7-2021

Reference Design of an Online Emulation and Hot-Patching Approach for Power Electronic Controller Validation

Estefano Soria Pearson
University of Arkansas, Fayetteville

Reference Design of an Online Emulation and Hot-Patching Approach for Power Electronic Controller Validation

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Electrical Engineering

by

Estefano Soria Pearson
University of Arkansas
Bachelor of Science in Engineering, 2019

July 2021
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

_____
H. Alan Mantooth, Ph.D.
Thesis Director

_____
Roy A. McCann, Ph.D.
Committee Member

_____
Chris Farnell, Ph.D.
Committee Member

ABSTRACT

This thesis aims to develop a reference design of an online security system approach embedded in a power electronic controller for cybersecurity purposes. Cybersecurity in power electronics focuses on reducing vulnerabilities in the system, where most reside in the communication with the hardware devices. Although methods to secure communications lessen the probability and effects of cyber-attacks, discovering vulnerabilities is inevitable. This thesis attempts to provide a fail-safe approach to securing the system by targeting the safety of the power-electronic controller. This approach applies an additional security layer in case of a malicious or accidental controller firmware malfunction.

The online security system is embedded in a controller board consisting of three main hardware components described in this thesis: Serial Communication Interface, Digital Signal Processors, and Field Programmable Gate Array. This security system consists of a real-time emulation and Hot-Patching approach to validate the power electronic controller firmware. This thesis will describe the fundamentals of real-time Hardware-In-Loop emulations and Hot-Patching and evaluate their combined contribution in securing this hardware controller. Under a controlled experiment, purposefully harmful firmware will be loaded and patched on the controller board to evaluate the efficacy of the controller malfunction detection. Parallel to the firmware validation process, the controller will continuously control a power electronics inverter with a known valid firmware. Conducting additional real-time test scenarios will demonstrate the efficacy of the emulation and Hot-Patch. The results from this experimental setup will establish the conclusions drawn, followed by recommendations for future work and discussion of enhancements.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. H. Alan Mantooth, for all the guidance and knowledge, that he provided for me to become a better engineer and a better person. His approach when mentoring his students and providing advice is more personal and different from others, making this opportunity so unique. I would also like to thank Dr. Chris Farnell for the guidance and advice that he has provided for me. Working closely with Dr. Chris Farnell has allowed me to learn more in this program than in the four years of my undergraduate. I would also like to thank Dr. Roy McCann for his advice and comments on the previous projects. Especially the comments about my undergraduate senior design presentation inspired me to improve my presentation and explanation skills. Lastly, I would like to thank Ammar Khan, Nick Blair, Justin Jackson, Brady McBride, Rosten Sweeting III, and Sloan Becker for the help, support, and knowledge.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

The electric power grid is a critical component in humans' everyday lives, as it is used for basic human needs such as water, light, temperature regulation, and more. The electric grid generates large amounts of electric power transmitted and distributed to communities through different power electronic devices. As the electric power grid increased in complexity, it has become more challenging to integrate and monitor grid-connected devices, which increased the potential of device failure. Therefore, grid-connected devices are becoming more connected as part of the Internet of Things (IoT), which allows for communication and monitorization of all connected devices through the Internet [1]. This integration has helped automate control for grid-connected devices to reduce potential device failures. Grid-connected devices consist of communication, control, and power electronics hardware. In this setup, commands are communicated to the power electronics controller, which adjusts the output of the power electronics hardware. In this three-layer setup, the communications layer has become essential to maintaining reliability within the electric power grid since it is the bridge between the user and the system. In addition, the communications layer monitors the system by communicating system events to the user.

Cyber threats focus on exposing the vulnerabilities of the communications layer. Cyber attackers gain access to the communication layer by connecting to the network via different methods, including Phishing, Man in the middle, Denial of service (DoS), Structured Query Language (SQL) injection, Domain Name System (DNS) Tunneling, and more. If an attacker gains control of the communications section, they can communicate malicious commands or

firmware to the controller, harming the power electronic device. There are different cybersecurity methods for the grid, which involve identifying the main types of attacks and creating risk assessments. Most cybersecurity methods focus on securing the communications layer since it is the primary contact with the system. However, vulnerabilities are constantly being discovered, bringing severe concern with the grid's reliability [2].

The severity of grid vulnerabilities is that cyber-attack threats can cause power outages throughout a city or a country, as happened in the Ukraine attacks in 2015 and 2016 [3]. Recently, the colonial pipeline in the south-east United States experienced a ransomware attack which caused the pipeline operations to shut down [4]. As [4] demonstrated, multiple cyber-attacks caused physical operations to shut down in 2020 by ransomware, showing the importance of cybersecurity. Currently, it is essential to implement cybersecurity in the design process of power electronics control systems. The electric grid is becoming more vulnerable to cyber-attacks, which are focused on communication networks. Cyber-attack examples in [3, 4] also demonstrate that the system control has little to no security when the communication networks are hacked. Once the communication networks are hacked, the whole system collapses, causing power outages or the Colonial Pipeline Co. to shut down, impacting millions of people. To secure against cyber threats, investigators focus on protecting the data flow through the communication networks by simulating attack scenarios and applying security on top of the physical system.

This work attempts to secure grid-connected devices by targeting the power electronics controller layer. Hot-Patching and Emulations inspire this work, two concepts applied in cybersecurity research for the electric power grid. These two concepts are embedded in the power electronics controller to secure the system by online detection, mitigation, and

performance validation of patched control firmware. This approach separates the vulnerabilities in the communication network from the vulnerabilities in the power electronic controller. The vulnerable power electronics controller would command the system to shut down during an attack. However, with the proposed approach, the controller rejects invalid firmware to be patched on the controller board and applies a known valid backup firmware on the controller board without interrupting the continuous control of the system. This approach assures that the system does not shut down during an attack.

1.1.1 Emulations and Digital Twins

Recently, emulations and digital twins have received much attention concerning cybersecurity for power electronics. The purpose of emulations and digital twin in this subject is to predict outcomes from cyber-attacks on a system by creating a digital version of the physical system. This digital version replicates the performance of the physical system as an identical twin. There are many definitions of digital twins, depending on the physical system under evaluation. [5] Describes the different levels and definitions of digital twins, where each level describes the modeling accuracy that imitates the physical system functionality and performance. The digital twin levels are labeled as the following: pre-Digital twin, Digital Twin, Adaptive Digital Twin, and the Intelligent Digital Twin [5]. These digital twins have five dimensions: a physical section, a virtual section, data, connections, and service modeling [6]. Digital twins are continuously being developed without including all the mentioned levels and dimensions. However, developments that include any of the levels and dimensions can be defined as digital twins.

With the use of digital twins, grid-connected devices are evaluated offline, conducting cyberattack scenarios and resembling the output of a physical system. This offline evaluation is a

method to secure the grid from cyberattacks; however, constantly discovering vulnerabilities in communications remains a concern. This work applies basic concepts of digital twins and emulations in the physical controller. Furthermore, this work attempts to provide additional firmware performance validation during a cyber-attack through the online emulation approach. The first step of a digital twin is a real-time emulation of the physical system. Therefore, the embedded approach consists of an online emulation that resembles a power electronic inverter for firmware performance validation.

1.1.2 Hot-Patch

Hot-Patching is a concept of updating the firmware of controller hardware without causing any downtime to the system. This concept has been part of multiple applications but with many restrictions. Hot-Patching contains many risks to systems because it can be unpredictable. As such, most attempts consist of specific applications with many restrictions. For example, [7] consists of a Hot-Patch that is possible with fixed parameters and specific program architecture. This Hot-Patch strategy was purposefully made this way to provide evidence that it is possible to Hot-Patch and represent a reference program structure for future Hot-Patching developments. However, these attempts demonstrate the difficulty of Hot-Patching using different firmware program structures. Furthermore, these attempts would be dangerous to a grid-connected power electronics controller board since the Hot-Patch would cause unpredictable outcomes on the system, including a shut-down.

Although the risks are a significant obstacle, Hot-Patching would bring tremendous value to grid-connected devices. Program vulnerabilities within grid-connected devices are discovered constantly and bring concern with cyber threats [2]. When vulnerabilities are discovered, a process is performed to patch the vulnerabilities, which takes time. This process includes

patching schedules that are used to patch vulnerabilities on multiple devices in the system. [8] Evaluates a patching schedule that reduces patching delay. However, these methods do not consider downtime during the firmware patch. Once the firmware patch has been created, the program must be loaded on one device at a time, which requires the device to be offline. Furthermore, it is impossible to turn all unpatched devices off simultaneously because it could cause extreme instability in the electric grid. Consequently, the unpatched devices are vulnerable to cyber-attack threats. In addition, discovered vulnerabilities can become public knowledge on the same day, making quick vulnerability patching essential to the electric grid [9].

During an attack, the communications layer may be compromised, which causes unpatched controllers to be tremendously vulnerable. The attacker could send hazardous commands to shut down vulnerable grid-connected devices. The Hot-Patching approach proposed in this work is a backup system that protects the controller from cyber-attacks. Although it may not be a complete solution that eradicates cyberattack threats, it provides time to gain back command of the communications layer from the cyber-attack while maintaining stability and security in the system. The construction of the Hot-Patch applies to different program structures, which makes this approach flexible and applicable to different power electronic devices. In other words, the firmware used to patch vulnerabilities does not need to be programmed in any specific way, making this approach flexible to patching various firmware architectures.

1.2 Thesis Objectives

The purpose of this research is to discover an alternative approach to secure a power electronics device from a cyber-attack through the controller board instead of the communications. This work is a reference design incorporating power electronic emulations,

Hot-Patching, and controller firmware validation that collaborate in securing the device online. This thesis is divided into chapters that describe the background that motivates this design approach, the architecture of the security design, and the experimental results that analyze each design feature. This work is created to reference how a multifunctional security system can be embedded in the controller board that validates the controller firmware online. This security design will validate control firmware before it is activated to control the power electronics. The validation process does not interrupt the continuous control of the power electronics with the known valid firmware.

1.3 Thesis Organization

The following list sequentially outlines this thesis:

- Chapter 1 motivates the approach, which consists of Digital Twin concepts and Hot-Patching concepts.
- Chapter 2 defines the background concepts that are incorporated in the design. This does not describe the primary components of the design, but it demonstrates the hardware architecture used and their combined contributions.
- Chapter 3 explains the driving components of the overall security reference design embedded in the power electronics controller board. All components are digitally programmed modules that are processed simultaneously, where all modules collaborate in securing the power electronics device from dangerous controller firmware patches. In addition, these modules collaboratively compose emulation, Hot-Patching, and other online security measures to validate the controller firmware functionality.

- Chapter 4 demonstrates an experimental setup that analyzes the efficacy of the validation process of this design. This setup examines the primary features of the design, which are emulation, Hot-Patching, and an additional firmware validation feature.
- Chapter 5 concludes the analysis of all security design features and recommends specific procedures when applying this design on different power electronic devices for future work.

1.4 References

[1] S. Singh and N. Singh, "Internet of Things (IoT): Security challenges, business opportunities & reference architecture for E-commerce," 2015 International Conference on Green Computing and Internet of Things (ICGCIoT), 2015, pp. 1577-1581, doi: 10.1109/ICGCIoT.2015.7380718.

[2] “National Vulnerability Database,” NVD, Jun-2021. [Online]. Available: https://nvd.nist.gov/vuln/full-listing. [Accessed: 07-Jun-2021].

[3] Q. Wang, W. Tai, Y. Tang and M. Ni, "Review of the false data injection attack against the cyber-physical power system", IET Cyber-Physical Systems: Theory & Applications, vol. 4, no. 2, pp. 101-107, 2019.

[4] “Colonial Pipeline Ransomware Attack Rattles Power Industry, Renews Vulnerability Concerns,” POWER Magazine, 2021. [Online]. Available: https://www.powermag.com/colonial-pipeline-ransomware-attack-rattles-power-industry-renews-vulnerability-concerns/. [Accessed: 26-Jun-2021].

[5] V. V. Makarov, Y. B. Frolov, I. S. Parshina and M. V. Ushakova, "The Design Concept of Digital Twin," 2019 Twelfth International Conference "Management of large-scale system development" (MLSD), 2019, pp. 1-4, doi: 10.1109/MLSD.2019.8911091.

[6] F. Tao, H. Zhang, A. Liu and A. Y. C. Nee, "Digital Twin in Industry: State-of-the-Art," in IEEE Transactions on Industrial Informatics, vol. 15, no. 4, pp. 2405-2415, April 2019, doi: 10.1109/TII.2018.2873186.

[7] H. Jeong, J. Baik and K. Kang, "Functional level hot-patching platform for executable and linkable format binaries," 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2017, pp. 489-494, doi: 10.1109/SMC.2017.8122653.Hot-Patch Examples

[8] F. Zhang and Q. Li, "Dynamic Risk-Aware Patch Scheduling," 2020 IEEE Conference on Communications and Network Security (CNS), Avignon, France, 2020, pp. 1-9, doi: 10.1109/CNS48642.2020.9162225.

[9] M. Shahzad, M. Z. Shafiq, and A. X. Liu, “A large scale exploratory analysis of software vulnerability life cycles,” in International Conference on Software Engineering (ICSE), IEEE, 2012. DOI: 10.1109/ ICSE.2012.6227141.

CHAPTER 2

BACKGROUND

2.1 Introduction

From the previous chapter, the motivation for this work is to apply an additional layer of security embedded in the power electronics controller. This reference design is motivated by two main concepts: emulation and Hot-Patching, which will perform online. The previous chapter mentioned how cybersecurity is focused on protecting communications with power electronic devices. The communication with the device goes through the power electronic controller that receives communicated commands and processes the control algorithm to manage the power electronics device's power output. The control process goes through the communications first, then through the controller board, and finally the power electronics device. This work applies security on the second step of the control process, the controller board, which means the focus of the design is not within the communications. The following sections will explain the concepts of the power electronics device and the architecture of the considered controller board. These sections will also explain Real-Time Simulations (RTS) applied to the power electronics device and the controller board. These concepts are used to build the online emulation since it is an RTS of the power electronics device and the control board. Also, the control architecture is specific to enable the incorporation of online emulations, Hot-Patching, and firmware validation in a single power electronics controller board.

2.2 Power Electronic Inverter Overview

The development of this controller security approach considers the vulnerabilities of grid-connected devices, specifically 3-Phase inverters. These inverters are power converters that input a Direct Current (DC) link supply to produce a 3-phase Alternating Current (AC) sinusoidal

output, where each phase is shifted 120 degrees from the other. Figure 2.1 shows a diagram of a 3-phase inverter consisting of four sections: the DC input voltage supply, six switches, a 3-phase filter, and a 3-phase resistive load. The six switches produce the 3-phase AC sinusoidal output in this setup, where each phase labeled as A, B, and C uses two switches. S1 and S2 are for phase A, S3 and S4 are for phase B, and S5 and S6 are for phase C.

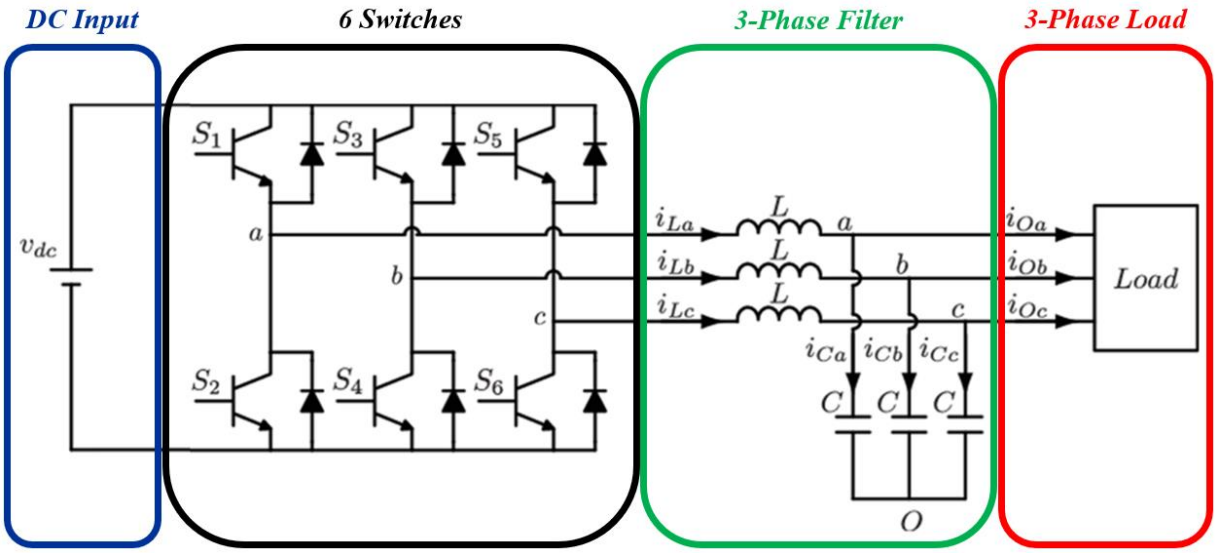

Figure 2.1. Fig3PhaseInvWithSections.

The inverter concept will initially be explained without the filter and for a single-phase, shown in Figure 2.2. There are three possible switching combinations for each phase, as shown in Table 2.1. From these combinations, the inverter will output either zero or the DC input voltage. From the switching combinations, S1 and S2 are never on simultaneously since that would cause a shoot-through. A shoot-through is an event where the positive and negative ports of the power source are connected without any resistance, causing the source to burst. A shoot-through event would be catastrophic in a high-power electronics device since it would explode, causing severe damage. Therefore, these switches are functioning in complimentary mode. The term on-time refers to the switch S1 being on because this switch causes the inverter to output

the DC input voltage. Figure 2.3 shows the AC voltage output of the inverter, where the on-time varies. This voltage output passes through a filter shown in Figure 2.4, soothes the voltage output to create the sinusoidal shape. This sinusoidal output is created by the combination of the fixed filter parameters and the varying on-time. As the on-time increases, the output voltage increases, and when the on-time decreases, the output voltage decreases. Therefore, the shape of the inverter output is controlled by the on-time. [1] can be referenced for additional information on the inverter filter design.

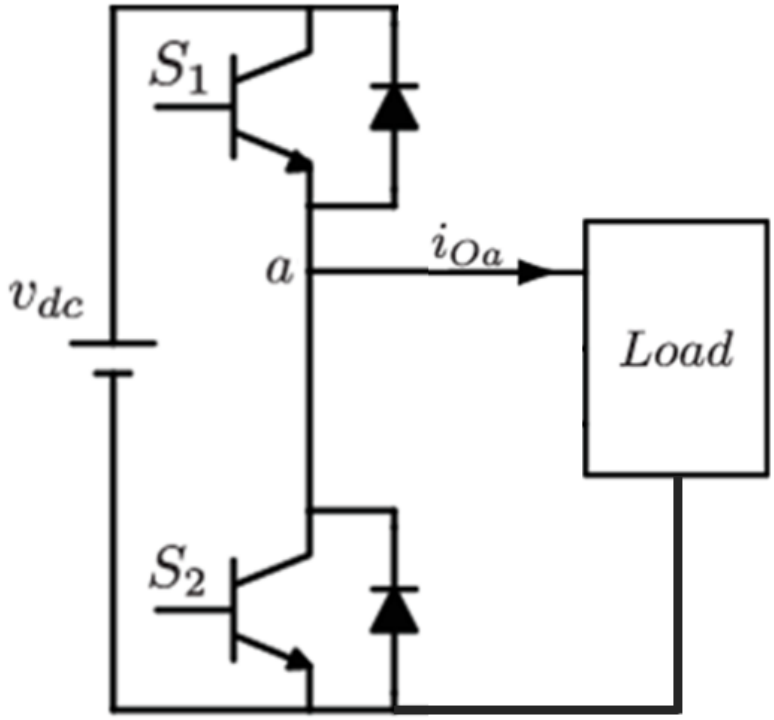

Figure 2.2. Single Phase Inverter

Table 2.1. Single-phase inverter switching combinations

Vin	SW1	SW2	Vout
Vdc	OFF (0)	OFF (0)	0
Vdc	OFF (0)	ON (1)	0
Vdc	ON (1)	OFF (0)	Vdc

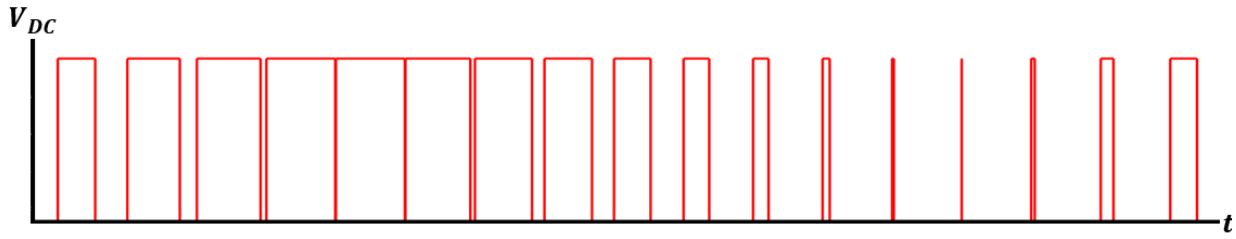

Figure 2.3. AC Inverter Output Voltage

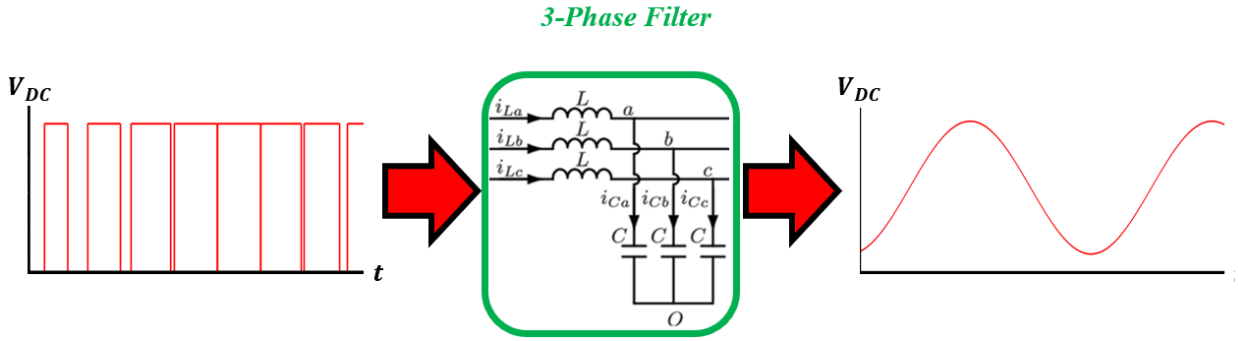

Figure 2.4. Filtered AC Inverter Output Voltage

A control signal must be applied to trigger the inverter switches to control the output voltage of the single-phase inverter. This control signal is known as Pulse Width Modulation (PWM). Figure 2.5 illustrates the PWM time durations: the on-time (Ton) and switching period (Tsw). On-time is the amount of time that the switch is on, and the switching period is the combined duration of the on-time and the off-time. When the single-phase inverter output is not filtered, the voltage output value will be the DC input voltage during the on-time. This non-filtered on-time voltage illustrated in Figure 2.3 has the same value as the DC input voltage, regardless of the on-time duration. However, the filter causes the on-time voltage value to be a percent of the DC input voltage. As the on-time increases, the filtered output voltage increases

until the on-time reaches a maximum limit of nearly 100%. At this instance, the filtered output voltage becomes the same value as the DC input voltage because the on-time is 100%, making the filtered output voltage 100% of the DC input voltage. By increasing and decreasing the on-time, the output voltage can be formed as a sinusoidal waveform, representing an AC output. Eq. (2-1) can be referenced to demonstrate the relationship between the on-time and switching period as a mathematical representation. This relationship is known as the Duty Cycle percent. This Duty Cycle percent is directly applied to the filtered output voltage of the inverter to gain a percent of the DC input voltage. If the Duty Cycle is at 100%, then the on-time is the same as the switching period, and the filtered output voltage of the inverter is the same as the DC input voltage.

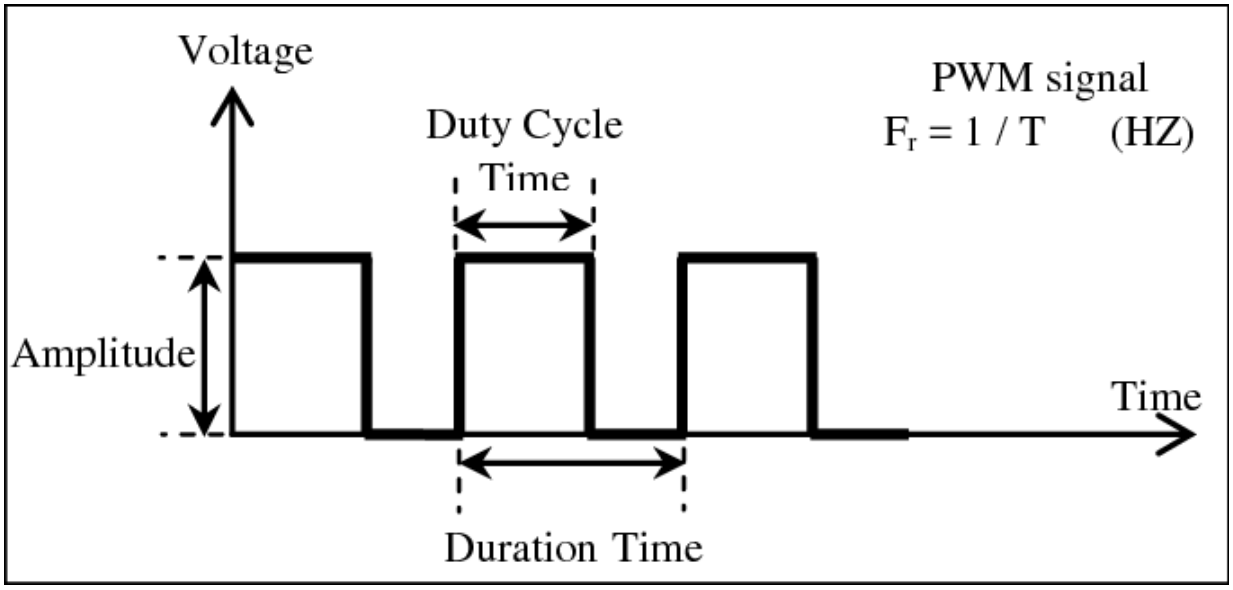

Figure 2.5. Pulse Width Modulation Time Components

$$Duty\ Cycle\ (\%) = \frac{T_{ON}}{T_{SW}} \qquad (2\text{-}1)$$

2.3 Power Electronic Controller Board

The considered power electronic control board is shown in Figure 2.6, referred to as the Universal Controller Board (UCB). The UCB includes different components and processors for various applications. The primary components of the UCB that are used for the security design approach will be described in more detail. These include a Digital Signal Processor (DSP), different communication ports, and a specific type of Programmable Logic Device (PLD) which can be referred to as a Complex Programmable Logic Device (CPLD) or Field Programmable Gate Array (FPGA). CPLDs and FPGAs have similarities and differences, but the specific device used applies features found on both, explained in more detail below.

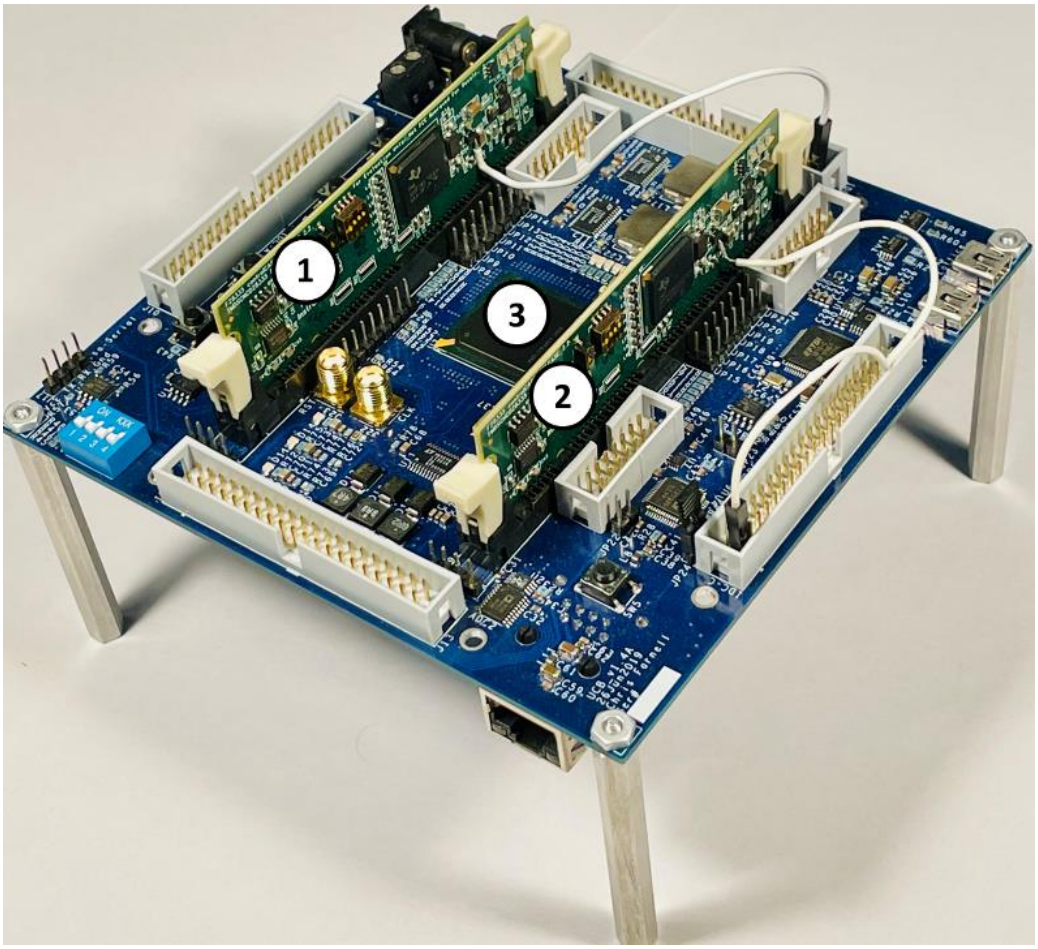

Figure 2.6. UCB

The communication ports consist of a Universal Asynchronous Transmitter and Receiver (UART). These ports are connected to the User Interface (UI) that resides in a computer. The UART consists of the receiver (RX) port and the transmitter (TX) port, where their connections are shown in Figure 2.7. The connection is via a USB cable that allows data to transfer between the UI and the UCB. The UART ports will transfer 10 bits of data one bit at a time. When data is communicated, the UART line goes “low,” corresponding to a digital 0. When no data is being sent, the UART lines are at a digital “high” state corresponding to a digital 1. The UART communicates data in a package consisting of a start bit, a data byte (8 bits), and a stop bit. The

start bit is a digital 0, and the stop bit is a digital 1. The UI and the UCB communication protocols are designed to “understand” what the start and stop bits are and use those bits to separate each byte of data. The receiver knows what bits from the whole package are the actual byte of data. The UART communication consists of two devices that perform in different clock frequencies, which means that both devices require a unique method of synchronizing the data transmission. This method is called baud rate, which means bits per second. Having different clock frequencies, both devices require a setup that will send the same number of bits per second. The specific communication protocol will be explained in chapter 3.

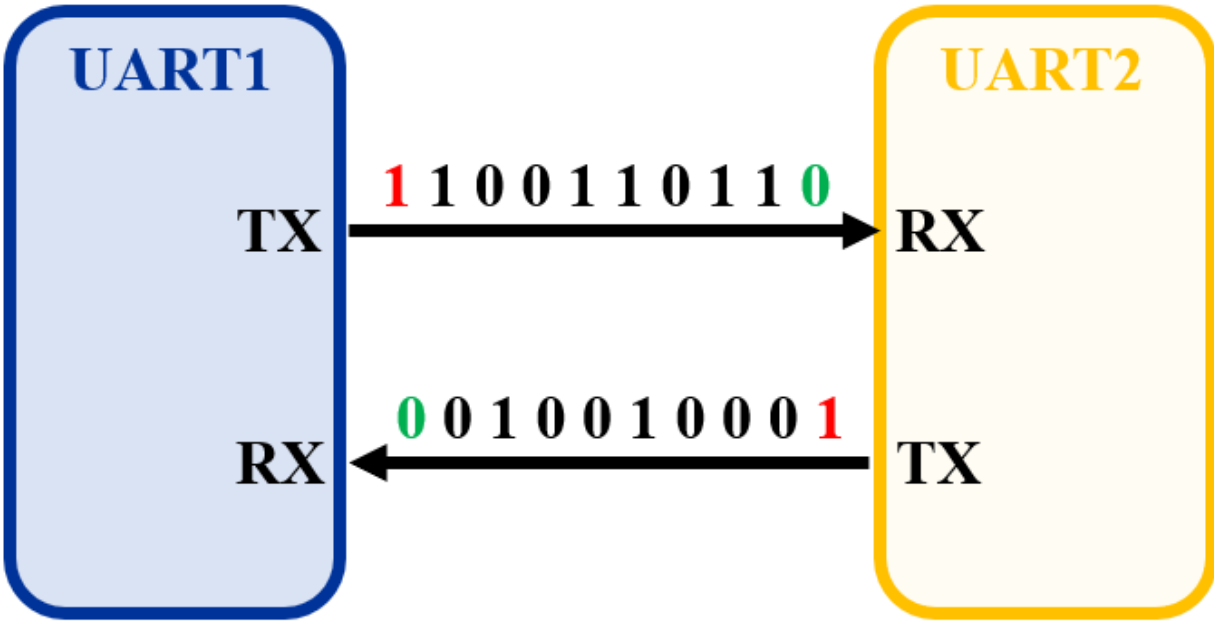

Figure 2.7. UART Connections and Data Structure

2.3.1 Field Programmable Gate Array

The security design is embedded in a Field Programmable Gate Array (FPGA) part of the UCB. This architecture includes the MachXO2 from Lattice, a non-volatile FPGA that includes features typically found on Complex Programmable Logic Devices (CPLD). Typical FPGAs are complex, volatile chips consisting of hundreds of thousands to millions of logic blocks in a single chip, making them physically large. FPGAs are programmed in a Static Random-Access Memory (SRAM), which does not save the program content once the device is off, making the device volatile. The architecture of FPGAs consists of input and output blocks, logic blocks, and additional hardware blocks such as Block RAM to store data and DSPs for mathematical computations. All blocks in the FPGA are interconnected through channeled wires that can be used as a routing fabric. The routing fabric is reprogrammable to connect different signals and ports to either logic blocks or I/O blocks. FPGAs take longer to be programmed and have a slower unpredictable processing time because of their complex structure. CPLDs are simpler, non-volatile chips that have far fewer logic blocks than FPGAs. CPLDs are programmed in an Erasable Programmable Read-Only Memory (EPROM), allowing the CPLD to keep the program data saved when it turns off. CPLDs have a global interconnection block that connects all the logic blocks and I/O blocks. CPLDs generally do not include other hardware blocks like the internal RAM and DSP blocks, simplifying the design capability. Their simple architecture allows them to perform fast with predictable timing. The MachXO2 is the considered CPLD because it consists of CPLD and FPGA characteristics.

The MachXO2 will be referred to as a CPLD for consistency. This chip is programmed in a Flash or EPROM, which makes it non-volatile and secure. The CPLD has security options that allow the CPLD flash memory to be inaccessible if needed, making it difficult to access the

programmed CPLD design. Figure 2.8 displays the top view of the CPLD architecture, which includes the following components: Configuration Flash Memory (CFM), System Clock, I/O Banks, Embedded Block RAM (EBR), Programmable Function Units (PFU)s, and more. CFM is the memory space where the design data is stored to configure the CPLD. This CPLD has an internal system clock with a maximum frequency of 400MHz. For the security reference design, the clock frequency is set at around 25MHz. This system clock includes a Phase Lock Loop (PLL) feature synthesizing different clock frequency domain in the design, making the timing predictable. More information about PLL features in the CPLD can be read in [2]. The EBR is a critical component used in this reference design since it is storage space for design parameters, communicated commands, and firmware files. The use of EBR is explained in chapter 3, which will consist of a Dual Port Block RAM. The PFUs are composed of 6864 Look-Up Tables (LUT) used to create complex digital designs. Each LUT is simple, but with thousands combined, they can be helpful in enhanced control designs. These PFUs are useful for timers, arithmetic, registers, and other design elements. All of the mentioned CPLD internal blocks are interconnected, allowing seamless rerouting of the signals and ports of the CPLD. This routing fabric is used on all components of the security reference design. This CPLD has hardwired ports for different communication methods such as I2C, SPI, and UART. The reference design consists of many counters, timers, different registers, logic gates, multiplexers, state machines, large data distribution, fast signal rerouting throughout the UCB, and much more. Therefore, the MachXO2 is applied to process all design needs fast, efficiently, and in a small chip. Additional information about the MachXO2 can be found on [3].

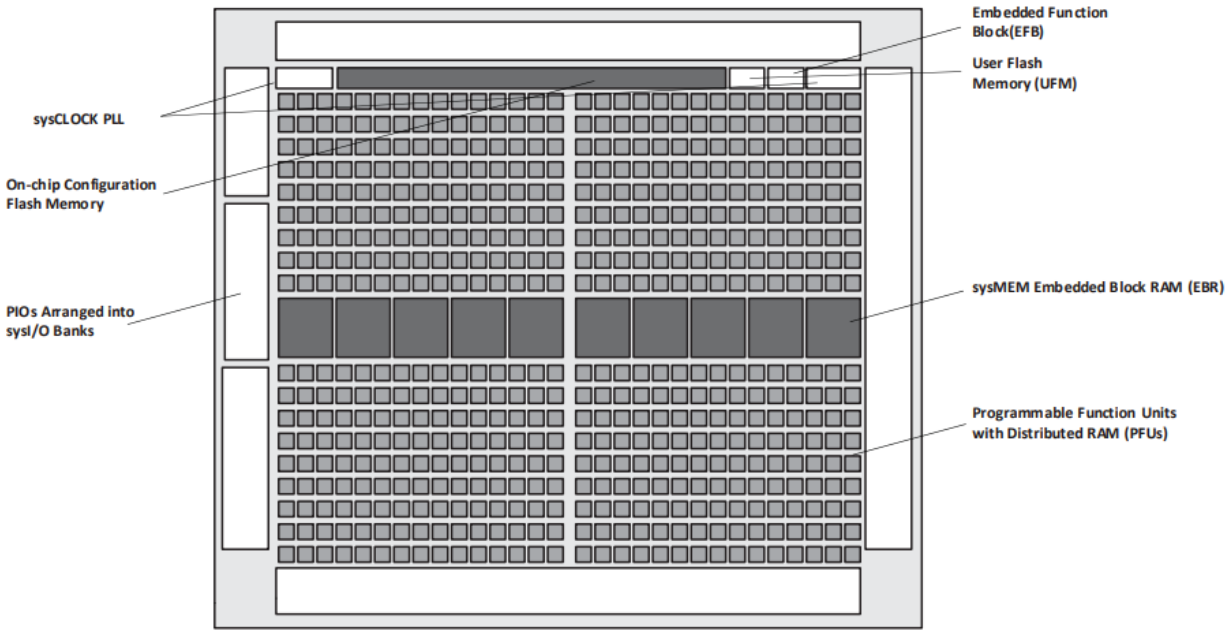

Figure 2.8. CPLD Top-Level Architecture

2.3.2 Digital Signal Processors

Power electronic controller boards regulate the behavior of power hardware devices. For this design, the considered power hardware device is a 3-Phase inverter regulated by a controller board. The controller board contains a program with the PWM specifications to trigger the inverter switches, regulating the AC output. The PWM generation is programmed in the DSP. As introduced, the UCB consists of two identical DSP cards. One DSP is used as the active controller for the inverter, and the second is for the firmware evaluation or backup.

PWM signals control the output of the inverter by turning the switches on and off. Therefore, the PWM signal sends two types of commands, one or zero, to trigger the switch on or off. Figure 2.9 illustrates the general concept of a Sinusoidal Pulse Width Modulation (SPWM) generation, containing a sinusoidal reference in red, a switching frequency reference in blue, and a fixed switching period (Tsw). The SPWM is generated by comparing a sinusoidal reference to a switching frequency reference, where both reference waveforms oscillate between

zero and 1. When the sinusoidal reference is higher than the switching frequency, the SPWM outputs a 1 to turn the switch on; otherwise, the SPWM outputs a 0 to turn the switch off. Thus, the sinusoidal waveform reference allows the SPWM on-time to vary, which causes the inverter to have an AC output.

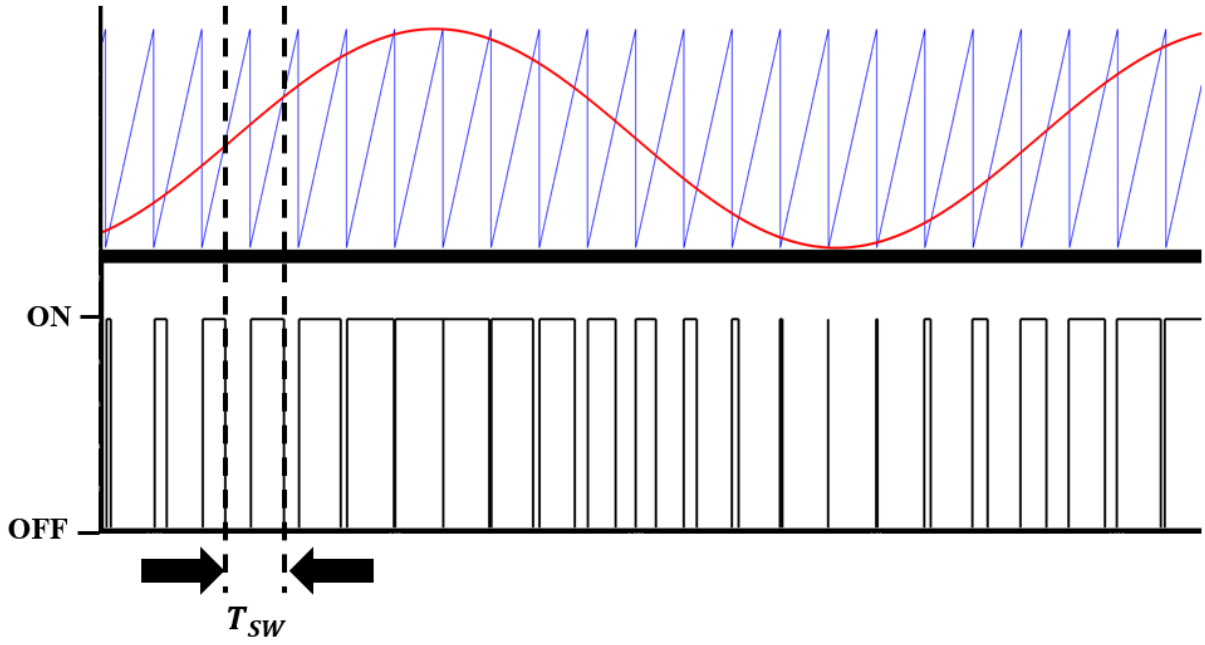

Figure 2.9. SPWM Generation Concept Waveform

The design process of control systems for power-electronic inverters considers the different outcomes of the inverter output and regulates those outcomes through control schemes. In other words, the focus of the control system design is to achieve the desired outcome of the inverter by controlling and regulating the inverter functionality. Conversely, the design process of this security system approach considers the effects of the control scheme on the inverter. Therefore, this design focuses on generating the inverter outcomes by evaluating the PWM input of the controller. The control scheme designs become more convoluted and are not the focus of the security design approach. The design incorporates a generic control scheme as an example to evaluate standard firmware functions. The validation can be applied to more enhanced control

schemes, but it should only be understood that the firmware provides PWM control to the power electronics for this reference design. More information on enhanced PWM control can be found in [4, 5], which provides examples of closed-loop control schemes and space vector modulation algorithms.

The PWM program is designed in Code Composer Studio (CCS), the programming software for control boards. The twin control boards in the UCB are Delfino F28335 Texas Instrument DSP cards. This DSP model is widely used for power electronics because of its capabilities. This DSP can process advanced control system programs for power electronics at high speed. For this reference design, the DSP control program is simple which does not require the full potential of this DSP. This program is only used in testing to demonstrate the functionality of the security design. However, the reference design may be incorporated with more advanced control system programs; therefore, the design will be embedded in hardware capable of processing different advanced control systems that may be used in future projects. This DSP also contains internal Analog to Digital Converter (ADC) peripherals to apply closed-loop control algorithms for power electronics. ADCs, take an analog signal which is a voltage or current measurement and convert the measurement value to a digital value to be processed in the DSP. The closed-loop control consists of reading the power electronic hardware measurements and applying them in the control algorithm. Closed-loop control is more resilient than open-loop control when stabilizing the inverter output power, especially when the inverter is grid-connected [6]. More information about the DSP can be found in [7], which provides an enhanced description of the Delfino F28335 Texas Instrument DSP specifications, features, and applicability.

2.4 Real-Time Simulations

Simulations, in general, are software models that interpret physical systems, processes, or environments. There are various software programs for simulations that can be used on almost any modern computer. Typical software programs used to simulate power electronics include LTSpice, Matlab/Simulink, Cadence, and more. These simulation programs are mathematically based algorithms that interpret the power electronics device or system. As such, these simulations are theoretical interpretations that differ from physically testing the device or system.

General simulations can execute and sample simulation data at a variable or fixed sample rate. The sample rate refers to the number of samples calculated in an amount of time, where the time can be fixed or varies in every sample interval [8]. The time of each sample is not the same as in real-time; instead, it is a simulation time. In the simulation "world," the whole simulation time is provided by the user. For example, if a rotating circle is simulated to rotate 360 degrees in 20 seconds, and 10 seconds of simulation time is set, the simulated circle will rotate halfway, as expected. The simulation illustrated what would happen to the circle in the given time parameters, but the simulation may have generated the results in less than one second. Assuming this simulated circle is highly complex, then the simulation would have taken hours to generate the same results. With this same example, the circle can be illustrated to rotate continuously or in steps, like an analog clock. The simulation uses the time steps to divide the whole simulation time. Therefore, the circle will change position at every time step. If the circle were to rotate at a fixed speed, the simulation could use a fixed time step, and if the rotating circle speed was varying, the simulation could use a variable time step.

Real-Time Simulations (RTS) are more enhanced than theoretically based simulation software programs. RTS combines software models with high-speed processors capable of

executing complex simulations in real-time. General simulations may take longer or shorter to generate the simulated data depending on the complexity of the model. However, RTS will generate the data at the same rate as real-world time because RTS high-speed processors essentially have a time step of nanoseconds. This timestep allows the simulation to seem real-time because the data is being generated in nanoseconds.

RTS use either an FPGA or a CPLD to process the simulation data in nanoseconds. As described in section 2.3.1, FPGAs and CPLDs can process separate tasks simultaneously in parallel. Therefore, FPGAs and CPLDs process complex simulation models as separate tasks in nanoseconds. The simulated data is more accurate than general simulations, and it is running in real-time. Different systems allow RTS, which include OPAL-RT, Typhoon HIL, Dspace. The security approach design includes real-time simulations as part of integration purposes, where OPAL-RT is the considered RTS system. Therefore, the application method of the OPAL-RT system, called Hardware-In-Loop (HIL), will need to be understood.

2.4.1 Hardware-In-Loop with OPAL-RT

The OPAL-RT consists of a software program called RT-LAB and a hardware device for RTS. OPAL-RT also partners with MATLAB/Simulink software to model the simulations. As a reference, Figure 2.10 illustrates a general simulation model of an inverter using Simulink model blocks. RT-LAB includes additional model blocks to the Simulink library for RTS compatibility. With these model blocks, RT-LAB compiles and prepares the simulation model for RTS. Once prepared, RT-LAB creates a bitstream file containing the simulation model data to program the OPAL-RT hardware. The OPAL-RT hardware consists of a CPU and an FPGA. The CPU communicates information with RT-LAB and with the FPGA. The simulation file generated in

RT-LAB is sent to the CPU of the OPAL-RT and then programmed in the FPGA to operate in real-time.

