can **finish building the packaging the image using the usual commands** if the default locations are used or with the additional specifiers shown pointing to the correct location or desired output for each included piece:

```
$ petalinux−build
$ petalinux−package −−boot −−fsbl −−u−boot −−pmufw −−fpga
    or
$ petalinux−package −−boot −−format BIN
−−fsbl images/linux/zynqmp_fsbl.elf −−u−boot images/linux/u−boot.elf
−−pmufw images/linux/pmufw.elf −−fpga images/linux/∗.bit −−force
```

   Again, note that the second packager command option is all on one line. All of these steps *should* build a working image.

7. Having build the images, the next step is code development. An introduction to setting up the Xilinx SDK is shown here. Notice that there are some differences between these steps and those described previously in this tutorial.

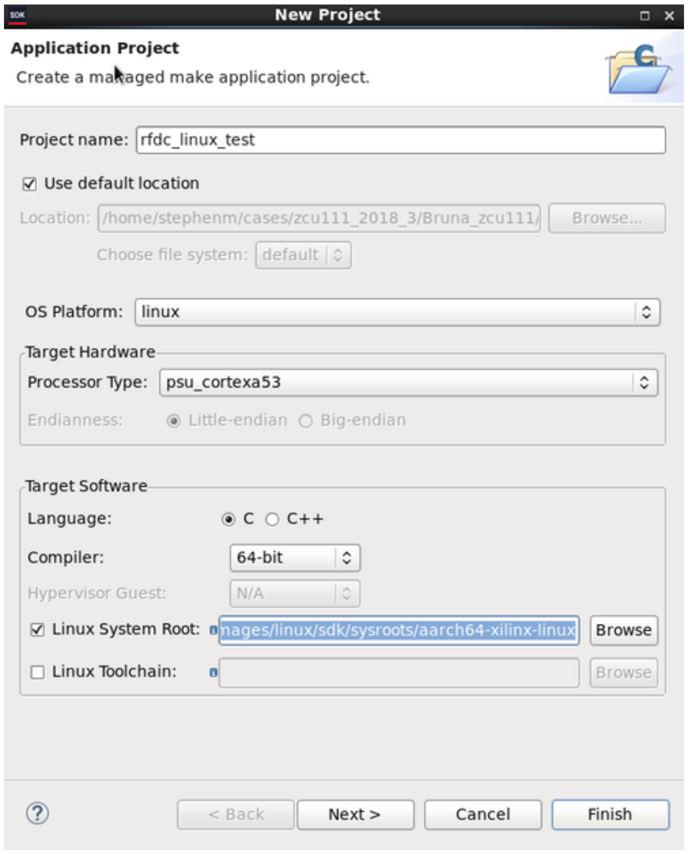
   (a) First, we will **build the SDK platform using the PetaLinux tools** by using the following commands:

```
$ petalinux−build −−sdk
$ petalinux−package −−sysroot
```

   Note that the build command will take a while again to complete, just as it does for the image build.

   (b) Next, **launch the SDK as usual**. Note that if you are using PetaLinux 2019.2 or later, you will have to navigate the Vitis tools, which are different from the traditional SDK and take more setup before an application project can be created. In the following steps we will show how to set up an application project, which for Vitis assumes that the project platform has already been set up.

(c) When the SDK opens **start a new application project and configure it as shown** in Figure 17.


New Project

Application Project

Create a managed make application project.

Project name: rfdc_linux_test

☑ Use default location

Location: /home/stephenm/cases/zcu111_2018_3/Bruna_zcu111/ Browse...