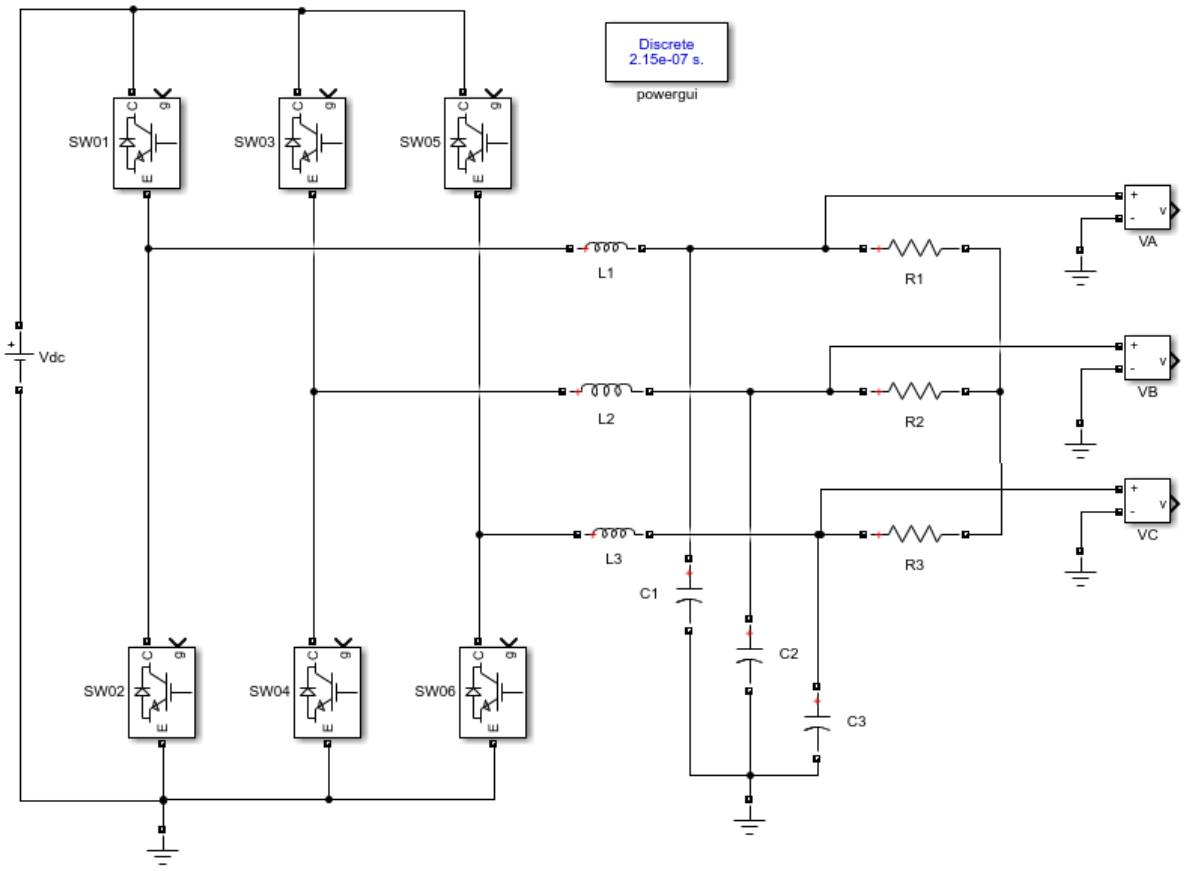

Figure 2.10. 3-Phase inverter Simulink model

RTS allows simulations to interact with physical hardware, creating an environment setup called Hardware-In-Loop (HIL). There are two HIL categories in power electronics: Controller-Hardware-In-Loop (CHIL) and Power-Hardware-In-Loop (PHIL). PHIL refers to a power electronic device, such as an inverter that receives control signals from the real-time simulated controller and sends voltage and current measurements back to the simulation. Figure 2.11 shows a PHIL diagram where a low-power inverter is connected to a real-time simulated controller. The real-time simulated controller model is loaded into the OPAL-RT, where the OPAL-RT

accurately interprets a hardware controller. The OPAL-RT sends real-time control signals to the physical inverter and receives inverter measurements such as voltage and current. The second category is CHIL, where a hardware control board, such as the UCB, interacts with a simulated inverter, as shown in Figure 2.12. In this setup, the inverter simulation model is loaded on the OPAL-RT, where the OPAL-RT accurately interprets a physical inverter. The control board sends control signals to the real-time simulated inverter and receives voltage and current measurements from the real-time simulation. The voltage and current measurements are part of the real-time simulation; however, the measurement values are physical outputs from the OPAL-RT system. One can measure the simulated voltage and current using an oscilloscope in the same way one measures voltage and current from a physical inverter.

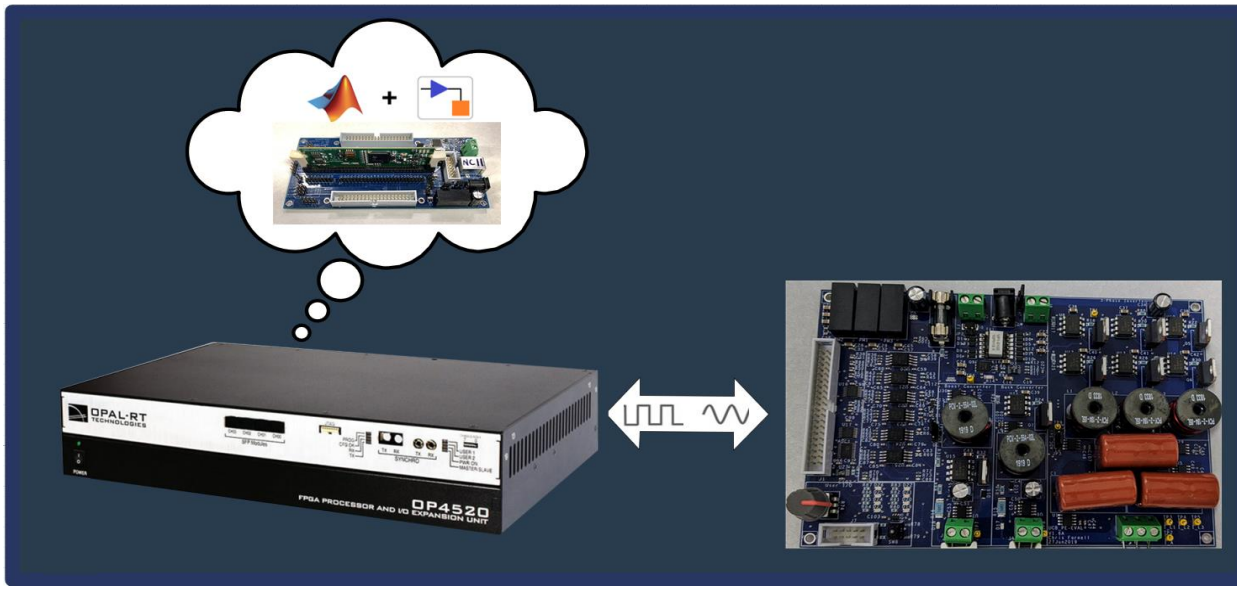

Figure 2.11. PHIL diagram

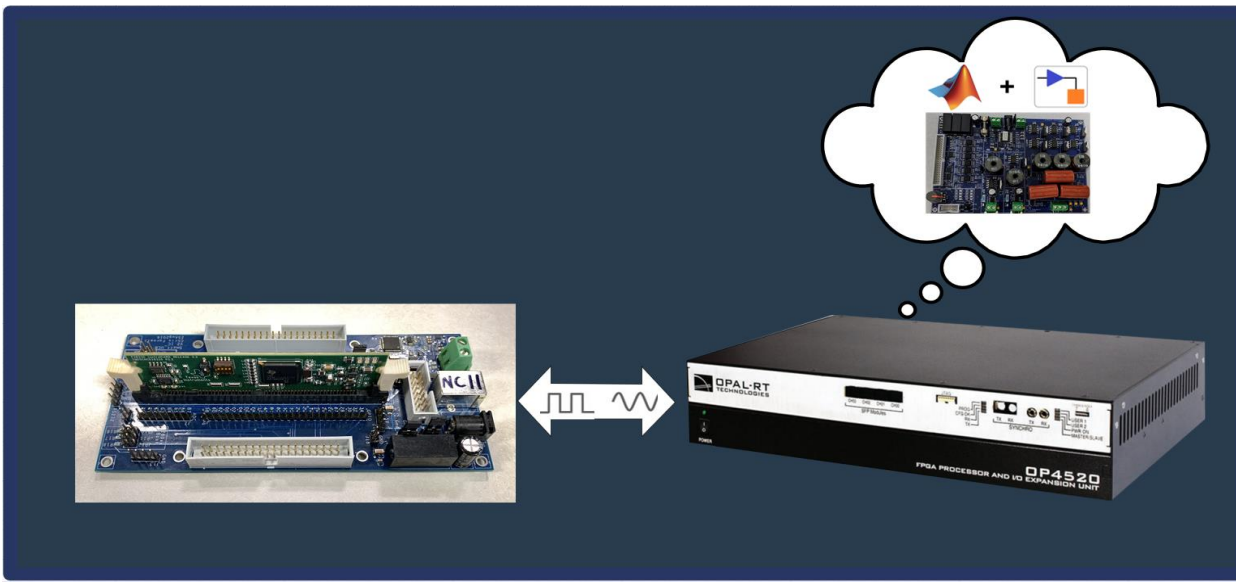

Figure 2.12. CHIL diagram

2.5 References

[1] Remus Teodorescu; Marco Liserre; Pedro Rodriguez, "Grid Filter Design," in Grid Converters for Photovoltaic and Wind Power Systems, IEEE, 2007, pp.289-312, doi: 10.1002/9780470667057.ch11.

[2] Lattice Semiconductor, “MachXO2 sysCLOCK PLL Design and Usage Guide,” LCMX02-7000HC Technical Note, March 2020.

[3] Lattice Semiconductor, “MachXO2 Family Data Sheet,” LCMX02-7000HC datasheet, December 2020.

[4] Bin Wu; Mehdi Narimani, "Diode-Clamped Multilevel Inverters," in High-Power Converters and AC Drives, IEEE, 2017, pp.143-183, doi: 10.1002/9781119156079.ch8.

[5] Remus Teodorescu; Marco Liserre; Pedro Rodriguez, "Grid Current Control," in Grid Converters for Photovoltaic and Wind Power Systems, IEEE, 2007, pp.313-354, doi: 10.1002/9780470667057.ch12.

[6] Remus Teodorescu; Marco Liserre; Pedro Rodriguez, "Control of Grid Converters under Grid Faults," in Grid Converters for Photovoltaic and Wind Power Systems, IEEE, 2007, pp.237-287, doi: 10.1002/9780470667057.ch10.

[7] Texas Instruments, “TMS320x2833x, TMS320x2823x Technical Reference Manual,” TPMS320F28335 Technical Reference Manual, March 2020.

[8] “Choose a Fixed-Step Solver,” Fixed Step Solvers in Simulink - MATLAB & Simulink. [Online]. Available: https://www.mathworks.com/help/simulink/ug/fixed-step-solvers-in-simulink.html. [Accessed: 06-Jun-2021].

CHAPTER 3

ONLINE SECURITY SYSTEM ARCHITECTURE

3.1 Introduction

The security reference design is embedded in the UCB, more specifically, the CPLD. The CPLD consists of six modules, which will be described in this section. The CPLD is the center of the UCB, which controls the signal routing, data flow, security measures, and firmware patching. Figure 3.1 illustrates the UCB components and the data flow diagram with the three main components: the two DSPs and the CPLD. This section will describe each CPLD module in-depth to understand how each module works and how they are integrated. In addition, this section will also discuss some general digital design components and concepts to gain a deeper understanding of the functionality of the modules.

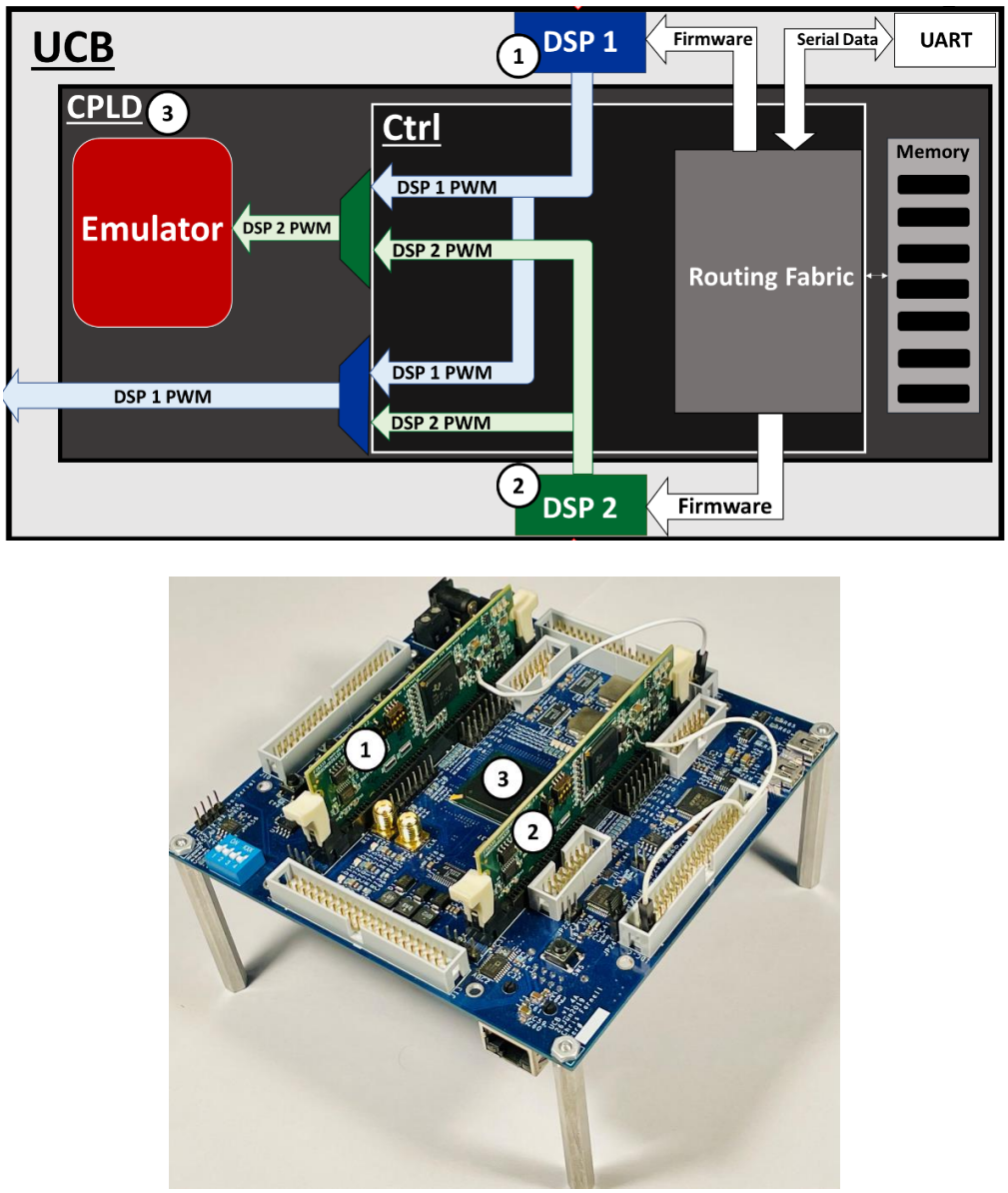

Figure 3.1. Controller board architecture and data flow

From Chapter 2, the explanation of the CPLD consisted of applying it for the overall security design. This security approach is integrated only in the CPLD instead of multiple devices. Having one device in charge of all the security allows more control over the system. The CPLD can process multiple tasks simultaneously, which means it will handle all of the security modules with ease. The CPLD will process two types of validation checks, the emulation and the

short circuit test. The user will communicate with the CPLD via the User Interface (UI) in LabVIEW software. The UI was not the focus of this work, but it was used to test the security design. The user sends firmware files to the CPLD and patches them on the standby DSP. The security system checks for short circuit hazards and then emulates the inverter using the new DSP firmware. The emulation is captured and viewed through the UI for additional validation. Once validated, the user may confirm the Hot-Patch through the UI, which swaps the DSPs. The other scenarios are when the firmware is invalid, and the user is notified that the firmware backup was patched because of a short circuit error. Details on how these modules work together in the CPLD to create the security design will be explained in the following sections. The demonstration of results using different scenarios will be displayed and explained in the next chapter.

3.2 System Components

The reference design is composed of digital logic programmed in the CPLD. This composition contains general components constructed into larger modules. The modules are as follows: Bus Interface, UART Interface, Firmware Validation, Bootloader, Emulation, and Hot-Patching. Each module will be described in the following separate sections. Some general components that construct the mentioned modules consist of data registers, Fist In First Out registers, counters, digital data structure, and more.

The CPLD processes digital data through the programmed digital components. Digital bits consist of 1’s and 0’s that make up a binary number representation. Table 3.1 illustrate binary number composition, which can also be represented in Hexadecimal, decimal, or other data representation. The number representation can be located in Table 3.2, the American Standard Code for Information Interchange (ASCII) table. More information about binary and

hexadecimal can be found in [1]. The core of digital data design uses logic gates, where a specific combination of the inputs will determine the output. Different logic gates require different input combinations for their output, where the combination may be all 1s, all 0s, or both, and the output will be a 1 or a 0. The input and output of digital gates are only 1s or 0s. These gates can be combined with thousands of other gates to create complex digital systems. These complex systems can be programmed using a Hardware Digital Logic (HDL) programming language. HDL programs the CPLD hardware to process the digital components and digital elements from the digital design.

Table 3.1. Binary and hexadecimal structure

Decimal	**14,956**															
Binary Bits	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
	0	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**0**
Hex	**3**				**A**				**6**				**C**			
Bit Structure	4-Bit				4-Bit				4-Bit				4-Bit			
	8-Bit (**Byte**)								8-Bit (**Byte**)							
	16-Bit (**Word**)															

Table 3.2. ASCII table

Decimal	Hex	Binary	Decimal	Hex	Binary
0	0	0	32	20	10 0000
1	1	1	33	21	10 0001
2	2	10	34	22	10 0010
3	3	11	35	23	10 0011
4	4	100	36	24	10 0100
5	5	101	37	25	10 0101
6	6	110	38	26	10 0110
7	7	111	39	27	10 0111
8	8	1000	40	28	10 1000
9	9	1001	41	29	10 1001
10	A	1010	42	2A	10 1010
11	B	1011	43	2B	10 1011
12	C	1100	44	2C	10 1100
13	D	1101	45	2D	10 1101
14	E	1110	46	2E	10 1110
15	F	1111	47	2F	10 1111
16	10	1 0000	48	30	11 0000
17	11	1 0001	49	31	11 0001
18	12	1 0010	50	32	11 0010
19	13	1 0011	51	33	11 0011
20	14	1 0100	52	34	11 0100
21	15	1 0101	53	35	11 0101
22	16	1 0110	54	36	11 0110
23	17	1 0111	55	37	11 0111
24	18	1 1000	56	38	11 1000
25	19	1 1001	57	39	11 1001
26	1A	1 1010	58	3A	11 1010
27	1B	1 1011	59	3B	11 1011
28	1C	1 1100	60	3C	11 1100
29	1D	1 1101	61	3D	11 1101
30	1E	1 1110	62	3E	11 1110
31	1F	1 1111	63	3F	11 1111

Table 3.2 (Cont.)

Decimal	Hex	Binary	Decimal	Hex	Binary
64	40	100 0000	96	60	110 0000
65	41	100 0001	97	61	110 0001
66	42	100 0010	98	62	110 0010
67	43	100 0011	99	63	110 0011
68	44	100 0100	100	64	110 0100
69	45	100 0101	101	65	110 0101
70	46	100 0110	102	66	110 0110
71	47	100 0111	103	67	110 0111
72	48	100 1000	104	68	110 1000
73	49	100 1001	105	69	110 1001
74	4A	100 1010	106	6A	110 1010
75	4B	100 1011	107	6B	110 1011
76	4C	100 1100	108	6C	110 1100
77	4D	100 1101	109	6D	110 1101
78	4E	100 1110	110	6E	110 1110
79	4F	100 1111	111	6F	110 1111
80	50	101 0000	112	70	111 0000
81	51	101 0001	113	71	111 0001
82	52	101 0010	114	72	111 0010
83	53	101 0011	115	73	111 0011
84	54	101 0100	116	74	111 0100
85	55	101 0101	117	75	111 0101
86	56	101 0110	118	76	111 0110
87	57	101 0111	119	77	111 0111
88	58	101 1000	120	78	111 1000
89	59	101 1001	121	79	111 1001
90	5A	101 1010	122	7A	111 1010
91	5B	101 1011	123	7B	111 1011
92	5C	101 1100	124	7C	111 1100
93	5D	101 1101	125	7D	111 1101
94	5E	101 1110	126	7E	111 1110
95	5F	101 1111	127	7F	111 1111

Registers are digital components that assign data to a signal in the CPLD. The data will remain the same on that signal until other data is loaded and latched. FIFO is a type of data register where the order of data saved will be the same order in which it is read. FIFOs are useful when modules need to temporarily store some data to be used by other modules or processes in the CPLD when needed. Another register type is the shift register, which moves every data bit in one place when a new bit is saved in the register. This register can be set to shift the bits to the right or the left, depending on the configuration. Counters are other digital components used to increment data in a signal at a specific rate. These counters can increment as fast as the clock speed of the CPLD. Another digital design concept is the state machines, which are processes in the CPLD that execute a function in a state then move to a different state to execute another function. The states generally change once every clock cycle. More on digital design can be found on [2, 3]. These digital components will be mentioned throughout the description of each module in the reference design.

3.2.1 Bus Interface

The Bus Interface is a component of the security design, which manages the access to the Random-Access Memory (RAM) of the CPLD. The Bus Interface is also programmed in the CPLD and collaborates with the other modules in the CPLD design. The modules in the CPLD consist of the Bust Interface, Serial Communication Interface, Firmware Validation, Bootloader, Hot-Patching, and Emulation. The RAM block of the CPLD contains two independent data ports that can be used simultaneously to read and write data to RAM. This RAM block is called Dual-Port RAM (DP-RAM), where the two ports are labeled as port 1 and port 2 to distinguish between them. The Serial Communication Interface, Firmware Validation, Hot-Patching, and Emulation modules will use port 1 of the DP-RAM to access the data. The Bootloader module

will use port 2 of the DP-RAM to access the data. The DP-RAM contains 12,288 registers with their specific location in the DP-RAM, pointed by an address. The address label convention is hexadecimal, containing a 16-bit word. Therefore, the address locations are ordered between X"0000" and X"3000, corresponding to 0 and 12,288 in decimal, respectively. The data in each DP-RAM register consists of 16-bit words. The DP-RAM contains allocated space to store data for each module. Table 3.3 shows the different allocated sections in the DP-RAM for the different modules.

Table 3.3. CPLD RAM register map distribution

Name	**RAM Address** (16-Bit Hex)	**Data** (16-Bit Hex)	**Description**
Emulation Va	**[0800 : 08FF]**	**16-Bit**	Emulated Inverter Voltage Phase A
Emulation Vb	**[0900 : 09FF]**	**16-Bit**	Emulated Inverter Voltage Phase B
Emulation Vc	**[0A00 : 0AFF]**	**16-Bit**	Emulated Inverter Voltage Phase C
Boot Start	**0B00**	**0**	**STOP**
		1	**START**
Hot-Patch Command	**0B01**	**0**	**N/A**
		1	**YES**
		2	**NO/Backup**
Emulation Start	**0B02**	**0**	**STOP**
		1	**START**
Boot Status	**0B10**	**0**	**OFF**
		1	**Working**
		2	**Backing up**

Table 3.3 (Cont.)

Name	RAM Address (16-Bit Hex)	Data (16-Bit Hex)	Description
HP Status	**0B11**	**0**	**OFF**
		1	**Ready**
		2	**Enabled**
		3	**ERROR**
Emu Status	**0B12**	**0**	**OFF**
		1	**Emulating**
		2	**Saving**
		3	**ERROR**
ERROR	**0B13**	**0**	**N/A**
		1	**Short Circuit**
FW Len	**1000**	**16-Bit**	Regs in Firmware
FW Data	**[1001 : 1FFF]**	**16-Bit**	Firmware
Backup Len	**2000**	**16-Bit**	Regs in Backup
Backup Data	**[2001 : 2FFF]**	**16-Bit**	Backup Firmware

The modules in the CPLD are processing data and functions simultaneously in parallel. Therefore, those that use port 1 of the DP-RAM may attempt to gain access to the stored data simultaneously. To gain RAM access, the signal ports between the RAM block and the module must physically connect. However, when multiple modules are connected to the RAM block to store data, it would cause an issue. If two signals attempt to assign data to one signal

simultaneously, the receiving signal will be unsure of what data to receive; therefore, the data received might not be what is intended by the designer.

Moreover, to fix this issue with the modules accessing RAM, a tri-state buffer will be applied. Tri-state buffers are digital components in the CPLD with multiple input data and output either one of the input data signals or a "high-impedance," which means it outputs nothing. The Bus Interface is a tri-state buffer that prioritizes RAM access between the modules. The Serial Communication Interface (SCI), Firmware Validation, Bootloader, Hot-Patching, and Emulation are the five modules in the design, each containing an independent Bus Interface used as "hand-raising" to request RAM access. The whole Bus Interface consists of the independent Bus Interface submodules used for "hand-raising," and a Bus Master used to grant RAM access. The independent Bus Interface modules temporarily store data in a register until the Bus Master grants the module RAM access. Once access is granted, the Bus Interface module saves the stored data in the appropriate RAM register.

The Bus Master grants RAM access to the modules using port 1 in the following order of highest priority to lowest priority: Bootloader control, Firmware Validation, Emulation, and SCI. The Bootloader consists of patching firmware on the standby DSP, pulling the firmware data from port 2 of the DP-RAM, and a Bootloader control that enables the patching process and informs the user about the patching status. The control saves the status information and reads the user commands to start the Bootloader through port 1 of the DP-RAM. The Bootloader control has the highest priority in case the user wants to patch a new firmware. The patching process of the bootloader uses port 2 of the DP-RAM because it needs to read the firmware data at a fixed rate. This bootloader portion is the only module that uses port 2, which means it does not need a Bus Interface. The Bus Interface applies a delay when accessing RAM, which could be

inefficient when patching firmware on the DSP. Therefore, the patching process is uninterrupted when pulling the firmware data from port 2 of the DP-RAM. The Bootloader functionality is detailed in the sections below.

3.2.2 Serial Communication Interface

The Serial Communication Interface (SCI) is designed to communicate data between the UCB and the UI through UART ports. The SCI is built in the CPLD of the UCB, and the UI is built using LabVIEW software on a computer. The UART ports of the UCB and the UI are connected via a USB cable. The data is communicated without a clock, which means asynchronous communication. Therefore, the UI and SCI agree on a fixed rate at which data will be transferred through the UART lines. This rate is known as the baud rate, which refers to bits per second. This baud rate is used instead of a clock. The UART ports are not the communication protocol, they are lines between the UI and the UCB used to communicate one data bit at a time. Therefore, a custom-made communication protocol is used to handle the communicated data through the UART lines. The communication protocol was designed to be universal, meaning it can be easily integrated into other protocols such as serial or Modbus with minor compatibility adjustments. For this design, the interface will be applied to serial communication protocol through the UART lines.

The SCI is used to communicate data between the DP-RAM of the CPLD and the UI. The DP-RAM contains allocated space for data that interacts with the different modules of the reference design. Moreover, the data stored from each module is to be communicated between the CPLD modules and the UI. This data consists of UI commands, collected data from the emulation, status updates from the CPLD modules, and more. The data described in the DP-

RAM is shown in Table 3.3. Figure 3.2 shows the UI in LabVIEW to send commands to the CPLD and read data from the CPLD.

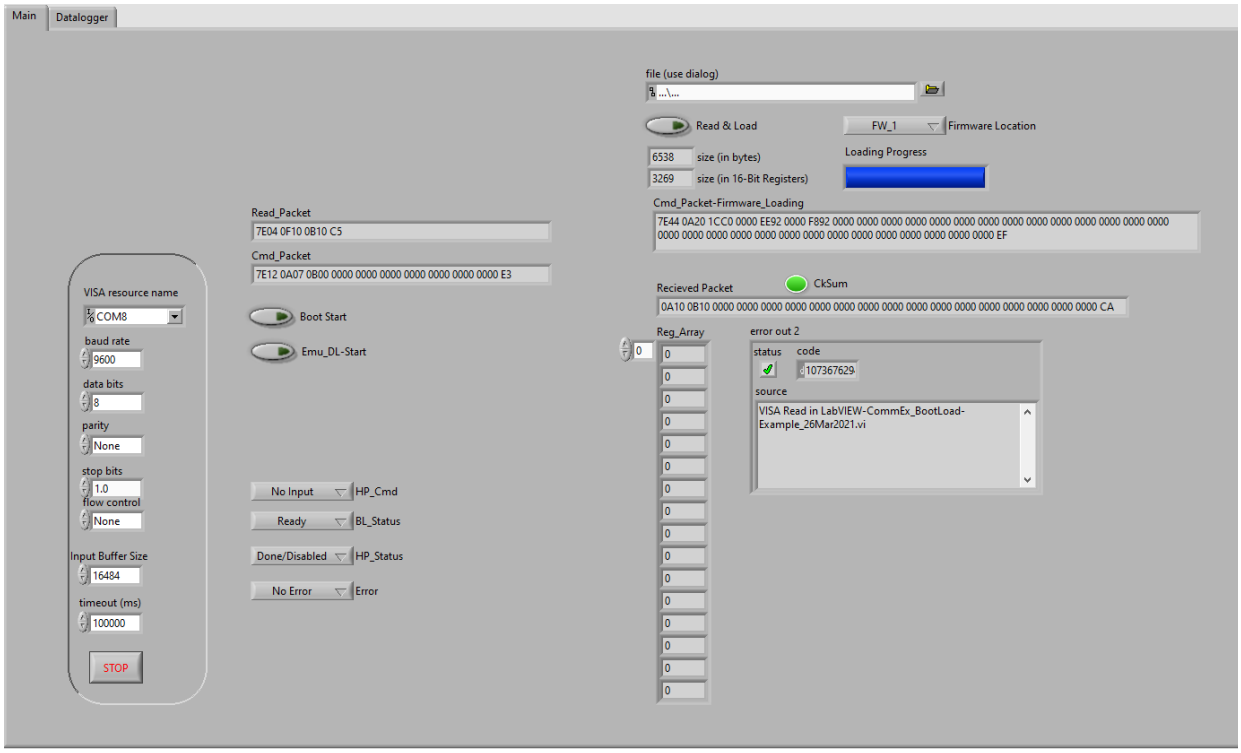

Figure 3.2. LabVIEW window

The communication protocol consists of a specific package of data, shown in Table 3.4. This table describes how the data is being communicated between the UI and the UCB. The packet is designed such that the receiver knows what to do with the packed data. The data packet is sent through the UART line in multiple bytes, each byte separated by a start and stop bit. The package consists of a start delimiter (Strt Del), package size (Pkt Len), operation ID (OPID), number of 16-bit register data sets (Reg Cnt), the starting address (Strt Addr), the data, and the checksum (ChkSum).

Table 3.4. Serial communication data packet structure

Write Packet				
Name	**Hex Value**	**Byte Split**	**Sending Order**	**Description**
StrtDel	**7E**	**7E**	**1**	New Packet Notification
PktLen	**08**	**08**	**2**	**D** Bytes + 4
OPID	**0A**	**0A**	**3**	Write (0A), Read(0F)
RegCnt	**02**	**02**	**4**	**D** Words
StrtAddr	**0011**	**00**	**5**	First Destination Address (High Byte)
		11	**6**	First Destination Address (Low Byte)
Data (D)	**3A6C4B7D**	**3A**	**7**	Data for Register 1 (High Byte)
		6C	**8**	Data for Register 1 (Low Byte)
		4B	**9**	Data for Register 2 (High Byte)
		7D	**10**	Data for Register 2 (Low Byte)
ChkSum	**84**	**84**	**11**	Packet Corruption Check

The start delimiter informs about a new packet being sent. The start delimiter is a fixed byte in hexadecimal, X”7E.” The package size is the total number of bytes that will be sent in the package. The package size does not include the start delimiter, package size, or the checksum. The OPID informs whether the package is a read or write command. If the OPID is X”0A,” then it is a write. If the OPID is X”0F, then it is a read. The register count is the number of 16-bit data sets included in the actual data being communicated. The starting address is the location of the data being read from or saved in the DP-RAM. The protocol will know that the first register will be saved in the starting address and will continue saving in the address locations above the

starting address. After the data, the checksum is sent. The checksum is the method of ensuring that the data was not corrupted during the communication process.

3.2.3 Firmware Validation

Firmware validation is a feature in the security reference design that focuses on specific firmware commands that would cause irreversible harm to the power electronic inverter. This reference design consists of one particular firmware command. The design of this feature is built such that the detection of additional malicious firmware commands can be applied to the main design with no difficulty. The focused firmware command to detect is a short circuit in the inverter. Figure 3.3 illustrates the switching combination that would cause a short circuit in the inverter. This combination wires the positive and negative terminals of the DC source, removing any resistance in the loop. Theoretically, this causes the current to reach infinity, but physically this causes the source to burst. If this happened to a grid-connected high-power inverter, the burst would be an explosion causing catastrophic damage to a section of the electric grid. Therefore, this design will focus on the specifics of the short circuit.

The design considers a 3-phase power electronic inverter with six switches, distributing two switches per phase leg. A short circuit in this inverter would be caused by turning on two switches in the same leg for whichever phase. Therefore, the firmware validation feature in the CPLD monitors the PWM signals from the stand-by DSP. This feature compares the PWM signals for all the switches in each phase leg. The comparison of each phase leg, shown in Table 3.5, is one of the parallel processes in the firmware validation performed in the CPLD.

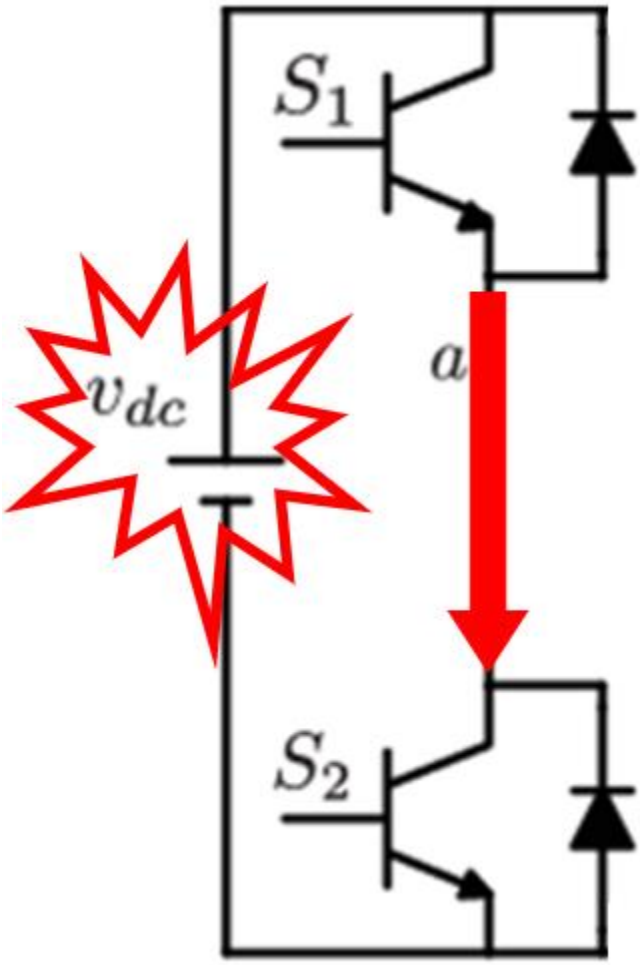

Figure 3.3. Shoot-through diagram

Table 3.5. Short circuit inverter switching combination

Vin	SW1	SW2	Vout
Vdc	ON (1)	ON (1)	Short Circuit
0	ON (1)	ON (1)	0

The firmware validation feature contains processes that go through the Bus Interface to access the DP-RAM registers. As shown in Table 3.3, there are registers assigned for the firmware validation feature, Hot-Patching, Bootloader, and Error. The firmware validation communicates directly with the Hot-Patching process, Bootloader process, and emulation process. After patching a new firmware on the stand-by DSP, the firmware validation checks for

a short circuit at every switching sequence before moving to the next task. If no short circuit was detected, the firmware validation allows the emulation to continue performing and allows the Hot-Patching process to begin. Hot-Patching, Bootloader, and emulation consist of multiple tasks that will be explained in the sections below. When the CPLD detects a short circuit, it will latch a signal labeled as Bad Firmware. The signal triggers an error function in the entire validation process, including the emulation, Hot-Patching, and Bootloader. The error function performs three tasks to inform, backup, and reset. First, the error function stops and resets the emulation. Second, error messages are saved in the allocated registers of the DP-RAM to inform the user. Finally, the Bootloader backup process begins automatically. The backup does not depend on the user commands. The demonstration of different firmware validation scenarios is displayed in chapter 4.

3.2.4 Bootloader

The Bootloader is a feature in the security design that is part of the Hot-Patching process. The Hot-Patching process will be explained in the section below. The purpose of the Bootloader is to load and patch the stand-by DSP with new firmware or a backup firmware. The Bootloader for the security design considers the specific Bootloading process that the DSP requires. As introduced in chapter 2, the security design is applied to the UCB, which includes two identical Delfino F28335 Texas Instrument DSP cards and a Lattice MachXO2 CPLD. The Bootload process of these DSP cards consists of a UART for communicating data and a reset signal. The UART signals are between the DSP cards and the CPLD. Data communicated to the DSP cards and from the DSP cards go through the CPLD. Therefore, the CPLD handles the Bootloader process for the DSP cards. The Bootloading process starts by sending a signal to the DSP reset

pin, which erases all DSP firmware and restarts the DSP. Once the DSP restarts with no firmware content, the Bootloader sends the new firmware data.

The firmware data is communicated in a specific packet format through the UART ports between the CPLD and the DSP. Do note that the CPLD has separate UART ports for the DSP cards and the UI. The UART between the DSP cards and the CPLD is interfaced via the Bootloader program. The UART between the UI and the CPLD is interfaced via the Serial Communication Interface (SCI). The packet data set consists of a start bit, one firmware data byte, and a stop bit. Once the DSP receives this, it waits for another data set in this packet format. Table 3.6 shows the arrangement of the firmware data sent to the DSP. The first data set is X”41,” which is 41 in hexadecimal or 65 decimal. This first data set configures the DSP to perform an auto baud rate. Auto baud rate means that the DSP will automatically adjust the baud rate of the receiving data from the transmitter. For the Bootloader, the data transmitter is the CPLD, and the receiver is the DSP. After sending the first data set, the firmware data packets can be sent to the DSP.

Table 3.6. Firmware patching data structure

RAM Address (16-Bit Hex)	**Firmware Data** (Hex)	**Data Splitting** (8-Bit Hex)	**Sending Order**
-	**41**	**41**	**1**
1001	**0A00**	**00**	**2**
		0A	**3**
1002	**6F20**	**20**	**4**
		6F	**5**
1003	**30D1**	**D1**	**6**
		30	**7**
. . .	. . .	. . .	. . .
12A4	**2C91**	**91**	**1352**
		2C	**1353**

When patching new firmware on the stand-by DSP, the user sends the firmware file to the CPLD through the SCI interface. The firmware file is in the form of a binary file, where the data is formatted to hexadecimal before sending it to the CPLD. The user interface exports the firmware data packets in a format compatible with the SCI of the CPLD. As previously mentioned in section 3.2.2, the communication with the SCI consists of specific packet formats. This formatting also includes a limit of 32 data registers in a single packet. These data registers have two bytes each, which are saved in the allocated space of the DP-RAM. As such, the user interface splits the firmware file into sets of 32 data registers and sends multiple packets to the CPLD until the entire firmware file is sent. It also sends the total number of data registers that

the firmware file contains to the CPLD, saved in the first allocated register of the DP-RAM. Once the new firmware file is saved in the DP-RAM, the Bootloader is enabled. The Bootloader first erases the DSP content, resets it, and sends the autobaud data to prepare the DSP for the new firmware. Then, the Bootloader pulls the firmware data from the allocated registers of the DP-RAM, splits the 16-bit register data into two 8-bit data sets, and formats the 10-bit packet, including the start and stop bits. The Bootloader uses the total number of firmware registers saved in the first allocated register of the DP-RAM, shown in Table 3.7, to assure that the entire firmware file is sent to the DSP. Once the firmware file is sent to the DSP, the Bootloader enables the firmware validation feature.

Table 3.7. CPLD RAM register distribution for new firmware

Name	**RAM Address** (16-Bit Hex)	**Data** (16-Bit Hex)	**Description**
FW Len	**1000**	**16-Bit**	Regs in Firmware
FW Data	**[1001 : 1FFF]**	**16-Bit**	Firmware

The firmware validation feature was described in the previous section, addressing the backup process when the new firmware file malfunctions. The backup process enables the Bootloader for backup mode, where it pulls the backup firmware data from the allocated registers in the DP-RAM. Table 3.8 shows the allocated space in the DP-RAM for the backup firmware data. The Bootload process for the backup is the same as the process for the new firmware.

Table 3.8. CPLD RAM register distribution for backup firmware

Name	RAM Address (16-Bit Hex)	Data (16-Bit Hex)	Description
Backup Len	**2000**	**16-Bit**	Regs in Backup
Backup Data	**[2001 : 2FFF]**	**16-Bit**	Backup Firmware

3.2.5 Hot-Patching

Hot-Patching is a term that refers to loading and installing a software program to a hardware device without forcing the device to shut down or restart. In other words, the device will continue performing its regular functions during the software installment; once it installs, the device will seamlessly begin performing the new functions commanded by the new software. This transition in software can be challenging to implement, especially for software programs with high complexity.

The Hot-Patch in the security reference design is possible with the controller architecture used. As described in chapter 2, the UCB contains two DSP cards and a CPLD. All of the digital signals flowing in the UCB go through the routing fabric of the CPLD. The key feature that defines this setup as a Hot-Patch is that while one of the two DSP cards is being patched with firmware, the other DSP card functions with no interruptions. The UCB is continuously running regardless of a software update, DSP malfunction, or any firmware error. Therefore, during the firmware patch, the stand-by DSP can be erased, turned off, and reset as needed, while the rest of the UCB functions uninterrupted. The Hot-Patch is processed entirely in the CPLD, which uses the routing fabric to create multiplexers.

A multiplexer is a digital logic operation that is used to select data to be transmitted. In this case, the multiplexer selects the data to be transmitted between the two DSP signals. There

are two multiplexers, one outputs the stand-by DSP signals to the validation and emulation, and the other multiplexer outputs the active DSP signals to the inverter. Each DSP contains six PWM signal ports, two UART signal ports for communication, and a reset port connected to the multiplexer's inputs. Figure 3.4 and Figure 3.5 illustrate a high-level Hot-Patch functionality in the security design. The second feature is the enable port, usually a single-bit signal that activates the multiplexer, toggling the output data. For example, DSP1 and DSP2 are inputs of the multiplexer, where DSP1 is the current multiplexer output; the enable signal will trigger the multiplexer to output DSP2. The enable signal is the same for both multiplexers to toggle them simultaneously. The firmware validation feature of the security design is the primary control of the Hot-Patch enable bit. The firmware validation assures that the firmware is not malfunctioning before allowing the user to confirm the Hot-Patch. The user sends the confirmation to the allocated register in the DP-RAM, where the security system reads the data. If the confirmation data represents a yes, the system will begin the Hot-Patch process; if the confirmation data represents a no, the system will begin the backup process. The firmware validation processes this decision, where it enables a single bit to the backup or a single bit to the Hot-Patch multiplexers. When the Hot-Patch enable bit is triggered, the Hot-Patch runs a process that assures that both multiplexers have triggered correctly. Once the Hot-Patch process is complete, a done signal is saved in the allocated register in the DP-RAM to notify the user.

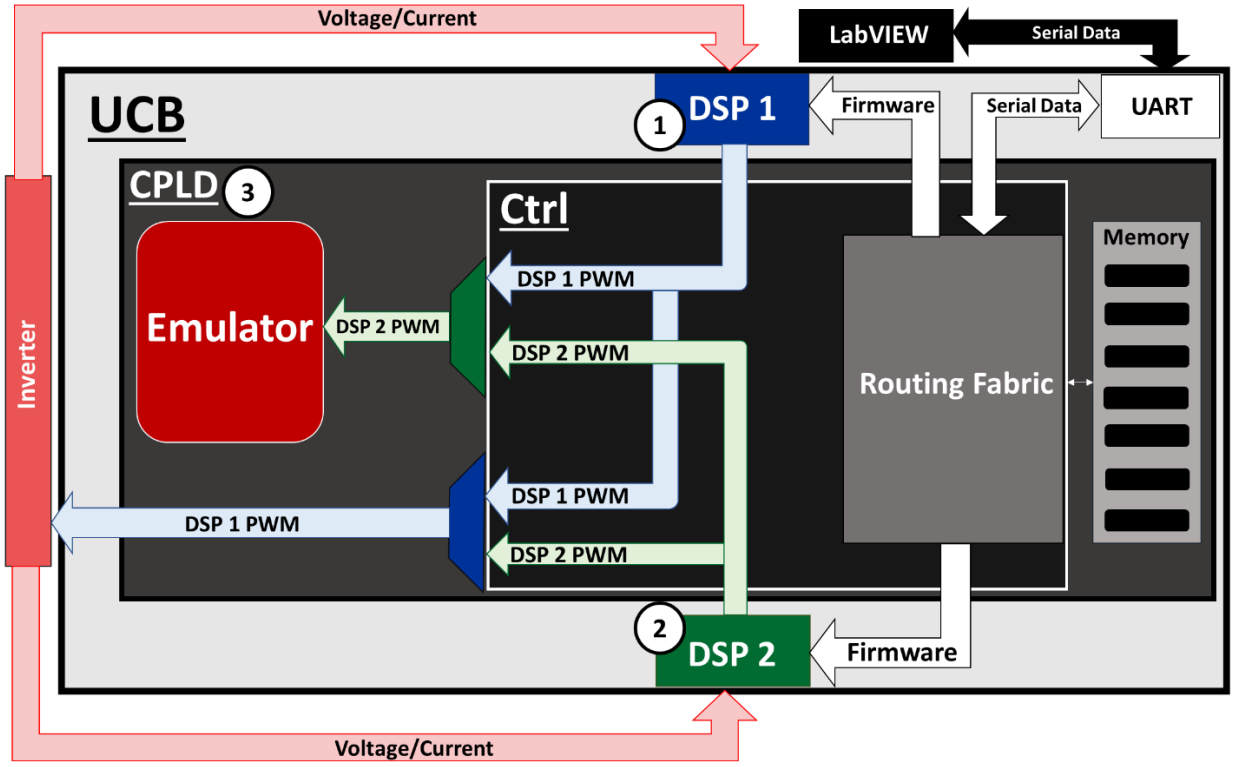

Figure 3.4. UCB architecture diagram before hot-patch

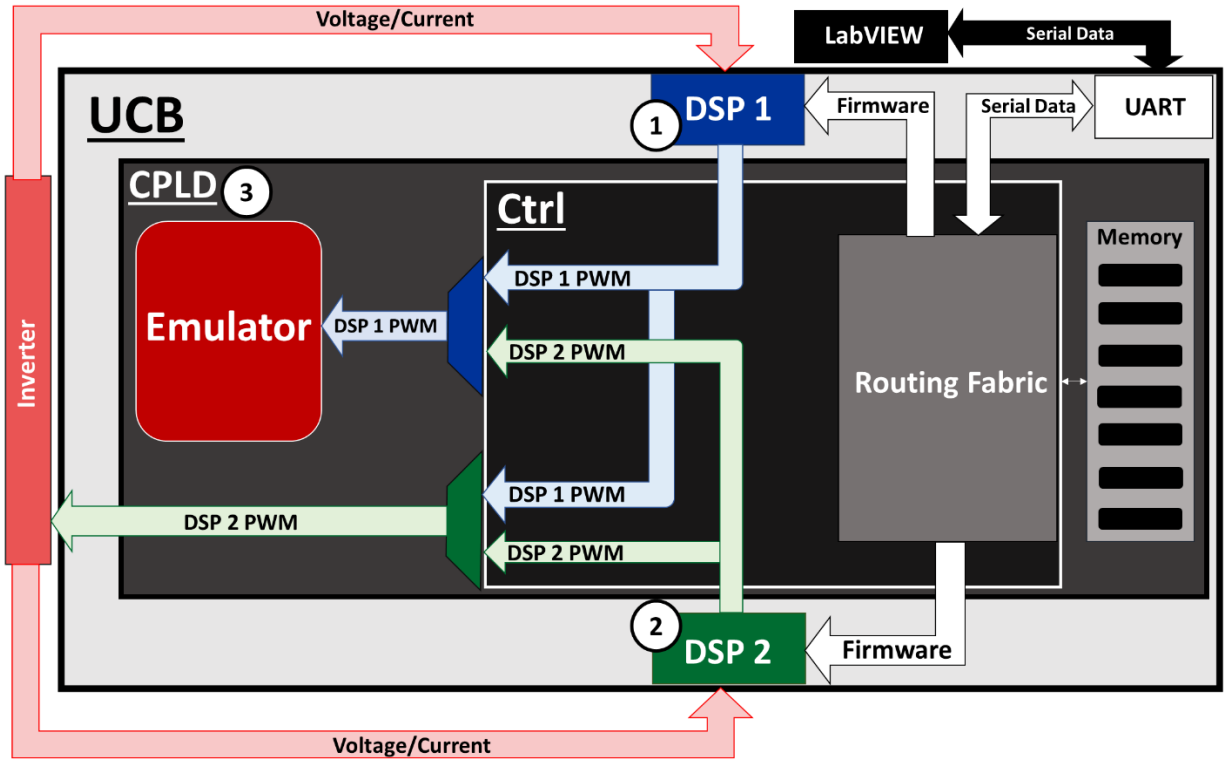

Figure 3.5. UCB architecture diagram after hot-patch

3.2.6 Emulation

The emulation is the third feature incorporated into the security design. The design approach consists of emulation and Hot-Patching, where the Hot-Patch was described in section 3.2.5. An emulation is a digital representation of a physical system or device performed through RTS. Section 2.4 describes the concepts of HIL with RTS and that they are effective validating tools for assessing power electronic systems without equipment damage hazards. Many engineers apply these tools offline to validate their system before applied on the physical hardware. These validate system performance and cybersecurity performance. However, situations where the power electronic controller firmware is updated without prior RTS and HIL validation, could be a severe hazard. Likewise, firmware patches that malfunction could be unintentional or malicious; either way, the outcome is a severe hazard. Moreover, the emulation

approach in this reference design is an additional validation tool that functions automatically. In other words, intentional or unintentional errors in the firmware update will be automatically detected online. This approach is embedding the RTS and HIL validation tools into the online physical power electronic controller board.

The purpose of the emulation in this security design is to represent a 3-Phase inverter, described in section 2.2, digitally. This inverter consists of a DC input source, and an AC output converted through the inverter's switching scheme. The controller board generates PWM signals to control the switches of the inverter. The emulation will process the PWM signals generated by the stand-by DSP, and it will reproduce an inverter output that is being controlled by these PWM signals. This emulation environment is processed in the CPLD of the UCB. Therefore, the emulation components will process with the internal clock speed of 40 nanoseconds. The core components of the emulation consist of state machines with multiple counters that can increment as fast as the 40-nanosecond clock speed. The PWM components are separated and processed with the counters in the different stages of the state machines. Once processed, the emulation will generate the inverter output referencing the AC voltage output of each inverter phase. This output visually demonstrates the accurateness of the emulated inverter output and the validity of the firmware performance. Some accuracy measures include phase shift between the 3 phases, the frequency of the AC output, and the amplitude of the output. The AC output must align with the 60Hz frequency standard for grid-connected inverters to avoid any hazards of unsynchronized components in the system [4]. Furthermore, the phase shift of the 3-Phase output must be 120 degrees to maintain balance in the system [4], which can be evaluated through online emulation. These evaluations are demonstrated in section 4.3.

The emulation consists of different tasks inside the CPLD that are performed in a sequence of steps. This sequence of tasks is explained and illustrated in multiple diagrams below to visualize the computational process of the emulation. State machines process a set of tasks in a sequence using Next State Logic (NSL) used in the computation. These tasks in the state machine are mainly a combination of counters and mathematical calculations programmed in the CPLD using VHDL. The diagrams showing the emulation computation steps will highlight only the components being used and processed at each step. The computation steps described represent one phase of the inverter. This same process is used for computing the other phases of the emulated inverter. Each phase is processed independently and simultaneously through the CPLD, allowing the output data to be an accurate imitation of the 3-phase inverter.

As introduced in section 2.3, PWM signals contain a switching frequency (Tsw), the combined duration of the ON time (Ton), and the OFF time (Toff). The PWM Tsw is approximately 33 microseconds. The CPLD processes these PWM signals as single-bit digital signals, 1 and 0. Therefore, to measure the duration of the Tsw, the CPLD counts the number of clock cycles between two PWM rising edges. This count makes up 825 clock cycles. Figure 3.6 illustrates the first step in the emulation process, Tsw counter. DSP1 is the stand-by DSP containing a new firmware in these diagrams, and DSP2 is actively controlling the inverter using a validated firmware. The second step is illustrated in Figure 3.7, which shows the emulation measuring Ton. The emulation process considers that the PWM signals for inverters have a variable Ton. Therefore, step 1b calculates a Ton value that would make up 1% of Tsw, measured in step 1a. This 1% value represents a fixed Ton reference used for the remaining steps in the emulation process, approximately 8 clock cycles.

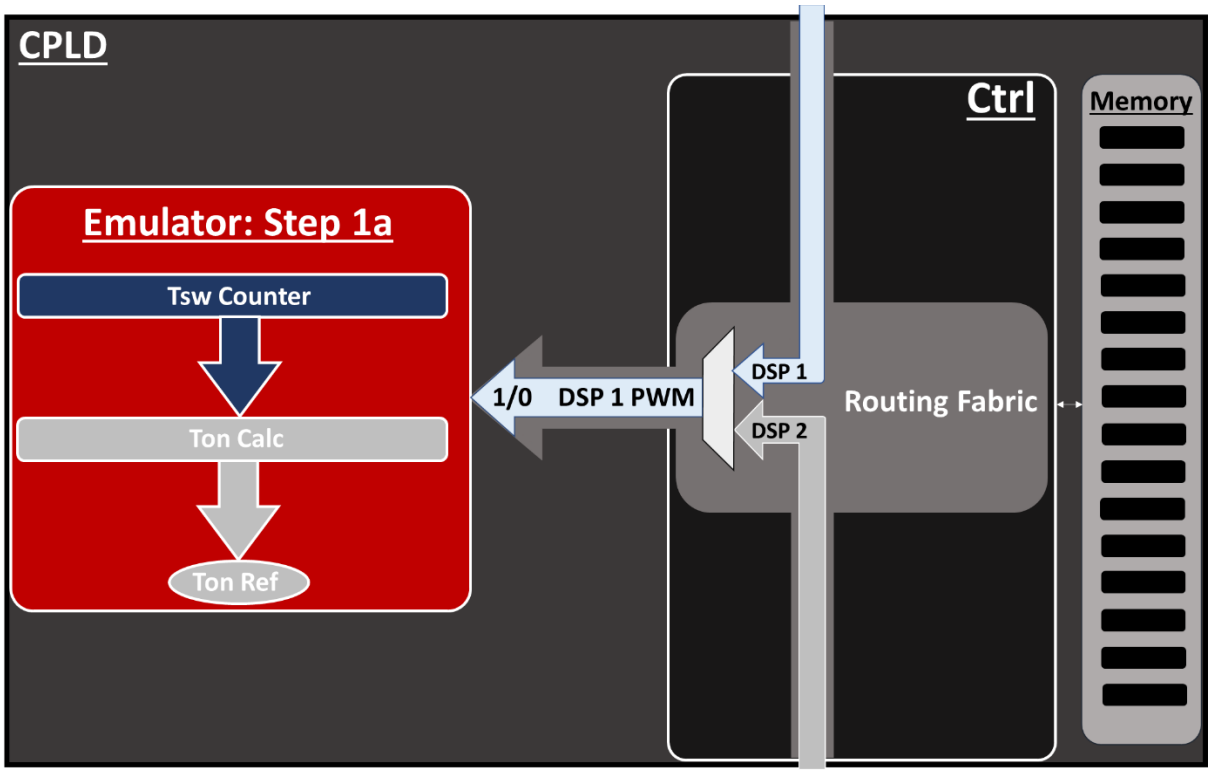

Figure 3.6. Emulation step 1a

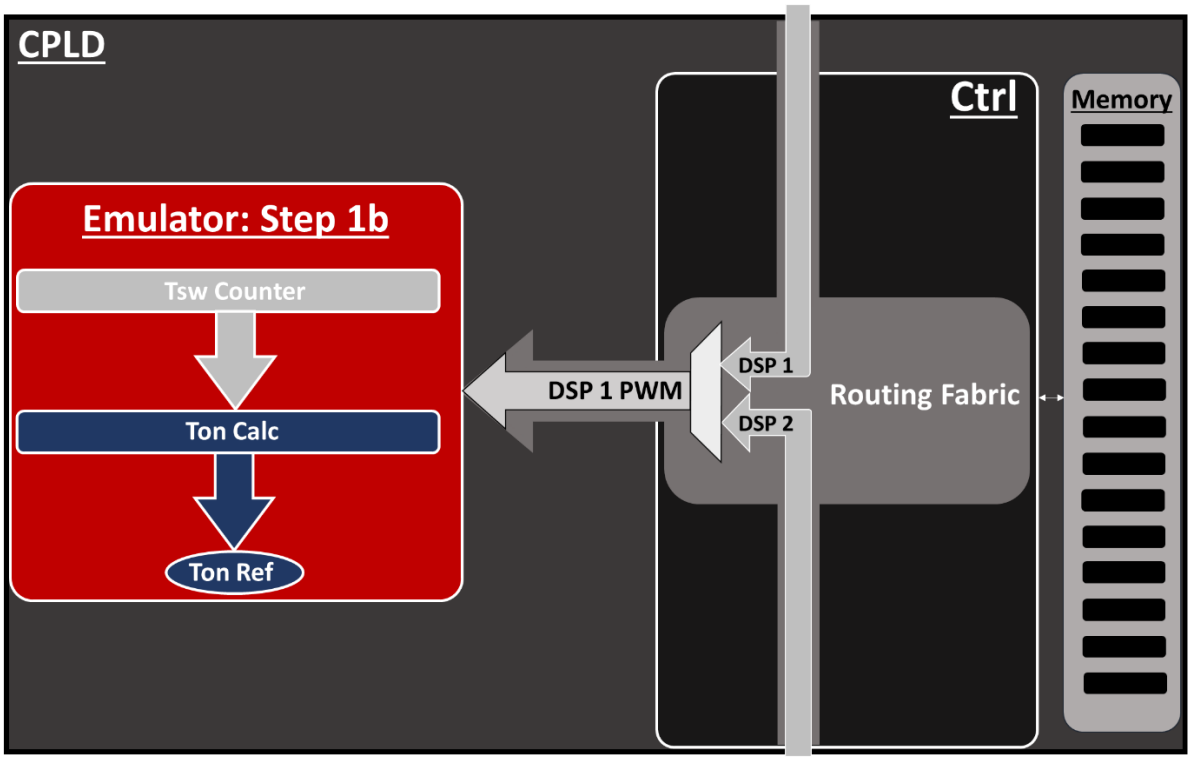

Figure 3.7. Emulation step 1b

The next step shown in Figure 3.8 illustrates a new set of tasks in the emulation process. This step consists of calculating the Duty Cycle using the Ton reference (8 clock cycles) and the PWM input from DSP1. This step uses two counters, labeled as Cnt1 and Cnt2, for this explanation. Cnt1 counts the number of clock cycles for the Ton duration and resets when the counter reaches the Ton reference (8 clock cycles). Cnt1 will keep counting and resetting in this format until the PWM signal from DSP1 is a digital 0. Every time Cnt1 equals the Ton reference (8 clock cycles), Cnt2 increments a count. In other words, Cnt2 counts the percent on time of every switching period, which makes up the Duty Cycle. When the PWM signal from DSP1 is 0, the emulation process goes to step 2b, shown in Figure 3.9. Step 2b consists of calculating the emulated inverter output voltage for each phase. This step uses the data from Cnt2 to calculate the output voltage of the inverter.

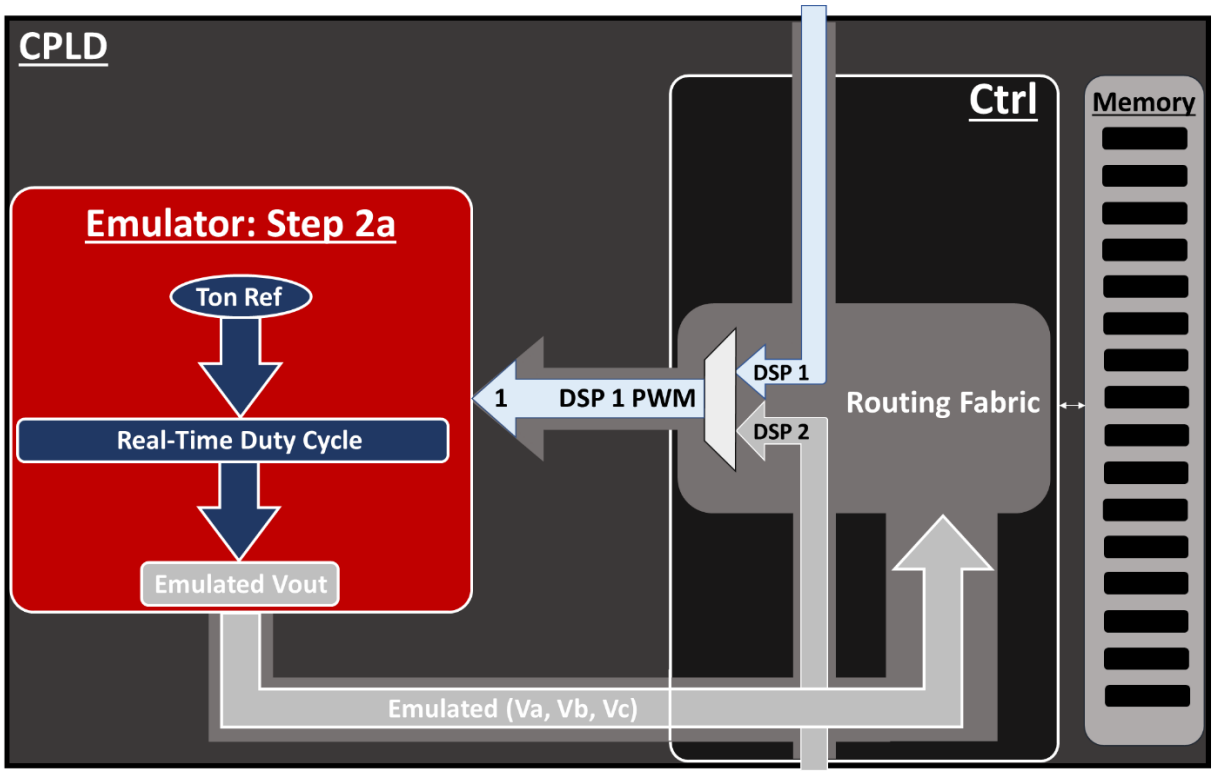

Figure 3.8. Emulation step 2a

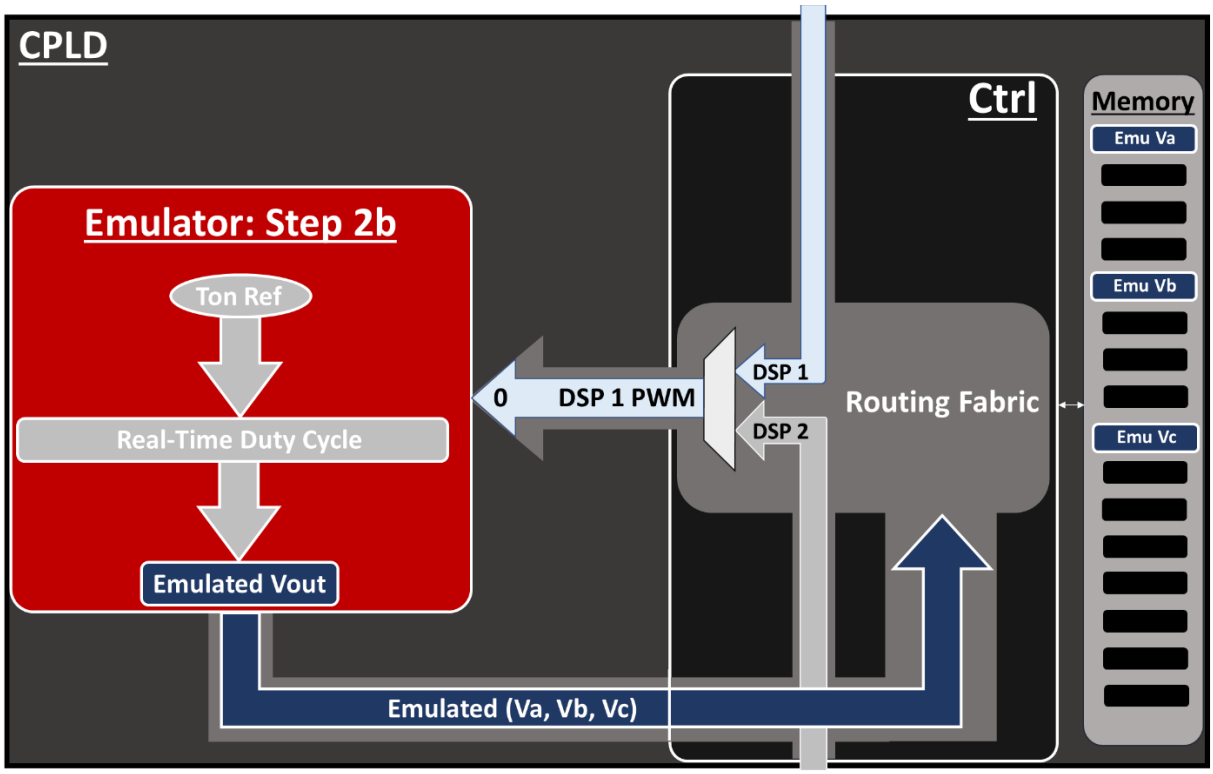

Figure 3.9. Emulation step 2b

The emulated output is calculated using a fixed-point approximation. The data in Cnt2 represents the Duty Cycle percent as a scaled whole number with no fractions. This scale also applies to the emulated voltage output data, which scales it up by a value of 100. This data is scaled back down once it reaches the user interface. The emulated voltage output data is sent to the DP-RAM of the CPLD through the Bus Interface. Once the Bus Interface allows access to the DP-RAM, the emulation saves data in the appropriate register. After saving one register, the emulation process loops back to step 2a. The process does not need to go through step 1 because the Tsw and reference Ton do not change throughout the emulation process. Therefore, the process cycles between step 2a and step 2b until the emulation is disabled. The emulator saves about 200 data sets for each phase at a specific rate to the allocated space in the DP-RAM. This saving method illustrates the output voltage for one sinusoidal cycle with a 60Hz frequency. The data is saved for all three phases simultaneously to their allocated registers to compare if all three

phases have the same frequency and have a 120 degrees phase shift. Figure 3.10 shows the diagram of Step 2b after the 200 emulated data sets have been filled for the three phases of the inverter. Once all these registers have saved data, the emulation stops saving until the user requests a new sample set. When the user requests a new sample set, the emulation process goes through steps 2a and 2b again, saving data to all the allocated registers.

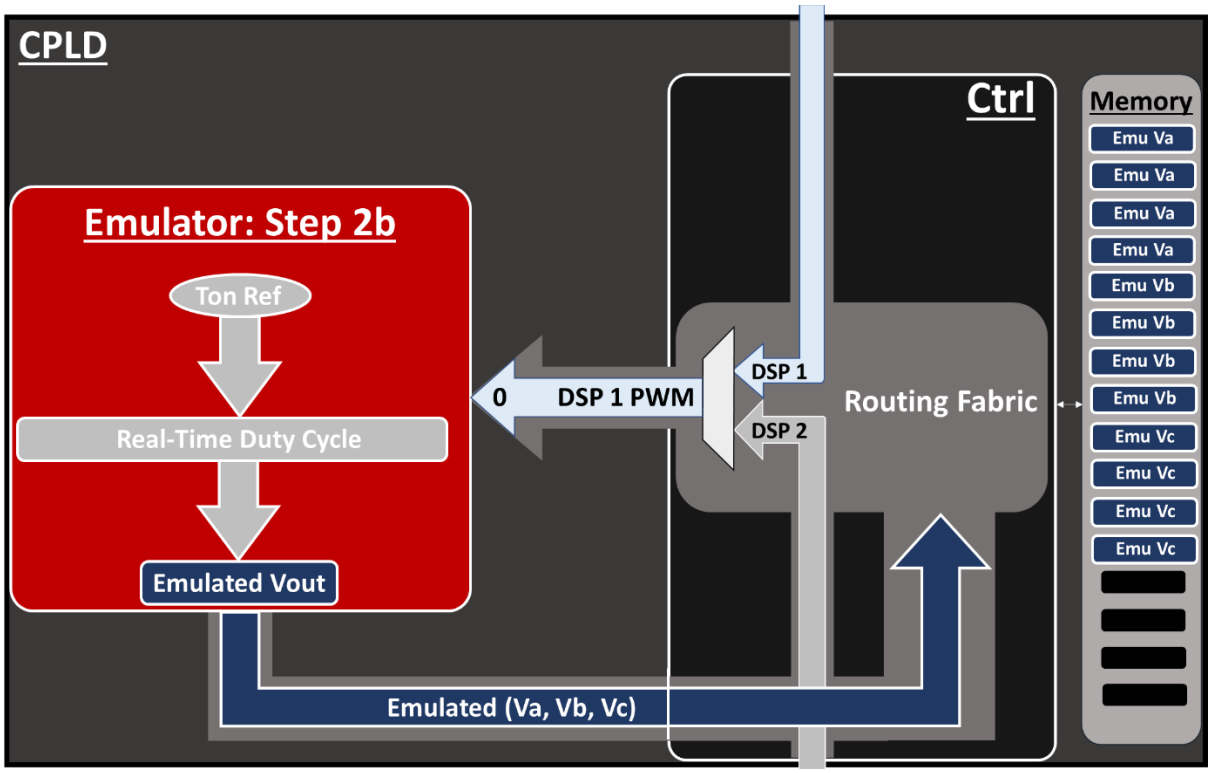

Figure 3.10. Emulation step 2b with full data saved

Two scenarios cause the emulation to completely restart from step 1a, Hot-Patching, and an error in the firmware. The emulation process begins at step 1a with DSP2 PWM signal inputs when re-enabling the emulation after a Hot-Patch procedure, shown in Figure 3.11. After a Hot-Patch was confirmed, the procedure caused the DSP swap making DSP2 the stand-by DSP used for the emulation. If the Hot-Patch was denied by the user or by a firmware validation error, DSP1 was backed up and remained in standby mode. The evaluation and results of the emulation

are demonstrated in section 4.3, providing a more visual understanding of the emulated output data through waveforms.

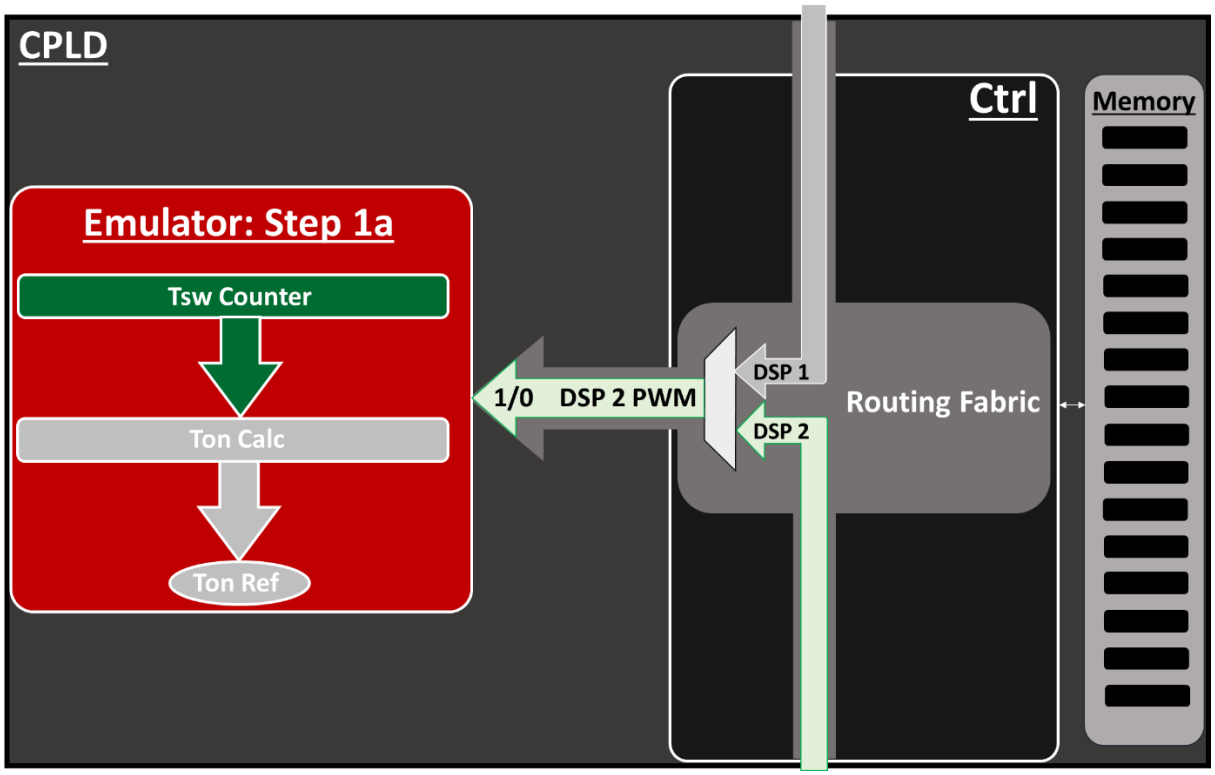

Figure 3.11. Emulation step 1a after hot-patch

3.3 References

[1] Ed Lipiansky, "Combinational Circuits," in Electrical, Electronics, and Digital Hardware Essentials for Scientists and Engineers, IEEE, 2013, pp.456-502, doi: 10.1002/9781118414552.ch7.

[2] Ed Lipiansky, "Digital Design Building Blocks and More Advanced Combinational Circuits," in Electrical, Electronics, and Digital Hardware Essentials for Scientists and Engineers, IEEE, 2013, pp.503-549, doi: 10.1002/9781118414552.ch8.DigitalDesignBookOrRef

[3] Ed Lipiansky, "Sequential Logic and State Machines," in Electrical, Electronics, and Digital Hardware Essentials for Scientists and Engineers, IEEE, 2013, pp.550-602, doi: 10.1002/9781118414552.ch9.

[4] "IEEE Standard Conformance Test Procedures for Equipment Interconnecting Distributed Energy Resources with Electric Power Systems and Associated Interfaces," in IEEE Std 1547.1-2020, vol., no., pp.1-282, 21 May 2020, doi: 10.1109/IEEESTD.2020.9097534.

CHAPTER 4

TEST SETUP AND EXPERIMENTAL RESULTS

4.1 Introduction

This research focuses on integrating a firmware validation security design with emulation and Hot-Patching approach methods. This design consists of an example power electronics topology that resembles the basic concepts of a power electronics inverter. The design is focused on the primary concepts of creating emulations and dangerous firmware command detections using a single controller board. This section will provide the results of an experimental setup that examines the efficacy of the security design in this research. The setup will examine all modules described in the previous chapter, and there will be individual performance results of the emulation, Hot-Patching, and other firmware validation components. These examination scenarios consist of different controller firmware programs that will be labeled as valid or invalid. The results with the valid firmware will be compared to the results with the invalid firmware. This experiment sets up an environment where the uploaded invalid firmware comes from an unintentional mistake or a malicious source. This design does not focus on the cause of the cyber-attack on the communications layer. Furthermore, it does not depend on the validity of the communications since it is an independent security method within the control layer. Therefore, if the invalid firmware is accidental or malicious, the results of the security design will be the same and will protect the power electronics device from harm.

4.2 Test Setup

The security design architecture was described in chapter 3, including six components: Bus Interface, Communication Interface, Bootloader, Hot-Patching, Firmware Validation, and Emulation. This experimental setup will test the Hot-Patching, Firmware Validation, and

Emulation separately since these are the most significant contributors to the security design. The experimental environment consists of the UCB with the online security system, the active inverter, a computer containing the user interface in LabVIEW, and the oscilloscope to view the measurements of the active inverter.

This test setup consists of the UCB controlling the active inverter that will be implemented with the OPAL-RT. As described in section 2.4, the OPAL-RT is a real-time simulator often used as a validation tool for power electronics. The setup between the OPAL-RT and the UCB is a CHIL simulation. In the CHIL simulation, the UCB sends the PWM control signals to the digital ports of the OPAL-RT and receives simulated inverter measurements from the analog ports of the OPAL-RT. The UCB contains an IDC-40 connection port with 40 pins that are used for analog and digital signals. The OPAL-RT analog and digital signal ports are separated; therefore, a custom interface board is placed between the OPAL-RT and the UCB. This interface board routs the separated digital and analog signal ports from the OPAL-RT to the single port of the UCB. The interface board will also connect the measurement probes from the oscilloscope to measure the voltage output of the OPAL-RT. This CHIL setup will have the active DSP from the UCB send PWM signals to control the RTS of the inverter in the OPAL-RT. The RTS results from the OPAL-RT are essentially equivalent to the results of a hardware test. The OPAL-RT contains safety measures that protect the CHIL devices in case of unexpected errors. The UCB input ports are rated at 3.3 volts, which means that the OPAL-RT inverter measurement outputs are scaled to the rated 3.3 volts, avoiding damage to the UCB. Applying the OPAL-RT in the test setup removes the hazard of damaging the UCB or a hardware inverter if the security design fails during the experiment.

The UART interface between the UCB and the user applies serial communication protocol. The user shares data with the UCB using LabVIEW as the user interface, shown in Figure 4.1. LabVIEW is used to send firmware files, Hot-Patching commands, Emulation commands, and Bootloading commands to the UCB. Also, LabVIEW is used to receive different messages, alerts, and emulation data from the UCB.

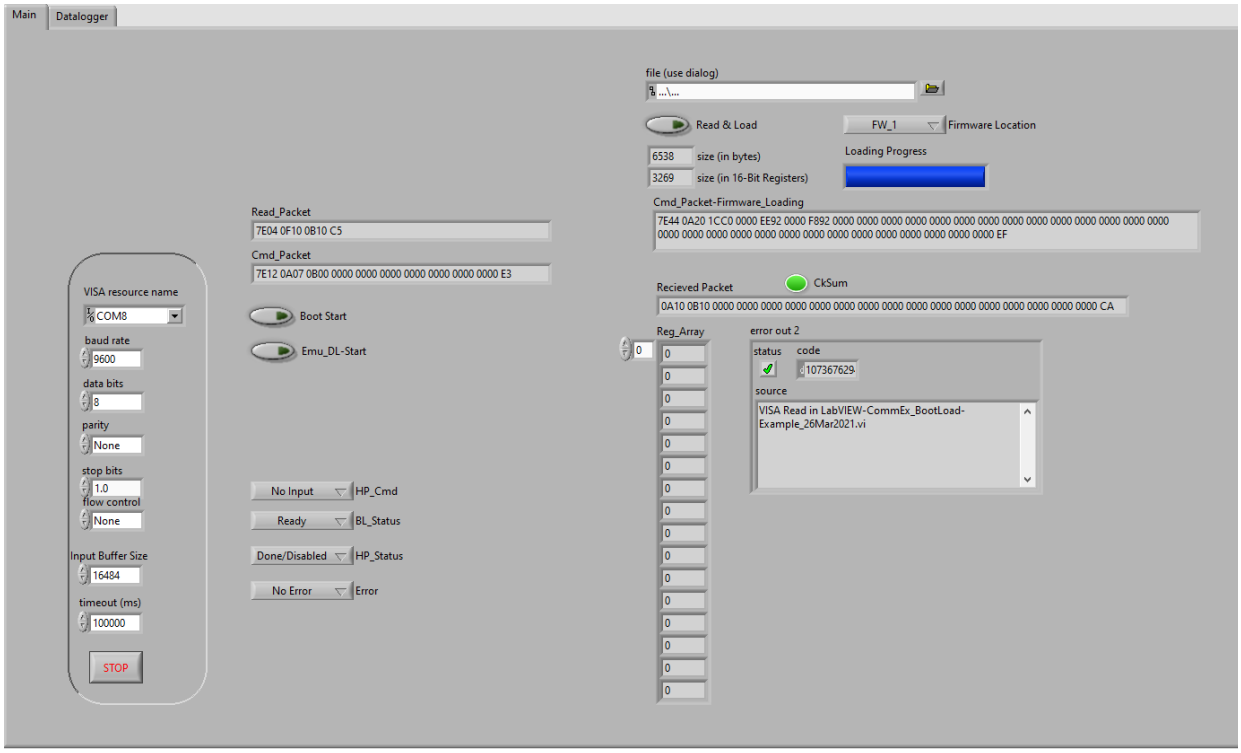

Figure 4.1. LabVIEW main window

The firmware files patched to the DSP contain control code that sends PWM signals to the inverter in the demonstration. Some files are purposely malicious to demonstrate the efficacy of the validation system; other firmware files are safe for the inverter to demonstrate the Hot-Patch and emulation. The safe firmware files configure the DSP with a control algorithm that generates the PWM signals. The control algorithm is exclusive to the inverter considered in this design, explained in section 2.3. The focus of the security design does not include enhanced control system algorithm design; therefore, the control is open loop consisting of a sinusoidal

reference and a fixed switching frequency of 30 kHz. Open-loop control differs from closed-loop by not incorporating the inverter measurement feedback into the control algorithm. Therefore, open-loop control does not account for changes in the inverter behavior, and this control assumes that the inverter output is balanced and consistent. As will be demonstrated in section 4.5, Hot-Patching will swap the DSP signals, affecting the inverter output. PLL integration in the control algorithm would compensate for the effects of the inverter output. Since this security design does not incorporate PLL in the DSP control algorithm, section 4.5 will highlight the effects of the inverter output caused by the Hot-Patch. The Hot-Patch demonstration focuses only on the effects to the inverter caused by the Hot-Patch; therefore, a simple method is used to align both DSP controllers during the Hot-Patch.

4.3 Emulation Demonstration and Results

The emulation feature aims to provide a visual representation of the firmware performance, which differs from the purpose of the firmware validation. The difference is that the firmware validation focuses on detecting control commands that would severely damage the inverter. In contrast, the emulation focuses on displaying the performance of the firmware after it has been verified as safe from severely damaging the inverter. The significance of the emulation approach in the security design will be demonstrated through different scenarios, which assess the validity of the firmware performance. This assessment consists of three categories: fundamental frequency, a phase shift between the three phases, and a specific amplitude range in volts. The emulation is designed to reference generic 3-Phase inverter emulations as an additional tool for firmware validation. Therefore, the focus is to provide a generic visual of this inverter emulation output. As such, the reference output of the emulation are three sinusoidal waveforms with a positive amplitude of 24 volts, a 60Hz fundamental frequency, and each phase

shifted 120 degrees from the other. This reference design may be adjusted and enhanced for more specific applications, as explained in section 5.2. Each category mentioned will show the invalid results displayed by the emulation and will be compared to Figure 4.2. This figure illustrates the emulation output controlled by a valid firmware program.

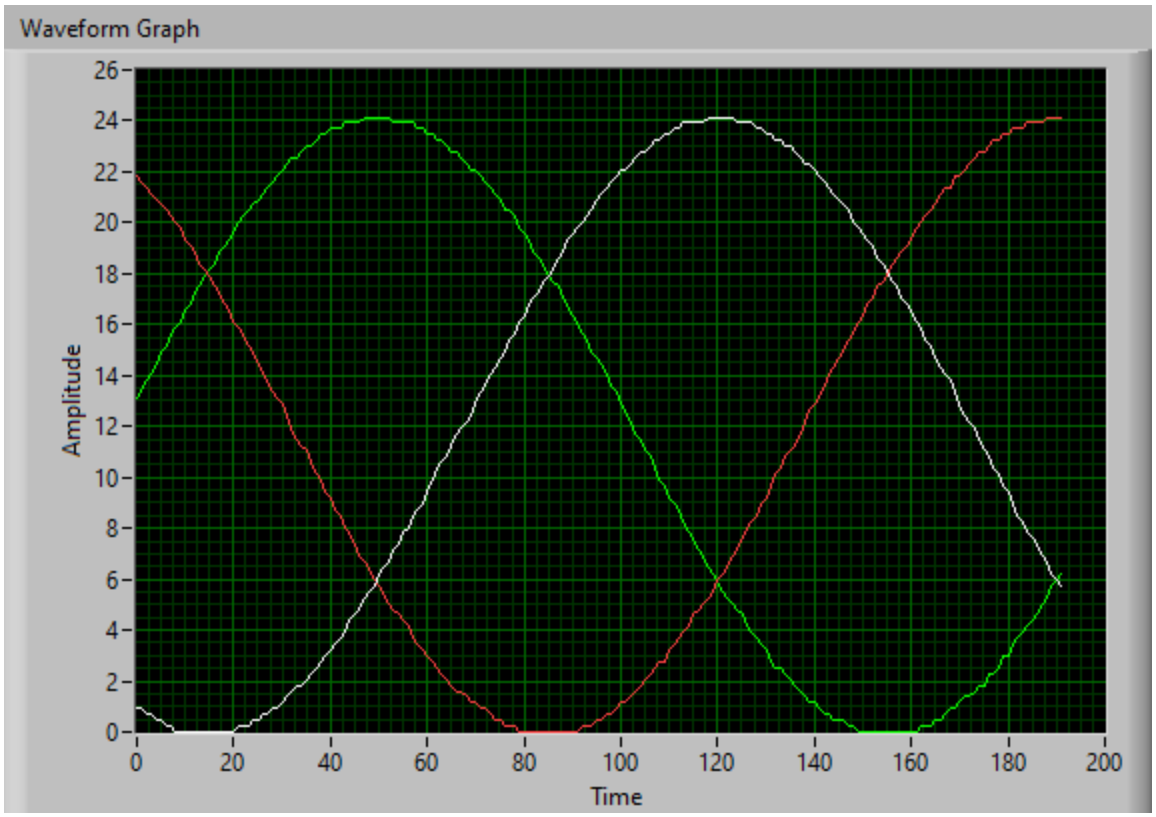

Figure 4.2. Emulation result reference

The emulation begins once a new firmware is patched in the stand-by DSP and verified that it would not damage the inverter. During this time, the emulation is continuously functioning in real-time. When the emulation enable button is pressed in LabVIEW, shown in Figure 4.3, the CPLD is notified to take the sample of the emulation and send it to the user interface. The emulation sample is collected by pressing the button in LabVIEW shown in Figure 4.4. Once the data is collected, the sample is displayed in the waveform graph, shown in Figure 4.5. This first emulation output illustrates each output phase of the 3-Phase inverter, where the output contains

a frequency of 30Hz instead of 60Hz. With this visual, the user can determine that this firmware is not performing correctly; therefore, the Hot-Patch must be denied adjusting the firmware. Once the firmware is adjusted, the same procedure is followed to view the emulation sample shown in Figure 4.2. This figure illustrates the emulation sample with the correct 60Hz frequency. At this time, the user can confirm the Hot-Patch procedure since the firmware is working correctly.

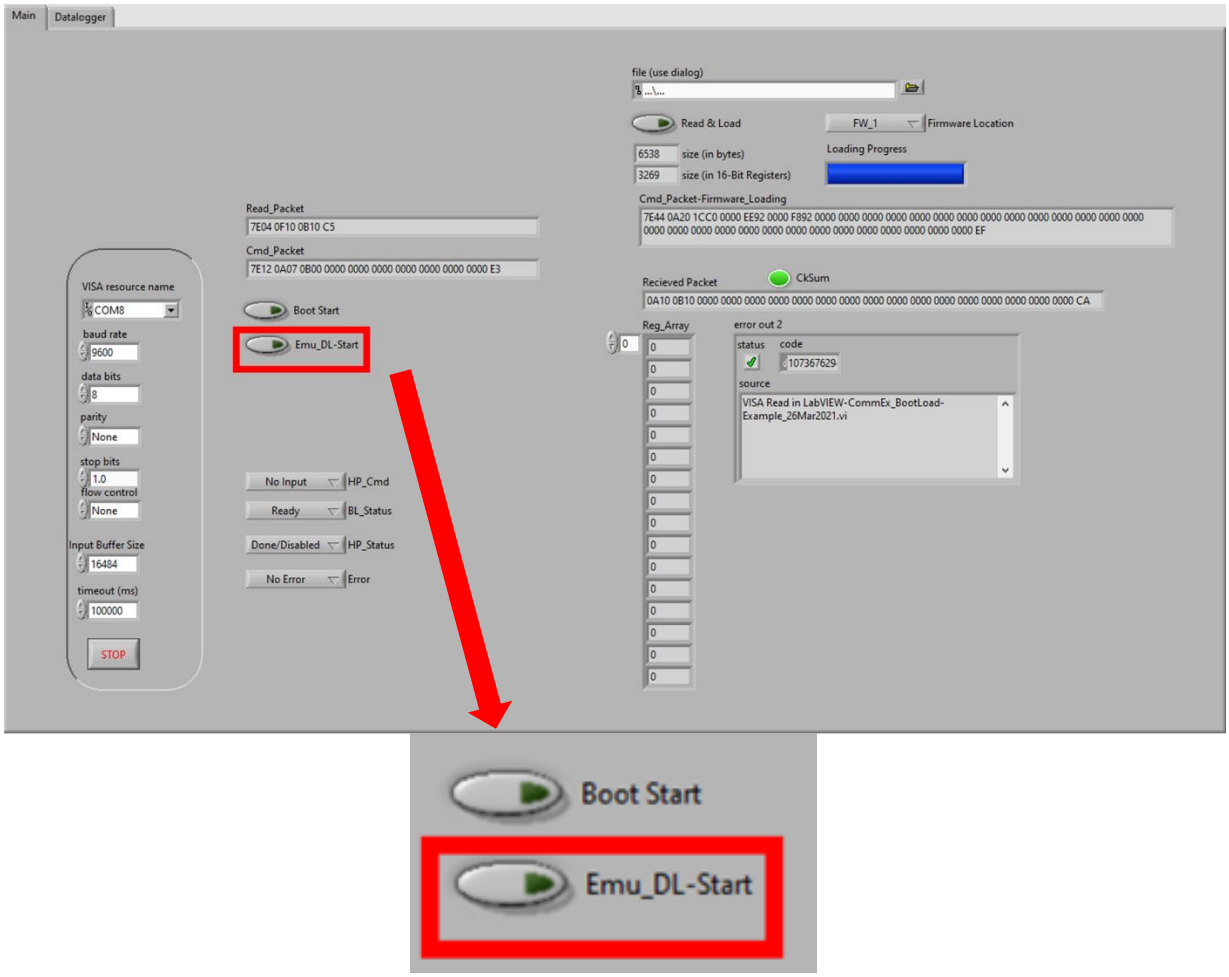

Figure 4.3. Start saving emulation data LabVIEW button

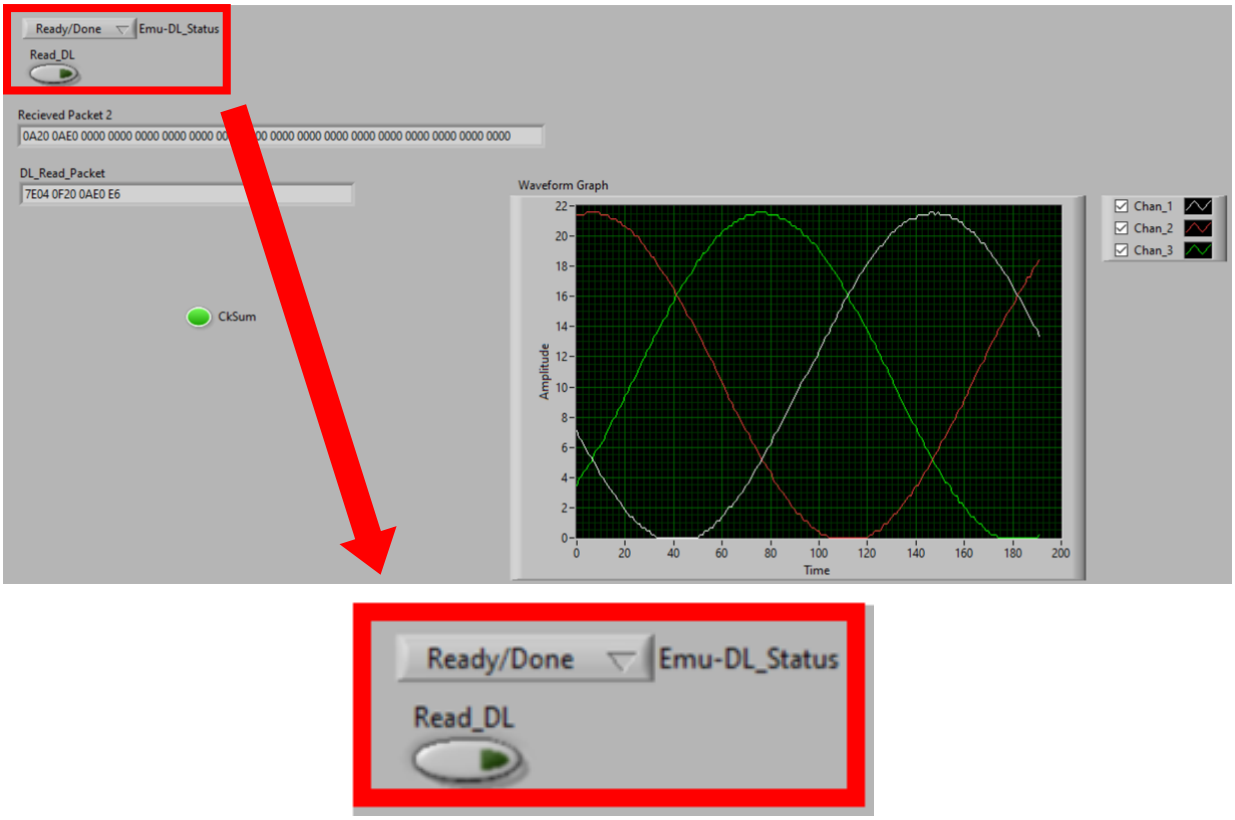

Figure 4.4. Emulation data capture LabVIEW button

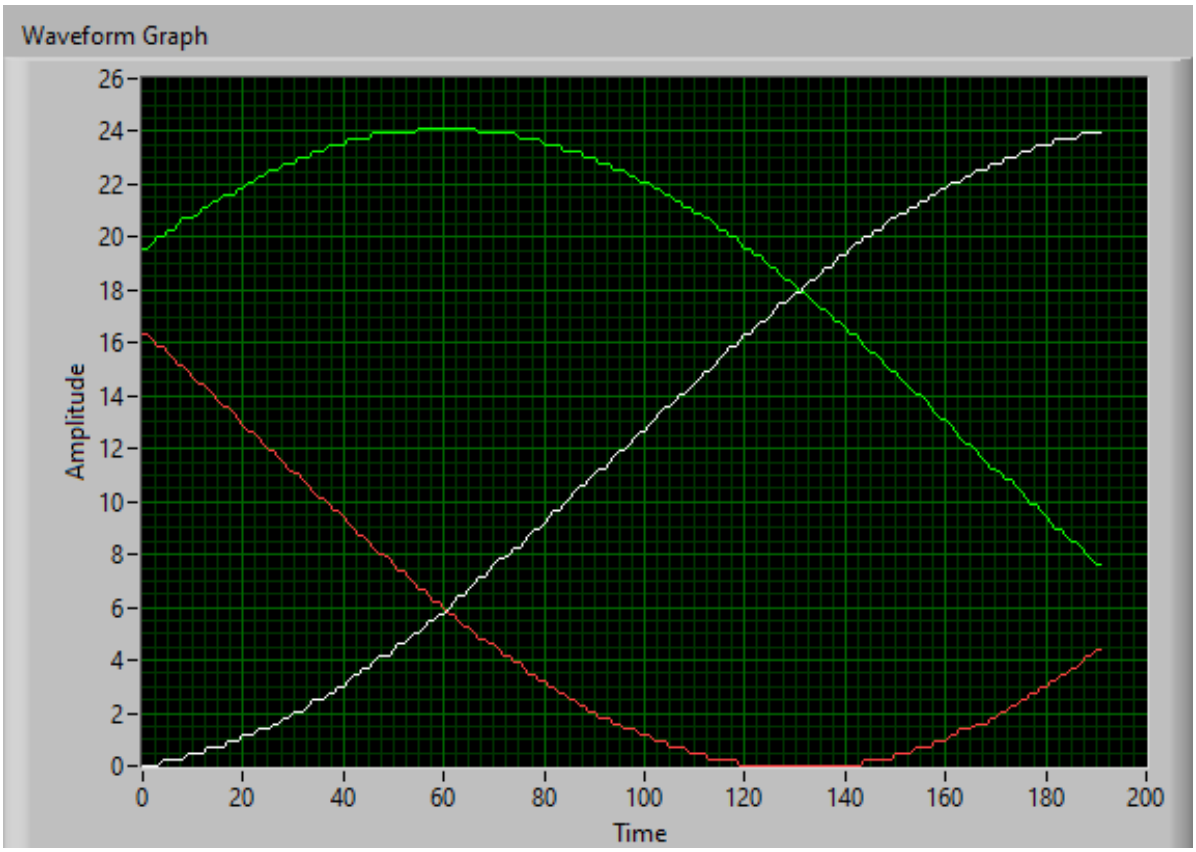

Figure 4.5. Emulation result with 30Hz frequency

The next category to be tested is the 120 degrees phase shift between each output phase of the 3-Phase inverter. The same procedure is followed to start and collect the emulation sampling process. For this experiment, two scenarios will be covered by applying different DSP firmware programs to the emulation. These firmware programs will not trigger an error in the firmware validation, but the emulation will illustrate the defects. The first firmware is purposely programmed with three unbalanced sinusoidal reference phase shifts for the PWM commands. This program causes the inverter output phases to be unbalanced, which will be evaluated through emulation. The output of the emulation should consist of three sinusoidal waveforms, each 120 degrees apart. As mentioned in section 3.2.6, the three-phase output of the inverter requires a balanced phase shift to keep the average inverter output balanced and stable.