Choose file system: default

OS Platform: linux

Target Hardware

Processor Type: psu_cortexa53

Endianness: ● Little-endian ○ Big-endian

Target Software

Language: ● C ○ C++

Compiler: 64-bit

Hypervisor Guest: N/A

☑ Linux System Root: nages/linux/sdk/sysroots/aarch64-xilinx-linux Browse

☐ Linux Toolchain: Browse

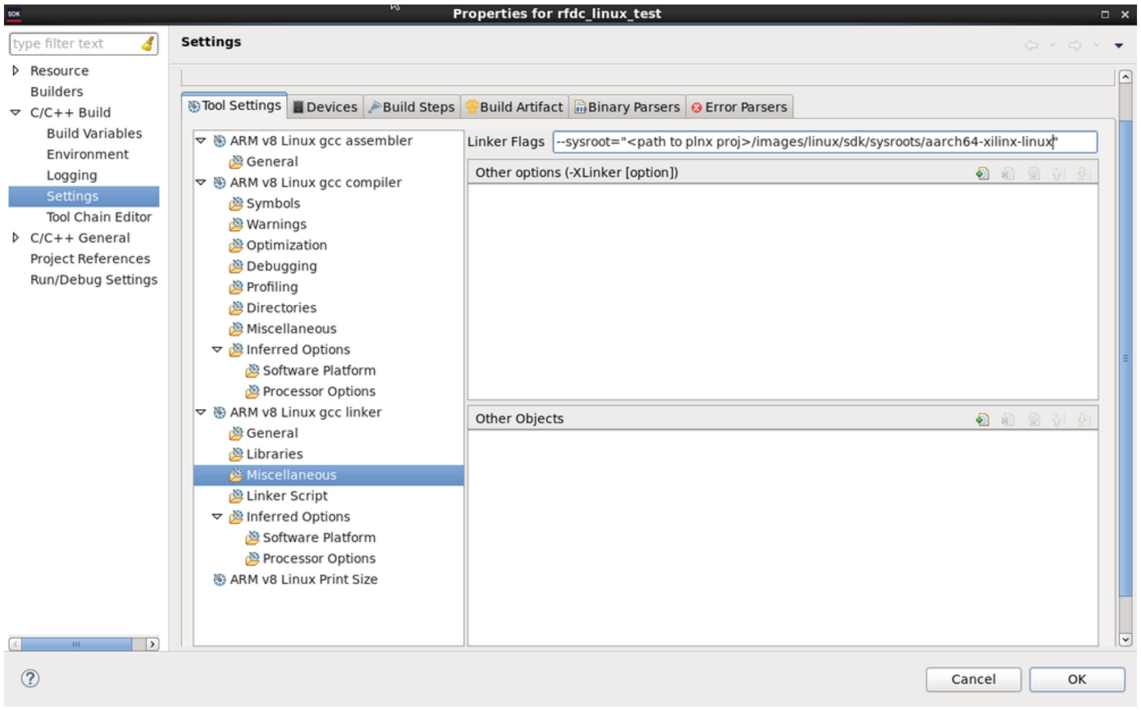
Figure 17

Notice in this configuration that in addition to the normal options used in other parts of this tutorial, it is also important to check the option "Linux System Root" and add the right directory, which is at /images/linux/sdk/sysroots/aarch64-xilinx-linux in the PetaLinux project. This will add additional libraries into the application based on the way that the image was configured.

(d) **Finish the configuration by selecting the option for an empty Linux application and clicking finish.**

(e) After the project is set up, **right click on the root project folder in the left window pane and click "C/C++ Build Settings"** toward the bottom of the dropdown menu.

(f) **Point the linker to the system root that was created by the PetaLinux tools** as shown in Figure 18. Notice that our addition is in the "Linker Flags" text box, which says −−sysroot="<path to plnx proj>/images/linux/sdk/sysroots/aarch64-xilinx-linux". <path to plnx proj> should be substituted for wherever your project root directory is located.