Otherwise, unstable inverter output commands could negatively impact other hardware devices or a section of the electric grid. The result of the emulation output controlled by the imbalanced PWM signals is illustrated in Figure 4.6. From this figure, it is noticeable that the sinusoidal waveforms are shifted at different degrees. From this visual, the Hot-Patch should be denied. Once the adjusting the firmware program, the emulation output is illustrated in Figure 4.2, representing a valid firmware.

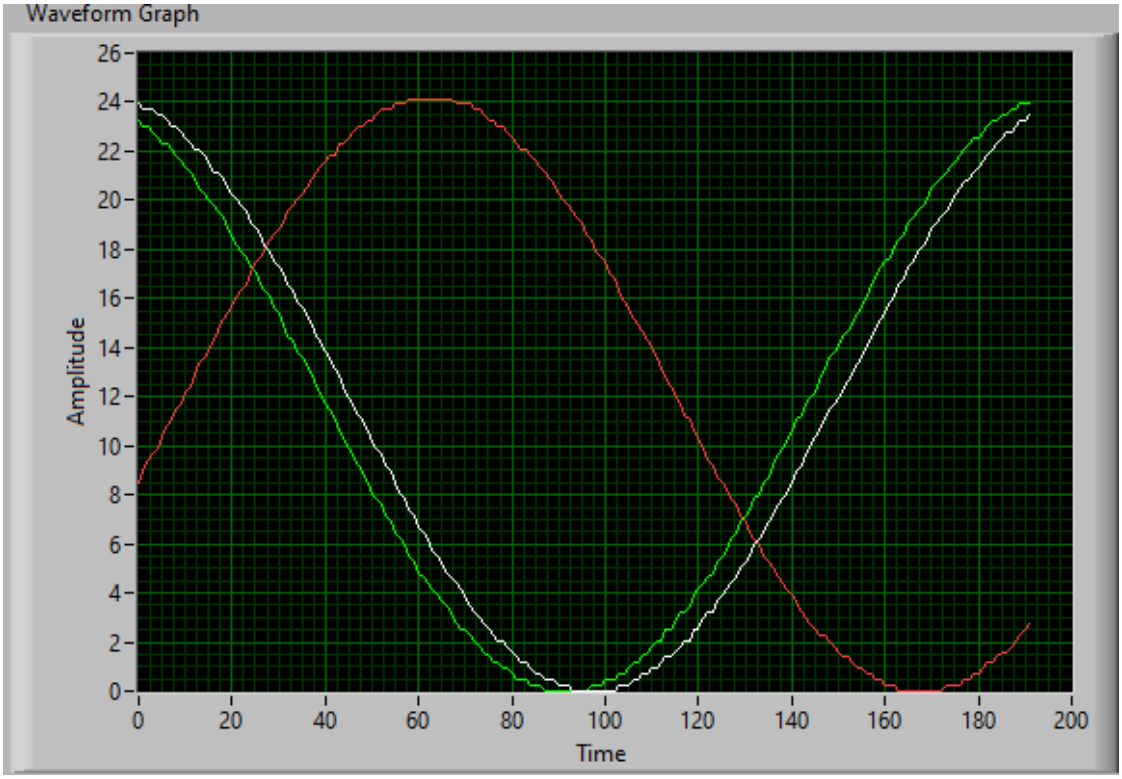

Figure 4.6. Emulation result with phase unbalance

The last category to evaluate consists of the emulated voltage amplitude. The amplitude range may vary depending on the application and is controlled in multiple methods. The output amplitude control may be through additional power electronics and electrical components or the control algorithm. For example, Photo Voltaic (PV) arrays use sunlight to generate DC voltage and are used as an input source to inverters. The PV arrays can connect to Boost converters to manage the DC input of the inverter, to manage the maximum amplitude of the inverter output.

More on this method is found in [2]. However, this design focuses on controlling the amplitude through general PWM signal generation, as described in section 2.3. The Duty Cycle percent manages the voltage output of the inverter; therefore, the maximum output can be 100% of the inverter input. Managing the amplitude depends on the application since some applications may require less maximum voltage than other applications. Moreover, this reference design allows for the user to verify if the control algorithm will provide the necessary inverter voltage output for their application. Figure 4.7 illustrates the emulation output with 50% voltage amplitude.

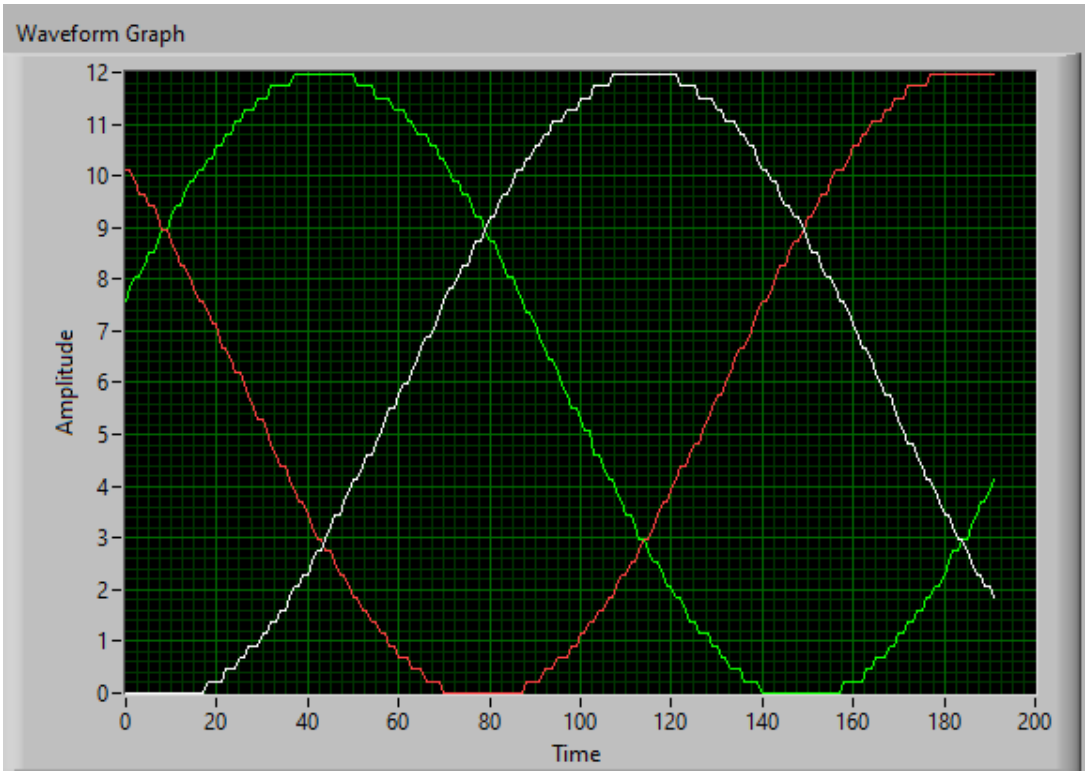

Figure 4.7. Emulation result with 50% maximum amplitude

The emulation results demonstrated are examples of how the output of the inverter may appear using a new firmware control. These emulation results are performed with no interruptions to the active controller. This reference emulation approach can be applied with other more enhanced control algorithms, including closed loop. These demonstrations provide evidence of the emulation generating realistic results of an inverter output in real-time. Using

these results, the user will determine if the emulated output is safe for the physical system before activating the firmware to control the inverter.

4.4 Firmware Validation Demonstration and Results

The firmware validation will be demonstrated in two ways, with a valid firmware and with malicious firmware. The firmware file is sent from LabVIEW to the UCB via the UART interface to begin the demonstration. Figure 4.8 shows the LabVIEW page where the user selects the firmware and a progress bar to know when the entire file has been sent to the UCB. When the firmware file is saved in the CPLD, the Bootload is enabled through LabVIEW, shown in Figure 4.9. LabVIEW sends the data to the allocated register in the DP-RAM when the Bootload enable button is pressed. The Bootloader feature in the CPLD pulls this data and begins the firmware patching process. When the Bootloader is enabled, the progress status is sent to LabVIEW stating that the Bootloader is Loading, shown in Figure 4.10. When the DSP is patched with the new firmware, the firmware validation begins. The process consists of evaluating the DSP PWM signals by detecting short circuit commands. For the inverter in this design, three potential switching modes cause a short circuit, illustrated in Table 4.1. Therefore, the short circuit detection compares the illustrated switching combinations. The duty cycle percent varies in every switching period, so the feature prechecks all the switching periods with different duty cycle percentages to detect a short circuit before moving to the next task. If the validation feature does not detect a short circuit, the user is notified that the Hot-Patch status is ready. Figure 4.11 shows the result of the first firmware validation where the Hot-Patch status is ready.

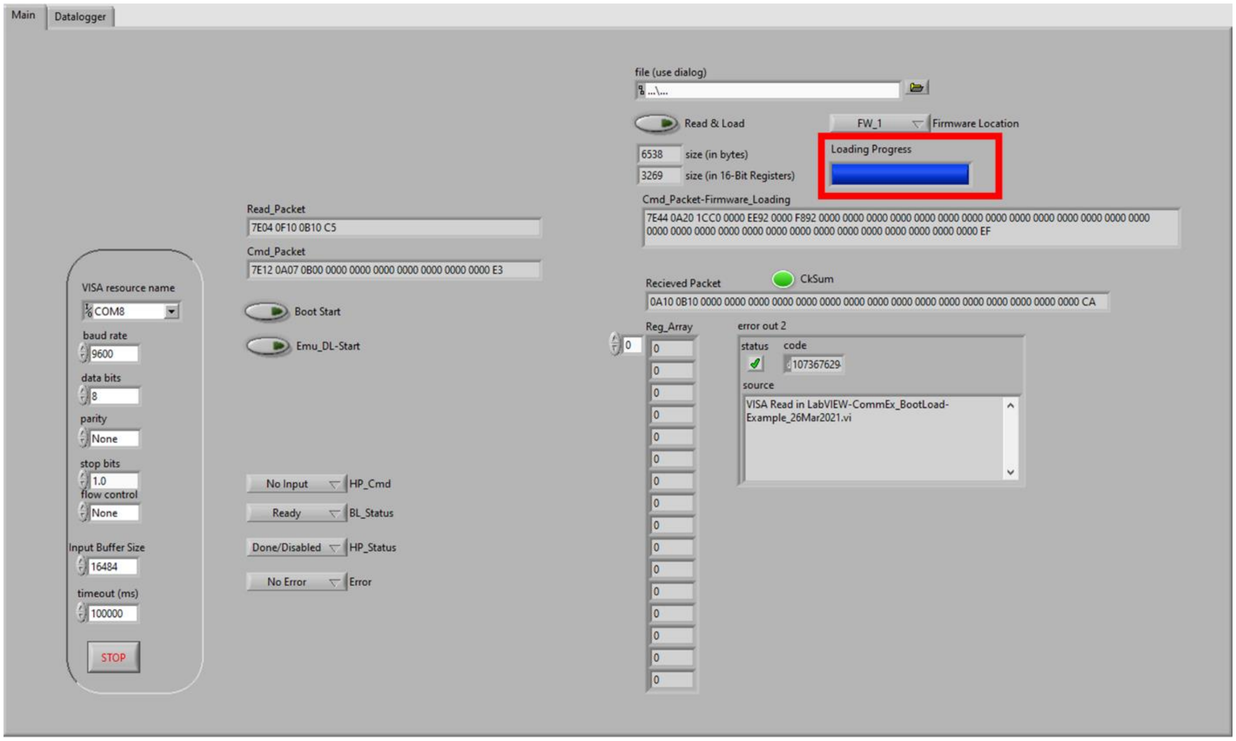

Figure 4.8. Firmware data sending progress bar

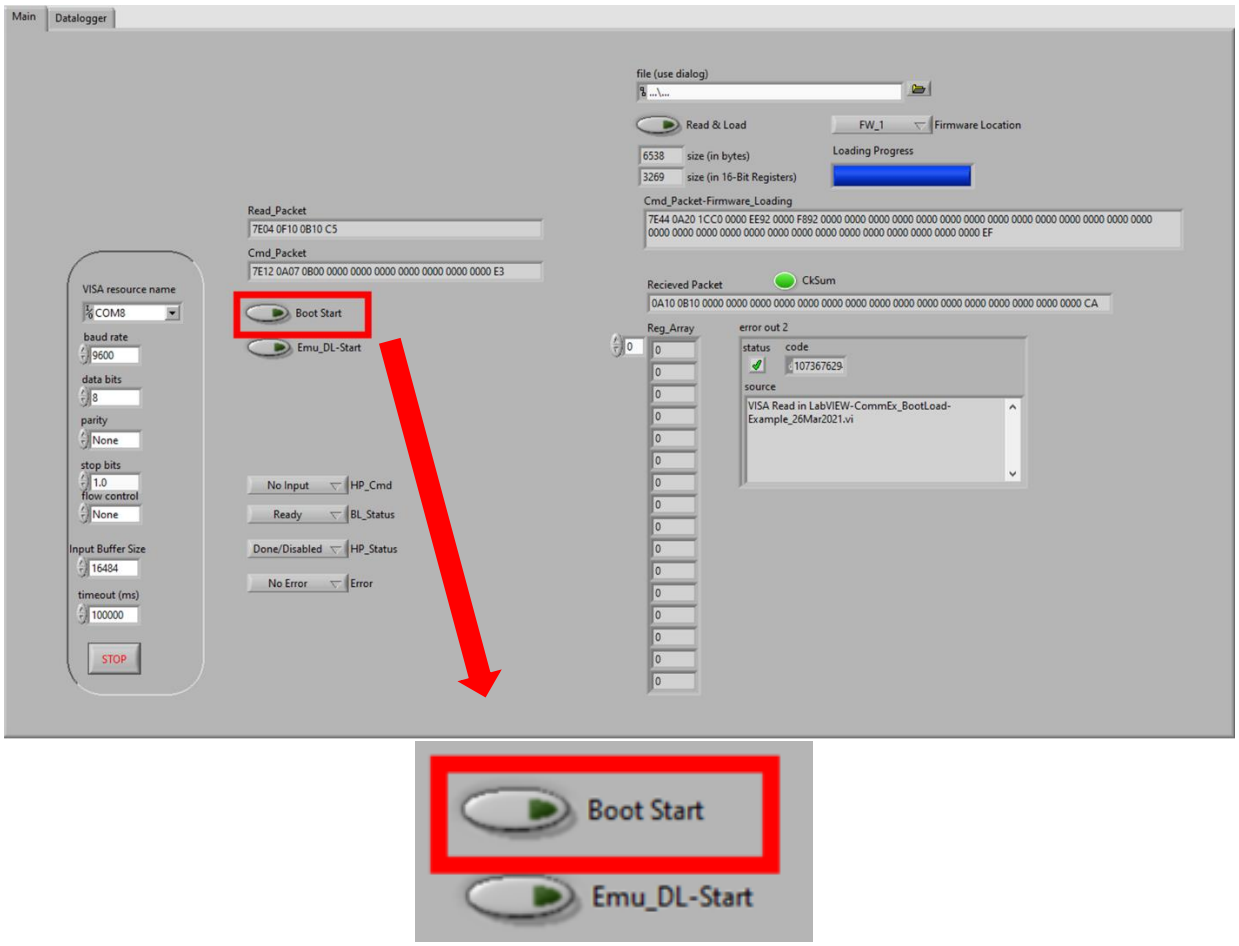

Figure 4.9. Bootload start button

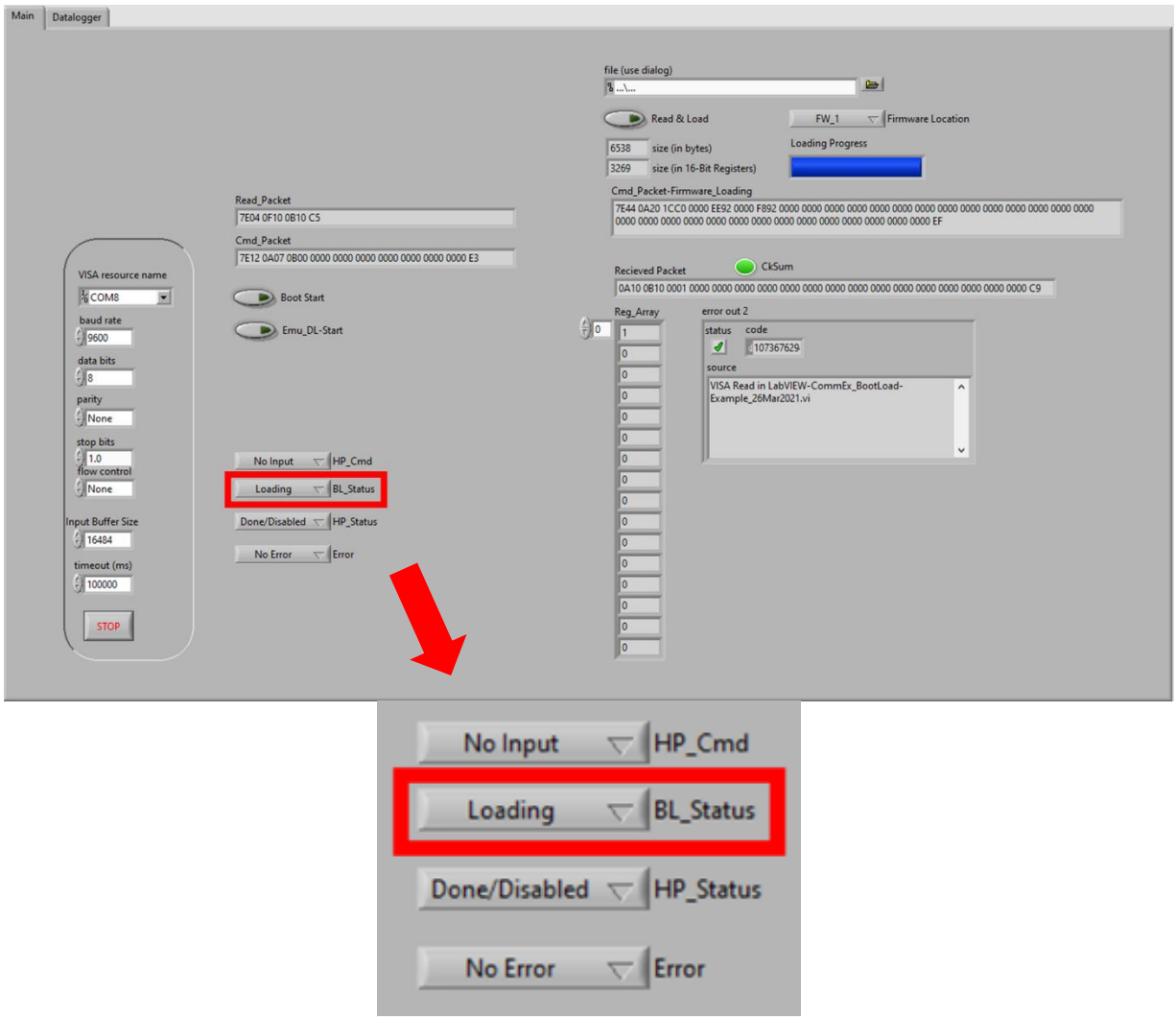

Figure 4.10. Bootload loading status

Table 4.1. 3-phase inverter short circuit switching combinations

Vin	**Phase A**		**Phase B**		**Phase C**		**Vout**		
Vin	**SW1**	**SW2**	**SW3**	**SW4**	**SW5**	**SW6**	**VA**	**VB**	**VC**
Vdc	ON (1)	ON (1)	OFF (0)	OFF (0)	OFF (0)	OFF (0)	Short Circuit	-	-
Vdc	OFF (0)	OFF (0)	ON (1)	ON (1)	OFF (0)	OFF (0)	-	Short Circuit	-
Vdc	OFF (0)	OFF (0)	OFF (0)	OFF (0)	ON (1)	ON (1)	-	-	Short Circuit

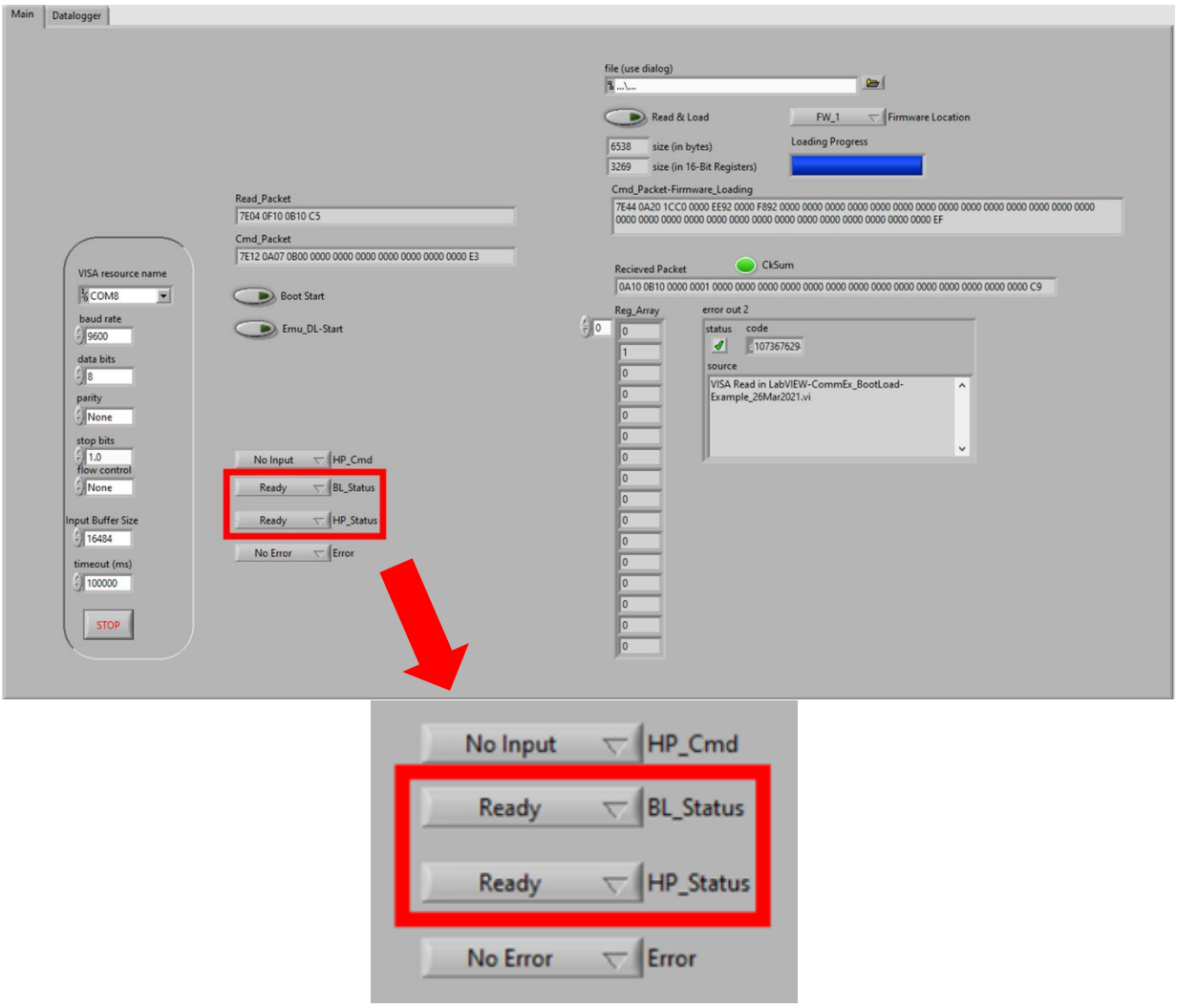

Figure 4.11. Hot-patch ready status

The second firmware validation test involves malicious firmware. This firmware programs the DSP to command a short circuit on the inverter, which the validation feature will detect before it is activated on the physical inverter. After patching the firmware in the DSP, the short circuit detection occurs immediately. The system notifies the user about the error in the Hot-Patch status, Bootloader backup status, and Error type status, shown in Figure 4.12. At this time, the validation feature enabled the backup mode on the Bootloader, where the DSP is patched with the backup firmware. The backup firmware is the same firmware patched on the

active DSP, which continuously controls the inverter uninterrupted. When this process is finished, the security system has restarted and is ready for new firmware.

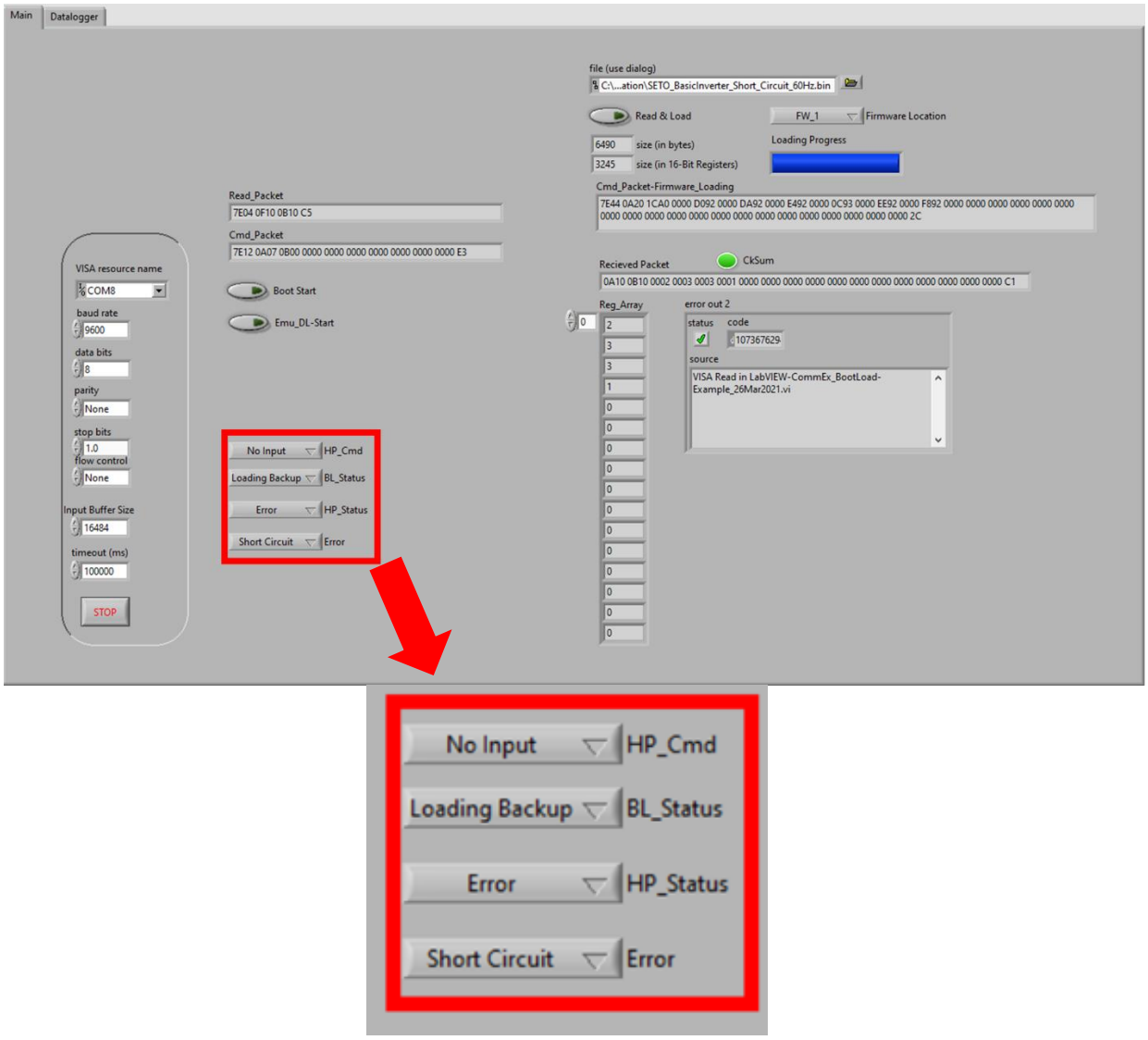

Figure 4.12. Short circuit error status

4.5 Hot-Patch Demonstration and Results

This section demonstrates Hot-Patching by detecting the exact moment of the DSP signal swap through the oscilloscope. As mentioned in the previous section, the Hot-Patch function is only enabled when the firmware validation process detects no errors. Therefore, the firmware

implemented in this demonstration will be considered safe for the inverter. The sole focus of the Hot-Patch demonstration is to illustrate the effects on the inverter output caused by the transition between the DSP signals. After the firmware is validated, the user is notified that the system is ready for a Hot-Patch command. The first scenario involves the user sending a Hot-Patch denial command to the CPLD, which triggers the backup feature on the Bootloader. The second scenario consists of the user sending a Hot-Patch confirmation command to the CPLD, which triggers the Hot-Patch process. The process includes an additional step that reduces the distortion during the DSP swap to compensate for the desynchronization of the two DSP controllers.

This demonstration implements an open-loop control code that does not include PLL algorithms. When the stand-by DSP is patched with the new firmware, the sinusoidal reference of the control will start from phase zero. In contrast, the sinusoidal reference of the active DSP control may be at a different phase since it has been functioning uninterrupted. Figure 4.13 demonstrates the impact of the Hot-Patch with desynchronized DSP controllers. The sinusoidal output after the swap is at a different phase than the output before the swap. This is because the Hot-Patch process includes a step that resets the sinusoidal reference of both DSP controllers to start from phase zero. This reset will cause the same distortion shown in Figure 4.13, but it will occur before the DSP swap. Therefore, the distortion is caused only by the active DSP and not the DSP swap.

Figure 4.13. Desynchronized 3-phase inverter output waveform during hot-patch

The Hot-Patch process in the CPLD triggers the reset of the sinusoidal reference. This trigger lasts one clock cycle in the CPLD, approximately 40 nanoseconds. After the reset, both DSP controllers are synchronized and functioning in the same phase. The Hot-Patch process proceeds to swap the synchronized DSP controllers, where the swap effects are shown in Figure 4.14. This figure illustrates the inverter output with no phase distortion throughout the transition of DSP controllers.

Figure 4.14. Synchronized 3-phase inverter output waveform during hot-patch

The method used in this reference design to synchronize the DSP controllers is not ideal in practical applications. However, the Hot-Patch demonstration does not focus on detecting the desynchronization, but it analyzes any distortions caused by swapping two identical DSP controllers. Therefore, this method allows for the assumption that both DSP controllers are synchronized. Moreover, the result from the Hot-Patch demonstrates that there are no effects in the DSP transition, which means that the Hot-Patch provides a seamless transition during normal conditions with a valid firmware.

4.6 References

[1] Remus Teodorescu; Marco Liserre; Pedro Rodriguez, "Photovoltaic Inverter Structures," in Grid Converters for Photovoltaic and Wind Power Systems, IEEE, 2007, pp.5-29, doi: 10.1002/9780470667057.ch2.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Summary of Conclusions

Cybersecurity for power electronic devices is commonly focused on protecting communications with the devices since it is the primary control. Discovering vulnerabilities in grid-connected devices is a common and serious concern. If communication with the devices is not secured, any discovered vulnerability can be a serious hazard during a cyber-attack. The devices consist of communications, a controller board, and a power electronics device. When a cyber-attack occurs, and the communications are compromised, the rest of the system is at risk. Therefore, this reference security design intends to apply an additional security layer embedded in the controller board. During a cyber-attack, the communications may be compromised, but this embedded security would help mitigate the damage caused by the attack. The reference design contains two types of security approach methods inspired by Digital Twins and Hot-Patching. The Digital Twin concept has multiple definitions describing a type of emulation with Hardware-In-Loop (HIL). This design is incorporated in a Universal Control Board consisting of multiple controller processors that collaborate in securing the power electronics inverter. The emulation and Hot-Patching concepts are embedded in the CPLD chip that is part of the UCB. The CPLD allows the security design to be fast, effective, and flexible to different applications. The design is focused on validating the power electronics control firmware online through the mentioned approach strategies while actively controlling the physical power electronics inverter. The design emulates the power electronics inverter and interacts with the control firmware for verification purposes. Different scenarios were evaluated for this design that consisted of

verifying different harmful firmware programs. This evaluated the effectiveness of the emulation, Hot-Patching, and overall firmware validation.

5.2 Recommendations for Future Work

The reference design was developed to accommodate different applications for power electronics. There are a few recommendations for implementing this design with a different application that should be addressed. The emulation is based on the fundamental concepts of a power electronics 3-Phase inverter. However, to emulate other inverter topologies, the design will need to be adjusted. When programming any emulation in the CPLD, this design may be used as a reference to calculate the emulated inverter output, and it may be included in parallel to the other design modules. This design is built to accommodate more modules as each module works independently. Therefore, the program would accept other modules added in parallel, like an emulation of a new inverter topology. This same recommendation is applied to new communication protocols. This reference design consists of a UART SCI module in the CPLD built with a custom data packet system. This packet setup is similar to other standard communication protocols. Therefore, if Modbus TCP is implemented, the reference design will need to be adjusted. This adjustment would require only the communication module to be changed and added parallel to the system's other modules. The design modules' adjustment depends on the required application, but this design references how to implement and add additional modules to the security system.