Properties for rfdc_linux_test

Settings

Tool Settings > ARM v8 Linux gcc linker > Miscellaneous

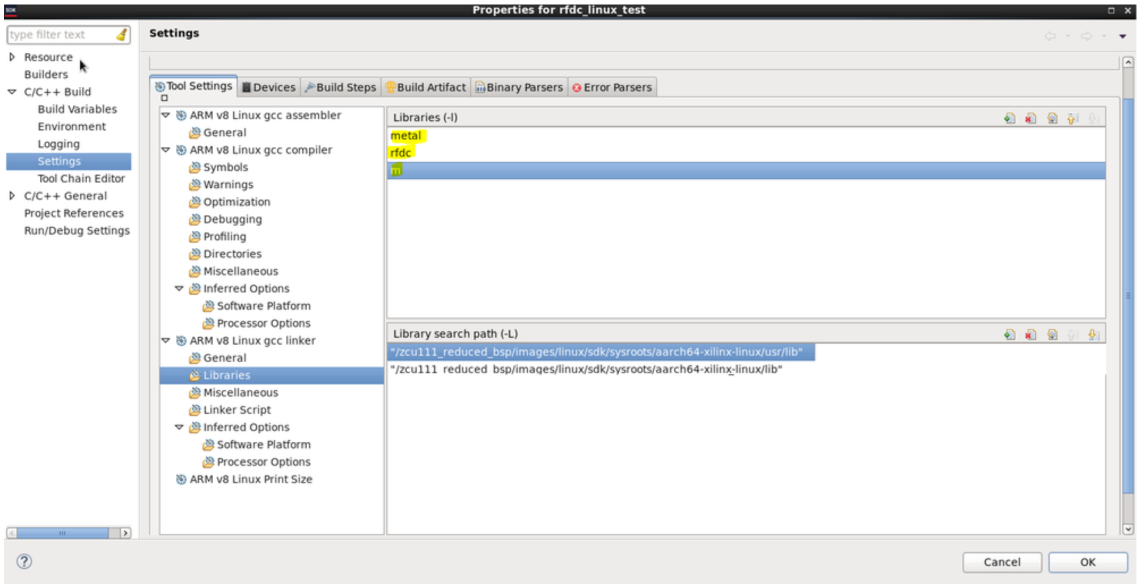
Linker Flags: `--sysroot="<path to plnx proj>/images/linux/sdk/sysroots/aarch64-xilinx-linux"`

Other options (-XLinker [option])

Other Objects

Figure 18

(g) Next, **add in the appropriate linker libraries** as shown in Figure 19, which are "metal," "rfdc," and "m;" and the appropriate search paths which are "<path to plnx project> /images/linux/sdk/sysroots/aarch64-xilinx-linux/usr" and "[...]/lib".


Properties for rfdc_linux_test

Settings

Tool Settings > ARM v8 Linux gcc linker > Libraries

Libraries (-l)

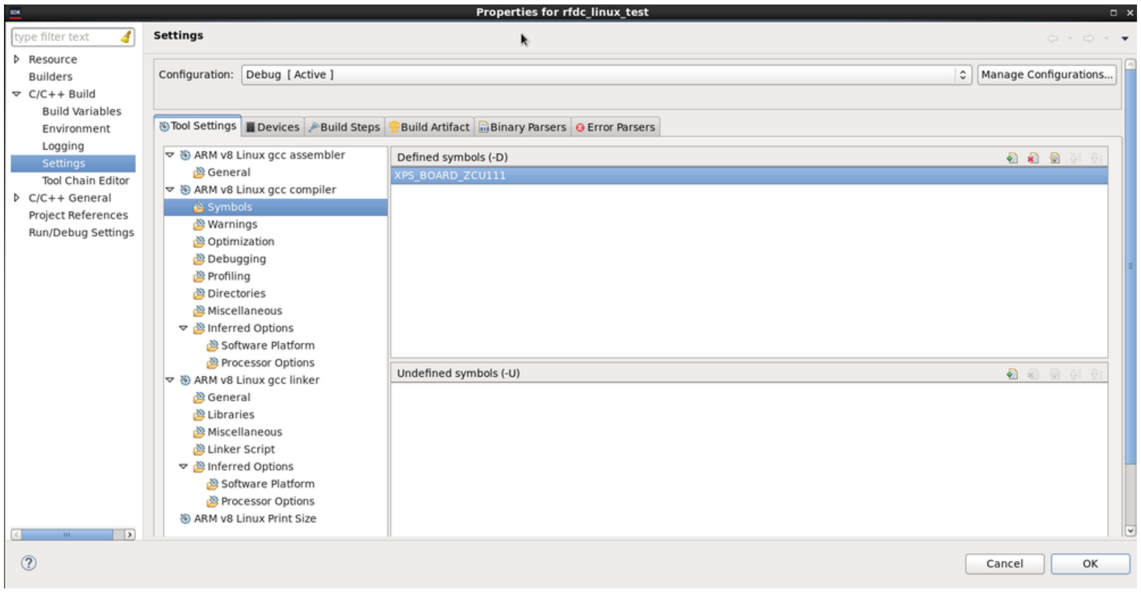
```
metal
rfdc
m
```

Library search path (-L)

```
"/zcu111_reduced_bsp/images/linux/sdk/sysroots/aarch64-xilinx-linux/usr/lib"
"/zcu111 reduced bsp/images/linux/sdk/sysroots/aarch64-xilinx-linux/lib"
```

Figure 19

(h) Last, all of the RFDC driver code require that a pre-processor macro identifying that the ZCU111 is the target board in order to compile. You can **either add the #define XPS_BOARD_ZCU111 macro to your code or you can use the build settings Symbols configuration** as shown in Figure 20 to pass XPS as a -D flag as part of the Makefile that XSDK generates behind the scene.


Properties for rfdc_linux_test

Settings

Configuration: Debug [Active]

Defined symbols (-D)

XPS_BOARD_ZCU111

Undefined symbols (-U)

Figure 20

(i) At this point the setup is complete and you are ready to begin building applications for the RFDC.

5 Using a Custom Carrier Card

The carrier card used for this tutorial was not the Avnet carrier card, though the layout for it was created by mimicking Avnet's version minus the majority of their additional peripherals. In our design we only really needed the Ethernet port, micro USB, and SD reader, as well as a couple of our own peripherals, so in the long term we needed to migrate away from Avnet's card anyway.

Though intuitive in some ways, it is worth saying that our custom carrier card was designed to mimic the Avnet card because doing so allows us to use the default configurations set up by Vivado, PetaLinux, and Avnet's UltraZed-7EV SOM board. If the Ethernet port, for example, were to be connected to different IO, that may also require changes in the configuration of the Vivado project (particularly if the Ethernet controller is in the PL), as well as the PetaLinux image. This alternative route is out of the scope of this tutorial, but we venture to point out that using the defaults where possible will reduce the amount of additional work required to set up a project, if a custom carrier card is to be used.

Furthermore, we note that when starting a Vivado project and selecting the default part to use, it can be confusing which option of the three to choose—the FPGA part alone, the UltraZed-7EV SOM or the UltraZed-7EV Carrier Card. In our projects, we have found that there is some crossover regarding which option works in which scenario, which is to say that the UltraZed-7EV SOM option will likely work for projects on the carrier card, the UltraZed-7EV Carrier Card option will likely work even if not on a carrier card, the FPGA part will likely work as well on and off the carrier card, etc. While it is hard to know all of what Vivado configures when a certain option is chosen, it is assumed that selecting the carrier card option will import the additional peripheral drivers that are present for use on the board such that the user can more easily interface with them. As long as these features are not needed, it shouldn't matter which option is chosen. For our configurations and using our custom carrier card, we have found it safe to use the UltraZed-7EV SOM option without issue (as of yet).

Acknowledgements

This tutorial is based heavily on Xilinx PetaLinux documentation (UG1144 and UG1157) and Xilinx Wiki posts (xilinx-wiki.atlassian.net/wiki/home). Various other resources were used including online sources, work done by various BYU students including David Buck and Mitch Burnett, and Xilinx employee contacts. Where helpful, these resources have been included as links in the tutorial.