APPENDICIES

APPENDIX A: CODE

A-1 DSP C Code

```
/
#define      DEBUG_EN           1            // Set to 1 for Debug Mode or 0 for
Flash Mode
#define Freq_Clk          150e6  // Clock Frequency (150 MHz)
#define      Freq_Sw                  30           // Set Switching Frequency (kHz)
#define      Freq_Fund_AC 60           // Set Fundamental Output Frequency (60 Hz)
#define AC_Ref                   340          // Set Peak Output AC Magnitude
Reference

//<! Includes
#include "DSP28x_Project.h"
#include "math.h"

// ADC start parameters
#if (CPU_FRQ_150MHZ)     // Default - 150 MHz SYSCLKOUT
  #define ADC_MODCLK 0x3 // HSPCLK = SYSCLKOUT/2*ADC_MODCLK2 = 150/(2*3)   = 25.0 MHz
#endif
#if (CPU_FRQ_100MHZ)
  #define ADC_MODCLK 0x2 // HSPCLK = SYSCLKOUT/2*ADC_MODCLK2 = 100/(2*2)   = 25.0 MHz
#endif
#define ADC_CKPS   0x1   // ADC module clock = HSPCLK/2*ADC_CKPS   = 25.0MHz/(1*2) =
12.5MHz


////Prototype statements for functions
void Gpio_Init(void);            //!< Initialize GPIO Outputs and Inputs
void ADC_Init(void);             //!< Initialize ADC
void InitEPWM1(void);            //!< Initialize EPWM1 Module
void InitEPWM2(void);            //!< Initialize EPWM2 Module
void InitEPWM3(void);            //!< Initialize EPWM3 Module
void InitEPWM4(void);            //!< Initialize EPWM4 Module
void InitFlash();                //!< Initialize Flash Module
void MemCopy();                        //!< Flash Memory Copy function


////Interrupts
interrupt void adc_isr(void);          //!< Interrupt routine for ADC

////External Functions
extern Uint16 RamfuncsLoadStart; //!< Variable for MemCopy for loading data into
flash.
extern Uint16 RamfuncsLoadEnd;         //!< Variable for MemCopy for loading data
into flash.
extern Uint16 RamfuncsRunStart;        //!< Variable for MemCopy for loading data
into flash.

////For Debug/Flash Selection
#if DEBUG_EN > 0
```

```
//debug mode
#else
#pragma CODE_SECTION(adc_isr, "ramfuncs"); // flash mode
#endif


////Global variables
float V_ADC_Max = 3.0;          //!< Max ADC Voltage for DSP ADC Modules
Uint16 D_ADC_Max =4095;         //!< Max digital represented ADC Voltage (2^12 -1)

Uint16 V_ADC[8];                //!<  ADC Register Values

//Registers
Uint16 Reg[64];                       //!< Registers for command and data


//// Constants for setting the correct parameters based on switching frequency////
//Prd_PWM = 2 x TBPRD × TTBCLK => TBPRD = [Freq_PWM/(2*TTBCLK)] -1
//Example1(30kHz): PWM_TBPRD = [(30e3^-1)/(2*60e6^-1)-1] = 999
//Example1(1kHz): PWM_TBPRD = [(1e3^-1)/(2*60e6^-1)-1] = 29999
float FreqClk;                        //!< Clock Freq (150 MHz)
float FreqPWM;                        //!< Switching Freq
int FreqFund_AC;                //!< Fundamental AC Freq

Uint16 PWM_TBPRD;               //!< PWM Period; \n Prd_PWM = 2 x TBPRD × TTBCLK =>
TBPRD = [Freq_PWM/(2*TTBCLK)] -1 \n Example1(30kHz): PWM_TBPRD = [(30e3^-1)/(2*60e6^-
1)-1] = 999 \n
float Prd_Sw;                   //!< Switching cycles per fundamental period Ex:(30
kHz / 60Hz = 500)
float Prd_Sw_d4;                //!< Switching cycles per fundamental period divided
by 4  Ex:((30 kHz / 60Hz)/4 = 125) \n This is used to sample at max and min AC
values.


int i; //!< Generic Counter variable

//ADC Values
float V_Buck;//!< Converted ADC Value for Buck
float V_Boost;      //!< Converted ADC Value for Boost
float V_PhA; //!< Converted ADC Value for Phase_A for 3-phase inverter
float V_PhB; //!< Converted ADC Value for Phase_B for 3-phase inverter
float V_PhC; //!< Converted ADC Value for Phase_C for 3-phase inverter
float V_DCin;//!< Converted ADC Value for VDC Input
float V_Pot; //!< Converted ADC Value for Potentiometer
float V_Scale1; //!< Scaling factor for ADC Values (V_ADC_Max/D_ADC_Max)
float V_Scale2; //!< Scaling factor for ADC Values including Op-Amp Scaling
V_Scale1*17.1551; // V_ADC*(3/4095) * (61.9e3/(1e6+61.9e3))^-1



//3-Phase Inverter Variables
float AC_Mag;       //!< AC Magnitude reference (V)
float AC_Freq_Fund; //!< AC Fundamental Frequency Ref (Hz)
float pi;                 //!< Value of PI (3.1415926535898)
float pi_2;               //!< Value of 2 x PI
```

```
float phase_offset; //!< Phase offset for 3-phase output voltages 120 degree offset
(2/3)*pi
float phase2_offset;
float Step_sin;           //!< Used in calculating the sin output for 3-phase
inverter; Step_sin = (Sin_Counter/Prd_Sw)*pi_2;
float Sine1;        //!< Duty to generate Duty Cycle for 3-Phase PWM
float Sine2;        //!< Duty to generate Duty Cycle for 3-Phase PWM
float Sine3;        //!< Duty to generate Duty Cycle for 3-Phase PWM
float Duty_AC;            //!< Duty Cycle for 3-Phase PWM; Based on AC Magnitude
reference and PWM Frequency.
Uint16 Sin_Counter; //!< Used to calculate the value of sin output; Based on number
of interrupts
Uint16 DeadTime;    //!< Deadtime; 30 clk cycles = (150e6^-1)*30 => 200ns


////Main function
void main(void)
{


    //Setup the initial Switching frequency
    //Prd_PWM = 2 x TBPRD × TTBCLK => TBPRD = [Freq_PWM/(2*TTBCLK)] -1
    //Example1(30kHz): PWM_TBPRD = [(30e3^-1)/(2*60e6^-1)-1] = 999
    //Example1(1kHz): PWM_TBPRD = [(1e3^-1)/(2*60e6^-1)-1] = 29999
    FreqClk = Freq_Clk;                         //Clock Freq (150 MHz)


    FreqPWM = 30e3;                                     //Switching Freq in Hz
    FreqFund_AC = Freq_Fund_AC;                               //Fundamental Freq
    PWM_TBPRD = (1/FreqPWM)/(2*(1/FreqClk))-1;      //Set initial PWM Period
    Prd_Sw = FreqPWM/FreqFund_AC;                             //Switching cycles per
fundamental period Ex:(30 kHz / 60Hz = 500)
    Prd_Sw_d4 = Prd_Sw/4;                                     //Switching cycles per 1/4
fundamental period
    DeadTime = 10;                                                  //Deadtime; 30 clk
cycles = (150e6^-1)*30 => 200ns

      pi=3.1415926535898;
      pi_2=2*pi;
      phase_offset=(2.0/3.0)*pi;
      //phase_offset=(7.0/9.0)*pi;
      //phase2_offset=(5.0/9.0)*pi;
      //phase2_offset=(1.0/18.0)*pi;


      // Step 1. Initialize System Control:
      // PLL, WatchDog, enable Peripheral Clocks
      InitSysCtrl();

      EALLOW;
      SysCtrlRegs.HISPCP.all = ADC_MODCLK;    // HSPCLK = SYSCLKOUT/ADC_MODCLK
      EDIS;

      // Step 2. Initialize GPIO:
      // InitGpio();
```

```
// For this example, only init the pins for the SCI-A port.
// This function is found in the DSP2833x_Sci.c file.
//InitSciaGpio();

// Step 3. Clear all interrupts and initialize PIE vector table:
// Disable CPU interrupts
DINT;

// Initialize the PIE control registers to their default state.
// The default state is all PIE interrupts disabled and flags
// are cleared.
// This function is found in the DSP2803x_PieCtrl.c file.
InitPieCtrl();

// Disable CPU interrupts and clear all CPU interrupt flags:
IER = 0x0000;
IFR = 0x0000;

// Initialize the PIE vector table with pointers to the shell Interrupt
// Service Routines (ISR).
// This will populate the entire table, even if the interrupt
// is not used in this example.  This is useful for debug purposes.
InitPieVectTable();

// Interrupts that are used in this example are re-mapped to
// ISR functions found within this file.
EALLOW;  // This is needed to write to EALLOW protected registers
//PieVectTable.EPWM1_INT = &epwm1_isr;
//PieVectTable.EPWM2_INT = &epwm2_isr;
//PieVectTable.EPWM3_INT = &epwm3_isr;
PieVectTable.ADCINT = &adc_isr;
EDIS;    // This is needed to disable write to EALLOW protected registers

EALLOW;
Gpio_Init(); //Initialize GPIO
//InitSciaGpio();
EDIS;


// Debug or flash mode selection
#if DEBUG_EN > 0
      // do nothing, as default debug mode
#else
      MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
      InitFlash();
#endif

ADC_Init(); // Initialize ADC

InitEPwm1Gpio();
InitEPwm2Gpio();
InitEPwm3Gpio();
InitEPwm4Gpio();

// Step 5. User specific code, enable interrupts:
```

```
    //scia_fifo_init();      // Initialize the SCI FIFO
    //scia_echoback_init();  // Initialize SCI for echoback

    //LSPCLK = 50MHZ
                //SpiaRegs.SPIBRR.bit.SPI_BIT_RATE = 0x0031; //set the SPI baud rate
                //SpiaRegs.SPIBRR.all = 0x007F;
                //SpiaRegs.SPIBRR.all = 0x007F;
                //7Fh (R/W) = SPI Baud Rate = LSPCLK/128
                //13h (R/W) = SPI Baud Rate = LSPCLK/20

EALLOW;

FreqPWM = 30e3;                              //Switching Freq in Hz
FreqFund_AC = Freq_Fund_AC;                  //Fundamental Freq
PWM_TBPRD = (1/FreqPWM)/(2*(1/FreqClk))-1;  //Set initial PWM Period
Prd_Sw = FreqPWM/FreqFund_AC;                //Switching cycles per fundamental period
Ex:(30 kHz / 60Hz = 500)
Prd_Sw_d4 = Prd_Sw/4;                        //Switching cycles per 1/4 fundamental
period
DeadTime = 30;                               //Deadtime; 30 clk cycles = (150e6^-1)*30
=> 200ns


      //Main Loop
      for(;;)
    {


          PWM_TBPRD = (1/FreqPWM)/(2*(1/FreqClk))-1;  //Set initial PWM Period


          //Update PWM Freq
          EPwm1Regs.TBPRD = PWM_TBPRD;
          EPwm2Regs.TBPRD = PWM_TBPRD;
          EPwm3Regs.TBPRD = PWM_TBPRD;
          EPwm4Regs.TBPRD = PWM_TBPRD;



            //End Update Values
    }
}

void ADC_Init(void)
{
      // Configure ADC
      EALLOW;
      PieVectTable.ADCINT = &adc_isr;
      EDIS;

      InitAdc();

      IER |= M_INT1;                                    // Enable CPU Interrupt 1
      //PieCtrlRegs.PIEIER1.bit.INTx1 = 1;    // Enable INT 1.1 in the PIE
      PieCtrlRegs.PIEIER1.bit.INTx6 = 1;
```

```
	EINT;                                                  // Enable Global interrupt
INTM
	ERTM;                                                  // Enable Global realtime
interrupt DBGM

	EALLOW;
	// Specific ADC setup for this example:
		//Sample and Hold window
		AdcRegs.ADCTRL1.bit.ACQ_PS = 0x1;              //Window length(1+#of
Cycles)
		//ADC Clock Rate
		AdcRegs.ADCTRL3.bit.ADCCLKPS = ADC_CKPS; //0 works faster but may result
in errors	// 12.5 MHz 80ns
		AdcRegs.ADCTRL1.bit.SEQ_CASC = 1;        // 1  Cascaded mode
		AdcRegs.ADCTRL1.bit.SEQ_OVRD = 0;
		//Specify # of Conversions
		AdcRegs.ADCMAXCONV.bit.MAX_CONV1 = 0x8; // convert only used channels

		//AdcRegs.ADCTRL3.bit.SMODE_SEL = 0; //setup ADC for sequential sampling

		//Convert used channels
		AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x0; //ADCA0
		AdcRegs.ADCCHSELSEQ1.bit.CONV01 = 0x1; //ADCA1
		AdcRegs.ADCCHSELSEQ1.bit.CONV02 = 0x2; //ADCA2
		AdcRegs.ADCCHSELSEQ1.bit.CONV03 = 0x3; //ADCA3
		AdcRegs.ADCCHSELSEQ2.bit.CONV04 = 0x4; //ADCA4
		AdcRegs.ADCCHSELSEQ2.bit.CONV05 = 0x5; //ADCA5
		AdcRegs.ADCCHSELSEQ2.bit.CONV06 = 0x6; //ADCA6
		AdcRegs.ADCCHSELSEQ2.bit.CONV07 = 0x7; //ADCA7

		AdcRegs.ADCTRL1.bit.CONT_RUN = 0;       // Setup start/stop run
		AdcRegs.ADCTRL2.bit.EPWM_SOCA_SEQ1 = 1;// Enable SOCA from ePWM to start
SEQ1
		AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1;  // Enable SEQ1 interrupt (every
End Of Sequence)
		AdcRegs.ADCTRL2.bit.INT_MOD_SEQ1 = 0;  //
	EDIS;

	//ADC Triggering Setup
	EPwm1Regs.ETSEL.bit.SOCAEN = 1;
	EPwm1Regs.ETSEL.bit.SOCASEL = 0x2;//4; // 2=Select SOC from CPMA on Period;
4=Select SOC from from CPMA on upcount
	EPwm1Regs.ETPS.bit.SOCAPRD = 1;        // Generate pulse every event
}

void Gpio_Init(void)
{
	// Step 2. Initalize GPIO:
	// Initialize GPIO
	EALLOW;

	GpioCtrlRegs.GPAMUX1.all = 0x0000;     // GPIO functionality GPIO0-GPIO15
	GpioCtrlRegs.GPAMUX2.all = 0x0000;     // GPIO functionality GPIO16-GPIO31
	GpioCtrlRegs.GPBMUX1.all = 0x0000;     // GPIO functionality GPIO32-GPIO44
```

```
        GpioCtrlRegs.GPADIR.all = 0xFFFF; //0x0000;      // GPIO0-GPIO31 are inputs
        GpioCtrlRegs.GPBDIR.all = 0xFFFF;      // GPIO32-GPIO44 are inputs

        GpioCtrlRegs.GPAQSEL1.all = 0x0000;    // GPIO0-GPIO15 Synch to SYSCLKOUT
        GpioCtrlRegs.GPAQSEL2.all = 0x0000;    // GPIO16-GPIO31 Synch to SYSCLKOUT
        GpioCtrlRegs.GPBQSEL1.all = 0x0000;    // GPIO32-GPIO44 Synch to SYSCLKOUT

        GpioCtrlRegs.GPAPUD.all = 0xFFFF; //0x0000;      // Pullup's disabled GPIO0-
GPIO31
        GpioCtrlRegs.GPBPUD.all = 0xFFFF; //0x0000;      // Pullup's disabled GPIO32-
GPIO44

        //Serial RX/TX Config
        GpioCtrlRegs.GPADIR.bit.GPIO28 = 0;    // Configure as input
        GpioCtrlRegs.GPADIR.bit.GPIO29 = 1;    // Configure as output
        GpioCtrlRegs.GPAPUD.bit.GPIO28 = 1;    // Disable internal pull-up
        GpioCtrlRegs.GPAPUD.bit.GPIO29 = 1;    // Disable internal pull-up

        //PushButton
        GpioCtrlRegs.GPAPUD.bit.GPIO30 = 1; // Disable internal pull-up


        //Switch Setup
        GpioCtrlRegs.GPBDIR.bit.GPIO58 = 0;    // Configure as input (SW1)
        GpioCtrlRegs.GPBDIR.bit.GPIO59 = 0;    // Configure as input (SW2)
        GpioCtrlRegs.GPBDIR.bit.GPIO60 = 0;    // Configure as input (SW3)
        GpioCtrlRegs.GPBDIR.bit.GPIO61 = 0;    // Configure as input (SW4)
        GpioCtrlRegs.GPBPUD.bit.GPIO58 = 0;    // Enable internal pull-up (SW1)
        GpioCtrlRegs.GPBPUD.bit.GPIO59 = 0;    // Enable internal pull-up (SW2)
        GpioCtrlRegs.GPBPUD.bit.GPIO60 = 0;    // Enable internal pull-up (SW3)
        GpioCtrlRegs.GPBPUD.bit.GPIO61 = 0;    // Enable internal pull-up (SW4)



        ////Configure LEDs
        GpioCtrlRegs.GPADIR.bit.GPIO31 = 1;    // Configure as output (Red_Led1 On
Board)
        GpioCtrlRegs.GPBDIR.bit.GPIO34 = 1;    // Configure as output (Red_Led2 On
Board)
        GpioCtrlRegs.GPBDIR.bit.GPIO32 = 1;    // Configure as output (Red_Led3 On
Demo Board)
        GpioCtrlRegs.GPAPUD.bit.GPIO31 = 1;    // Disable internal pull-up
        GpioCtrlRegs.GPBPUD.bit.GPIO34 = 1;    // Disable internal pull-up
        GpioCtrlRegs.GPBPUD.bit.GPIO32 = 1;    // Disable internal pull-up
        GpioDataRegs.GPACLEAR.bit.GPIO31 = 1;  //initialize to zero
        GpioDataRegs.GPBCLEAR.bit.GPIO34 = 1;  //initialize to zero
        GpioDataRegs.GPBCLEAR.bit.GPIO32 = 1;  //initialize to zero


        //Configure SPI
    GpioCtrlRegs.GPAPUD.bit.GPIO16 = 0;   // Enable pull-up on GPIO16 (SPISIMOA)
    GpioCtrlRegs.GPAPUD.bit.GPIO17 = 0;   // Enable pull-up on GPIO17 (SPISOMIA)
    GpioCtrlRegs.GPAPUD.bit.GPIO18 = 0;   // Enable pull-up on GPIO18 (SPICLKA)
    GpioCtrlRegs.GPAPUD.bit.GPIO19 = 0;   // Enable pull-up on GPIO19 (SPISTEA)
```

```
    //GpioCtrlRegs.GPAQSEL2.bit.GPIO16 = 3; // Asynch input GPIO16 (SPISIMOA)
    GpioCtrlRegs.GPAQSEL2.bit.GPIO17 = 3; // Asynch input GPIO17 (SPISOMIA)
    //GpioCtrlRegs.GPAQSEL2.bit.GPIO18 = 3; // Asynch input GPIO18 (SPICLKA)
    //GpioCtrlRegs.GPAQSEL2.bit.GPIO19 = 3; // Asynch input GPIO19 (SPISTEA)

    GpioCtrlRegs.GPAMUX2.bit.GPIO16 = 1; // Configure GPIO16 as SPISIMOA
    GpioCtrlRegs.GPAMUX2.bit.GPIO17 = 1; // Configure GPIO17 as SPISOMIA
    GpioCtrlRegs.GPAMUX2.bit.GPIO18 = 1; // Configure GPIO18 as SPICLKA
    GpioCtrlRegs.GPAMUX2.bit.GPIO19 = 1; // Configure GPIO19 as SPISTEA

    GpioCtrlRegs.GPADIR.bit.GPIO19 = 1;// Configure as output
    GpioCtrlRegs.GPADIR.bit.GPIO16 = 1;// Configure as output
    GpioCtrlRegs.GPADIR.bit.GPIO18 = 1;// Configure as output
    GpioCtrlRegs.GPADIR.bit.GPIO17 = 0;// Configure as input



       EDIS;

       InitEPwm1Gpio();
       InitEPwm2Gpio();
       InitEPwm3Gpio();
       InitEPwm4Gpio();

       EALLOW;
       SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;
       EDIS;

       InitEPWM1();
       InitEPWM2();
       InitEPWM3();
       InitEPWM4();

       EALLOW;
       SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;
       EDIS;
}

interrupt void  adc_isr(void)
{
       // Sample ADCs and converter variables to real world values
       V_ADC[0] = AdcRegs.ADCRESULT0 >>4;
       V_ADC[1] = AdcRegs.ADCRESULT1 >>4;
       V_ADC[2] = AdcRegs.ADCRESULT2 >>4;
       V_ADC[3] = AdcRegs.ADCRESULT3 >>4;
       V_ADC[4] = AdcRegs.ADCRESULT4 >>4;
       V_ADC[5] = AdcRegs.ADCRESULT5 >>4;
       V_ADC[6] = AdcRegs.ADCRESULT6 >>4;
       V_ADC[7] = AdcRegs.ADCRESULT7 >>4;

       //Store ADC Values in Register Bank (DSP)
       Reg[32]=V_ADC[0];
       Reg[33]=V_ADC[1];
       Reg[34]=V_ADC[2];
       Reg[35]=V_ADC[3];
```

```
	Reg[36]=V_ADC[4];
	Reg[37]=V_ADC[5];
	Reg[38]=V_ADC[6];
	Reg[39]=V_ADC[7];

	// Reconstruct ADC values to real world values
	V_Scale1 = (V_ADC_Max/D_ADC_Max);
	V_Scale2 = V_Scale1*17.1551;
	V_Buck = V_ADC[0] * V_Scale2;		// V_ADC*(3/4095) *
(61.9e3/(1e6+61.9e3))^-1
	V_Boost = V_ADC[1] * V_Scale2;		// V_ADC*(3/4095) *
(61.9e3/(1e6+61.9e3))^-1
	V_PhA = V_ADC[2] * V_Scale2;		// V_ADC*(3/4095) *
(61.9e3/(1e6+61.9e3))^-1
	V_PhB = V_ADC[3] * V_Scale2;		// V_ADC*(3/4095) *
(61.9e3/(1e6+61.9e3))^-1
	V_PhC = V_ADC[4] * V_Scale2;		// V_ADC*(3/4095) *
(61.9e3/(1e6+61.9e3))^-1
	V_DCin = V_ADC[5] * V_Scale2;		// V_ADC*(3/4095) *
(61.9e3/(1e6+61.9e3))^-1
	V_Pot = V_ADC[6] * V_Scale1;		// V_ADC*(3/4095)



	// Create an AC reference

	if(Sin_Counter >= Prd_Sw || (GpioDataRegs.GPADAT.bit.GPIO30 == 0))	//here
we count to a number based on how many switching cycles occur for every fundamental
cycle
	{
		Sin_Counter = 0;
	}
	else
	{
		Sin_Counter++;
	}

	Duty_AC = 1;//0.9;

	Step_sin = (Sin_Counter/Prd_Sw)*pi_2;
	Duty_AC = Duty_AC*PWM_TBPRD;
	Sine1 = ((sin(Step_sin+0)+1)/2)*Duty_AC;
	Sine2 = ((sin(Step_sin-phase_offset)+1)/2)*Duty_AC;
	Sine3 = ((sin(Step_sin+phase_offset)+1)/2)*Duty_AC;
	//Sine3 = ((sin(Step_sin+phase2_offset)+1)/2)*Duty_AC;

	//Update Duty Cycle for Buck Converter
	EPwm1Regs.CMPA.half.CMPA= 0;
	//Update Duty Cycle for Boost Converter
	EPwm1Regs.CMPB = 0;

	//Update Duty Cycle for 3-phase inverter
	EPwm2Regs.CMPA.half.CMPA = (Uint16) Sine1;
	EPwm3Regs.CMPA.half.CMPA = (Uint16) Sine2;
	EPwm4Regs.CMPA.half.CMPA = (Uint16) Sine3;
```

```
	// Reinitialize for next ADC sequence
	AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;         // Reset SEQ1
	AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;       // Clear INT SEQ1 bit
	PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;   // Acknowledge interrupt to PIE

	return;
}

void InitEPWM1()
{
	EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;
	EPwm1Regs.TBPRD = PWM_TBPRD;                          // Period = 2* 1000 TBCLK
counts => 30 kHz

	EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE;               // Enable phase loading
	EPwm1Regs.TBPHS.half.TBPHS = 0;
	EPwm1Regs.TBCTL.bit.PHSDIR = TB_DOWN;
	EPwm1Regs.TBCTL.bit.PRDLD = TB_SHADOW;
	EPwm1Regs.TBCTL.bit.SYNCOSEL = TB_CTR_ZERO;   //Sync down-stream module
	EPwm1Regs.TBCTR = 0x0000;                             // Clear counter
	EPwm1Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;              // Clock ratio to
SYSCLKOUT
	EPwm1Regs.TBCTL.bit.CLKDIV = TB_DIV1;

	EPwm1Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;           // Load registers every
ZERO
	EPwm1Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;
	EPwm1Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;
	EPwm1Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;

	// Setup compare
	EPwm1Regs.CMPA.half.CMPA = 0;
	EPwm1Regs.CMPB = 0;

	// Set actions
	EPwm1Regs.AQCTLA.bit.CAU = AQ_CLEAR;
	EPwm1Regs.AQCTLA.bit.CAD = AQ_SET;
	EPwm1Regs.AQCTLB.bit.CBU = AQ_CLEAR;
	EPwm1Regs.AQCTLB.bit.CBD = AQ_SET;

	/*
	// Setup Deadband
	EPwm1Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE;
	EPwm1Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC;
	EPwm1Regs.DBCTL.bit.IN_MODE = DBA_ALL;
	EPwm1Regs.DBRED = 45;  //30;
	EPwm1Regs.DBFED = 45;  //30;
	*/

	//ADC Triggering Setup
	EPwm1Regs.ETSEL.bit.SOCAEN = 1;
	EPwm1Regs.ETSEL.bit.SOCASEL = 0x2;//2;//0x2;       // Select SOC from from
CPMA on Period
	EPwm1Regs.ETPS.bit.SOCAPRD = 1;        // Generate pulse every event
}
```

```
void InitEPWM2()
{
       EPwm2Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;
       EPwm2Regs.TBPRD = PWM_TBPRD;                                  // Period = 2*1000
TBCLK counts => 30 kHz
       EPwm2Regs.TBCTL.bit.PHSEN = TB_DISABLE;                       // Enable phase
loading
       EPwm2Regs.TBPHS.half.TBPHS = 0;
       EPwm2Regs.TBCTL.bit.PHSDIR = TB_DOWN;
       EPwm2Regs.TBCTL.bit.PRDLD = TB_SHADOW;
       EPwm2Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN; // sync flow-through
       EPwm2Regs.TBCTR = 0x0000;                    // Clear counter
       EPwm2Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;       // Clock ratio to SYSCLKOUT
       EPwm2Regs.TBCTL.bit.CLKDIV = TB_DIV1;

       EPwm2Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;    // Load registers every ZERO
       EPwm2Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;
       EPwm2Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;
       EPwm2Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;

       // Setup compare
       EPwm2Regs.CMPA.half.CMPA = 0; //fixed 50% duty cycle
       EPwm2Regs.CMPB = 0;

       // Set actions qualifiers
       EPwm2Regs.AQCTLA.bit.CAU = AQ_CLEAR;
       EPwm2Regs.AQCTLA.bit.CAD = AQ_SET;
       EPwm2Regs.AQCTLB.bit.CBU = AQ_CLEAR;
       EPwm2Regs.AQCTLB.bit.CBD = AQ_SET;

       //EPwm2Regs.ETSEL.bit.SOCAEN = 0;        // Disable SOC on A group


       // Setup Deadband
    EPwm2Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE;
       EPwm2Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC;
       EPwm2Regs.DBCTL.bit.IN_MODE = DBA_ALL;
       EPwm2Regs.DBRED = DeadTime;
       EPwm2Regs.DBFED = DeadTime;

}

void InitEPWM3()
{
       EPwm3Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;
       EPwm3Regs.TBPRD = PWM_TBPRD;                // Period = 2*1000 TBCLK counts =>
30 kHz
       EPwm3Regs.TBCTL.bit.PHSEN = TB_DISABLE;
       EPwm3Regs.TBPHS.half.TBPHS = 0;
       EPwm3Regs.TBCTL.bit.PHSDIR = TB_DOWN;
       EPwm3Regs.TBCTL.bit.PRDLD = TB_SHADOW;
       EPwm3Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN;    // sync flow-through
       EPwm3Regs.TBCTR = 0x0000;                     // Clear counter
       EPwm3Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;      // Clock ratio to SYSCLKOUT
       EPwm3Regs.TBCTL.bit.CLKDIV = TB_DIV1;
```

```
        // Setup shadow register load on ZERO
        EPwm3Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;
        EPwm3Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;
        EPwm3Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;
        EPwm3Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;

        // Set Compare values
        EPwm3Regs.CMPA.half.CMPA = 0;    // Set compare A value
        EPwm3Regs.CMPB = 0;

        // Set action qualifiers
        EPwm3Regs.AQCTLA.bit.CAU = AQ_CLEAR;
        EPwm3Regs.AQCTLA.bit.CAD = AQ_SET;
        EPwm3Regs.AQCTLB.bit.CBU = AQ_CLEAR;
        EPwm3Regs.AQCTLB.bit.CBD = AQ_SET;

        //EPwm3Regs.AQCTLA.bit.CAU = AQ_SET;
        //EPwm3Regs.AQCTLA.bit.CAD = AQ_CLEAR;
        //EPwm3Regs.AQCTLB.bit.CBU = AQ_SET;
        //EPwm3Regs.AQCTLB.bit.CBD = AQ_CLEAR;

        // Setup Deadband
        EPwm3Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE;
        EPwm3Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC;
        EPwm3Regs.DBCTL.bit.IN_MODE = DBA_ALL;
        EPwm3Regs.DBRED = DeadTime;
        EPwm3Regs.DBFED = DeadTime;

}


void InitEPWM4()
{
        EPwm4Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;
        EPwm4Regs.TBPRD = PWM_TBPRD;                // Period = 2*1000 TBCLK counts =>
30 kHz
        EPwm4Regs.TBCTL.bit.PHSEN = TB_DISABLE;
        EPwm4Regs.TBPHS.half.TBPHS = 0;
        EPwm4Regs.TBCTL.bit.PHSDIR = TB_DOWN;
        EPwm4Regs.TBCTL.bit.PRDLD = TB_SHADOW;
        EPwm4Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN;    // sync flow-through
        EPwm4Regs.TBCTR = 0x0000;                     // Clear counter
        EPwm4Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;      // Clock ratio to SYSCLKOUT
        EPwm4Regs.TBCTL.bit.CLKDIV = TB_DIV1;

        // Setup shadow register load on ZERO
        EPwm4Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;
        EPwm4Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;
        EPwm4Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;
        EPwm4Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;

        // Set Compare values
        EPwm4Regs.CMPA.half.CMPA = 0;    // Set compare A value
        EPwm4Regs.CMPB = 0;
```

```
        // Set action qualifiers
        EPwm4Regs.AQCTLA.bit.CAU = AQ_CLEAR;
        EPwm4Regs.AQCTLA.bit.CAD = AQ_SET;
        EPwm4Regs.AQCTLB.bit.CBU = AQ_CLEAR;
        EPwm4Regs.AQCTLB.bit.CBD = AQ_SET;

        //EPwm4Regs.AQCTLA.bit.CAU = AQ_SET;
        //EPwm4Regs.AQCTLA.bit.CAD = AQ_CLEAR;
        //EPwm4Regs.AQCTLB.bit.CBU = AQ_SET;
        //EPwm4Regs.AQCTLB.bit.CBD = AQ_CLEAR;

        // Setup Deadband
        EPwm4Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE;
        EPwm4Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC;
        EPwm4Regs.DBCTL.bit.IN_MODE = DBA_ALL;
        EPwm4Regs.DBRED = DeadTime;
        EPwm4Regs.DBFED = DeadTime;

}
```

A-2 CPLD VHDL Code

A-2-1 Main Top Function

```
----------------------------------------------------------------------------------

----------------------------------------------------------------------------------

----------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Estefano Soria
--
-- Create Date:                26Dec2020
-- Design Name:                Digital_Twin_Top_V0
-- Module Name:                Digital_Twin_Top_V0 - Behavioral
-- Project Name:               Digital_Twin_Top_V0
-- Target Devices:             LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:              Lattice Diamond_x64 Build 3.11
-- Description:
--
---- PinOut:
--
-- Revision
--
--
--
-- Additional Comments:
--
--
----------------------------------------------------------------------------------


Library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library machxo2;
use machxo2.all;

library work;
use work.Test2_DT_Common.all;

entity Test2_DT_Top is
    Port (
                                --clk : in std_logic;
```

```
                --rst : in std_logic;

                --Backup_FW_IN : in std_logic;
        ---- THIS INPUT SIGNAL IS ONLY FOR THIS TEST, AND IT SHOULD BE 1 THE
WHOLE TIME TO KEEP THE BACKUP FILE DISSABLED----

                SCI_RX : in std_logic;          -- This is the input Serial--
        --FW_BIT_IN : in std_logic;            ---- ONLY FOR THIS TEST ----
                SCI_TX : out std_logic;         -- This is the output Serial--
        --FW_BIT_OUT : out std_logic;                --- ONLY FOR THIS TEST ----

                --Command_EN_Byte_OUT : out std_logic_vector (7 downto 0);
---- ONLY USED FOR THIS TEST. THIS OUTPUTS THE DATA BYTE THAT WOULD GO
THROUGH SERIAL BACK TO THE USR TO ALLOW THEM TO SEND COMMANDS TO
THE SYSTEM ----

                --Command_EN_Bit_OUT : out std_logic;
            ---- ONLY FOR THIS TEST ----

                IDC_B_GPIO_00 : out STD_LOGIC;
        ---- USUALLY THIS OUTPUT SIGNAL IS THE GPIO THAT CONNECTS TO THE
SOLDERED WIRE ON THE DSP THAT ALLOWS IT TO RESET ----
                IDC_C_GPIO_00 : out STD_LOGIC;
        ---- USUALLY THIS OUTPUT SIGNAL IS THE GPIO THAT CONNECTS TO THE
SOLDERED WIRE ON THE DSP THAT ALLOWS IT TO RESET ----

                DIMM_B_SCI_RX : out STD_LOGIC;
        ---- THIS IS THE PIN THAT CONNECTS TO THE SERIAL PIN OF THE DSP,
WHICH IS USED TO SEND THE FIRMWARE FILE DATA FOR BOOTLOAD ----
                DIMM_C_SCI_RX : out STD_LOGIC;

                DIMM_B_SCI_TX : in STD_LOGIC;
                DIMM_C_SCI_TX : in STD_LOGIC;

                DIMM_B_GPIO30 : out STD_LOGIC;
                DIMM_C_GPIO30 : out STD_LOGIC;
                --RAM_Din : out std_logic_vector (15 downto 0);
                --RAM_WE : out std_logic;
                --UART_New_FW_EN : out std_logic;
                --BM_Back_FW_EN :          out std_logic;
                --Bootload_EN :         out std_logic;
                --FW_BYTE_OUT : out std_logic_vector (7 downto 0)
        ---- THIS OUTPUT SIGNAL IS ONLY FOR THIS TEST, AND IT IS THE BYTE
ECHO FROM THE SERIAL INPUT FW ----
```

```
                DSP1_01 : in  STD_LOGIC;          -- DIMM_B Connector GPIO
                DSP1_02 : in  STD_LOGIC;          -- DIMM_B Connector GPIO
                DSP1_03 : in  STD_LOGIC;          -- DIMM_B Connector GPIO
                DSP1_04 : in  STD_LOGIC;          -- DIMM_B Connector GPIO
                DSP1_05 : in  STD_LOGIC;          -- DIMM_B Connector GPIO
                DSP1_06 : in  STD_LOGIC;          -- DIMM_B Connector GPIO

                DSP2_01 : in  STD_LOGIC;          -- DIMM_B Connector GPIO
                DSP2_02 : in  STD_LOGIC;          -- DIMM_B Connector GPIO
                DSP2_03 : in  STD_LOGIC;          -- DIMM_B Connector GPIO
                DSP2_04 : in  STD_LOGIC;          -- DIMM_B Connector GPIO
                DSP2_05 : in  STD_LOGIC;          -- DIMM_B Connector GPIO
                DSP2_06 : in  STD_LOGIC;          -- DIMM_B Connector GPIO

                LED_A : out STD_LOGIC;
                LED_B : out STD_LOGIC;

                IDC_B_GPIO_02 : out STD_LOGIC;

                LED_C : out STD_LOGIC;
                LED_D : out STD_LOGIC;
                LED_E : out STD_LOGIC;
                LED_F : out STD_LOGIC;
                LED_G : out STD_LOGIC;
                LED_H : out STD_LOGIC;

                Btn1 : in STD_LOGIC;
                Btn2 : in STD_LOGIC;
                SW : in STD_LOGIC;

                SW01 : out std_logic;
                SW02 : out std_logic;
                SW03 : out std_logic;
                SW04 : out std_logic;
                SW05 : out std_logic;
                SW06 : out std_logic

                );
end Test2_DT_Top;

architecture Behavioral of Test2_DT_Top is

        -- Oscillator
        signal  OSC_Stdby : std_logic := '0';
        signal  OSC_Out : std_logic := '0';
        signal  OSC_SEDSTDBY : std_logic := '0';
```

```
-- PLL
--signal  OSC_Out : std_logic := '0';
signal  clk : std_logic := '0';
signal  Pll_Lock : std_logic := '0';

-- Bus Master
signal Xrqst                : std_logic := '0';
signal XDat                 : std_logic := '0';
signal YDat                 : std_logic := '0';
signal DSP_RAM_clk          : std_logic := '0';
signal DSP_RAM_clk_en : std_logic := '0';
signal DSP_RAM_rst          : std_logic := '0';
signal DSP_RAM_we           : std_logic := '0';
signal Data  : std_logic_vector (15 downto 0) := (others => '0');
signal Addr  : std_logic_vector (15 downto 0) := (others => '0');
signal BusRqst  : std_logic_vector (9 downto 0) := (others => '0');
signal BusCtrl  : std_logic_vector (9 downto 0) := (others => '0');
signal DSP_RAM_addr  : std_logic_vector (15 downto 0) := (others => '0');
signal DSP_RAM_din   : std_logic_vector (15 downto 0) := (others => '0');
signal DSP_RAM_dout  : std_logic_vector (15 downto 0) := (others => '0');


-- Bootloader

signal  Bootload_EN : std_logic := '0';
signal  FW_Type             : std_logic := '0';
signal  DSP_rcv         : std_logic := '0';
signal  DSP_xmt         : std_logic := '0';
signal  DSP_Rst         : std_logic := '0';

-- Other
signal  rs232_rcv : std_logic := '0';
signal  rs232_xmt : std_logic := '0';
signal  Error : std_logic := '0';
signal  Boot_Wrkn : std_logic := '0';
signal  Boot_Done : std_logic := '0';
signal  HP_EN : std_logic := '0';
signal  HP_Done : std_logic := '0';
signal  Emu_EN : std_logic := '0';
signal  Reset_Cnt_rst : std_logic := '0';
signal  Reset_Cnt_INC : std_logic := '0';
signal  System_rst : std_logic := '0';
signal  Nothing : std_logic := '0';
signal  Emu_SW01 : std_logic := '0';
signal  Emu_SW02 : std_logic := '0';
signal  Emu_SW03 : std_logic := '0';
```

```
        signal  Emu_SW04 : std_logic := '0';
        signal  Emu_SW05 : std_logic := '0';
        signal  Emu_SW06 : std_logic := '0';
        signal  DSP1_Act : std_logic := '0';
        signal  DSP1_Act_HP_Out : std_logic := '0';
        signal  DSP_Sync_HP : std_logic := '0';
        signal  Reset_Cnt_out : std_logic_vector (7 downto 0) := (others => '0');




        -- Declare Internal Oscillator
        COMPONENT OSCH
        GENERIC (NOM_FREQ: string := "8.31");
        PORT ( STDBY :IN std_logic;
                OSC :OUT std_logic;
                SEDSTDBY :OUT std_logic
                );
        END COMPONENT;

        -- Declare PLL
COMPONENT PLL_Clk_Test2
        PORT(
                ClkI: in  std_logic;
                ClkOP: out std_logic;
                Lock: out std_logic
                );
END COMPONENT;




        -- Declare Bus_Master
COMPONENT Bus_Master
PORT(
    clk : IN  std_logic;
    rst : IN  std_logic;
    Data : INOUT  std_logic_vector(15 downto 0);
    Addr : IN  std_logic_vector(15 downto 0);
    Xrqst : IN  std_logic;
    XDat : OUT  std_logic;
    YDat : IN  std_logic;
    BusRqst : IN  std_logic_vector(9 downto 0);
    BusCtrl : OUT  std_logic_vector(9 downto 0);
                DSP_RAM_clk : in  STD_LOGIC;
```

```
            DSP_RAM_clk_en : in  STD_LOGIC;
            DSP_RAM_rst : in  STD_LOGIC;
            DSP_RAM_we : in  STD_LOGIC;
            DSP_RAM_addr : in  STD_LOGIC_VECTOR (15 downto 0);
            DSP_RAM_din : in  STD_LOGIC_VECTOR (15 downto 0);
            DSP_RAM_dout : out  STD_LOGIC_VECTOR (15 downto 0)
            );
END COMPONENT;


    -- Declare RS232_Usr_Int
     COMPONENT RS232_Usr_Int
            Generic (
            Baud        : integer;          -- Baud Rate
            clk_in      : integer           -- Input Clk
            );
PORT(
   clk : IN  std_logic;
   rst : IN  std_logic;
   rs232_rcv : IN  std_logic;
   rs232_xmt : OUT  std_logic;
   Data : INOUT  std_logic_vector(15 downto 0);
   Addr : OUT  std_logic_vector(15 downto 0);
   Xrqst : OUT  std_logic;
   XDat : IN  std_logic;
   YDat : OUT  std_logic;
   BusRqst : OUT  std_logic;
   BusCtrl : IN  std_logic
  );
END COMPONENT;


    ---- Test2_DT_Emu_Ctrl ----
    component Test2_DT_Emu_Ctrl
    Port (
                        clk         : in STD_LOGIC;
                        rst         : in STD_LOGIC;

                        Emu_EN          : in std_logic;

                        Data        : INOUT  std_logic_vector(15 downto 0);
                        Addr        : OUT  std_logic_vector(15 downto 0);
                        Xrqst       : OUT  std_logic;
                        XDat        : IN  std_logic;
                        YDat        : OUT  std_logic;
                        BusRqst     : OUT  std_logic;
```

```
                BusCtrl         : IN  std_logic;

                Emu_SW01    : in  STD_LOGIC;
                Emu_SW02    : in  STD_LOGIC;
                Emu_SW03    : in  STD_LOGIC;
                Emu_SW04    : in  STD_LOGIC;
                Emu_SW05    : in  STD_LOGIC;
                Emu_SW06    : in  STD_LOGIC;

                Error           : in STD_LOGIC;
                HP_EN                   : in STD_LOGIC

                --** Missing Ports for Bus Interface, and some others **


                );
end component;


------ Test1_DT_Boot_Ctrl -----
component Test2_DT_Boot_Ctrl
Port (
                --IN
                clk             : in  STD_LOGIC;
                rst             : in  STD_LOGIC;

                Data            : INOUT  std_logic_vector(15 downto 0);
                Addr            : OUT  std_logic_vector(15 downto 0);
                Xrqst           : OUT  std_logic;
                XDat            : IN  std_logic;
                YDat            : OUT  std_logic;
                BusRqst         : OUT  std_logic;
                BusCtrl         : IN  std_logic;

                Error           :in STD_LOGIC;
                Boot_Wrkn   :in STD_LOGIC;
                Boot_Done   :in STD_LOGIC;
                HP_EN                   :in STD_LOGIC;

                --OUT
                Bootload_EN  :out STD_LOGIC;
                FW_Type                 :out STD_LOGIC


        );
end component;
```

```
------ DT_Bootloader_Test Component-----
        component DT_Bootloader
        generic(
                        Baud : integer;                 --9,600 bps
                        clk_in : integer);              --25MHz
   Port (
                    clk                 : IN  std_logic;
                    rst                 : IN  std_logic;
                    Bootload_EN  : IN STD_LOGIC;
                    FW_Type                 : IN STD_LOGIC;
                    Bootload_Wrkn       : OUT STD_LOGIC;
                    Bootload_Done       : OUT STD_LOGIC;
                    DSP_rcv             : OUT std_logic;
                    DSP_xmt             : IN  std_logic;
                    DSP_RAM_clk         : OUT  STD_LOGIC;
                    DSP_RAM_clk_en      : OUT  STD_LOGIC;
                    DSP_RAM_rst         : OUT  STD_LOGIC;
                    DSP_RAM_we              : OUT  STD_LOGIC;
                    DSP_RAM_addr        : OUT  STD_LOGIC_VECTOR (15 downto 0);
                    DSP_RAM_din         : OUT  STD_LOGIC_VECTOR (15 downto 0);
                    DSP_RAM_dout        : IN  STD_LOGIC_VECTOR (15 downto 0);
                    DSP_Rst             : OUT  STD_LOGIC


                     );


        end component;


------ Test1_DT_Firmware_Validation -----
        component Test2_DT_Firmware_Validation
                Port (
                        clk             : in STD_LOGIC;
                        rst             : in STD_LOGIC;

                        Data            : INOUT  std_logic_vector(15 downto 0);
                        Addr            : OUT  std_logic_vector(15 downto 0);
                        Xrqst           : OUT  std_logic;
                        XDat            : IN  std_logic;
                        YDat            : OUT  std_logic;
                        BusRqst         : OUT  std_logic;
                        BusCtrl         : IN  std_logic;

                        Emu_SW01    : in std_logic;
                        Emu_SW02    : in std_logic;
```

```
                                Emu_SW03    : in std_logic;
                                Emu_SW04    : in std_logic;
                                Emu_SW05    : in std_logic;
                                Emu_SW06    : in std_logic;


                                HP_Done             : in std_logic;         ---- Signal coming
from DSP_Hot-Patch module saying that HP process is complete
                                Boot_Done   : in std_logic;
                                Boot_Wrkn   : in std_logic;
                                Emu_EN              : out std_logic;
                                HP_EN               : out std_logic;        ---- Signal sent to
DSP_Hot-Patch module to enable HP process
                                Error           : out std_logic

                                );

        end component;




        -- Declare Std_Counter Component
        component Std_Counter is
        generic
        (
                Width : integer                 -- width of counter
        );
        port(INC,rst,clk: in std_logic;
                 Count: out STD_LOGIC_VECTOR(Width-1 downto 0));
        end component;




------ DSP_Hot_Patch Component-----
        component Test2_DT_HP_Ctrl
   Port (

                                clk : in std_logic;
                                rst : in std_logic;
                                EN : in std_logic;
                                DSP1_Act_Out : out std_logic;
                                DSP_Sync : out std_logic;
```

```
                        Done : out std_logic


                                );
    end component;

-------- DeadTime Component --------
    component Test2_DT_DeadTime
        Port (
                        clk                 : in STD_LOGIC;
                        rst                 : in STD_LOGIC;
                        EN                          : in STD_LOGIC;
                        Emu_SW01            : in std_logic;
                        Emu_SW02            : in std_logic;
                        Emu_SW03            : in std_logic;
                        Emu_SW04            : in std_logic;
                        Emu_SW05            : in std_logic;
                        Emu_SW06            : in std_logic;
                        BtnRst              : in std_logic;

                        DeadTimeError       : out std_logic

                        );
    end component;


begin




    -- Instantiate Internal Oscillator
    Int_OSC: OSCH PORT MAP (
                        STDBY => OSC_Stdby,
                        OSC => OSC_Out,
                        SEDSTDBY => OSC_SEDSTDBY
                    );


    -- Instantiate PLL
    PLL_1: PLL_Clk_Test2 PORT MAP (
                        ClkI => OSC_Out,
                        ClkOP => clk,
```

```
                Lock =>Pll_Lock
        );
```

```
-- Instantiate DeadTime
DeadTChk: Test2_DT_DeadTime PORT MAP (
        clk             => clk,
        rst             => System_rst,
        EN                      => Btn1,
        Emu_SW01        => Emu_SW01,
        Emu_SW02        => Emu_SW02,
        Emu_SW03        => Emu_SW03,
        Emu_SW04        => Emu_SW04,
        Emu_SW05        => Emu_SW05,
        Emu_SW06        => Emu_SW06,
        BtnRst          => Btn2,
        DeadTimeError   => LED_D
        );
```

```
-- Instantiate Bus_Master
BM: Bus_Master PORT MAP (
        clk             => clk,
        rst             => System_rst,
        Data            => Data,
        Addr            => Addr,
        Xrqst           => Xrqst,
        XDat            => XDat,
        YDat            => YDat,
        BusRqst         => BusRqst,
```

```
                BusCtrl             => BusCtrl,
                --Added for DSP Firmware Loader
                DSP_RAM_clk         => DSP_RAM_clk,
                DSP_RAM_clk_en      => DSP_RAM_clk_en,
                DSP_RAM_rst         => DSP_RAM_rst,
                DSP_RAM_we                  => DSP_RAM_we,
                DSP_RAM_addr        => DSP_RAM_addr,
                DSP_RAM_din         => DSP_RAM_din,
                DSP_RAM_dout        => DSP_RAM_dout
);


   -- Instantiate RS232_Usr_Int
   RS232_Usr: RS232_Usr_Int
   Generic Map
    (
          Baud   => 9600,        -- Baud Rate
          Clk_In => Clk_Freq   --        Input Clk
    )
   PORT MAP (
 clk => clk,
 rst => System_rst,
 rs232_rcv => SCI_RX,
 rs232_xmt => SCI_TX,
           --rs232_xmt => Temp_Debug,
 Data => Data,
 Addr => Addr,
 Xrqst => Xrqst,
 XDat => XDat,
 YDat => YDat,
 BusRqst => BusRqst(3),
 BusCtrl => BusCtrl(3)
);



   -- Instantiate Emu_Ctrl
   Emu_Ctrl: Test2_DT_Emu_Ctrl
   port map (
                        clk       => clk,
                        rst       => System_rst,

                        Emu_EN        => Emu_EN,

                        Data      => Data,
                        Addr      => Addr,
```

```
                Xrqst    => Xrqst,
                XDat     => XDat,
                YDat     => YDat,
                BusRqst  => BusRqst(2),
                BusCtrl  => BusCtrl(2),

                Emu_SW01 => Emu_SW01,
                Emu_SW02 => Emu_SW02,
                Emu_SW03 => Emu_SW03,
                Emu_SW04 => Emu_SW04,
                Emu_SW05 => Emu_SW05,
                Emu_SW06 => Emu_SW06,

                Error    => Error,
                HP_EN          => HP_EN
        );



-- Instantiate Boot_Ctrl
Boot_Ctrl: Test2_DT_Boot_Ctrl
port map (

                clk             => clk,
                rst             => System_rst,
                Data            => Data,
                Addr            => Addr,
                Xrqst           => Xrqst,
                XDat            => XDat,
                YDat            => YDat,
                BusRqst         => BusRqst(0),
                BusCtrl         => BusCtrl(0),
                Error           => Error,
                Boot_Wrkn       => Boot_Wrkn,
                Boot_Done       => Boot_Done,
                HP_EN                   => HP_EN,
                Bootload_EN     => Bootload_EN,
                FW_Type                 => FW_Type
        );
```

```
-- Instantiate Bootloader
Bootload: DT_Bootloader
generic map
(
                Baud            => 9600,        --9,600 bps
                clk_in          => Clk_Freq --25MHz
)
port map (
                clk                     => clk,
                rst                     => System_rst,
                Bootload_EN     => Bootload_EN,
                FW_Type                 => FW_Type,
                Bootload_Wrkn           => Boot_Wrkn,
                Bootload_Done           => Boot_Done,
                DSP_rcv                 =>      DSP_rcv,
        --FW_BIT_OUT,       ---- THIS FW_BIT_OUT SIGNAL IS ONLY
USED FOR THIS TEST, USUALLY THIS CONNECTS TO THE SERIAL PORT OF THE
DSP THROUGH DIMM B OR DIMM C DEPENDING ON THE DSP, AND IT IS NOW
CONNECTED THROUGH THE HP PROCESS BELOW ----
                DSP_xmt                 =>      DSP_xmt,
        --xmt,          ---- THIS xmt SIGNAL IS ONLY FOR THIS TEST, AND
NEEDS TO BE INITIALIZED TO 1 ----
                DSP_RAM_clk             => DSP_RAM_clk,
                DSP_RAM_clk_en          => DSP_RAM_clk_en,
                DSP_RAM_rst             => DSP_RAM_rst,
                DSP_RAM_we                      => DSP_RAM_we,
                DSP_RAM_addr            => DSP_RAM_addr,
                DSP_RAM_din             => DSP_RAM_din,
                DSP_RAM_dout            => DSP_RAM_dout,
                DSP_Rst                 =>      DSP_Rst
        --DSP_Rst,          ---- ONLY FOR THIS TEST, USUALLY
CONNECTS TO THE EXTERNAL GPIO PIN THAT IS SOLDERED TO THE DSP TO BE
ABLE TO RESET IT, AND IT IS NOW CONNECTED THROUGH THE HP PROCESS
BELOW ----

        );



-- Instantiate Firmware Validation_EN
FW_Valid: Test2_DT_Firmware_Validation
port map(

                clk             => clk,
                rst             => System_rst,
                Data            => Data,
```

```
                        Addr            => Addr,
                        Xrqst           => Xrqst,
                        XDat            => XDat,
                        YDat            => YDat,
                        BusRqst         => BusRqst(1),
                        BusCtrl         => BusCtrl(1),
                        Emu_SW01        => Emu_SW01,
                        Emu_SW02        => Emu_SW02,
                        Emu_SW03        => Emu_SW03,
                        Emu_SW04        => Emu_SW04,
                        Emu_SW05        => Emu_SW05,
                        Emu_SW06        => Emu_SW06,
                        HP_Done                 => HP_Done,
                        Boot_Done       => Boot_Done,
    Boot_Wrkn => Boot_Wrkn,
    Emu_EN              => Emu_EN,
    HP_EN               => HP_EN,
    Error               => Error
                );
```

```
-- Instantiate Reset_Cnt_8
Reset_Cnt: Std_Counter
generic map
(
        Width => 8
)
port map(
        clk => OSC_Out,
        rst=> Reset_Cnt_rst,
        INC=> Reset_Cnt_INC,
        Count=> Reset_Cnt_out
);
```

```
-- Instantiate HP
HP_Set: Test2_DT_HP_Ctrl
PORT MAP (
                clk => clk,
                rst => System_rst,
                EN => HP_EN,
                DSP1_Act_Out => DSP1_Act_HP_Out,
                DSP_Sync => DSP_Sync_HP,
                Done => HP_Done
        );



--  Oscillator
OSC_Stdby <= '0';


-- Tie unused ports to '0'

BusRqst(9 downto 4) <= (others => '0');
--DSP_RAM_rst <= '0';

-- Reset Block1
Reset_Blk1: process
begin
        wait until OSC_Out'event and OSC_Out = '1';
                if(PLL_Lock ='0') then
                        Reset_Cnt_rst <= '0';
                else
                        Reset_Cnt_rst <= '1';
                end if;
end process;

-- Reset Block
Reset_Blk: process
begin
        wait until OSC_Out'event and OSC_Out = '1';
                if(Reset_Cnt_out < X"7F") then
                        System_rst <= '0';
                        Reset_Cnt_INC <='1';
                else
                        System_rst <= '1';
                        Reset_Cnt_INC <='0';
                end if;
end process;
```

```
--Setting DSP1 assignment
DSP1_Act_Set: process
begin
	wait until clk'event and clk = '1';
		if (System_rst = '0') then
			DSP1_Act <= '1';
		else
			DSP1_Act <= DSP1_Act_HP_Out;
		end if;
end process;




-- Main Routing Process (Combinatorial)

process(SCI_RX, DSP_Rst, DSP_rcv, DSP1_Act, DSP1_01, DSP1_02, DSP1_03,
DSP1_04, DSP1_05, DSP1_06, DSP2_01, DSP2_02, DSP2_03, DSP2_04, DSP2_05, DSP2_06)
		begin
			if (DSP1_Act = '1') then

				SW01 <= DSP1_01;
				SW02 <= DSP1_02;
				SW03 <= DSP1_03;
				SW04 <= DSP1_04;
				SW05 <= DSP1_05;
				SW06 <= DSP1_06;


				Emu_SW01 <= DSP2_01;
				Emu_SW02 <= DSP2_02;
				Emu_SW03 <= DSP2_03;
				Emu_SW04 <= DSP2_04;
				Emu_SW05 <= DSP2_05;
				Emu_SW06 <= DSP2_06;

				DIMM_C_GPIO30 <= DSP_Sync_HP;
				DIMM_B_GPIO30 <= DSP_Sync_HP;
```

```
                --DSP1_Act_HP_In <= '1';

                LED_A <= '0';
                LED_B <= '1';

                IDC_B_GPIO_02 <= '0';
                LED_C <= '1';
                --LED_C <= not(DSP1_01);
                --LED_D <= not(DSP1_02);
                --LED_C <= '1';
                --LED_D <= '1';
                LED_E <= '1';
                LED_F <= '1';
                LED_G <= '1';
                LED_H <= '1';

---------- IDC pins are different than the Bootloader code that Chris did----------
------------------------------

                IDC_B_GPIO_00 <= '1';                   ---- Reset is active
low, and 1(NO Reset) is routed to pin 00 of IDC B (DSP1 is Active)
                IDC_C_GPIO_00 <= DSP_Rst;               ---- DSP_Rst
signal(Bootloader) routed to pin 00 of IDC C (DSP2 is Stand-By)

                DIMM_B_SCI_RX <= '1';                   ---- Stop bit is high,
and is sent to the serial receiver of DIMM B (DSP1 is Active)
                DIMM_C_SCI_RX <= DSP_rcv;       ---- DSP_rsv
signal(Bootloader) is routed to the serial receiver of DIMM C (DSP2 is Stand-By)

                Nothing <= DIMM_B_SCI_TX;
                DSP_xmt <= DIMM_C_SCI_TX;
        else

                SW01 <= DSP2_01;
                SW02 <= DSP2_02;
                SW03 <= DSP2_03;
                SW04 <= DSP2_04;
                SW05 <= DSP2_05;
                SW06 <= DSP2_06;


                Emu_SW01 <= DSP1_01;
                Emu_SW02 <= DSP1_02;
                Emu_SW03 <= DSP1_03;
                Emu_SW04 <= DSP1_04;
                Emu_SW05 <= DSP1_05;
```

```
                    Emu_SW06 <= DSP1_06;

                    DIMM_C_GPIO30 <= DSP_Sync_HP;
                    DIMM_B_GPIO30 <= DSP_Sync_HP;

                    --DSP1_Act_HP_In <= '0';

                    LED_A <= '1';
                    LED_B <= '0';
                    IDC_B_GPIO_02 <= '1';
                    LED_C <= '1';
                    --LED_C <= not(DSP2_01);
                    --LED_D <= not(DSP2_02);
                    --LED_C <= '0';
                    --LED_D <= '1';
                    LED_E <= '1';
                    LED_F <= '1';
                    LED_G <= '1';
                    LED_H <= '1';

                    IDC_B_GPIO_00 <= DSP_Rst;          ---- DSP_Rst
signal(Bootloader) routed to pin 00 of IDC B (DSP1 is Stand-By)
                    IDC_C_GPIO_00 <= '1';    ---- Reset is active low, and 1(NO
Reset) is routed to pin 00 of IDC C (DSP2 is Active)

                    DIMM_B_SCI_RX <= DSP_rcv;          ---- DSP_rsv
signal(Bootloader) is routed to the serial receiver of DIMM B (DSP1 is Stand-By)
                    DIMM_C_SCI_RX <= '1';          ---- Stop bit is high, and is
sent to the serial receiver of DIMM C (DSP2 is Active)

                    DSP_xmt <= DIMM_B_SCI_TX;
                    Nothing <= DIMM_C_SCI_TX;


                end if;



    end process;

end Behavioral;
```

A-2-2 Boot_Ctrl

```
----------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Estefano Soria
--
-- Create Date:                4Jan2021
-- Design Name:                Test2_DT_Boot_Ctrl
-- Module Name:                Test2_DT_Boot_Ctrl - Behavioral
-- Project Name:               Test2_DT_Boot_Ctrl
-- Target Devices:             LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:              Lattice Diamond_x64 Build 3.11
-- Description:
--
---- PinOut:
--
-- Revision
--
--
--
-- Additional Comments:
--
--
----------------------------------------------------------------------------------



Library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library machxo2;
use machxo2.all;

library work;
use work.Test2_DT_Common.all;

entity Test2_DT_Boot_Ctrl is
   Port (
                           --IN
                           clk : in  STD_LOGIC;
                           rst : in  STD_LOGIC;

                           Data : INOUT  std_logic_vector(15 downto 0);
                           Addr : OUT  std_logic_vector(15 downto 0);
```

```
                Xrqst : OUT  std_logic;
                XDat : IN  std_logic;
                YDat : OUT  std_logic;
                BusRqst : OUT  std_logic;
                BusCtrl : IN  std_logic;

                Error           :in STD_LOGIC;
                Boot_Wrkn       :in STD_LOGIC;
                Boot_Done       :in STD_LOGIC;
                HP_EN                   :in STD_LOGIC;

                --OUT
                Bootload_EN  :out STD_LOGIC;
                FW_Type                 :out STD_LOGIC


        );
end Test2_DT_Boot_Ctrl;


architecture Behavioral of Test2_DT_Boot_Ctrl is

------NSL signals (5 NSL) ----------------------------------------------------------------------
        type state_type is
(S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,S20,S21,S22,S23,
S24,S25,S26,S27,S28,S29,S30,S31,S32,S33,S34,S35,S36,S37,S38,S39,S40,S41,S42,S43,S44,S4
5,S46,S47,S48,S49,S50);
        signal CS, NS : state_type;




        --FW_Type = 0 (New FW), FW_Type = 1 (Backup FW)
        --Bootload_EN = 0 (Enabled), Bootload_EN = 1 (Dissabled)
        --Bootload is in a separate file because it needs direct access to DP_RAM Port 2.
        -- FW_Type, and Bootload_EN are Ports
        -- Boot_Start is a signal that holds the data from the Bootloader Start RAM Reg
        -- Boot_Start = 0 (Dissabled), Boot_Start = 1 (Enabled)

   ---- Registers ----

        signal   LD_Boot_Wrkn           : std_logic := '0';
        signal      Boot_Wrkn_reg_o  : std_logic := '0';

        signal   LD_Boot_Done           : std_logic := '0';
        signal      Boot_Done_reg_o  : std_logic := '0';
```

```
signal   LD_HP_EN            : std_logic := '0';
signal     HP_EN_reg_o       : std_logic := '0';

signal   LD_Boot_Start               : std_logic := '0';
signal Temp_Boot_Start               : std_logic := '0';
signal           Boot_Start          : std_logic := '0';

signal   LD_Vrble_Data               : std_logic := '0';
signal Temp_Vrble_Data               : std_logic_vector (15 downto 0) := (others => '0');
signal           Vrble_Data          : std_logic_vector (15 downto 0) := (others => '0');

signal   LD_Bootload_EN     : std_logic := '0';
signal Temp_Bootload_EN     : std_logic := '0';

signal   LD_FW_Type                  : std_logic := '0';
signal Temp_FW_Type                  : std_logic := '0';



---- Counters ----

signal CntBus_INC : std_logic := '0';
signal CntBus_Rst : std_logic := '0';
signal CntBus_Out : std_logic_vector(15 downto 0) := (others => '0');

signal CntDelay_INC : std_logic := '0';
signal CntDelay_Rst : std_logic := '0';
signal CntDelay_Out : std_logic_vector(7 downto 0) := (others => '0');



---- Bus Interface Signals ----

signal  Busy : std_logic := '0';
signal  WE : std_logic := '0';
signal  RE : std_logic := '0';
signal  AddrIn : std_logic_vector (15 downto 0) := (others => '0');
signal  DataIn : std_logic_vector (15 downto 0) := (others => '0');
signal  DataOut : std_logic_vector (15 downto 0) := (others => '0');

--Bus Interface Signals
signal  Bus_Int1_Busy : std_logic := '0';
signal  Bus_Int1_WE : std_logic := '0';
signal  Bus_Int1_RE : std_logic := '0';
signal  Bus_Int1_AddrIn : std_logic_vector (15 downto 0) := (others => '0');
signal  Bus_Int1_DataIn : std_logic_vector (15 downto 0) := (others => '0');
```

```
        signal  Bus_Int1_DataOut : std_logic_vector (15 downto 0) := (others => '0');


        -- Compontents --

        --declare Std_Counter Component
        component Std_Counter is
        generic
        (
                Width : integer                 --width of counter
        );
        port(INC,rst,clk: in std_logic;
                 Count: out STD_LOGIC_VECTOR(Width-1 downto 0));
        end component;


        --declare Bus Interface
        COMPONENT Bus_Int
  PORT(
                clk : IN  std_logic;
                rst : IN  std_logic;
                DataIn : IN  std_logic_vector(15 downto 0);
                DataOut : OUT  std_logic_vector(15 downto 0);
                AddrIn : IN  std_logic_vector(15 downto 0);
                WE : IN  std_logic;
                RE : IN  std_logic;
                Busy : OUT  std_logic;
                Data : INOUT  std_logic_vector(15 downto 0);
                Addr : OUT  std_logic_vector(15 downto 0);
                Xrqst : OUT  std_logic;
                XDat : IN  std_logic;
                YDat : OUT  std_logic;
                BusRqst : OUT  std_logic;
                BusCtrl : IN  std_logic
  );
   END COMPONENT;




begin

        --instantiate Delay_Cnt
        Delay_Cnt: Std_Counter
```

```
generic map
(
        Width => 8
)
port map(
        clk => clk,
        rst=> CntDelay_rst,
        INC=> CntDelay_INC,
        Count=> CntDelay_Out
);


--instantiate Bus_Cnt
Bus_Cnt: Std_Counter
generic map
(
        Width => 16
)
port map
(
        clk => clk,
        rst=> CntBus_rst,
        INC=> CntBus_INC,
        Count=>CntBus_Out
);



 --Instantiate Bus Interface
 Bus_Int1: Bus_Int PORT MAP (
clk => clk,
rst => rst,
DataIn => Bus_Int1_DataIn,
DataOut => Bus_Int1_DataOut,
AddrIn => Bus_Int1_AddrIn,
WE => Bus_Int1_WE,
RE => Bus_Int1_RE,
Busy => Bus_Int1_Busy,
Data => Data,
Addr => Addr,
Xrqst => Xrqst,
XDat => XDat,
YDat => YDat,
BusRqst => BusRqst,
BusCtrl => BusCtrl
```

```
    );


        ----Registers
        Reg_Proc: process
        begin
                wait until clk'event and clk = '1';
                if rst = '0' then

                        Boot_Wrkn_reg_o     <= '0';
                        Boot_Done_reg_o     <= '0';
                        HP_EN_reg_o         <= '0';
                        Boot_Start      <= '0';
                        Vrble_Data          <= (others => '0');
                        Bootload_EN         <= '0';
                        FW_Type             <= '0';

                else

                        if (LD_Boot_Wrkn = '1')       then Boot_Wrkn_reg_o <= Boot_Wrkn;
end if;
                        if (LD_Boot_Done = '1')       then Boot_Done_reg_o <= Boot_Done;
          end if;
                        if (LD_HP_EN = '1')                 then HP_EN_reg_o     <= HP_EN;
          end if;
                        if (LD_Boot_Start = '1')      then Boot_Start         <=
Temp_Boot_Start;  end if;
                        if (LD_Vrble_Data = '1')      then Vrble_Data         <=
Temp_Vrble_Data;  end if;
                        if (LD_Bootload_EN = '1')   then Bootload_EN        <=
Temp_Bootload_EN; end if;
                        if (LD_FW_Type = '1')         then FW_Type                <=
Temp_FW_Type;     end if;

        end if;
        end process;
        ----End Registers




        Bootload_Enable: process(CS, Bus_Int1_Busy, Bus_Int1_DataOut, CntDelay_Out,
CntBus_Out, Error, Vrble_Data, Boot_Wrkn_reg_o, Boot_Done_reg_o, Boot_Start,
HP_EN_reg_o)
```

```
begin

        CntBus_Rst <='1';
        CntDelay_Rst <='1';
        CntBus_INC <='0';
        CntDelay_INC <='0';

        Bus_Int1_AddrIn <= (others => '0');
        Bus_Int1_RE <='0';
        Bus_Int1_DataIn <= (others => '0');
        Bus_Int1_WE <='0';

        Temp_Boot_Start <= '0';
        LD_Boot_Start <= '0';

        Temp_Vrble_Data <= (others => '0');
        LD_Vrble_Data <= '0';

        Temp_Bootload_EN <= '1'; -- Active Low
        LD_Bootload_EN <= '0';

        LD_Boot_Wrkn <= '0';

        LD_Boot_Done <= '0';

        LD_FW_Type <= '0';

        LD_HP_EN <= '0';

        case CS is
                when S0=>
                        -- Reset Register and Counter signals
                        CntBus_Rst <='0';               -- Reset Bus Counter
                        CntDelay_Rst <='0';             -- Reset Delay Counter
                        Temp_Boot_Start <= '0';
                        LD_Boot_Start <= '1';
                        Temp_Vrble_Data <= (others => '0');
                        LD_Vrble_Data <= '1';
                        Temp_Bootload_EN <= '1'; -- Active Low
                        LD_Bootload_EN <= '1';
                        NS <= S1;

                when S1=>
                        if(CntDelay_Out < 40) then
                                NS<=S1;
                        else
```

```
                        NS<=S2;
                end if;
                CntDelay_INC<='1';

        when S2=>                                               --Wait
                if(CntBus_Out < 128) then
                        NS<=S2;
                else
                        NS<=S3;
                end if;
                CntBus_INC<='1';

        when S3 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S3;
                else
                        NS <=S4;
                end if;
                CntBus_Rst <='0';               -- Reset Bus Counter
                CntDelay_Rst <='0';             -- Reset Delay Counter

        when S4 =>
                Bus_Int1_AddrIn <= Addr_Bootloader_Start; --
Addr_Bootloader_Start is a constant from Common file
                Bus_Int1_RE <='1';
                NS <= S5;

        when S5 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S5;
                else

                        NS <=S6;
                end if;
                Temp_Vrble_Data <= Bus_Int1_DataOut;
                LD_Vrble_Data <= '1';

        when S6 =>
                if(Error = '1') then
                        Temp_Boot_Start <= '0';
----SE
                        NS <= S7;
                else
                        Temp_Boot_Start <= Vrble_Data(0);
----Sa
                        NS <= S20;
```

```
                        end if;
                        LD_Boot_Start <= '1';

        ---------------------------------- Start ERROR Procedure ----------------------------------------
-


----SE
                    when S7 =>
                        Temp_FW_Type <= '1';        -- 1 is for Backup firmware type
                        LD_FW_Type <= '1';
                        LD_Boot_Wrkn <= '1';
                        NS <= S8;

                    when S8 =>              -- Enable Bootload until Boot_Wrkn equals 1.
                        if(Boot_Wrkn_reg_o = '0')then
                            Temp_Bootload_EN <= '0';   -- Active Low
                            NS <= S8;
                        else
                            Temp_Bootload_EN <= '1';   -- Active Low
                            NS <= S9;
                        end if;
                        LD_Bootload_EN <= '1';
                        LD_Boot_Wrkn <= '1';
                        Temp_FW_Type <= '1';        -- 1 is for Backup firmware type
                        LD_FW_Type <= '1';

                    when S9 =>
                        if(Bus_Int1_Busy = '1') then
                            NS <= S9;
                        else

                            NS <=S10;
                        end if;

                    when S10 =>
                        Bus_Int1_AddrIn <= Addr_Bootloader_Status; --
Addr_Bootloader_Status is a constant from Common file
                        Bus_Int1_DataIn <= X"0002"; -- Bootloader_Status = 2 (Backup
Working)
                        Bus_Int1_WE <='1';
                        NS <= S11;

                    when S11 =>
                        if(Bus_Int1_Busy = '1') then
                            NS <= S11;
```

```
                else

                        NS <=S12;
                end if;
                LD_Boot_Done <= '1';

        when S12 =>
                if(Boot_Done_reg_o = '0') then

                        NS <= S12;
                else

                        NS <= S13;
                end if;
                LD_Boot_Done <= '1';

        when S13 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S13;
                else

                        NS <=S14;
                end if;

        when S14 =>
                Bus_Int1_AddrIn <= Addr_Bootloader_Status; --
Addr_Bootloader_Status is a constant from Common file
                Bus_Int1_DataIn <= X"0000"; -- Bootloader_Status = 0
(Ready/Done)
                Bus_Int1_WE <='1';
                NS <= S15;

        when S15 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S15;
                else

                        NS <=S16;
                end if;

        when S16=>
                if(Error = '1')then

                        NS <= S16;
                else
```

```
                        NS <= S17;
                end if;

        when S17 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S17;
                else

                        NS <=S18;
                end if;

        when S18 =>
                Bus_Int1_AddrIn <= Addr_Bootloader_Start; --
Addr_Bootloader_Start is a constant from Common file
                Bus_Int1_DataIn <= X"0000";
                Bus_Int1_WE <='1';
                NS <= S19;

        when S19 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S19;
                else

                        NS <=S0;
                end if;
                Temp_Boot_Start <= '0';
                LD_Boot_Start <= '1';
                Temp_Vrble_Data <= (others => '0');
                LD_Vrble_Data <= '1';


        ---------------------------------- End ERROR Procedure ----------------------------------------



------Sa
        when S20 =>
                NS <= S21;


        -- Check if Boot_Start is ON
        when S21 =>
                if(Boot_Start = '1') then
```

```
                NS <= S22;
            else
                NS <= S0;
            end if;


        when S22 =>
            Temp_FW_Type <= '0';        -- 0 is for New firmware type
            LD_FW_Type <= '1';
            LD_Boot_Wrkn <= '1';
            NS <= S23;

        when S23 =>             -- Enable Bootload until Boot_Wrkn equals 1.
            if(Boot_Wrkn_reg_o = '0')then
                Temp_Bootload_EN <= '0';    -- Active Low
                NS <= S23;
            else
                Temp_Bootload_EN <= '1';    -- Active Low
                NS <= S24;
            end if;
            LD_Bootload_EN <= '1';
            LD_Boot_Wrkn <= '1';
            Temp_FW_Type <= '0';        -- 0 is for New firmware type
            LD_FW_Type <= '1';

        when S24 =>
            if(Bus_Int1_Busy = '1') then
                NS <= S24;
            else

                NS <=S25;
            end if;

        when S25 =>
            Bus_Int1_AddrIn <= Addr_Bootloader_Status; --
Addr_Bootloader_Status is a constant from Common file
            Bus_Int1_DataIn <= X"0001"; -- Bootloader_Status = 1 (Working)
            Bus_Int1_WE <='1';
            NS <= S26;

        when S26 =>
            if(Bus_Int1_Busy = '1') then
                NS <= S26;
            else

                NS <=S27;
```

```
                end if;
                LD_Boot_Done <= '1';

            when S27 =>
                if(Boot_Done_reg_o = '0') then

                    NS <= S27;
                else

                    NS <= S28;
                end if;
                LD_Boot_Done <= '1';

            when S28 =>
                if(Bus_Int1_Busy = '1') then
                    NS <= S28;
                else

                    NS <=S29;
                end if;

            when S29 =>
                Bus_Int1_AddrIn <= Addr_Validation_Start; --
Addr_Validation_Start is a constant from Common file
                Bus_Int1_DataIn <= X"0001";
                Bus_Int1_WE <='1';
                NS <= S30;

            when S30 =>
                if(Bus_Int1_Busy = '1') then
                    NS <= S30;
                else

                    NS <=S31;
                end if;

            when S31 =>
                Bus_Int1_AddrIn <= Addr_Bootloader_Status; --
Addr_Bootloader_Status is a constant from Common file
                Bus_Int1_DataIn <= X"0000"; -- Bootloader_Status = 0
(Ready/Done)
                Bus_Int1_WE <='1';
                NS <= S32;

            when S32 =>
                if(Bus_Int1_Busy = '1') then
```

```
                                        NS <= S32;
                                    else

                                        NS <=S33;
                                    end if;

----Sback
                            when S33 =>
                                    if(Error = '1')then

-----SE
                                        NS <= S7;
                                    else

                                        NS <= S34;
                                    end if;
                                    LD_HP_EN <= '1';

                            when S34 =>
                                    if(HP_EN_reg_o = '0')then

----Sback
                                        NS <= S33;

                                    else

                                        NS <= S35;
                                    end if;
                                    LD_HP_EN <= '1';

                            when S35 =>
                                    if(Bus_Int1_Busy = '1') then
                                        NS <= S35;
                                    else

                                        NS <=S36;
                                    end if;


                            -- Reset Bootload Start
                            when S36 =>
                                    Bus_Int1_AddrIn <= Addr_Bootloader_Start; --
Addr_Bootloader_Start is a constant from Common file
                                    Bus_Int1_DataIn <= X"0000";
                                    Bus_Int1_WE <='1';
                                    NS <= S37;
```

```
                when S37 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S37;
                        else

                                NS <=S0;
                        end if;
                        Temp_Boot_Start <= '0';
                        LD_Boot_Start <= '1';
                        Temp_Vrble_Data <= (others => '0');
                        LD_Vrble_Data <= '1';









                when others =>
                        NS <= S0;

                end case;
end process;









----State Sync
sync_States: process
begin
        wait until clk'event and clk = '1';
```

```
		if rst = '0' then
			CS <= S0;

		else
			CS <= NS;

		end if;
	end process;
	----End State Sync


end Behavioral;
```

A-2-3 Bootloader

```
----------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Chris Farnell, Estefano Soria
-- Created by: Chris Farnell
-- Edited by: Estefano Soria
--
-- Create Date:		14Jan2021
-- Design Name:		DT_Bootloader
-- Module Name:		DT_Bootloader
-- Project Name:		DT_Bootloader
-- Target Devices:		LCMXO2-7000HC-4FG484C (UCB v1.3a)
-- Tool versions:		Lattice Diamond_x64 Build 3.10.2.115.1
-- Description:
-- This module is responsible for the actual selection and loading for the DSP Firmware via a
serial UART Interface.

-- Revisions:--
--

--
--
-- Additional Comments:
--
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_arith.all;
```

```
library machxo2;
use machxo2.all;

library work;
use work.Test2_DT_Common.all;

entity DT_Bootloader is
	generic(
				Baud : integer := 9600;				--9,600 bps
				clk_in : integer := 25000000);			--25MHz
   Port (
			clk : IN  std_logic;
			rst : IN  std_logic;

			--SW_1 : in  STD_LOGIC;				-- Board DIP Switch
			--BTN_1 : in  STD_LOGIC;				-- Board Push Button
			--FW_Load_EN : in STD_LOGIC;
			--Backup_FW_EN : in STD_LOGIC;
			Bootload_EN : IN STD_LOGIC;
			FW_Type	: IN STD_LOGIC;
			Bootload_Wrkn : OUT STD_LOGIC;
			Bootload_Done : OUT STD_LOGIC;

			DSP_rcv : OUT std_logic;
			DSP_xmt : IN  std_logic;
			DSP_RAM_clk : OUT  STD_LOGIC;
			DSP_RAM_clk_en : OUT  STD_LOGIC;
			DSP_RAM_rst : OUT  STD_LOGIC;
			DSP_RAM_we : OUT  STD_LOGIC;
			DSP_RAM_addr : OUT  STD_LOGIC_VECTOR (15 downto 0);
			DSP_RAM_din : OUT  STD_LOGIC_VECTOR (15 downto 0);
			DSP_RAM_dout : IN  STD_LOGIC_VECTOR (15 downto 0);
			DSP_Rst : OUT  STD_LOGIC
			--Val_EN : OUT STD_LOGIC
			 );
end DT_Bootloader;



architecture Behavioral of DT_Bootloader is

	-- Declare Std_Counter Component
	component Std_Counter is
	generic
	(
		Width : integer			-- width of counter
```

```
	);
	port(INC,rst,clk: in std_logic;
		Count: out STD_LOGIC_VECTOR(Width-1 downto 0));
	end component;


	type state_type is
(S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,S20,S21);
	signal CS_RS232_W, NS_RS232_W : state_type;
	signal tx_done :STD_LOGIC:= '0';
	signal i,j: STD_LOGIC_VECTOR (15 downto 0):= (others => '0');
	signal uartclk : STD_LOGIC:= '0';
	signal u: integer;
	signal rs232_xmt : STD_LOGIC:= '1';
	signal txbuff: STD_LOGIC_VECTOR(9 downto 0):= (others => '1');		--buff used to
transmit 1 bytes with start and stop bits

	--signal Val_EN_Temp: std_logic:= '0';
	--signal LD_Val_EN: std_logic:= '0';
	signal Bootload_Wrkn_Temp: std_logic:= '0';
	signal LD_Bootload_Wrkn: std_logic:= '0';
	signal Bootload_Done_Temp: std_logic:= '0';
	signal LD_Bootload_Done: std_logic:= '0';




	--Declare Signals for Registers
	signal LD_busy,LD_busy2,LD_rx,LD_tx,LD_temp_data,LD_temp2: STD_LOGIC:= '0';
	signal LD_Temp_Addr: STD_LOGIC:= '0';
	signal Ld_temp_cmd: STD_LOGIC:= '0';
	signal busy,busy_reg_o,busy2,busy2_reg_o,rx,rx_reg_o,tx,tx_reg_o: STD_LOGIC:= '0';
	signal temp2_reg_o, temp2: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
	signal Temp_Data_reg_o, Temp_Data: STD_LOGIC_VECTOR(15 downto 0):= (others
=> '0');
	signal Temp_Cmd_reg_o, Temp_Cmd: STD_LOGIC_VECTOR(7 downto 0):= (others
=> '0');
	--Added for DSP Loading
	signal
LD_DSP_FW_Config,LD_DSP_Reset,LD_Addr_DSP_FW_Len,LD_Addr_DSP_FW_Start :
STD_LOGIC:= '0';
	signal DSP_FW_Config_reg_o,DSP_Reset,DSP_Reset_reg_o: STD_LOGIC:='0';
	signal Addr_DSP_FW_Len,Addr_DSP_FW_Len_reg_o : STD_LOGIC_VECTOR(15
downto 0):= (others => '0');
```

```
	signal Addr_DSP_FW_Start,Addr_DSP_FW_Start_reg_o : STD_LOGIC_VECTOR(15
downto 0):= (others => '0');
	signal LD_DSP_FW_Len : STD_LOGIC:= '0';
	signal DSP_FW_Len,DSP_FW_Len_reg_o  : STD_LOGIC_VECTOR(15 downto 0):=
(others => '0');

	-- Program Counter
	signal Program_Cnt_INC, Program_Cnt_rst: std_logic := '0';
	signal Program_Cnt_out: std_logic_vector(15 downto 0):= (others => '0');

	-- Wait Counter (DSP Reset and Load)
	signal Wait_Cnt_INC, Wait_Cnt_rst: std_logic := '0';
	signal Wait_Cnt_out: std_logic_vector(31 downto 0):= (others => '0');



	----User defined variables
	-- CM is the Clock Divder 25MHz/CM=115,200 Baud
	constant CM : integer := clk_in/Baud;
	-- CN is the read offset for serial input
	constant CN : integer :=CM/2;
	----End User defined variables


begin

	-- Instantiate Program_Cnt_16
	Program_Cnt: Std_Counter
	generic map
	(
		Width => 16
	)
	port map(
		clk => clk,
		rst=> Program_Cnt_rst,
		INC=> Program_Cnt_INC,
		Count=> Program_Cnt_Out
	);

	-- Instantiate Wait_Cnt_32
	Wait_Cnt: Std_Counter
	generic map
	(
		Width => 32
	)
	port map(
```

```
		clk => clk,
		rst=> Wait_Cnt_rst,
		INC=> Wait_Cnt_INC,
		Count=> Wait_Cnt_Out
	);



	----Registers
	Reg_Proc: process
	begin
		wait until clk'event and clk = '1';
		if rst = '0' then
			--busy_reg_o <= '0';
			busy2_reg_o <='0';
			--rx_reg_o <= '0';
			tx_reg_o <= '0';
			temp_data_reg_o <= (others => '0');
			temp2_reg_o <= (others => '0');
			DSP_FW_Config_reg_o <= '0';
			DSP_Reset_reg_o <= '0'; -- Active Low
			Addr_DSP_FW_Len_reg_o <= (others => '0');
			Addr_DSP_FW_Start_reg_o <= (others => '0');
			DSP_FW_Len_reg_o <= (others => '0');
			--Val_EN <= '0';
		else
		--if (LD_busy = '1') then busy_reg_o <= busy; end if;
		if (LD_busy2 = '1') then busy2_reg_o <= busy2; end if;
		--if (LD_rx = '1') then rx_reg_o <= rx; end if;
		if (LD_tx = '1') then tx_reg_o <= tx; end if;
		if (LD_temp_data = '1') then temp_data_reg_o <= temp_data; end if;
		if (LD_temp2 = '1') then temp2_reg_o <= temp2; end if;
		if (LD_DSP_FW_Config = '1') then DSP_FW_Config_reg_o <= FW_Type; end
if;--Backup_FW_EN; end if;
		if (LD_DSP_Reset = '1') then DSP_Reset_reg_o <= DSP_Reset; end if;
		if (LD_Addr_DSP_FW_Len = '1') then Addr_DSP_FW_Len_reg_o <=
Addr_DSP_FW_Len; end if;
		if (LD_Addr_DSP_FW_Start = '1') then Addr_DSP_FW_Start_reg_o <=
Addr_DSP_FW_Start; end if;
		if (LD_DSP_FW_Len = '1') then DSP_FW_Len_reg_o <= DSP_RAM_dout; end
if;
		if (LD_Bootload_Wrkn = '1') then Bootload_Wrkn <= Bootload_Wrkn_Temp;
end if;
		if (LD_Bootload_Done = '1') then Bootload_Done <= Bootload_Done_Temp; end
if;
```

```
            --if (LD_Val_EN = '1') then Val_EN <= Val_EN_Temp; end if;

        end if;
        end process;
        ----End Registers



        ----Next State Logic for Serial Interface Write
        NSL_RS232_W:
process(CS_RS232_W,Bootload_EN,FW_Type,DSP_FW_Config_reg_o,Addr_DSP_FW_Len_r
eg_o,Addr_DSP_FW_Len_reg_o,Addr_DSP_FW_Start_reg_o,DSP_FW_Len,DSP_RAM_dout,
tx_done,Program_Cnt_Out,Wait_Cnt_Out)
        begin

            ----Default States to remove latches
            tx<='0';
            NS_RS232_W <= S0;
            temp2 <= (others => '0');
            LD_tx <= '0';
            LD_temp2<='0';
            Busy2 <='0';
            LD_Busy2<='0';
            --Added for DSP Firmware Loading
            LD_DSP_FW_Config <= '0';
            DSP_Reset <= '1';       -- Active Low
            LD_DSP_Reset <= '0';
            LD_Addr_DSP_FW_Len <= '0';
            LD_Addr_DSP_FW_Start <= '0';
            DSP_RAM_addr <= (others => '0');
            LD_DSP_FW_Len <= '0';
            Program_Cnt_rst <= '1'; -- Active Low
            Program_Cnt_INC <= '0';
            Wait_Cnt_rst <= '1'; -- Active Low
            Wait_Cnt_INC <= '0';

            --LD_Val_EN <= '0';


            ----------------New Unlatches--------------------

            -- Notify Bus Master that Bootload is working and to unlatch the Bootload EN.
            Bootload_Wrkn_Temp <= '0';
            LD_Bootload_Wrkn   <= '0';

            Bootload_Done_Temp <= '0';
```

```
                LD_Bootload_Done   <= '0';
                ------------------------------------------------

                case CS_RS232_W is

                        when S0=>
                                if(Bootload_EN = '1') then
                                        NS_RS232_W<=S0;
                                else
                                        NS_RS232_W<=S1;
                                end if;
                                busy2 <='0';                                    -- the busy signal
stops the baud generator
                                tx <= '0';                                              -- signals to
stop sending data
                                LD_tx <= '1';
                                LD_busy2 <= '1';
                                LD_DSP_FW_Config <= '1';          -- Load DSP Firmware
Configuration from Backup_FW_EN
                                Program_Cnt_rst <= '0';                 -- Active Low
                                Wait_Cnt_rst <= '0';              -- Active Low
                                --Val_EN_Temp <= '0';                   -- Validation Enable register
is off
                                --LD_Val_EN <= '1';                     -- Loading the validation
enable register to be turned off
                                LD_DSP_Reset <= '1';
                            ------------------------------------ THIS SIGNAL NEEDS TO BE EN SO
THAT DSP_Reset IS 1 (DSP NOT RESET), OTHERWISE IT WILL STAY AT 0 (DSP IN
RESET)----
                                Bootload_Wrkn_Temp <= '0';          -- Notify Bus Master that
Bootload is NOT working yet and is ready for Bootload EN.
                                LD_Bootload_Wrkn <= '1';
                                Bootload_Done_Temp <= '1';-- Bootload Done signal stays on
until Bootload is enabled again, once enabled, the Done signal turns OFF immediately
                                LD_Bootload_Done   <= '1';

                        when S1=>                                                       -- Set DSP
Firmware addresses based on Backup_FW_EN position from S0 and Reset DSP
                                if(DSP_FW_Config_reg_o = '0') then
                                        Addr_DSP_FW_Len <= Addr_DSP_FW1_Len;
                                        Addr_DSP_FW_Start <= Addr_DSP_FW1_Start;
                                        --Val_EN_Temp <= '1';                   -- Validation Enable
register is on
                                else
                                        Addr_DSP_FW_Len <= Addr_DSP_FW2_Len;
                                        Addr_DSP_FW_Start <= Addr_DSP_FW2_Start;
```

```
                                --Val_EN_Temp <= '0';                -- Validation Enable
register is off
                        end if;
                        DSP_Reset <= '0';                            --Reset DSP (Active
Low)
                        LD_Addr_DSP_FW_Len <= '1';
                        LD_Addr_DSP_FW_Start <= '1';
                        LD_DSP_Reset <= '1';
                        Bootload_Wrkn_Temp <= '0';          -- Notify Bus Master that
Bootload is NOT working yet and is ready for Bootload EN.
                        LD_Bootload_Wrkn <= '1';
                        Bootload_Done_Temp <= '0';-- Bootload Done signal stays on
until Bootload is enabled again, once enabled, the Done signal turns OFF immediately
                        LD_Bootload_Done   <= '1';
                        NS_RS232_W <= S2;

                when S2 =>                                                --Delay for
longer DSP Reset ( Wait 1.3 ms for DSP to reset.)
                        if(Wait_Cnt_Out < X"00007FFF") then
                                Wait_Cnt_INC <= '1';
                                NS_RS232_W<=S2;
                        else
                                NS_RS232_W<=S3;
                        end if;
                        DSP_RAM_addr <= Addr_DSP_FW_Len;   --Pre-Load Address

                when S3 =>                                                --Read MSB
of Firmware Length (Step 2: Read Data and Store)
                        DSP_RAM_addr <= Addr_DSP_FW_Len;
                        LD_DSP_FW_Len <= '1';
                        DSP_Reset <= '1';                            -- Disable Reset DSP
(Active Low)
                        LD_DSP_Reset <= '1';
                        Wait_Cnt_rst <= '0';              -- Reset Counter (Active Low)
                        Bootload_Wrkn_Temp <= '1';          -- Notify Bus Master that
Bootload is working and to unlatch the Bootload EN.
                        LD_Bootload_Wrkn <= '1';
                        Bootload_Done_Temp <= '0';-- Bootload Done signal stays on
until Bootload is enabled again, once enabled, the Done signal turns OFF immediately
                        LD_Bootload_Done   <= '1';
                        NS_RS232_W<=S4;


                when S4 =>                                                --Set Length
Variable ( Wait 1s for DSP to Boot.)
                        if(Wait_Cnt_Out < X"017D7840") then
```

```
                    Wait_Cnt_INC <= '1';
                    NS_RS232_W<=S4;
                else
                    NS_RS232_W<=S7;
                end if;
                DSP_RAM_addr <= Addr_DSP_FW_Len;
                LD_DSP_FW_Len <= '1';
                Bootload_Wrkn_Temp <= '1';          -- Notify Bus Master that
Bootload is working and to unlatch the Bootload EN.
                LD_Bootload_Wrkn <= '1';
                Bootload_Done_Temp <= '0';-- Bootload Done signal stays on
until Bootload is enabled again, once enabled, the Done signal turns OFF immediately
                LD_Bootload_Done   <= '1';


            when S7 =>                                          -- Load "A"
Data and prepare to send to DSP
                temp2 <= X"41";                                 -- Send "A" to
DSP for Autobaud configuration
                LD_temp2 <= '1';
                Program_Cnt_rst <= '0';             -- Reset Counter (Active
Low)
                Wait_Cnt_rst <= '0';            -- Reset Counter (Active Low)
                NS_RS232_W<=S8;

            when S8=>
                busy2 <='1';                            -- the busy signal
starts the baud generator
                tx<='1';                                        -- signals to
start sending data
                LD_tx<='1';
                LD_busy2 <= '1';
                NS_RS232_W<=S9;

            when S9=>                                           -- Wait until
Data is sent
                if(tx_done='0') then
                    NS_RS232_W <=S9;
                else
                    NS_RS232_W <=S10;
                end if;


            when S10=>                                          -- Data has
finished sending; stop transmission
```

```
                busy2 <='0';                    -- the busy signal
stops the baud generator
                tx <= '0';                      -- signals to
stop sending data
                LD_tx <= '1';
                LD_busy2 <= '1';
                NS_RS232_W<=S11;

            when S11=>
            DSP_RAM_addr <= Addr_DSP_FW_Start_reg_o + Program_Cnt_Out;
            NS_RS232_W<=S12;

            when S12=>
            DSP_RAM_addr <= Addr_DSP_FW_Start_reg_o + Program_Cnt_Out;
            temp2 <= DSP_RAM_dout(15 downto 8);
        -- Send Program Data via Serial
            LD_temp2 <= '1';
            NS_RS232_W<=S13;

            when S13=>
                busy2 <='1';                    -- the busy signal
starts the baud generator
                tx<='1';                        --
signals to start sending data
                LD_tx<='1';
                LD_busy2 <= '1';
                NS_RS232_W<=S14;

            when S14=>                          -- Wait until
Data is sent
                if(tx_done='0') then
                    NS_RS232_W <=S14;
                else
                    NS_RS232_W <=S15;
                end if;

            when S15=>                          -- Data has
finished sending; stop transmission
                DSP_RAM_addr <= Addr_DSP_FW_Start_reg_o +
Program_Cnt_Out;
                busy2 <='0';                    -- the busy signal
stops the baud generator
                tx <= '0';                      -- signals to
stop sending data
                LD_tx <= '1';
                LD_busy2 <= '1';
```

```
                NS_RS232_W<=S16;



            when S16=>
            DSP_RAM_addr <= Addr_DSP_FW_Start_reg_o + Program_Cnt_Out;
            temp2 <= DSP_RAM_dout(7 downto 0);
        -- Send Program Data via Serial
            LD_temp2 <= '1';
            NS_RS232_W<=S17;

            when S17=>
                busy2 <='1';                                -- the busy signal
starts the baud generator
                tx<='1';                                                    --
signals to start sending data
                LD_tx<='1';
                LD_busy2 <= '1';
                NS_RS232_W<=S18;

            when S18=>                                              -- Wait until
Data is sent
                if(tx_done='0') then
                    NS_RS232_W <=S18;
                else
                    NS_RS232_W <=S19;
                end if;

            when S19=>                                              -- Data has
finished sending; stop transmission
                busy2 <='0';                                -- the busy signal
stops the baud generator
                tx <= '0';                                  -- signals to
stop sending data
                LD_tx <= '1';
                LD_busy2 <= '1';
                NS_RS232_W<=S20;

            when S20=>                                              -- Wait until
Data is sent
                if(Program_Cnt_Out < DSP_FW_Len_reg_o - 1 ) then
                    Program_Cnt_INC <= '1';                 -- Increment
Program Cnt
                    NS_RS232_W <=S11;
                else
```

```
                                        --LD_Val_EN <= '1';                     -- Loading the
validation enable register
                                        NS_RS232_W <=S21;
                                end if;

                        when S21 =>
                                --LD_Val_EN <= '1';
                                Bootload_Wrkn_Temp <= '1';          -- Notify Bus Master that
Bootload is working and to unlatch the Bootload EN.
                                LD_Bootload_Wrkn <= '1';
                                Bootload_Done_Temp <= '0';-- Bootload Done signal stays on
until Bootload is enabled again, once enabled, the Done signal turns OFF immediately
                                LD_Bootload_Done   <= '1';
                                NS_RS232_W <= S0;

                        when others =>
                                NS_RS232_W <=S0;

                end case;
        end process;
        ----End Next State Logic for Serial Interface Write


--
--




        ----UART Clock Divider
        UART_Clk: process
        begin
                wait until clk'event and clk = '1';

                if(rst = '0' or (busy_reg_o = '0' and busy2_reg_o = '0')) then
                        uartclk <='0';
                        i <= CONV_STD_LOGIC_VECTOR(CN,16);
                elsif( i = CM ) then
                                uartclk <= '1';
                                i <= X"0000";
                else
                                i <= i+1;
                                uartclk<='0';
                        end if;
```

```
end process;
---- End UART Clock Divider



-----UART_Xmit
UART_Xmit: process
begin
	wait until clk'event and clk = '1';
	if (rst = '0' or tx_reg_o='0') then
		rs232_xmt<='1';
		tx_done <='0';
		u<=0;

		--structure the 10-bit frame to be sent
		txbuff(9)<='1'; --stopbit 2
		txbuff(8 downto 1) <= temp2_reg_o;
		txbuff(0)<='0'; --startbit 2
	else
		if uartclk = '1' then
			if(u<10) then
				rs232_xmt<= txbuff(0);
				txbuff(8 downto 0) <= txbuff(9 downto 1);
				tx_done<='0';
				u<=u+1;
			else
				u<=0;
				tx_done<='1';
			end if;
		end if;
	end if;
end process;
-----End UART_Xmit


----State Sync
sync_States: process
begin
	wait until clk'event and clk = '1';
	if rst = '0' then
		CS_RS232_W <= S0;
	else
		CS_RS232_W <= NS_RS232_W;
	end if;
end process;
----End State Sync
```

```
        ---- Combinatorial Logic
        comb: process (rs232_xmt,clk,DSP_Reset_reg_o)
        begin
                DSP_rcv <= rs232_xmt;                   -- Map TX of Module to RX pin of
DSP
                --DSP_xmt <= '1';                              -- Tied High, not used
                DSP_RAM_we <= '0';                            -- Disable Write Enable
                DSP_RAM_clk <= clk;                           -- Tie to System Clock
                DSP_RAM_clk_en <= '1';                  -- Enable RAM Clock
                DSP_RAM_rst <= '0';                     -- Disable Reset
                DSP_RAM_din <= (others => '0');    -- Not Used
                DSP_Rst <=DSP_Reset_reg_o;              -- DSP Reset Pin
        end process;

end Behavioral;
```

A-2-4 Firmware Validation

```
----------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Estefano Soria
--
-- Create Date:                26Dec2020
-- Design Name:                DSP_Firmware_Validation
-- Module Name:                DSP_Firmware_Validation - Behavioral
-- Project Name:               DSP_Firmware_Validation
-- Target Devices:             LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:              Lattice Diamond_x64 Build 3.11
-- Description:
--
---- PinOut:
--
-- Revision: V1.1
--
--
--
-- Additional Comments:
--
--                          -
--                          -
--                          -
--                          -
--                          -
--                          -
--                          -
--                          -
```

```
----------------------------------------------------------------------------


Library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library machxo2;
use machxo2.all;

library work;
use work.Test2_DT_Common.all;


entity Test2_DT_Firmware_Validation is
   Port (
                              clk : in STD_LOGIC;
                              rst : in STD_LOGIC;

                              Data : INOUT  std_logic_vector(15 downto 0);
                              Addr : OUT  std_logic_vector(15 downto 0);
                              Xrqst : OUT  std_logic;
                              XDat : IN  std_logic;
                              YDat : OUT  std_logic;
                              BusRqst : OUT  std_logic;
                              BusCtrl : IN  std_logic;

                              Emu_SW01    : in std_logic;
                              Emu_SW02    : in std_logic;
                              Emu_SW03    : in std_logic;
                              Emu_SW04    : in std_logic;
                              Emu_SW05    : in std_logic;
                              Emu_SW06    : in std_logic;


                              HP_Done           : in std_logic;         ---- Signal coming
from DSP_Hot-Patch module saying that HP process is complete
                              Boot_Done   : in std_logic;
                              Boot_Wrkn   : in std_logic;
                              Emu_EN            : out std_logic;
                              HP_EN             : out std_logic;        ---- Signal sent to
DSP_Hot-Patch module to enable HP process
                              Error       : out std_logic

                              );
```

```
end Test2_DT_Firmware_Validation;

architecture Behavioral of Test2_DT_Firmware_Validation is

	type state_type is
(S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,S20,S21,S22,S23,
S24,S25,S26,S27,S28,S29,S30,S31,S32,S33,S34,S35,S36,S37,S38,S39,S40,S41,S42,S43,S44,S4
5,S46,S47,S48,S49,S50,S51,S52,S53,S54,S55,S56,S57,S58,S59,S60);
	signal CS, NS, CS_Chk, NS_Chk, CS_ShCrk, NS_ShCrk, CS_DeadT, NS_DeadT :
state_type;

	signal SW01_t : std_logic := '0';
	signal SW01   : std_logic := '0';
	signal SW02_t : std_logic := '0';
	signal SW02   : std_logic := '0';
	signal SW03_t : std_logic := '0';
	signal SW03   : std_logic := '0';
	signal SW04_t : std_logic := '0';
	signal SW04   : std_logic := '0';
	signal SW05_t : std_logic := '0';
	signal SW05   : std_logic := '0';
	signal SW06_t : std_logic := '0';
	signal SW06   : std_logic := '0';


	---Signals for emergency Limit Exceeded process----
	signal  eq : std_logic := '0';                          ---- If all Bad_FW are OFF
then eq is OFF. If it is ever ON, then backup FW is EN.
	--signal  Bad_FW1 : std_logic := '0';                   ---- Bad FW for Short Circuit
check
	--signal  Bad_FW2 : std_logic := '0';                   ---- Bad FW for  check
	--signal  Bad_FW3 : std_logic := '0';                   ---- Bad FW for  check
	signal dt_done : STD_LOGIC := '0';



	--Bus Interface Signals
	signal  Bus_Int1_Busy : std_logic := '0';
	signal  Bus_Int1_WE : std_logic := '0';
	signal  Bus_Int1_RE : std_logic := '0';
	signal  Bus_Int1_AddrIn : std_logic_vector (15 downto 0) := (others => '0');
	signal  Bus_Int1_DataIn : std_logic_vector (15 downto 0) := (others => '0');
	signal  Bus_Int1_DataOut : std_logic_vector (15 downto 0) := (others => '0');
```

```
-- Registers
signal   LD_HP_EN            : std_logic := '0';
signal Temp_HP_EN            : std_logic := '0';

signal   LD_HP_Done                  : std_logic := '0';
signal      HP_Done_reg_o    : std_logic := '0';

signal   LD_Bad_FW1          : std_logic := '0';
signal Temp_Bad_FW1          : std_logic := '0';
signal           Bad_FW1             : std_logic := '0';

signal   LD_Bad_FW2          : std_logic := '0';
signal Temp_Bad_FW2          : std_logic := '0';
signal           Bad_FW2             : std_logic := '0';

signal   LD_Bad_FW3          : std_logic := '0';
signal Temp_Bad_FW3          : std_logic := '0';
signal           Bad_FW3             : std_logic := '0';

signal   LD_EN_Chk           : std_logic := '0';
signal      EN_Chk_reg_o     : std_logic := '0';
signal           EN_Chk              : std_logic := '0';

signal   LD_Stop_Chk                 : std_logic := '0';
signal Temp_Stop_Chk                 : std_logic := '0';
signal           Stop_Chk            : std_logic := '0';

signal   LD_EN_Dead_Time_Chk             : std_logic := '0';
signal      EN_Dead_Time_Chk_reg_o       : std_logic := '0';
signal           EN_Dead_Time_Chk        : std_logic := '0';

signal   LD_Boot_Done                : std_logic := '0';
signal      Boot_Done_reg_o  : std_logic := '0';

signal   LD_Boot_Wrkn                : std_logic := '0';
signal      Boot_Wrkn_reg_o  : std_logic := '0';

signal   LD_Emu_EN           : std_logic := '0';
signal Temp_Emu_EN                   : std_logic := '0';

signal   LD_HP_Cmd           : std_logic := '0';
signal Temp_HP_Cmd                   : std_logic_vector (15 downto 0) := (others => '0');
signal           HP_Cmd              : std_logic_vector (15 downto 0) := (others => '0');

signal   LD_Error            : std_logic := '0';
signal Temp_Error            : std_logic := '0';
```

```
signal   LD_Err_Type              : std_logic := '0';
signal Temp_Err_Type              : std_logic_vector (15 downto 0) := (others => '0');
signal           Err_Type         : std_logic_vector (15 downto 0) := (others => '0');

signal   LD_Vrble_Data            : std_logic := '0';
signal Temp_Vrble_Data            : std_logic_vector (15 downto 0) := (others => '0');
signal           Vrble_Data       : std_logic_vector (15 downto 0) := (others => '0');

signal   LD_Val_Start             : std_logic := '0';
signal Temp_Val_Start             : std_logic := '0';
signal           Val_Start        : std_logic := '0';


-- Counters
signal CntBus_INC : std_logic := '0';
signal CntBus_Rst : std_logic := '0';
signal CntBus_Out : std_logic_vector(15 downto 0) := (others => '0');

signal CntDelay_INC : std_logic := '0';
signal CntDelay_Rst : std_logic := '0';
signal CntDelay_Out : std_logic_vector(7 downto 0) := (others => '0');

signal Cnt_PreChk_INC : std_logic := '0';
signal Cnt_PreChk_Rst : std_logic := '0';
signal Cnt_PreChk_Out : std_logic_vector(31 downto 0) := (others => '0');




signal Cnt_DeadT_INC : std_logic := '0';
signal Cnt_DeadT_Rst : std_logic := '0';
signal Cnt_DeadT_Out : std_logic_vector(7 downto 0) := (others => '0');
```