Appendix

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/device.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kacen Moody");
MODULE_DESCRIPTION("Reserved memory access driver (RMD)");
MODULE_VERSION("1.00");

#define DEVICE_NAME "rmd"
#define MESSAGE "SUCCESS! This string is stored in driver memory."
#define MSG_SIZE 256

#define DMAJOR 242
#define DMINOR 0
#define DCOUNT 3

/* Prototypes for device functions */
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);

// static int major_num;
static int device_open_count = 0;
static char msg_buffer[MSG_SIZE];
static char *msg_ptr;
static struct class *device_class;
struct cdev device_cdev;
static dev_t first;

/* This structure points to all of the device functions */
static struct file_operations file_ops = {
  .owner = THIS_MODULE,
  .read = device_read,
  .open = device_open,
  .release = device_release
};


/* When a process reads from our device, this gets called. */
static ssize_t device_read(struct file *flip, char *buffer, size_t len, loff_t *offset) {
  int bytes_read = 0;
  /* If we're at the end, loop back to the beginning */
  if (*msg_ptr == 0) {
    msg_ptr = msg_buffer;
  }
  /* Put data in the buffer */
  while (len && *msg_ptr) {
    /* Buffer is in user data, not kernel, so you can't just reference
     * with a pointer. The function put_user handles this for us */
    put_user(*(msg_ptr++), buffer++);
    len--;
    bytes_read++;
  }
  return bytes_read;
}


/* Called when a process opens our device */
static int device_open(struct inode *inode, struct file *file) {
  /* If device is open, return busy */
  if (device_open_count) {
    printk(KERN_INFO "RMD is already open");
    return -EBUSY;
  }
```

Figure 21

```c
  device_open_count++;
  try_module_get(THIS_MODULE);
  return 0;
}


/* Called when a process closes our device */
static int device_release(struct inode *inode, struct file *file) {
  /* Decrement the open counter and usage count. Without this, the module would not unload. */
  device_open_count--;
  module_put(THIS_MODULE);
  return 0;
}


static int __init RMD_init(void) {
  printk(KERN_INFO "Initializing RMD\n");

  /* Fill buffer with our message */
  strncpy(msg_buffer, MESSAGE, MSG_SIZE);
  /* Set the msg_ptr to the buffer */
  msg_ptr = msg_buffer;

  // allocate region for character device
  alloc_chrdev_region(&first, DMINOR, DCOUNT, "rmd");

  // make sure that region is valid
  if (first < 0) {
    printk(KERN_ALERT "Could not register RMD device: %d\n", first);
    return first;
  }
  printk(KERN_INFO "RMD module loaded with device major number %d\n", first);

  // create a device class with this module
  device_class = class_create(THIS_MODULE, "rmd");
  if (IS_ERR(device_class)) {
    unregister_chrdev_region(first, DEVICE_NAME);
    printk(KERN_ALERT "Could not make RMD device class\n");
    return PTR_ERR(device_class);
  }
  printk(KERN_INFO "Device class created\n");

  // link the new device with the allocated region, and adde the character device to the kernel system (which creates a node for use)
  struct device *dev;
  dev = device_create(device_class, NULL, first, NULL, "rmd");
  cdev_init(&device_cdev, &file_ops);
  cdev_add(&device_cdev, first, 1);


  printk(KERN_INFO "Device node created\n");

  return 0;
}


static void __exit RMD_exit(void) {
  printk(KERN_INFO "Closing RMD and cleaning up\n");

  device_destroy(device_class, first);
  class_destroy(device_class);
  /* Remember -- we have to clean up after ourselves. Unregister the character device. */
  cdev_del(&device_cdev);
  unregister_chrdev_region(first, DCOUNT);
}

module_init(RMD_init);
module_exit(RMD_exit);
```

Figure 22