-----** SHORT CIRCUIT CHECK AND DEAD TIME CHECK WILL BE 2 SEPARATE PROCESSES IN THIS FILE. NO INSTANTIATION **

```
-- Compontents --

--declare Std_Counter Component
component Std_Counter is
```

```
        generic
        (
                Width : integer                 --width of counter
        );
        port(INC,rst,clk: in std_logic;
                 Count: out STD_LOGIC_VECTOR(Width-1 downto 0));
        end component;


        --declare Bus Interface
        COMPONENT Bus_Int
  PORT(
                clk : IN  std_logic;
                rst : IN  std_logic;
                DataIn : IN  std_logic_vector(15 downto 0);
                DataOut : OUT  std_logic_vector(15 downto 0);
                AddrIn : IN  std_logic_vector(15 downto 0);
                WE : IN  std_logic;
                RE : IN  std_logic;
                Busy : OUT  std_logic;
                Data : INOUT  std_logic_vector(15 downto 0);
                Addr : OUT  std_logic_vector(15 downto 0);
                Xrqst : OUT  std_logic;
                XDat : IN  std_logic;
                YDat : OUT  std_logic;
                BusRqst : OUT  std_logic;
                BusCtrl : IN  std_logic
   );
   END COMPONENT;



begin


        --instantiate Delay_Cnt
        Delay_Cnt: Std_Counter
        generic map
        (
                Width => 8
        )
        port map(
                clk => clk,
                rst=> CntDelay_rst,
                INC=> CntDelay_INC,
                Count=> CntDelay_Out
```

```
);


--instantiate Bus_Cnt
Bus_Cnt: Std_Counter
generic map
(
        Width => 16
)
port map
(
        clk => clk,
        rst=> CntBus_rst,
        INC=> CntBus_INC,
        Count=>CntBus_Out
);


-- Instantiate Wait_Cnt_32
Cnt_PreChk: Std_Counter
generic map
(
        Width => 32
)
port map(
        clk => clk,
        rst=> Cnt_PreChk_rst,
        INC=> Cnt_PreChk_INC,
        Count=> Cnt_PreChk_Out
);


Det_Cnt: Std_Counter
generic map
(
        Width => 8
)
port map(
        clk => clk,
        rst=> Cnt_DeadT_Rst,
        INC=> Cnt_DeadT_INC,
        Count=> Cnt_DeadT_Out
);


--Instantiate Bus Interface
```

```
  Bus_Int1: Bus_Int PORT MAP (
 clk => clk,
 rst => rst,
 DataIn => Bus_Int1_DataIn,
 DataOut => Bus_Int1_DataOut,
 AddrIn => Bus_Int1_AddrIn,
 WE => Bus_Int1_WE,
 RE => Bus_Int1_RE,
 Busy => Bus_Int1_Busy,
 Data => Data,
 Addr => Addr,
 Xrqst => Xrqst,
 XDat => XDat,
 YDat => YDat,
 BusRqst => BusRqst,
 BusCtrl => BusCtrl
);




  ----Registers
  Reg_Proc: process
  begin
        wait until clk'event and clk = '1';
        if rst = '0' then


              HP_Cmd <= (others => '0');
              Err_Type <= (others => '0');
              Vrble_Data<= (others => '0');

              HP_EN <= '0';
              HP_Done_reg_o <= '0';
              --EN_SC_Chk_reg_o <= '0';
              --Stop_SC_Chk <= '0';
              EN_Chk_reg_o <= '0';
              Stop_Chk <= '0';
              EN_Dead_Time_Chk_reg_o <= '0';
              Boot_Done_reg_o <= '0';
              Boot_Wrkn_reg_o <= '0';
              Emu_EN <= '0';
              Error <= '0';
              Val_Start <= '0';
              Bad_FW1 <= '0';
```

```
                    Bad_FW2 <= '0';
                    Bad_FW3 <= '0';

                else

                    if (LD_HP_EN                    = '1') then  HP_EN
                <= Temp_HP_EN;              end if;
                    if (LD_HP_Done                  = '1') then  HP_Done_reg_o
            <= HP_Done;             end if;
                    --if (LD_EN_SC_Chk              = '1') then  EN_SC_Chk_reg_o
            <= EN_SC_Chk;               end if;
                    --if (LD_Stop_SC_Chk     = '1') then  Stop_SC_Chk              <=
Temp_Stop_Chk;       end if;
                    if (LD_EN_Dead_Time_Chk = '1') then  EN_Dead_Time_Chk_reg_o <=
EN_Dead_Time_Chk;  end if;
                    if (LD_Boot_Done            = '1') then  Boot_Done_reg_o
            <= Boot_Done;           end if;
                    if (LD_Boot_Wrkn            = '1') then  Boot_Wrkn_reg_o
            <= Boot_Wrkn;           end if;
                    if (LD_Emu_EN                   = '1') then  Emu_EN
            <= Temp_Emu_EN;             end if;
                    if (LD_HP_Cmd                   = '1') then  HP_Cmd
            <= Temp_HP_Cmd;             end if;
                    if (LD_Error                    = '1') then  Error
            <= Temp_Error;              end if;
                    if (LD_Err_Type             = '1') then  Err_Type
            <= Temp_Err_Type;    end if;
                    if (LD_Vrble_Data           = '1') then  Vrble_Data
<= Temp_Vrble_Data;         end if;
                    if (LD_Val_Start            = '1') then  Val_Start                <=
Temp_Val_Start;       end if;
                    if (LD_Bad_FW1                  = '1') then  Bad_FW1
<= Temp_Bad_FW1;   end if;
                    if (LD_Bad_FW2                  = '1') then  Bad_FW2
<= Temp_Bad_FW2;   end if;
                    if (LD_Bad_FW3                  = '1') then  Bad_FW3
<= Temp_Bad_FW3;   end if;
                    if (LD_EN_Chk               = '1') then  EN_Chk_reg_o
            <= EN_Chk;                  end if;
                    if (LD_Stop_Chk       = '1') then  Stop_Chk                 <=
Temp_Stop_Chk;       end if;


            end if;
            end process;
            ----End Registers
```

```
--	Limit_Exceeded_Emergency_Process : process--(Bad_FW1, Bad_FW2, Bad_FW3)
--	begin
--
--		wait until clk'event and clk = '1';
--		if (Bad_FW1 = '0' and Bad_FW2 = '0' and Bad_FW3 = '0') then
--			eq <= '1';
--		else
--			eq <= '0';
--		end if;
--
--
--
--	end process;


	Bad_FW_Chk : process (CS_Chk, EN_Chk_reg_o, Bad_FW1, Bad_FW2, Bad_FW3)
	begin
		LD_Bad_FW1 <= '0';
		LD_Bad_FW2 <= '0';
		LD_Bad_FW3 <= '0';


		case CS_Chk is
			when S0 =>

				if(EN_Chk_reg_o = '0')then
					NS_Chk <= S0;
				else
					NS_Chk <= S1;
				end if;
				eq <= '1';
				LD_Bad_FW1 <= '1';
				LD_Bad_FW2 <= '1';
				LD_Bad_FW3 <= '1';
			when S1 =>

				if (Bad_FW1 = '0' and Bad_FW2 = '0' and Bad_FW3 = '0') then
					eq <= '1';
					NS_Chk <= S2;
				else
					eq <= '0';

					NS_Chk <= S3;
				end if;
				--LD_Bad_FW1 <= '1';
				--LD_Bad_FW2 <= '1';
```

```
                        --LD_Bad_FW3 <= '1';
                    when S2 =>

                        if (Stop_Chk = '0')then
                            LD_Bad_FW1 <= '1';
                            LD_Bad_FW2 <= '1';
                            LD_Bad_FW3 <= '1';
                            NS_Chk <= S1;
                        else
                            eq <= '1';
                            NS_Chk <= S0;
                        end if;
                        --LD_Bad_FW1 <= '1';
                        --LD_Bad_FW2 <= '1';
                        --LD_Bad_FW3 <= '1';
                    when S3 =>

                        if (Stop_Chk = '0')then
                            eq <= '0';
                            NS_Chk <= S3;
                        else
                            eq <= '1';
                            --LD_Bad_FW1 <= '1';
                            --LD_Bad_FW2 <= '1';
                            --LD_Bad_FW3 <= '1';
                            NS_Chk <= S0;
                        end if;
                    when others =>
                        NS_Chk <= S0;

            end case;
        end process;


        --Short_Circuit : process (CS_ShCrk, Emu_SW01, Emu_SW02, Emu_SW03,
Emu_SW04, Emu_SW05, Emu_SW06)
        --begin

            --Temp_Bad_FW1 <= '0';
            --LD_Bad_FW1 <= '0';



            --case CS_ShCrk is
                --when S0 =>
```

```
                --if((Emu_SW01 = '1') and (Emu_SW02 = '1'))then

                        --NS_ShCrk <= S1;
                --elsif((Emu_SW03 = '1') and (Emu_SW04 = '1'))then

                        --NS_ShCrk <= S1;
                --elsif((Emu_SW05 = '1') and (Emu_SW06 = '1'))then

                        --NS_ShCrk <= S1;
                --else

                        --NS_ShCrk <= S0;
                --end if;

                --Temp_Bad_FW1 <= '0';
                --LD_Bad_FW1 <= '1';

        --when S1 =>
                --if(EN_Chk_reg_o = '1')then
                        --Temp_Bad_FW1 <= '1';
                        --LD_Bad_FW1 <= '1';
                        --NS_ShCrk <= S2;
                --else
                        --Temp_Bad_FW1 <= '0';
                        --LD_Bad_FW1 <= '1';
                        --NS_ShCrk <= S0;
                --end if;

        --when S2 =>
                --if(Stop_Chk = '0')then
                        --Temp_Bad_FW1 <= '1';
                        --LD_Bad_FW1 <= '1';
                        --NS_ShCrk <= S2;
                --else
                        --Temp_Bad_FW1 <= '0';
                        --LD_Bad_FW1 <= '1';
                        --NS_ShCrk <= S0;
                --end if;


        --when others =>
                --NS_ShCrk <= S0;
    --end case;
--end process;
```

```
        Short_Circuit : process (Emu_SW01, Emu_SW02, Emu_SW03, Emu_SW04,
Emu_SW05, Emu_SW06)
    begin

        --if((SW01 = '1') and (SW02 = '1'))then
        if((Emu_SW01='1') and (Emu_SW02 = '1'))then
            Temp_Bad_FW1 <= '1';
        --elsif((SW03 = '1') and (SW04 = '1'))then
        elsif((Emu_SW03='1') and (Emu_SW04 = '1'))then
            Temp_Bad_FW1 <= '1';
        --elsif((SW05 = '1') and (SW06 = '1'))then
        elsif((Emu_SW05='1') and (Emu_SW06 = '1'))then
            Temp_Bad_FW1 <= '1';
        else
            Temp_Bad_FW1 <= '0';
        end if;


    end process;
```

```
------------------------------ DAEAD TIME WORKS ONLY ONE TIME, AND STOPS
DETECTING DEAD TIME AFTER ONE TIME ------------------------------------


--      Dead_Time : process(CS_DeadT, Emu_SW01, Emu_SW02, dt_done)--, EN_Chk_reg_o)
--      begin
--
--              Cnt_DeadT_Rst <= '1';
--              Cnt_DeadT_INC <= '0';
--
--              case CS_DeadT is
--                      when S0 =>
--
--                              Cnt_DeadT_Rst <= '0';
--                              Cnt_DeadT_INC <= '0';
--                              NS_DeadT <= S1;
--
--                      when S1 =>
--                              if Emu_SW01 = '1' then
--                                      NS_DeadT <= S2;
--                              elsif Emu_SW02 = '1' then
--                                      NS_DeadT <= S4;
--                              else
--                                      NS_DeadT <= S1;
--                              end if;
--                              Cnt_DeadT_Rst <= '1';
--
--
--                      when S2 =>
--                              if Emu_SW01 = '0' then
--                                      Cnt_DeadT_INC <= '1';
--                                      NS_DeadT <= S3;
--                              else
--                                      NS_DeadT <= S2;
--                              end if;
--
--
--
--                      when S3 =>
--                              if Emu_SW02 = '1' then
--                                      Temp_Bad_FW2 <= '1';
--
--                                      NS_DeadT <= S0;
--                              elsif dt_done = '1' then
--                                      Temp_Bad_FW2 <= '0';
--
```

```
--                                      NS_DeadT <= S0;
--                              else
--                                      NS_DeadT <= S3;
--                              end if;
--
--
--
--                      when S4 =>
--                              if Emu_SW02 = '0' then
--                                      Cnt_DeadT_INC <= '1';
--                                      NS_DeadT <= S5;
--                              else
--                                      NS_DeadT <= S4;
--                              end if;
--
--
--
--                      when S5 =>
--
--                              if Emu_SW01 = '1' then
--                                      Temp_Bad_FW2 <= '1';
--
--                                      NS_DeadT <= S0;
--                              elsif dt_done = '1' then
--                                      Temp_Bad_FW2 <= '0';
--
--                                      NS_DeadT <= S0;
--                              else
--                                      NS_DeadT <= S5;
--                              end if;
--
--
--                      when others =>
--                              NS_DeadT <= S0;
--                      end case;
--      end process;
--
--      dt_counter_check : process(Cnt_DeadT_Out)
--      begin
--
--              if (Cnt_DeadT_Out = X"03") then
--                      dt_done <= '1';
--              else
--                      dt_done <= '0';
--              end if;
--      end process;
```

```
------------------------------ DAEAD TIME WORKS ONLY ONE TIME, AND STOPS
DETECTING DEAD TIME AFTER ONE TIME ------------------------------------


---- Sync SW signal Inputs before applying to any process ----
SW_Sync: process
begin

        wait until clk'event and clk = '1';
        SW01_t <= Emu_SW01;
        SW01   <=   SW01_t;

        SW02_t <= Emu_SW02;
        SW02   <=   SW02_t;

        SW03_t <= Emu_SW03;
        SW03   <=   SW03_t;

        SW04_t <= Emu_SW04;
        SW04   <=   SW04_t;

        SW05_t <= Emu_SW05;
        SW05   <=   SW05_t;

        SW06_t <= Emu_SW06;
        SW06   <=   SW06_t;
```

```
	end process;


	process(CS, Bus_Int1_Busy, Bus_Int1_DataOut, CntDelay_Out, CntBus_Out,
Cnt_PreChk_Out, Vrble_Data, Val_Start, eq, Boot_Wrkn_reg_o, Boot_Done_reg_o, Bad_FW1,
Bad_FW2, Bad_FW3, Err_Type, HP_Cmd, HP_Done_reg_o)
	begin

		CntBus_Rst <='1';
		CntDelay_Rst <='1';
		CntBus_INC <='0';
		CntDelay_INC <='0';
		Cnt_PreChk_INC <='0';
		Cnt_PreChk_Rst <='1';

		Bus_Int1_AddrIn <= (others => '0');
		Bus_Int1_RE <='0';
		Bus_Int1_DataIn <= (others => '0');
		Bus_Int1_WE <='0';

		Temp_HP_EN <= '0'; --Port
		LD_HP_EN <= '0';

		LD_HP_Done <= '0';  --Port

		Temp_Emu_EN <= '0'; --Port
		LD_Emu_EN <= '0';

		EN_Chk <= '0';
		LD_EN_Chk <= '0';

		EN_Dead_Time_Chk <= '0';
		LD_EN_Dead_Time_Chk <= '0';

		LD_Boot_Done <= '0';    --Port

		LD_Boot_Wrkn <= '0';          --Port

		Temp_Stop_Chk <= '0';
		LD_Stop_Chk <= '0';
		--LD_Stop_SC_Chk <= '0';

		Temp_HP_Cmd <= (others => '0');
		LD_HP_Cmd <= '0';

		Temp_Error <= '0';     --Port
```

```
LD_Error <= '0';

LD_Err_Type   <= '0';
Temp_Err_Type <= (others => '0');

LD_Vrble_Data   <= '0';
Temp_Vrble_Data <= (others => '0');

LD_Val_Start   <= '0';
Temp_Val_Start <= '0';



case CS is

        when S0 =>

                CntBus_INC <='0';
                CntBus_Rst <='0';

                CntDelay_INC <='0';
                CntDelay_Rst <='0';

                Cnt_PreChk_INC <='0';
                Cnt_PreChk_Rst <='0';

                Temp_HP_EN <= '0';
                LD_HP_EN <= '1';

                Temp_Emu_EN <= '0';
                LD_Emu_EN <= '1';

                EN_Chk <= '0';
                LD_EN_Chk <= '1';

                EN_Dead_Time_Chk <= '0';
                LD_EN_Dead_Time_Chk <= '1';

                Temp_Stop_Chk <= '0';
                LD_Stop_Chk <= '1';
                --LD_Stop_SC_Chk <= '1';

                Temp_HP_Cmd <= (others => '0');
                LD_HP_Cmd <= '1';
```

```
                Temp_Error <= '0';
                LD_Error <= '1';

                Temp_Err_Type <= (others => '0');
                LD_Err_Type <= '1';



                NS <= S1;
        when S1=>
                if(CntDelay_Out < 40) then
                        NS<=S1;
                else
                        NS<=S2;
                end if;
                CntDelay_INC<='1';

        when S2=>                                               --Wait
                if(CntBus_Out < 128) then
                        NS<=S2;
                else
                        NS<=S3;
                end if;
                CntBus_INC<='1';

        when S3 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S3;
                else
                        NS <=S4;
                end if;
                CntBus_Rst <='0';               -- Reset Bus Counter
                CntDelay_Rst <='0';
        when S4 =>
                Bus_Int1_AddrIn <= Addr_Validation_Start; --
Addr_Validation_Start is a constant from Common file
                Bus_Int1_RE <='1';
                NS <= S5;

        when S5 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S5;
                else

                        NS <=S6;
                end if;
```

```
                Temp_Vrble_Data <= Bus_Int1_DataOut;
                LD_Vrble_Data <= '1';

        when S6 =>
                Temp_Val_Start <= Vrble_Data(0);
                LD_Val_Start <= '1';
                NS <= S7;

        when S7=>
                Bus_Int1_AddrIn <= Addr_HP_Status; --Addr_HP_Status is a
constant from Common file
                Bus_Int1_DataIn <= X"0000"; -- HP_Stat = 0 (Done/Dissabled)
                Bus_Int1_WE <='1';
                NS <= S8;

        when S8 =>
                if(Val_Start = '1') then
                        NS <= S9;
                else
                        NS <= S0;
                end if;

        when S9=>
                Temp_HP_EN <= '0';
                LD_HP_EN <= '1';

                Temp_Emu_EN <= '1';
                LD_Emu_EN <= '1';

                EN_Chk <= '1';
                LD_EN_Chk <= '1';

                EN_Dead_Time_Chk <= '1';
                LD_EN_Dead_Time_Chk <= '1';

                NS <= S10;


        when S10=>
                if(Cnt_PreChk_Out < X"CAE18")then --Cnt_PreChk is a 20 bit
counter. X"CAE18" is equivalent to 831000 clk cycles which is 30Hz. X"6570C" is equivalent to
415500 clk cycles which is one 60Hz cycle.
                        Cnt_PreChk_INC <= '1';
                        NS <= S10;
                else
                        Cnt_PreChk_INC <= '0';
```

```
                        NS <= S11;
                    end if;
                    Temp_HP_EN <= '0';
                    LD_HP_EN <= '1';

                -- Check for Error. Error signal goes to all Modules
                when S11=>
                    if (eq = '0') then          -- eq = 0 means ERROR in a Chk.
                        Temp_Error <= '1';
                        LD_Error <= '1';
                        NS <= S12;
                    else
                        Temp_Error <= '0';
                        LD_Error <= '1';
------Sa
                        NS <= S27;
                    end if;
                    Cnt_PreChk_Rst <= '0';

        ---------------------------------- Start ERROR Procedure ----------------------------------------
-

                -- Start Bootload Backup if needed. Bootloader ctrl may be able to handle
the situation if it receives the Error signal
                --when S =>









                -- Set the Error Type from Bad_FW# into Err_Type
-----SE
                when S12 =>
                    if(Boot_Wrkn_reg_o = '0')then
                        NS <= S12;
                    else
                        NS <= S13;
                    end if;
                    Temp_Error <= '1';
                    LD_Error <= '1';
                    Temp_HP_EN <= '0';
```

```
                    LD_HP_EN <= '1';
                -- May not need to put '1' on (0) for Backup EN status, and may send
status from Bootloader instead.
                -- This is in case HP_N='1' and only Backup needs to go through without
setting an Error status,
                -- then the Err_Type would be 0 only if I get rid of the '1' on (0).

                    --Temp_Err_Type(0) <= '1';          -- Err_Type needs to be 16
bit signal
                    Temp_Err_Type(0) <= Bad_FW1;
                    Temp_Err_Type(1) <= Bad_FW2;
                    Temp_Err_Type(2) <= Bad_FW3;
                    LD_Err_Type <= '1';

                    LD_Boot_Wrkn <= '1';

                when S13 =>
                    if(Bus_Int1_Busy = '1') then
                        NS <= S13;
                    else

                        NS <=S14;
                    end if;
                -- Set the Error Type RAM Reg
                when S14 =>
                    Bus_Int1_AddrIn <= Addr_ERROR; --Addr_ERROR is a constant
from Common file
                    Bus_Int1_DataIn <= Err_Type;
                    Bus_Int1_WE <='1';
                    NS <= S15;

                when S15 =>
                    if(Bus_Int1_Busy = '1') then
                        NS <= S15;
                    else

                        NS <=S16;
                    end if;

                -- Set the Error bit on HP RAM Reg
                when S16=>
                    Bus_Int1_AddrIn <= Addr_HP_Status; --Addr_HP_Status is a
constant from Common file
                    Bus_Int1_DataIn <= X"0003"; -- HP_Stat = 3 (ERROR)
                    Bus_Int1_WE <='1';
                    NS <= S17;
```

```
when S17 =>
        if(Bus_Int1_Busy = '1') then
                NS <= S17;
        else

                NS <=S18;
        end if;


-- Wait until Bootload is done with Backup
when S18 =>
        if(Boot_Done_reg_o = '0')then
                NS <= S18;
        else
                NS <= S19;
        end if;
        Temp_HP_EN <= '0';
        LD_HP_EN <= '1';
        LD_Boot_Done <= '1';
        EN_Chk <= '0';
        LD_EN_Chk <= '1';
        EN_Dead_Time_Chk <= '0';
        LD_EN_Dead_Time_Chk <= '1';
        Temp_Emu_EN <= '0';
        LD_Emu_EN <= '1';
-- Reset Error to all Modules, Reset Err_Type, and Send Stop_Chk to all
Checks
when S19 =>
        Temp_Error <= '0';
        LD_Error <= '1';
        Temp_Err_Type <= (others => '0');
        LD_Err_Type <= '1';
        Temp_Stop_Chk <= '1';
        LD_Stop_Chk <= '1';

        NS <= S20;


-- After Error is OFF, Reset (Val_Start, HP_Cmds, and HP_Stat) RAM
Regs.
-- Do not reset Clear_ERROR RAM Reg, only User can Reset that.

-- Reset Val_Start RAM Reg
when S20=>
        Bus_Int1_AddrIn <= Addr_Validation_Start; --
Addr_Validation_Start is a constant from Common file
```

```
                Bus_Int1_DataIn <= X"0000";
                Bus_Int1_WE <='1';
                NS <= S21;

        when S21 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S21;
                else

                        NS <=S22;
                end if;
                Temp_Val_Start <= '0';
                LD_Val_Start <= '1';
        -- Reset HP_Cmd RAM Reg
        when S22=>
                Bus_Int1_AddrIn <= Addr_HP_Cmd; --Addr_HP_Cmd is a
constant from Common file
                Bus_Int1_DataIn <= X"0000";
                Bus_Int1_WE <='1';
                NS <= S23;

        when S23 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S23;
                else

                        NS <=S24;
                end if;
                Temp_HP_Cmd <= (others => '0');
                LD_HP_Cmd <= '0';

        -- Reset HP_Stat RAM Reg
        when S24=>
                Bus_Int1_AddrIn <= Addr_HP_Status; --Addr_HP_Status is a
constant from Common file
                Bus_Int1_DataIn <= X"0000"; -- HP_Stat = 0 (Done/Dissabled)
                Bus_Int1_WE <='1';
                NS <= S25;

        when S25 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S25;
                else

                        NS <=S26;
                end if;
```

```
                        -- Check if eq is 0 (ERROR) or 1 (No ERROR) before moving to S0.
                        when S26 =>
                                if(eq = '0')then
                                                                        -- Stop_Chk = 1 will
tell Check modules to restart and reset their Bad_FW signal to 0.
                                        Temp_Stop_Chk <= '1';
                                        LD_Stop_Chk <= '1';
                                        NS <= S26;
                                else
                                        Temp_Stop_Chk <= '0';
                                        LD_Stop_Chk <= '1';
                                        NS <= S0;
                                end if;

        ------------------------------------ End ERROR Procedure ----------------------------------------
--




-----Sa
                        when S27=>
                                if(Bus_Int1_Busy = '1') then
                                        NS <= S27;
                                else

                                        NS <=S28;
                                end if;

                        when S28=>
                                Bus_Int1_AddrIn <= Addr_HP_Status; --Addr_HP_Status is a
constant from Common file
                                Bus_Int1_DataIn <= X"0001"; -- HP_Stat = 1 (Ready)
                                Bus_Int1_WE <='1';
                                NS <= S29;

                        when S29 =>
                                if(Bus_Int1_Busy = '1') then
                                        NS <= S29;
                                else

                                        NS <=S30;
                                end if;
```

```
                when S30 =>
                        Bus_Int1_AddrIn <= Addr_HP_Cmd; --Addr_HP_Cmd is a
constant from Common file
                        Bus_Int1_RE <='1';
                        NS <= S31;

                when S31 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S31;
                        else

                                NS <=S32;
                        end if;
                        Temp_Vrble_Data <= Bus_Int1_DataOut;
                        LD_Vrble_Data <= '1';

                -- Check for Error. Error signal goes to all Modules
                when S32=>
                        if (eq = '0') then          -- eq = 0 means ERROR in a Chk.
                                Temp_Error <= '1';
                                LD_Error <= '1';
------SE
                                NS <= S12;
                        else
                                Temp_Error <= '0';
                                LD_Error <= '1';
                                NS <= S33;
                        end if;


                when S33 =>
                        Temp_HP_Cmd <= Vrble_Data;
                        LD_HP_Cmd <= '1';
                        NS <= S34;

                when S34 =>
                        NS <= S35;

                when S35 =>
                        if(HP_Cmd > X"0000")then
                                NS <= S36;
                        else
------Sa
                                NS <= S27;
                        end if;
                        Temp_HP_EN <= '0';
```

```
                    LD_HP_EN <= '1';

                -- Check for Error. Error signal goes to all Modules
                when S36=>
                    if (eq = '0') then          -- eq = 0 means ERROR in a Chk.
                        Temp_Error <= '1';
                        LD_Error <= '1';
------SE
                        NS <= S12;
                    else
                        Temp_Error <= '0';
                        LD_Error <= '1';
                        NS <= S37;
                    end if;
                    Temp_HP_EN <= '0';
                    LD_HP_EN <= '1';

                when S37=>
                    if(HP_Cmd = X"0001")then   -- HP_Cmd = 1 (HP_Y)
                        Temp_Error <= '0';
                        LD_Error <= '1';
                        NS <= S38;
                    else
                                                                                --
HP_Cmd = 2 (HP_N), but HP_Cmd was seen to be > 0 in state above,
                    --elsif(HP_Cmd = X"0002")then       -- so here if HP_Cmd is not =
1, then it will be an Error regardless.
                        Temp_Error <= '1';
                        LD_Error <= '1';
------SE
                        NS <= S12;
                    end if;
                    Temp_HP_EN <= '0';
                    LD_HP_EN <= '1';

                -- Check for Error. Error signal goes to all Modules
                when S38=>
                    if (eq = '0') then          -- eq = 0 means ERROR in a Chk.
                        Temp_Error <= '1';
                        LD_Error <= '1';
------SE
                        NS <= S12;
                    else
                        Temp_Error <= '0';
                        LD_Error <= '1';
                        NS <= S39;
```

```
            end if;
            Temp_HP_EN <= '0';
            LD_HP_EN <= '1';

        when S39=>
            Temp_HP_EN <= '0';
            LD_HP_EN <= '1';

            Temp_Emu_EN <= '0';
            LD_Emu_EN <= '1';

            EN_Chk <= '0';
            LD_EN_Chk <= '1';

            EN_Dead_Time_Chk <= '0';
            LD_EN_Dead_Time_Chk <= '1';

            Temp_Stop_Chk <= '1';
            LD_Stop_Chk <= '1';
            --LD_Stop_SC_Chk <= '1';

            NS <= S40;

        when S40 =>
            Temp_HP_EN <= '1';
            LD_HP_EN <= '1';
            LD_HP_Done <= '1';
            NS <= S41;



        when S41 =>
            LD_HP_Done <= '1';
            NS <= S42;

        when S42 =>
            if (HP_Done_reg_o = '0')then
                Temp_HP_EN <= '1';
                NS <= S42;
            else
                Temp_HP_EN <= '0';
                NS <= S43;
            end if;
            LD_HP_EN <= '1';
            LD_HP_Done <= '1';
```

```
                -- Reset HP_Stat RAM Reg

                when S43 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S43;
                        else

                                NS <=S44;
                        end if;

                when S44=>
                        Bus_Int1_AddrIn <= Addr_HP_Status; --Addr_HP_Status is a
constant from Common file
                        Bus_Int1_DataIn <= X"0000"; -- HP_Stat = 0 (Done/Dissabled)
                        Bus_Int1_WE <='1';
                        NS <= S45;

                when S45 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S45;
                        else

                                NS <=S46;
                        end if;

                -- Reset Val_Start RAM Reg
                when S46=>
                        Bus_Int1_AddrIn <= Addr_Validation_Start; --
Addr_Validation_Start is a constant from Common file
                        Bus_Int1_DataIn <= X"0000";
                        Bus_Int1_WE <='1';
                        NS <= S47;

                when S47 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S47;
                        else

                                NS <=S48;
                        end if;
                        Temp_Val_Start <= '0';
                        LD_Val_Start <= '1';

                -- Reset HP_Cmd RAM Reg
                when S48=>
```

```
                Bus_Int1_AddrIn <= Addr_HP_Cmd; --Addr_HP_Cmd is a
constant from Common file
                Bus_Int1_DataIn <= X"0000";
                Bus_Int1_WE <='1';
                NS <= S49;

            when S49 =>
                if(Bus_Int1_Busy = '1') then
                    NS <= S49;
                else

                    NS <=S50;
                    --NS <= S0;
                end if;
                Temp_HP_Cmd <= (others => '0');
                LD_HP_Cmd <= '0';




            -- Reset ERROR RAM Reg
            when S50 =>
                Temp_Err_Type <= (others => '0');
                --Temp_Err_Type(1) <= Bad_FW2;
                --Temp_Err_Type(2) <= Bad_FW3;
                LD_Err_Type <= '1';
                NS <= S51;

            when S51 =>
                if(Bus_Int1_Busy = '1') then
                    NS <= S51;
                else

                    NS <=S52;
                end if;


            -- Set the Error Type RAM Reg
            when S52 =>
                Bus_Int1_AddrIn <= Addr_ERROR; --Addr_ERROR is a constant
from Common file
                Bus_Int1_DataIn <= Err_Type;
                Bus_Int1_WE <='1';
                NS <= S53;
```

```
                when S53 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S53;
                        else

                                NS <=S0;
                        end if;



                when others =>
                        NS <= S0;


        end case;
    end process;




    ----State Sync
    sync_States: process
    begin
        wait until clk'event and clk = '1';
        if rst = '0' then
                CS <= S0;
                CS_Chk <= S0;
                CS_ShCrk <= S0;
                CS_DeadT <= S0;
        else
                CS <= NS;
                CS_Chk <= NS_Chk;
                CS_ShCrk <= NS_ShCrk;
                CS_DeadT <= NS_DeadT;
        end if;
    end process;
    ----End State Sync


end Behavioral;
```

A-2-5 Emulation

```
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Estefano Soria
--
-- Create Date:                4Jan2021
-- Design Name:                DSP_Short_Circuit_Validation
-- Module Name:                DSP_Short_Circuit_Validation - Behavioral
-- Project Name:               DSP_Short_Circuit_Validation
-- Target Devices:             LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:              Lattice Diamond_x64 Build 3.11
-- Description:
--
---- PinOut:
--
-- Revision
--
--
--
-- Additional Comments:
--
--
----------------------------------------------------------------------------------


Library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library machxo2;
use machxo2.all;

library work;
use work.Test2_DT_Common.all;

entity Test2_DT_Emu_Ctrl is
    Port (
                                clk : in STD_LOGIC;
                                rst : in STD_LOGIC;

                                Emu_EN : in std_logic;

                                Data : INOUT  std_logic_vector(15 downto 0);
                                Addr : OUT  std_logic_vector(15 downto 0);
                                Xrqst : OUT  std_logic;
                                XDat : IN  std_logic;
                                YDat : OUT  std_logic;
```

```
                        BusRqst : OUT  std_logic;
                        BusCtrl : IN  std_logic;

                        Emu_SW01 : in  STD_LOGIC;
                        Emu_SW02 : in  STD_LOGIC;
                        Emu_SW03 : in  STD_LOGIC;
                        Emu_SW04 : in  STD_LOGIC;
                        Emu_SW05 : in  STD_LOGIC;
                        Emu_SW06 : in  STD_LOGIC;

                        Error : in STD_LOGIC;
                        HP_EN : in STD_LOGIC

                        --** Missing Ports for Bus Interface, and some others **


                        );
end Test2_DT_Emu_Ctrl;

architecture Behavioral of Test2_DT_Emu_Ctrl is

---------------------------------------------------------- START SIGNAL AND COMPONENT
DECLARATIONS-----------------------------------------------------------

        type state_type is
(S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,S20,S21,S22,S23,
S24,S25,S26,S27,S28,S29,S30,S31,S32,S33,S34,S35,S36,S37,S38,S39,S40,S41,S42,S43,S44,S4
5,S46,S47,S48,S49,S50,S51,S52,S53,S54,S55,S56,S57,S58,S59,S60,S61,S62,S63,S64,S65,S66,
S67);
        signal CS, NS, CSA, NSA, CSB, NSB, CSC, NSC, CS_Fsw, NS_Fsw, CSab, NSab,
CSbc, NSbc, CSca, NSca : state_type;



        ---------------------------- Reference -----------------------------

        --signal   LD_          : std_logic := '0';
        --signal            _reg_o       : std_logic_vector ( downto 0) := (others => '0');
        --signal Temp_                    : std_logic_vector ( downto 0) := (others => '0');
        --signal                                  : std_logic_vector ( downto 0) := (others => '0');
        --
        --
        --signal   LD_          : std_logic := '0';
        --signal            _reg_o       : std_logic_vector ( downto 0) := (others => '0');
        --signal                                  : std_logic_vector ( downto 0) := (others => '0');
        --
```

```
        --signal   LD_            : std_logic := '0';
        --signal Temp_                    : std_logic_vector ( downto 0) := (others => '0');
        --signal                          : std_logic_vector ( downto 0) := (others => '0');
        --
        --
        --
        --signal   : std_logic := '0';
        --signal   : std_logic_vector ( downto 0) := (others => '0');
        --
        --
        --signal Cnt_INC : std_logic := '0';
        --signal Cnt_Rst : std_logic := '0';
        --signal Cnt_Out : std_logic_vector( downto 0) := (others => '0');


----------------------------------------Other Signals----------------------------------------


        signal SW01_t : std_logic := '0';
        signal SW01 : std_logic := '0';

        signal SW03_t : std_logic := '0';
        signal SW03 : std_logic := '0';

        signal SW05_t : std_logic := '0';
        signal SW05 : std_logic := '0';

        signal EN : std_logic := '0';

        --Bus Interface Signals
        signal  Bus_Int1_WE : std_logic := '0';
        signal  Bus_Int1_RE : std_logic := '0';
        signal  Bus_Int1_Busy : std_logic := '0';
        signal  Bus_Int1_AddrIn : std_logic_vector (15 downto 0) := (others => '0');
        signal  Bus_Int1_DataIn : std_logic_vector (15 downto 0) := (others => '0');
        signal  Bus_Int1_DataOut : std_logic_vector (15 downto 0) := (others => '0');

        -- Va FIFO Signals
        signal  STD_FIFO_Va_Full : std_logic := '0';
        signal  STD_FIFO_Va_Empty : std_logic := '0';
        signal  STD_FIFO_Va_WriteEn  : std_logic := '0';
        signal  STD_FIFO_Va_ReadEn : std_logic := '0';
        signal  STD_FIFO_Va_DataIn : std_logic_vector (15 downto 0) := (others => '0');
        signal  STD_FIFO_Va_DataOut  : std_logic_vector (15 downto 0) := (others => '0');

        -- Vb FIFO Signals
```

```
signal  STD_FIFO_Vb_Full : std_logic := '0';
signal  STD_FIFO_Vb_Empty : std_logic := '0';
signal  STD_FIFO_Vb_WriteEn  : std_logic := '0';
signal  STD_FIFO_Vb_ReadEn : std_logic := '0';
signal  STD_FIFO_Vb_DataIn : std_logic_vector (15 downto 0) := (others => '0');
signal  STD_FIFO_Vb_DataOut  : std_logic_vector (15 downto 0) := (others => '0');

-- Vc FIFO Signals
signal  STD_FIFO_Vc_Full : std_logic := '0';
signal  STD_FIFO_Vc_Empty : std_logic := '0';
signal  STD_FIFO_Vc_WriteEn  : std_logic := '0';
signal  STD_FIFO_Vc_ReadEn : std_logic := '0';
signal  STD_FIFO_Vc_DataIn : std_logic_vector (15 downto 0) := (others => '0');
signal  STD_FIFO_Vc_DataOut  : std_logic_vector (15 downto 0) := (others => '0');

------------------------------------------- Counters -----------------------------------------


---------------------- Freq Calculation Counters ----------------------

--Fsw and 1%Ton Counters
--16 bit
signal CntTsw_INC : std_logic := '0';
signal CntTsw_Rst : std_logic := '0';
signal CntTsw_Out : std_logic_vector(15 downto 0) := (others => '0');
--16 bit
signal CntTon_INC : std_logic := '0';
signal CntTon_Rst : std_logic := '0';
signal CntTon_Out : std_logic_vector(15 downto 0) := (others => '0');



---------- VA, VB, VC Duty Cycle % Calculation Counters --------------

-- A Phase Ton % Counters
--8 bit
signal CntS1on_INC : std_logic := '0';
signal CntS1on_Rst : std_logic := '0';
signal CntS1on_Out : std_logic_vector(7 downto 0) := (others => '0');
--16 bit
signal CntA1_INC : std_logic := '0';
signal CntA1_Rst : std_logic := '0';
signal CntA1_Out : std_logic_vector(15 downto 0) := (others => '0');
--16 bit
```

```
signal CntA2_INC : std_logic := '0';
signal CntA2_Rst : std_logic := '0';
signal CntA2_Out : std_logic_vector(15 downto 0) := (others => '0');


-- B Phase Ton % Counters
--8 bit
signal CntS3on_INC : std_logic := '0';
signal CntS3on_Rst : std_logic := '0';
signal CntS3on_Out : std_logic_vector(7 downto 0) := (others => '0');
--16 bit
signal CntB1_INC : std_logic := '0';
signal CntB1_Rst : std_logic := '0';
signal CntB1_Out : std_logic_vector(15 downto 0) := (others => '0');
--16 bit
signal CntB2_INC : std_logic := '0';
signal CntB2_Rst : std_logic := '0';
signal CntB2_Out : std_logic_vector(15 downto 0) := (others => '0');


-- C Phase Ton % Counters
--8 bit
signal CntS5on_INC : std_logic := '0';
signal CntS5on_Rst : std_logic := '0';
signal CntS5on_Out : std_logic_vector(7 downto 0) := (others => '0');
--16 bit
signal CntC1_INC : std_logic := '0';
signal CntC1_Rst : std_logic := '0';
signal CntC1_Out : std_logic_vector(15 downto 0) := (others => '0');
--16 bit
signal CntC2_INC : std_logic := '0';
signal CntC2_Rst : std_logic := '0';
signal CntC2_Out : std_logic_vector(15 downto 0) := (others => '0');




---------------------- Data Distribution Counters -------------------

-- Bus Counter Delay
-- 8 bit
signal CntBus_INC : std_logic := '0';
signal CntBus_Rst : std_logic := '0';
signal CntBus_Out : std_logic_vector(7 downto 0) := (others => '0');

-- Start Data Traffic Counter Delay
```

```
	-- 8 bit
	signal CntDelay_INC : std_logic := '0';
	signal CntDelay_Rst : std_logic := '0';
	signal CntDelay_Out : std_logic_vector(7 downto 0) := (others => '0');

	-- 192 FIFO Reg Counter to Save Emu Data
	-- 8 bit
	signal Cnt_LeadReg_INC : std_logic := '0';
	signal Cnt_LeadReg_Rst : std_logic := '0';
	signal Cnt_LeadReg_Out : std_logic_vector(7 downto 0) := (others => '0');

	-- 192 Reg Counter to Save Emu Data from FIFO to RAM
	--8 bit
	signal Cnt_FollowReg_INC : std_logic := '0';
	signal Cnt_FollowReg_Rst : std_logic := '0';
	signal Cnt_FollowReg_Out : std_logic_vector(7 downto 0) := (others => '0');

	-- PreScale Counter
	-- 16 bit
	signal Cnt_Scale_INC : std_logic := '0';
	signal Cnt_Scale_Rst : std_logic := '0';
	signal Cnt_Scale_Out : std_logic_vector(15 downto 0) := (others => '0');




---------------------------------------- Registers ----------------------------------------


   ------------------------------ Freq Calculations ------------------------------

	-- Div and Var work together to divide Tsw/100; round to nearest whole number.
	signal   LD_Div                  : std_logic := '0';
	signal      Div_reg_o   : std_logic_vector (15 downto 0) := (others => '0');
	signal            Div                        : std_logic_vector (15 downto 0) := (others => '0');

	signal   LD_Var                  : std_logic := '0';
	signal      Var_reg_o   : std_logic_vector (15 downto 0) := (others => '0');
	signal            Var                        : std_logic_vector (15 downto 0) := (others => '0');


	-- Ref_Ton = Tsw/100, which will be the 1% Ton reference
	signal   LD_Ref_Ton  : std_logic := '0';
	signal Temp_Ref_Ton          : std_logic_vector (15 downto 0) := (others => '0');
	signal            Ref_Ton                  : std_logic_vector (15 downto 0) := (others => '0');
```

```
        -- Tsw is the # of clk cycles from the measured switching period
        signal   LD_Tsw                 : std_logic := '0';
        signal      Tsw_reg_o  : std_logic_vector (15 downto 0) := (others => '0');
        signal            Tsw                       : std_logic_vector (15 downto 0) := (others => '0');


        -- Dec is the calculation of the Tsw/100 rounding.
        signal   LD_Dec                 : std_logic := '0';
        signal      Dec_reg_o  : std_logic_vector (15 downto 0) := (others => '0');
        signal            Dec                       : std_logic_vector (15 downto 0) := (others => '0');



    --------------------------------VAN %DC-------------------------------------
        -- Van Duty Cycle
        signal   LD_Van_DC          : std_logic := '0';
        signal Temp_Van_DC                  : std_logic_vector (7 downto 0) := (others => '0');
        signal            Van_DC                    : std_logic_vector (7 downto 0) := (others => '0');


    --------------------------------VBN %DC-------------------------------------
        -- Vbn Duty Cycle
        signal   LD_Vbn_DC          : std_logic := '0';
        signal Temp_Vbn_DC                  : std_logic_vector (7 downto 0) := (others => '0');
        signal            Vbn_DC                    : std_logic_vector (7 downto 0) := (others => '0');


    --------------------------------VCN %DC-------------------------------------
        -- Vcn Duty Cycle
        signal   LD_Vcn_DC          : std_logic := '0';
        signal Temp_Vcn_DC                  : std_logic_vector (7 downto 0) := (others => '0');
        signal            Vcn_DC                    : std_logic_vector (7 downto 0) := (others => '0');



        -------------------------------VA %DC-------------------------------------
        -- Va Duty Cycle
        signal   LD_Va_DC           : std_logic := '0';
        --signal Temp_Va_DC                 : std_logic_vector (23 downto 0) := (others => '0');
        --signal                  Va_DC                     : std_logic_vector (23 downto 0) := (others
=> '0');
        signal Temp_Va_DC           : std_logic_vector (15 downto 0) := (others => '0');
        signal            Va_DC                     : std_logic_vector (15 downto 0) := (others => '0');
```

```
    ---------------------------------VB %DC-------------------------------------
        -- Vb Duty Cycle
        signal   LD_Vb_DC           : std_logic := '0';
        signal Temp_Vb_DC           : std_logic_vector (15 downto 0) := (others => '0');
        signal          Vb_DC               : std_logic_vector (15 downto 0) := (others => '0');


    ---------------------------------VC %DC-------------------------------------
        -- Vc Duty Cycle
        signal   LD_Vc_DC           : std_logic := '0';
        signal Temp_Vc_DC           : std_logic_vector (15 downto 0) := (others => '0');
        signal          Vc_DC               : std_logic_vector (15 downto 0) := (others => '0');




        ---------------------------------VAB %DC-------------------------------------
        -- Vab Duty Cycle
        signal   LD_Vab_DC          : std_logic := '0';
        --signal Temp_Vab_DC                : std_logic_vector (23 downto 0) := (others => '0');
        --signal                Vab_DC                : std_logic_vector (23 downto 0) := (others
=> '0');
        signal Temp_Vab_DC                : std_logic_vector (15 downto 0) := (others => '0');
        signal          Vab_DC              : std_logic_vector (15 downto 0) := (others => '0');
        ---------------------------------VBC %DC-------------------------------------
        -- Vbc Duty Cycle
        signal   LD_Vbc_DC          : std_logic := '0';
        signal Temp_Vbc_DC                : std_logic_vector (15 downto 0) := (others => '0');
        signal          Vbc_DC              : std_logic_vector (15 downto 0) := (others => '0');

        ---------------------------------VCA %DC-------------------------------------
        -- Vca Duty Cycle
        signal   LD_Vca_DC          : std_logic := '0';
        signal Temp_Vca_DC                : std_logic_vector (15 downto 0) := (others => '0');
        signal          Vca_DC              : std_logic_vector (15 downto 0) := (others => '0');




        ---------------------------------- VAB -------------------------------------
        -- Vab
        signal   LD_Vab             : std_logic := '0';
        signal Temp_Vab             : std_logic_vector (11 downto 0) := (others => '0');
        signal          Vab                 : std_logic_vector (11 downto 0) := (others => '0');
```

```
-------------------------------- VBC ------------------------------------
-- Vbc
signal   LD_Vbc              : std_logic := '0';
signal Temp_Vbc              : std_logic_vector (11 downto 0) := (others => '0');
signal          Vbc                  : std_logic_vector (11 downto 0) := (others => '0');

-------------------------------- VCA ------------------------------------
-- Vca
signal   LD_Vca              : std_logic := '0';
signal Temp_Vca              : std_logic_vector (11 downto 0) := (others => '0');
signal          Vca                  : std_logic_vector (11 downto 0) := (others => '0');



------------------- V1-V2=Vab, V3-V4=Vbc, V5-V6=Vca ------------------------
-- V1=Van
signal   LD_V1_Dat           : std_logic := '0';
signal Temp_V1_Dat           : std_logic_vector (7 downto 0) := (others => '0');
signal          V1_Dat               : std_logic_vector (7 downto 0) := (others => '0');

-- V2=Vbn
signal   LD_V2_Dat           : std_logic := '0';
signal Temp_V2_Dat           : std_logic_vector (7 downto 0) := (others => '0');
signal          V2_Dat               : std_logic_vector (7 downto 0) := (others => '0');

-- V3=Vbn
signal   LD_V3_Dat           : std_logic := '0';
signal Temp_V3_Dat           : std_logic_vector (7 downto 0) := (others => '0');
signal          V3_Dat               : std_logic_vector (7 downto 0) := (others => '0');

-- V4=Vcn
signal   LD_V4_Dat           : std_logic := '0';
signal Temp_V4_Dat           : std_logic_vector (7 downto 0) := (others => '0');
signal          V4_Dat               : std_logic_vector (7 downto 0) := (others => '0');

-- V5=Vcn
signal   LD_V5_Dat           : std_logic := '0';
signal Temp_V5_Dat           : std_logic_vector (7 downto 0) := (others => '0');
signal          V5_Dat               : std_logic_vector (7 downto 0) := (others => '0');

-- V6=Van
signal   LD_V6_Dat           : std_logic := '0';
signal Temp_V6_Dat           : std_logic_vector (7 downto 0) := (others => '0');
signal          V6_Dat               : std_logic_vector (7 downto 0) := (others => '0');
```

```
---------------------------- Data Distribution ---------------------------------

-- Variable Data Register
signal   LD_Vrble_Data              : std_logic := '0';
signal Temp_Vrble_Data              : std_logic_vector (15 downto 0) := (others => '0');
signal          Vrble_Data          : std_logic_vector (15 downto 0) := (others => '0');


-- Start Emu DataLogging Register
signal   LD_Emu_DL_Start   : std_logic := '0';
signal Temp_Emu_DL_Start  : std_logic := '0';
signal          Emu_DL_Start        : std_logic := '0';


-- PreScale Register
signal   LD_PreScale          : std_logic := '0';
signal Temp_PreScale                : std_logic_vector (15 downto 0) := (others => '0');
signal          PreScale            : std_logic_vector (15 downto 0) := (others => '0');


-- Trigger Register
signal   LD_Trigger           : std_logic := '0';
signal Temp_Trigger           : std_logic_vector (15 downto 0) := (others => '0');
signal          Trigger             : std_logic_vector (15 downto 0) := (others => '0');


-- Emu Output Type (VLL or VLN) Register
signal   LD_Emu_V_Type              : std_logic := '0';
signal Temp_Emu_V_Type              : std_logic_vector (7 downto 0) := (others => '0');
signal          Emu_V_Type          : std_logic_vector (7 downto 0) := (others => '0');


-- Scale Ref Register Used for Scale Counter (Latched from PreScale Reg)
signal   LD_Scale_Ref               : std_logic := '0';
signal Temp_Scale_Ref               : std_logic_vector (15 downto 0) := (others => '0');
signal          Scale_Ref           : std_logic_vector (15 downto 0) := (others => '0');


-- Emu Va, Vb, Vc Sampling Registers (Sample based on Scale Counter)
signal   LD_Va_Samp                 : std_logic := '0';
signal Temp_Va_Samp                 : std_logic_vector (23 downto 0) := (others => '0');
signal          Va_Samp                     : std_logic_vector (23 downto 0) := (others
=> '0');

signal   LD_Vb_Samp                 : std_logic := '0';
signal Temp_Vb_Samp                 : std_logic_vector (23 downto 0) := (others => '0');
```

```
	signal		Vb_Samp			: std_logic_vector (23 downto 0) := (others
=> '0');

	signal	LD_Vc_Samp		: std_logic := '0';
	signal Temp_Vc_Samp		: std_logic_vector (23 downto 0) := (others => '0');
	signal		Vc_Samp			: std_logic_vector (23 downto 0) := (others
=> '0');


	-- Emu Va, Vb, Vc RAM Starting Address Latched from Common Constants
	signal	LD_Addr_Va_Start	: std_logic := '0';
	signal Temp_Addr_Va_Start	: std_logic_vector (15 downto 0) := (others => '0');
	signal		Addr_Va_Start		: std_logic_vector (15 downto 0) := (others
=> '0');

	signal	LD_Addr_Vb_Start	: std_logic := '0';
	signal Temp_Addr_Vb_Start	: std_logic_vector (15 downto 0) := (others => '0');
	signal		Addr_Vb_Start		: std_logic_vector (15 downto 0) := (others
=> '0');

	signal	LD_Addr_Vc_Start	: std_logic := '0';
	signal Temp_Addr_Vc_Start	: std_logic_vector (15 downto 0) := (others => '0');
	signal		Addr_Vc_Start		: std_logic_vector (15 downto 0) := (others
=> '0');

	-- Process EN Registers
	signal	LD_EN		: std_logic := '0';
	signal Temp_EN		: std_logic := '0';
	--signal		EN		: std_logic := '0';

------------------- Component Declarations (FIFO, Bus_Int, Counters) ----------------------------

	--declare STD_FIFO
	COMPONENT STD_FIFO
	Generic (
		DATA_WIDTH		: integer;		-- Width of FIFO
		FIFO_DEPTH		: integer;		--	Depth of FIFO
		FIFO_ADDR_LEN	: integer		-- Required number of bits to represent
FIFO_Depth
	);
	Port (
		CLK	: in  STD_LOGIC;
		RST	: in  STD_LOGIC;
		WriteEn : in  STD_LOGIC;
		DataIn  : in  STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0);
		ReadEn  : in  STD_LOGIC;
```

```
            DataOut : out STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0);
            Empty   : out STD_LOGIC;
            Full    : out STD_LOGIC
        );
        end COMPONENT;

        --declare Bus Interface
        COMPONENT Bus_Int
  PORT(
            clk : IN  std_logic;
            rst : IN  std_logic;
            DataIn : IN  std_logic_vector(15 downto 0);
            DataOut : OUT  std_logic_vector(15 downto 0);
            AddrIn : IN  std_logic_vector(15 downto 0);
            WE : IN  std_logic;
            RE : IN  std_logic;
            Busy : OUT  std_logic;
            Data : INOUT  std_logic_vector(15 downto 0);
            Addr : OUT  std_logic_vector(15 downto 0);
            Xrqst : OUT  std_logic;
            XDat : IN  std_logic;
            YDat : OUT  std_logic;
            BusRqst : OUT  std_logic;
            BusCtrl : IN  std_logic
  );
   END COMPONENT;


        --Declare Counter Component
        component Std_Counter
        generic
        (
            Width : integer                 --width of counter
        );
        port(INC,rst,clk: in std_logic;
             Count: out STD_LOGIC_VECTOR(Width-1 downto 0));
        end component;


---------------------------------------------------------- END SIGNAL AND COMPONENT
DECLARATIONS------------------------------------------------------------

begin


        --Instantiate STD_FIFO for Va
```

```
STD_FIFO_Va: STD_FIFO
Generic Map
(
        DATA_WIDTH      => 16,          -- Width of FIFO
        FIFO_DEPTH      => 200,         --      Depth of FIFO
        FIFO_ADDR_LEN => 9      -- Required number of bits to represent FIFO_Depth
)
Port Map
(
        CLK => clk,
        RST => rst,
        WriteEn => STD_FIFO_Va_WriteEn,
        DataIn  => STD_FIFO_Va_DataIn,
        ReadEn  => STD_FIFO_Va_ReadEn,
        DataOut => STD_FIFO_Va_DataOut,
        Empty   => STD_FIFO_Va_Empty,
        Full    => STD_FIFO_Va_Full
        );

--Instantiate STD_FIFO for Vb
STD_FIFO_Vb: STD_FIFO
Generic Map
(
        DATA_WIDTH      => 16,          -- Width of FIFO
        FIFO_DEPTH      => 200,         --      Depth of FIFO
        FIFO_ADDR_LEN => 9      -- Required number of bits to represent FIFO_Depth
)
Port Map
(
        CLK => clk,
        RST => rst,
        WriteEn => STD_FIFO_Vb_WriteEn,
        DataIn  => STD_FIFO_Vb_DataIn,
        ReadEn  => STD_FIFO_Vb_ReadEn,
        DataOut => STD_FIFO_Vb_DataOut,
        Empty   => STD_FIFO_Vb_Empty,
        Full    => STD_FIFO_Vb_Full
        );


--Instantiate STD_FIFO for Vc
STD_FIFO_Vc: STD_FIFO
Generic Map
(
        DATA_WIDTH      => 16,          -- Width of FIFO
        FIFO_DEPTH      => 200,         --      Depth of FIFO
```

```
        FIFO_ADDR_LEN => 9       -- Required number of bits to represent FIFO_Depth
    )
    Port Map
    (
        CLK => clk,
        RST => rst,
        WriteEn => STD_FIFO_Vc_WriteEn,
        DataIn  => STD_FIFO_Vc_DataIn,
        ReadEn  => STD_FIFO_Vc_ReadEn,
        DataOut => STD_FIFO_Vc_DataOut,
        Empty   => STD_FIFO_Vc_Empty,
        Full    => STD_FIFO_Vc_Full
        );



   --Instantiate Bus Interface
   Bus_Int1: Bus_Int PORT MAP (
  clk => clk,
  rst => rst,
  DataIn => Bus_Int1_DataIn,
  DataOut => Bus_Int1_DataOut,
  AddrIn => Bus_Int1_AddrIn,
  WE => Bus_Int1_WE,
  RE => Bus_Int1_RE,
  Busy => Bus_Int1_Busy,
  Data => Data,
  Addr => Addr,
  Xrqst => Xrqst,
  XDat => XDat,
  YDat => YDat,
  BusRqst => BusRqst,
  BusCtrl => BusCtrl
);



   --Inst Counter for
   CounterTsw: Std_Counter
   generic map
   (
        Width => 16
   )
   port map(
        INC     => CntTsw_INC,
        rst     => CntTsw_Rst,
```

```
        clk     => clk,
        Count   => CntTsw_Out
        );



--Inst Counter for
CounterTon: Std_Counter
generic map
(
        Width => 16
)
port map(
        INC     => CntTon_INC,
        rst     => CntTon_Rst,
        clk     => clk,
        Count   => CntTon_Out
        );



--Inst Counter for
CounterS1on: Std_Counter
generic map
(
        Width => 8
)
port map(
        INC     => CntS1on_INC,
        rst     => CntS1on_Rst,
        clk     => clk,
        Count   => CntS1on_Out
        );



--Inst Counter for
CounterA1: Std_Counter
generic map
(
```

```
        Width => 16
)
port map(
        INC     => CntA1_INC,
        rst     => CntA1_Rst,
        clk     => clk,
        Count   => CntA1_Out
        );



--Inst Counter for
CounterA2: Std_Counter
generic map
(
        Width => 16
)
port map(
        INC     => CntA2_INC,
        rst     => CntA2_Rst,
        clk     => clk,
        Count   => CntA2_Out
        );



--Inst Counter for
CounterS3on: Std_Counter
generic map
(
        Width => 8
)
port map(
        INC     => CntS3on_INC,
        rst     => CntS3on_Rst,
        clk     => clk,
        Count   => CntS3on_Out
        );
```

```
--Inst Counter for
CounterB1: Std_Counter
generic map
(
        Width => 16
)
port map(
        INC     => CntB1_INC,
        rst     => CntB1_Rst,
        clk     => clk,
        Count   => CntB1_Out
        );



--Inst Counter for
CounterB2: Std_Counter
generic map
(
        Width => 16
)
port map(
        INC     => CntB2_INC,
        rst     => CntB2_Rst,
        clk     => clk,
        Count   => CntB2_Out
        );



--Inst Counter for
CounterS5on: Std_Counter
generic map
(
        Width => 8
)
port map(
        INC     => CntS5on_INC,
        rst     => CntS5on_Rst,
        clk     => clk,
        Count   => CntS5on_Out
        );
```

```
--Inst Counter for
CounterC1: Std_Counter
generic map
(
        Width => 16
)
port map(
        INC     => CntC1_INC,
        rst     => CntC1_Rst,
        clk     => clk,
        Count   => CntC1_Out
        );



--Inst Counter for
CounterC2: Std_Counter
generic map
(
        Width => 16
)
port map(
        INC     => CntC2_INC,
        rst     => CntC2_Rst,
        clk     => clk,
        Count   => CntC2_Out
        );



--Inst Counter for
CounterBus: Std_Counter
generic map
(
        Width => 8
)
port map(
        INC     => CntBus_INC,
        rst     => CntBus_Rst,
        clk     => clk,
        Count   => CntBus_Out
        );
```

```
--Inst Counter for
CounterDelay: Std_Counter
generic map
(
        Width => 8
)
port map(
        INC     => CntDelay_INC,
        rst     => CntDelay_Rst,
        clk     => clk,
        Count   => CntDelay_Out
        );



--Inst Counter for
Counter_LeadReg: Std_Counter
generic map
(
        Width => 8
)
port map(
        INC     => Cnt_LeadReg_INC,
        rst     => Cnt_LeadReg_Rst,
        clk     => clk,
        Count   => Cnt_LeadReg_Out
        );



--Inst Counter for
Counter_Scale: Std_Counter
generic map
(
        Width => 16
)
port map(
        INC     => Cnt_Scale_INC,
        rst     => Cnt_Scale_Rst,
        clk     => clk,
        Count   => Cnt_Scale_Out
        );
```

```
--Inst Counter for
Counter_FollowReg: Std_Counter
generic map
(
        Width => 8
)
port map(
        INC     => Cnt_FollowReg_INC,
        rst     => Cnt_FollowReg_Rst,
        clk     => clk,
        Count   => Cnt_FollowReg_Out
        );




----Registers
Reg_Proc: process
begin
        wait until clk'event and clk = '1';
        if rst = '0' then

                --Calculations
                Tsw_reg_o <= (others => '0');
                Div_reg_o<= (others => '0');
                Var_reg_o<= (others => '0');
                Dec_reg_o <= (others => '0');
                Ref_Ton<= (others => '0');
                Van_DC <= (others => '0');
                Vbn_DC <= (others => '0');
                Vcn_DC <= (others => '0');
                Va_DC <= (others => '0');
                Vb_DC <= (others => '0');
                Vc_DC <= (others => '0');
                Vab_DC<= (others => '0');
                Vbc_DC<= (others => '0');
                Vca_DC<= (others => '0');
                Vab     <= (others => '0');
                Vbc     <= (others => '0');
                Vca     <= (others => '0');
                V1_Dat<= (others => '0');
                V2_Dat<= (others => '0');
```

```
V3_Dat<= (others => '0');
V4_Dat<= (others => '0');
V5_Dat<= (others => '0');
V6_Dat<= (others => '0');

--Data Distribution
Vrble_Data<= (others => '0');
PreScale<= (others => '0');
Trigger<= (others => '0');
Emu_V_Type<= (others => '0');
Scale_Ref<= (others => '0');
Va_Samp<= (others => '0');
Vb_Samp<= (others => '0');
Vc_Samp<= (others => '0');
Addr_Va_Start<= (others => '0');
Addr_Vb_Start<= (others => '0');
Addr_Vc_Start<= (others => '0');
Emu_DL_Start<= '0';

--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--<= (others => '0');
--
EN <= '0';
--<= '0';
--<= '0';
--<= '0';
--<= '0';
--<= '0';
```

else

```
                    --Calculations
                    if (LD_Tsw = '1')    then  Tsw_reg_o <= Tsw;                    end if;
                    if (LD_Div = '1')    then  Div_reg_o <= Div;                    end if;
                    if (LD_Var = '1')    then  Var_reg_o <= Var;                    end if;
                    if (LD_Dec = '1')         then  Dec_reg_o <= Dec;               end if;
                    if (LD_Ref_Ton = '1') then  Ref_Ton   <= Temp_Ref_Ton; end if;
                    if (LD_Van_DC = '1')   then  Van_DC    <= Temp_Van_DC;   end if;
                    if (LD_Vbn_DC = '1')   then  Vbn_DC    <= Temp_Vbn_DC;   end if;
                    if (LD_Vcn_DC = '1')   then  Vcn_DC    <= Temp_Vcn_DC;   end if;
                    if (LD_Va_DC = '1')    then  Va_DC     <= Temp_Va_DC;   end if;
                    if (LD_Vb_DC = '1')    then  Vb_DC     <= Temp_Vb_DC;   end if;
                    if (LD_Vc_DC = '1')    then  Vc_DC     <= Temp_Vc_DC;   end if;
                    if (LD_Vab_DC = '1')   then  Vab_DC                <= Temp_Vab_DC;
end if;
                    if (LD_Vbc_DC = '1')   then  Vbc_DC        <= Temp_Vbc_DC; end if;
                    if (LD_Vca_DC = '1')   then  Vca_DC        <= Temp_Vca_DC; end if;
                    if (LD_Vab      = '1')   then  Vab             <= Temp_Vab  ; end if;
                    if (LD_Vbc      = '1')   then  Vbc             <= Temp_Vbc  ; end if;
                    if (LD_Vca      = '1')   then  Vca             <= Temp_Vca  ; end if;
                    if (LD_V1_Dat = '1')   then  V1_Dat    <= Temp_V1_Dat;   end if;
                    if (LD_V2_Dat = '1')   then  V2_Dat    <= Temp_V2_Dat;   end if;
                    if (LD_V3_Dat = '1')   then  V3_Dat    <= Temp_V3_Dat;   end if;
                    if (LD_V4_Dat = '1')   then  V4_Dat    <= Temp_V4_Dat;   end if;
                    if (LD_V5_Dat = '1')   then  V5_Dat    <= Temp_V5_Dat;   end if;
                    if (LD_V6_Dat = '1')   then  V6_Dat    <= Temp_V6_Dat;   end if;

                    --Data Distribution
                    if (LD_Vrble_Data = '1')   then    Vrble_Data   <= Temp_Vrble_Data;
end if;
                    if (LD_Emu_DL_Start = '1')   then   Emu_DL_Start  <=
Temp_Emu_DL_Start;   end if;
                    if (LD_PreScale = '1')   then    PreScale    <= Temp_PreScale;   end if;
                    if (LD_Trigger = '1')   then    Trigger    <= Temp_Trigger;   end if;
                    if (LD_Emu_V_Type = '1')   then    Emu_V_Type    <=
Temp_Emu_V_Type;   end if;
                    if (LD_Scale_Ref = '1')   then    Scale_Ref    <= Temp_Scale_Ref;   end if;
                    if (LD_Va_Samp = '1') then  Va_Samp <= Temp_Va_Samp; end if;
                    if (LD_Vb_Samp = '1') then  Vb_Samp <= Temp_Vb_Samp; end if;
                    if (LD_Vc_Samp = '1') then  Vc_Samp <= Temp_Vc_Samp; end if;
                    if (LD_Addr_Va_Start = '1') then Addr_Va_Start  <=
Temp_Addr_Va_Start; end if;
                    if (LD_Addr_Vb_Start = '1') then Addr_Vb_Start  <=
Temp_Addr_Vb_Start; end if;
                    if (LD_Addr_Vc_Start = '1') then Addr_Vc_Start  <=
Temp_Addr_Vc_Start; end if;
                    if (LD_EN = '1') then EN  <= Temp_EN ; end if;
```

```
                --if ( = '1') then   <= ; end if;
                --if ( = '1') then   <= ; end if;
                --if ( = '1') then   <= ; end if;
                --if ( = '1') then   <= ; end if;
                --if ( = '1') then   <= ; end if;
                --if ( = '1') then   <= ; end if;
                --if ( = '1') then   <= ; end if;
                --if ( = '1') then   <= ; end if;
                --if ( = '1') then   <= ; end if;
                --if ( = '1') then   <= ; end if;
                --if ( = '1') then   <= ; end if;

        end if;
        end process;
        ----End Registers




        Fsw_and_1pCent_Duty_Cycle: process(CS_Fsw, Emu_EN, SW01, CntTsw_Out,
CntTon_Out, Div_reg_o, Var_reg_o, Dec_reg_o, HP_EN, Error)
        begin


                CntTsw_INC <= '0';
                CntTsw_Rst <= '1';

                CntTon_INC <= '0';
                CntTon_Rst <= '1';

                LD_Tsw <= '0';
                LD_Div <= '0';
                LD_Var <= '0';
                LD_Dec <= '0';
                LD_Ref_Ton <= '0';

                Tsw <= (others => '0');
                Div <= (others => '0');
                Var <= (others => '0');
                Dec <= (others => '0');
                Temp_Ref_Ton <= (others => '0');
```

```
		Temp_EN <= '0';
		LD_EN <= '0';


-------START: Calculate Tsw and 1% On Time for SW01 using Clk Cycles--------
		case CS_Fsw is
			when S0 =>
				if(Emu_EN = '0')then
					Temp_EN <= '0';
					LD_EN <= '1';
					NS_Fsw <= S0;
				else
					NS_Fsw <= S1;

				end if;

				CntTsw_Rst <= '0';
				CntTon_Rst <= '0';
				--Tsw <= (others => '0');
				--LD_Tsw <= '1';

			when S1 =>
				if (SW01 = '1') then

					NS_Fsw <= S1;
				else

					NS_Fsw <= S2;
				end if;

				--CntTsw_Rst <= '0';

			when S2 =>
				if (SW01 = '0') then

					NS_Fsw <= S2;
				else


					CntTsw_INC <= '1';
					NS_Fsw <= S3;
				end if;

			when S3 =>
				if(SW01 = '1') then
```

```
                NS_Fsw <= S3;
        else
                NS_Fsw <= S4;
        end if;
        CntTsw_INC <= '1';
        Tsw <= CntTsw_Out;
        LD_Tsw <= '1';

when S4 =>
        if(SW01 = '0') then
                CntTsw_INC <= '1';
                NS_Fsw <= S4;
        else
                CntTsw_INC <= '0';
                NS_Fsw <= S5;
        end if;
        Tsw <= CntTsw_Out;
        LD_Tsw <= '1';


when S5 =>
        Tsw <= CntTsw_Out;
        LD_Tsw <= '1';
        Div <= X"0064";
        LD_Div <= '1';
        Var <= X"0064";
        LD_Var <= '1';
        --CntTon_Rst <= '0';
        NS_Fsw <= S6;

when S6 =>
        NS_Fsw <= S7;

when S7 =>
        if(Div_reg_o < (Tsw_reg_o + 1)) then
                NS_Fsw <= S8;
        else
                NS_Fsw <= S12;
        end if;

when S8 =>
        Var <= Var_reg_o + X"0064";
        LD_Var <= '1';
        NS_Fsw <= S9;

when S9 =>
```

```
                CntTon_INC <= '1';
                NS_Fsw <= S10;

        when S10 =>
                Div <= Var_reg_o;
                LD_Div <= '1';
                NS_Fsw <= S11;

        when S11 =>
                NS_Fsw <= S7;
--*
        when S12 =>
                Dec <= Div_reg_o - Tsw_reg_o;
                LD_Dec <= '1';
                NS_Fsw <= S13;
        when S13 =>
                NS_Fsw <= S14;

        when S14 =>
                if(Dec_reg_o < X"0032")then
                        CntTon_INC <= '1';
                else
                        CntTon_INC <= '0';
                end if;
                NS_Fsw <= S15;

        when S15 =>
                NS_Fsw <= S16;

        when S16 =>
                Temp_Ref_Ton <= CntTon_Out;
                LD_Ref_Ton <= '1';
                NS_Fsw <= S17;

        when S17 =>
                if((Error = '0') and (HP_EN = '0'))then
                        --EN <= '1';
                        NS_Fsw <= S17;
                else
                        --EN <= '0';
                        NS_Fsw <= S0;
                end if;
                        Temp_EN <= '1';
                        LD_EN <= '1';
        when others =>
                NS_Fsw <= S0;
```

```
-------STOP: Calculate Tsw and 1% On Time for SW01 using Clk Cycles--------

        end case;
end process;









Va_Duty_Cycle: process(CSA, EN, SW01, CntA1_Out, CntA2_Out, CntS1on_Out)
begin

        CntA1_INC <= '0';
        CntA1_Rst <= '1';

        CntA2_INC <= '0';
        CntA2_Rst <= '1';

        CntS1on_INC <= '0';
        CntS1on_Rst <= '1';

        LD_Van_DC <= '0';

        Temp_Van_DC <= (others => '0');




-------START: Calculate Duty Cycle % of SW01 --------
        case CSA is
                when S0 =>
                        if(EN <= '0')then
                                NSA<=S0;
                        else
                                NSA<=S1;
                        end if;


                when S1 =>
```

```
                        if (SW01 = '1') then            -- Check to assure that SW01 is not
ON yet
                                NSA <= S1;
                        else
                                NSA <= S2;
                        end if;
                        CntS1on_Rst <= '0';
                        CntA1_Rst <= '0';
                        CntA2_Rst <= '0';

                when S2 =>
                        if (SW01 = '0') then            -- Once SW01 is OFF, wait until it
turns ON
                                NSA <= S2;
                        else
                                CntA1_INC <= '1';               -- Once SW01 is ON, start
incrementing counter A1
                                NSA <= S3;
                        end if;

                when S3 =>                              -- S13 and S14 alternate to count
(CntA1, CntA2) until 1% ON Time is reached
                        if ((SW01 = '1') and (CntA1_Out = Ref_Ton)) then
                                CntA2_INC <= '1';
                                CntS1on_INC <='1';                      --Cnt1 counts only
when 1% ON Time is reached
                                CntA1_Rst <= '0';
                                NSA <= S4;
                        elsif(SW01 = '1') then
                                CntA1_INC <= '1';               -- CntA1 counts to 1% ON
Time in S13 then resets in S14
                                CntA2_Rst <= '0';               -- While CntA1 counts to 1%
ON Time, CntA2 will reset to 0
                                NSA <= S3;
                        else
                                NSA <= S5;
                        end if;

                when S4 =>
                        if ((SW01 = '1') and (CntA2_Out = Ref_Ton)) then
                                CntA1_INC <= '1';
                                CntS1on_INC <='1';                      --Cnt1 counts only
when 1% ON Time is reached
                                CntA2_Rst <= '0';
                                NSA <= S3;
                        elsif(SW01 = '1') then
```

```
                                        CntA2_INC <= '1';               -- CntA2 counts to 1% ON
Time in S14 then resets in S13
                                        CntA1_Rst <= '0';               -- While CntA2 counts to 1%
ON Time, CntA1 will reset to 0
                                        NSA <= S4;
                                else
                                        NSA <= S5;
                                end if;

                        when S5 =>                              -- In case of 0% < %ON Time < 1%,
round up extra 1% for DutyCycle (Temp_Van_DC)
                                if(CntA1_Out < X"0064")then
                                        if ((CntA1_Out > X"0002") or (CntA2_Out > X"0002"))
then
                                                Temp_Van_DC <= CntS1on_Out + 1;--X"01";

                                        else
                                                Temp_Van_DC <= CntS1on_Out;
                                        end if;
                                else
                                        Temp_Van_DC <= X"63";
                                end if;

                                LD_Van_DC <= '1';
                                NSA <= S0;

                        when others=>
                                NSA <= S0;


        -------STOP: Calculate Duty Cycle % of SW01 --------

                end case;
        end process;



        Vb_Duty_Cycle: process(CSB, EN, SW03, CntB1_Out, CntB2_Out, CntS3on_Out)
        begin

                CntB1_INC <= '0';
                CntB1_Rst <= '1';

                CntB2_INC <= '0';
                CntB2_Rst <= '1';
```

```
		CntS3on_INC <= '0';
		CntS3on_Rst <= '1';

		LD_Vbn_DC <= '0';

		Temp_Vbn_DC <= (others => '0');




	-------START: Calculate Duty Cycle % of SW03 --------
		case CSB is
			when S0 =>
				if(EN <= '0')then
					NSB<=S0;
				else
					NSB<=S1;
				end if;


			when S1 =>
				if (SW03 = '1') then		-- Check to assure that SW03 is not
ON yet
					NSB <= S1;
				else
					NSB <= S2;
				end if;
				CntS3on_Rst <= '0';
				CntB1_Rst <= '0';
				CntB2_Rst <= '0';

			when S2 =>
				if (SW03 = '0') then		-- Once SW03 is OFF, wait until it
turns ON
					NSB <= S2;
				else
					CntB1_INC <= '1';		-- Once SW03 is ON, start
incrementing counter B1
					NSB <= S3;
				end if;

			when S3 =>				-- S13 and S14 alternate to count
(CntB1, CntB2) until 1% ON Time is reached
				if ((SW03 = '1') and (CntB1_Out = Ref_Ton)) then
```

```
                        CntB2_INC <= '1';
                        CntS3on_INC <='1';              --Cnt1 counts only
when 1% ON Time is reached
                        CntB1_Rst <= '0';
                        NSB <= S4;
                    elsif(SW03 = '1') then
                        CntB1_INC <= '1';           -- CntB1 counts to 1% ON
Time in S13 then resets in S14
                        CntB2_Rst <= '0';           -- While CntB1 counts to 1%
ON Time, CntB2 will reset to 0
                        NSB <= S3;
                    else
                        NSB <= S5;
                    end if;

                when S4 =>
                    if ((SW03 = '1') and (CntB2_Out = Ref_Ton)) then
                        CntB1_INC <= '1';
                        CntS3on_INC <='1';              --Cnt1 counts only
when 1% ON Time is reached
                        CntB2_Rst <= '0';
                        NSB <= S3;
                    elsif(SW03 = '1') then
                        CntB2_INC <= '1';           -- CntB2 counts to 1% ON
Time in S14 then resets in S13
                        CntB1_Rst <= '0';           -- While CntB2 counts to 1%
ON Time, CntB1 will reset to 0
                        NSB <= S4;
                    else
                        NSB <= S5;
                    end if;

                when S5 =>                          -- In case of 0% < %ON Time < 1%,
round up extra 1% for DutyCycle (Temp_Vbn_DC)
                    if ((CntB1_Out > X"0002") or (CntB2_Out > X"0002")) then
                        Temp_Vbn_DC <= CntS3on_Out + 1;--X"01";

                    else
                        Temp_Vbn_DC <= CntS3on_Out;
                    end if;
                    LD_Vbn_DC <= '1';
                    NSB <= S0;

                when others=>
                    NSB <= S0;
```

```
-------STOP: Calculate Duty Cycle % of SW03 --------

        end case;
end process;


Vc_Duty_Cycle: process(CSC, EN, SW05, CntC1_Out, CntC2_Out, CntS5on_Out)
begin

        CntC1_INC <= '0';
        CntC1_Rst <= '1';

        CntC2_INC <= '0';
        CntC2_Rst <= '1';

        CntS5on_INC <= '0';
        CntS5on_Rst <= '1';

        LD_Vcn_DC <= '0';

        Temp_Vcn_DC <= (others => '0');




-------START: Calculate Duty Cycle % of SW05 --------
        case CSC is
                when S0 =>
                        if(EN <= '0')then
                                NSC<=S0;
                        else
                                NSC<=S1;
                        end if;


                when S1 =>
                        if (SW05 = '1') then            -- Check to assure that SW05 is not
ON yet
                                NSC <= S1;
                        else
                                NSC <= S2;
                        end if;
                        CntS5on_Rst <= '0';
                        CntC1_Rst <= '0';
```

```
                CntC2_Rst <= '0';

            when S2 =>
                if (SW05 = '0') then            -- Once SW05 is OFF, wait until it
turns ON
                    NSC <= S2;
                else
                    CntC1_INC <= '1';           -- Once SW05 is ON, start
incrementing counter C1
                    NSC <= S3;
                end if;

            when S3 =>                          -- S13 and S14 alternate to count
(CntC1, CntC2) until 1% ON Time is reached
                if ((SW05 = '1') and (CntC1_Out = Ref_Ton)) then
                    CntC2_INC <= '1';
                    CntS5on_INC <='1';                  --Cnt1 counts only
when 1% ON Time is reached
                    CntC1_Rst <= '0';
                    NSC <= S4;
                elsif(SW05 = '1') then
                    CntC1_INC <= '1';           -- CntC1 counts to 1% ON
Time in S13 then resets in S14
                    CntC2_Rst <= '0';           -- While CntC1 counts to 1%
ON Time, CntC2 will reset to 0
                    NSC <= S3;
                else
                    NSC <= S5;
                end if;

            when S4 =>
                if ((SW05 = '1') and (CntC2_Out = Ref_Ton)) then
                    CntC1_INC <= '1';
                    CntS5on_INC <='1';                  --Cnt1 counts only
when 1% ON Time is reached
                    CntC2_Rst <= '0';
                    NSC <= S3;
                elsif(SW05 = '1') then
                    CntC2_INC <= '1';           -- CntC2 counts to 1% ON
Time in S14 then resets in S13
                    CntC1_Rst <= '0';           -- While CntC2 counts to 1%
ON Time, CntC1 will reset to 0
                    NSC <= S4;
                else
                    NSC <= S5;
                end if;
```

```
                when S5 =>                          -- In case of 0% < %ON Time < 1%,
round up extra 1% for DutyCycle (Temp_Vcn_DC)
                        if ((CntC1_Out > X"0002") or (CntC2_Out > X"0002")) then
                                Temp_Vcn_DC <= CntS5on_Out + 1;--X"01";

                        else
                                Temp_Vcn_DC <= CntS5on_Out;
                        end if;
                        LD_Vcn_DC <= '1';
                        NSC <= S0;

                when others=>
                        NSC <= S0;


        -------STOP: Calculate Duty Cycle % of SW05 --------

                end case;
        end process;


        Vab_Calculation: process(CSab, EN, V1_Dat, V2_Dat, Emu_V_Type, Vab_DC,
Van_DC, Vbn_DC, Vab)
        begin

                Temp_V1_Dat<= (others => '0');
                Temp_V2_Dat<= (others => '0');
                Temp_Vab_DC     <= (others => '0');
                Temp_Va_DC      <= (others => '0');
                Temp_Vab    <= (others => '0');

                LD_V1_Dat   <= '0';
                LD_V2_Dat   <= '0';
                LD_Vab_DC   <= '0';
                LD_Va_DC    <= '0';
                LD_Vab          <= '0';



--*Vab_DC needs to have more bits to allow V1-V2 to fit. V1 and V2 are 8 bits. Vab_DC is 16
bits

                case CSab is
                        when S0 =>
```

```
        if(EN = '0')then
                NSab <= S0;
        else
                NSab <= S1;
        end if;
        Temp_V1_Dat <= Van_DC;
        LD_V1_Dat <= '1';
        Temp_V2_Dat <= Vbn_DC;
        LD_V2_Dat <= '1';

when S1 =>
        --LD_Vab_DC <= '1';
        NSab<=S2;

when S2 =>
        if(V1_Dat > V2_Dat)then
                Temp_Vab <= V1_Dat - V2_Dat;
                --Temp_Vab_DC <= (others => '0');

        else
                Temp_Vab <= V2_Dat - V1_Dat;
                --Temp_Vab_DC(15) <= '1';

        end if;
        --LD_Vab_DC <= '1';
        LD_Vab <= '1';
        NSab <= S3;
when S3=>
        if(V1_Dat > V2_Dat)then

                Temp_Vab_DC(11 downto 0) <= Vab;

        else

                Temp_Vab_DC <= Vab + X"8000";

        end if;
        LD_Vab_DC <= '1';

        NSab <= S4;
when S4 =>
        --Temp_Vab_DC(11 downto 0) <= Vab;
        --Temp_Vab_DC <= Vab + X"0000";-- * Vdc;
        --LD_Vab_DC <= '1';
        if (Emu_V_Type < X"02")then
                Temp_Va_DC(7 downto 0) <= Van_DC;
```

```
                else
                        Temp_Va_DC <= Vab_DC;
                end if;
                LD_Va_DC <= '1';
                NSab <= S0;



        --when S5=>
                --if(V1_Dat > V2_Dat)then
                        --Temp_Vab_DC <= Vab_DC;-- X"00F3";-- + X"4000";
                        --Temp_Vab_DC(11 downto 0) <= Vab;

                --else

                        --Temp_Vab_DC <= Vab_DC + X"8000";

                --end if;
                --LD_Vab_DC <= '1';
                --NSab <= S6;

        --when S6 =>
                --if (Emu_V_Type < X"02")then
                        --Temp_Va_DC(7 downto 0) <= Van_DC;
                        --Temp_Va_DC <= Van_DC*Vdc;

                --else
                        --Temp_Va_DC <= Vab_DC;
                --end if;
                --LD_Va_DC <= '1';
                --NSab <= S0;
        when others =>
                NSab <= S0;

    end case;
end process;
```

```
Vbc_Calculation: process(CSbc, EN, V3_Dat, V4_Dat, Emu_V_Type, Vbc_DC,
Vbn_DC, Vcn_DC, Vbc)
```

```
begin

        Temp_V3_Dat<= (others => '0');
        Temp_V4_Dat<= (others => '0');
        Temp_Vbc_DC     <= (others => '0');
        Temp_Vb_DC      <= (others => '0');
        Temp_Vbc    <= (others => '0');

        LD_V3_Dat   <= '0';
        LD_V4_Dat   <= '0';
        LD_Vbc_DC   <= '0';
        LD_Vb_DC    <= '0';
        LD_Vbc          <= '0';


--*Vbc_DC needs to have more bits to allow V3-V4 to fit. V3 and V4 are 8 bits. Vbc_DC is 16 bits

        case CSbc is
            when S0 =>
                if(EN = '0')then
                    NSbc <= S0;
                else
                    NSbc <= S1;
                end if;
                Temp_V3_Dat <= Vbn_DC;
                LD_V3_Dat <= '1';
                Temp_V4_Dat <= Vcn_DC;
                LD_V4_Dat <= '1';

            when S1 =>
                --LD_Vbc_DC <= '1';
                NSbc<=S2;

            when S2 =>
                if(V3_Dat > V4_Dat)then
                    Temp_Vbc <= V3_Dat - V4_Dat;
                    --Temp_Vbc_DC <= (others => '0');

                else
                    Temp_Vbc <= V4_Dat - V3_Dat;
                    --Temp_Vbc_DC(15) <= '1';

                end if;
                --LD_Vbc_DC <= '1';
                LD_Vbc <= '1';
```

```
                        NSbc <= S3;
                when S3=>
                        if(V3_Dat > V4_Dat)then

                                Temp_Vbc_DC(11 downto 0) <= Vbc;

                        else

                                Temp_Vbc_DC <= Vbc + X"8000";

                        end if;
                        LD_Vbc_DC <= '1';
                        NSbc <= S4;

                when S4 =>
                        if (Emu_V_Type < X"02")then
                                Temp_Vb_DC(7 downto 0) <= Vbn_DC;

                        else
                                Temp_Vb_DC <= Vbc_DC;
                        end if;
                        LD_Vb_DC <= '1';
                        NSbc <= S0;
                when others =>
                        NSbc <= S0;

        end case;
    end process;



    Vca_Calculation: process(CSca, EN, V5_Dat, V6_Dat, Emu_V_Type, Vca_DC,
Vcn_DC, Van_DC, Vca)
    begin

        Temp_V5_Dat<= (others => '0');
        Temp_V6_Dat<= (others => '0');
        Temp_Vca_DC     <= (others => '0');
        Temp_Vc_DC      <= (others => '0');
        Temp_Vca    <= (others => '0');

        LD_V5_Dat   <= '0';
        LD_V6_Dat   <= '0';
        LD_Vca_DC   <= '0';
        LD_Vc_DC    <= '0';
        LD_Vca          <= '0';
```

```
--*Vca_DC needs to have more bits to allow V5-V6 to fit. V5 and V6 are 8 bits. Vca_DC is 16
bits

                case CSca is
                        when S0 =>
                                if(EN = '0')then
                                        NSca <= S0;
                                else
                                        NSca <= S1;
                                end if;
                                Temp_V5_Dat <= Vcn_DC;
                                LD_V5_Dat <= '1';
                                Temp_V6_Dat <= Van_DC;
                                LD_V6_Dat <= '1';

                        when S1 =>
                                --LD_Vca_DC <= '1';
                                NSca<=S2;

                        when S2 =>
                                if(V5_Dat > V6_Dat)then
                                        Temp_Vca <= V5_Dat - V6_Dat;
                                        --Temp_Vca_DC <= (others => '0');

                                else
                                        Temp_Vca <= V6_Dat - V5_Dat;
                                        --Temp_Vca_DC(15) <= '1';

                                end if;
                                --LD_Vca_DC <= '1';
                                LD_Vca <= '1';
                                NSca <= S3;
                        when S3=>
                                if(V5_Dat > V6_Dat)then

                                        Temp_Vca_DC(11 downto 0) <= Vca;

                                else

                                        Temp_Vca_DC <= Vca + X"8000";

                                end if;
                                LD_Vca_DC <= '1';
                                NSca <= S4;

                        when S4 =>
```

```
                        if (Emu_V_Type < X"02")then
                                Temp_Vc_DC(7 downto 0) <= Vcn_DC;

                        else
                                Temp_Vc_DC <= Vca_DC;
                        end if;
                        LD_Vc_DC <= '1';
                        NSca <= S0;
                when others =>
                        NSca <= S0;

        end case;
end process;
```

---------- ** EMU DATA DISTRIBUTION GOES HERE, BUT WILL BE EDITED IN SEPARATE FILE FOR SIMPLICITY AND THEN COPED HERE **

```
        Emu_Data_Traffic : process(CS, CntDelay_Out, CntBus_Out, Bus_Int1_Busy,
Bus_Int1_DataOut, Vrble_Data, Error, Emu_DL_Start, HP_EN, EN, PreScale, Tsw_reg_o,
Cnt_LeadReg_Out, Scale_Ref, Cnt_Scale_Out, Va_DC, Vb_DC, Vc_DC, Va_Samp, Vb_Samp,
Vc_Samp, Cnt_FollowReg_Out, STD_FIFO_Va_Full, STD_FIFO_Va_Empty,
STD_FIFO_Va_DataOut, STD_FIFO_Vb_Full, STD_FIFO_Vb_Empty,
```

```
STD_FIFO_Vb_DataOut, STD_FIFO_Vc_Full, STD_FIFO_Vc_Empty,
STD_FIFO_Vc_DataOut)
        begin

                CntBus_Rst <= '1';
                CntDelay_Rst <= '1';
                Cnt_LeadReg_Rst <= '1';
                Cnt_Scale_Rst <= '1';
                Cnt_FollowReg_Rst <= '1';
                CntBus_INC <= '0';
                CntDelay_INC <= '0';
                Cnt_LeadReg_INC <= '0';
                Cnt_Scale_INC <= '0';
                Cnt_FollowReg_INC <= '0';

                LD_Addr_Va_Start <= '0';
                LD_Addr_Vb_Start <= '0';
                LD_Addr_Vc_Start <= '0';
                Temp_Addr_Va_Start <= (others => '0');
                Temp_Addr_Vb_Start <= (others => '0');
                Temp_Addr_Vc_Start <= (others => '0');

                LD_Vrble_Data <= '0';
                Temp_Vrble_Data <= (others => '0');

                LD_Emu_DL_Start <= '0';
                Temp_Emu_DL_Start <= '0';

                LD_Emu_V_Type <= '0';
                Temp_Emu_V_Type <= (others => '0');

                LD_PreScale <= '0';
                Temp_PreScale <= (others => '0');

                LD_Trigger <= '0';
                Temp_Trigger <= (others => '0');

                LD_Scale_Ref <= '0';
                Temp_Scale_Ref <= (others => '0');

                Temp_Va_Samp <= (others => '0');
                Temp_Vb_Samp <= (others => '0');
                Temp_Vc_Samp <= (others => '0');
                LD_Va_Samp <= '0';
                LD_Vb_Samp <= '0';
                LD_Vc_Samp <= '0';
```

```
Bus_Int1_AddrIn <= (others => '0');
Bus_Int1_RE <='0';
Bus_Int1_DataIn <= (others => '0');
Bus_Int1_WE <='0';

STD_FIFO_Va_WriteEn <='0';
STD_FIFO_Va_DataIn <= (others => '0');
STD_FIFO_Va_ReadEn <='0';

STD_FIFO_Vb_WriteEn <='0';
STD_FIFO_Vb_DataIn <= (others => '0');
STD_FIFO_Vb_ReadEn <='0';

STD_FIFO_Vc_WriteEn <='0';
STD_FIFO_Vc_DataIn <= (others => '0');
STD_FIFO_Vc_ReadEn <='0';



case CS is
	when S0 =>
		CntBus_Rst <='0';		-- Reset Bus Counter
		CntDelay_Rst <='0';		-- Reset Delay Counter
		Cnt_LeadReg_Rst <= '0';
		Cnt_Scale_Rst <= '0';
		Cnt_FollowReg_Rst <= '0';
		Temp_Addr_Va_Start <= Addr0_Emu_Va;
		Temp_Addr_Vb_Start <= Addr0_Emu_Vb;
		Temp_Addr_Vc_Start <= Addr0_Emu_Vc;
		LD_Addr_Va_Start <= '1';
		LD_Addr_Vb_Start <= '1';
		LD_Addr_Vc_Start <= '1';

		NS <= S1;
	when S1=>
		if(CntDelay_Out < 40) then
			NS<=S1;
		else
			NS<=S2;
		end if;
		CntDelay_INC<='1';

	when S2=>                                                    --Wait
		if(CntBus_Out < 128) then
			NS<=S2;
		else
```

```
                                NS<=S3;
                        end if;
                        CntBus_INC<='1';

                when S3 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S3;
                        else
                                NS <=S4;
                        end if;
                        CntBus_Rst <='0';               -- Reset Bus Counter

                when S4 =>
                        Bus_Int1_AddrIn <= Addr_Emu_DL_Start; --
Addr_Emu_DL_Start is a constant from Common file
                        Bus_Int1_RE <='1';
                        NS <= S5;

                when S5 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S5;
                        else

                                NS <=S6;
                        end if;
                        Temp_Vrble_Data <= Bus_Int1_DataOut;
                        LD_Vrble_Data <= '1';

                when S6 =>
                        if(Error = '1') then
                                Temp_Emu_DL_Start <= '0';
----SE
                                NS <= S7;
                        else
                                Temp_Emu_DL_Start <= Vrble_Data(0);
----Sa
                                NS <= S19;
                        end if;
                        LD_Emu_DL_Start <= '1';


        ---------------------------------- Start ERROR Procedure ----------------------------------------
-
                -- Set the Error bit in Emu DL Stat RAM Reg
----SE
                when S7 =>
```

```
                    Bus_Int1_AddrIn <= Addr_Emu_DL_Status; --
Addr_Emu_DL_Status is a constant from Common file
                    Bus_Int1_DataIn <= X"0003"; -- Emu_DL_Stat = 3 (ERROR)
                    Bus_Int1_WE <='1';
                    NS <= S8;

                when S8 =>
                    if(Bus_Int1_Busy = '1') then
                        NS <= S8;
                    else

                        NS <=S9;
                    end if;

                -- Check if Error is still ON
                when S9 =>
                    if (Error = '1') then
                        NS <= S9;
                    else
                        NS <= S10;
                    end if;


                --After Error is Cleared, Reset (Emu_DL_Status Error bit, Emu_DL_Start,
Emu_V_OP, Emu_Prescale, and Emu_Trigger) RAM Regs


                -- Reset Emu DL Start RAM Reg
                when S10 =>
                    Bus_Int1_AddrIn <= Addr_Emu_DL_Start; --
Addr_Emu_DL_Start is a constant from Common file
                    Bus_Int1_DataIn <= X"0000";
                    Bus_Int1_WE <='1';
                    NS <= S11;

                when S11 =>
                    if(Bus_Int1_Busy = '1') then
                        NS <= S11;
                    else

                        NS <=S12;
                    end if;
                    Temp_Emu_DL_Start <= '0';
                    LD_Emu_DL_Start <= '1';
```

```
                -- Reset Emu Vout OP RAM Reg
                when S12 =>
                        Bus_Int1_AddrIn <= Addr_Emu_V_OP; --Addr_Emu_V_OP is a
constant from Common file
                        Bus_Int1_DataIn <= X"0000";
                        Bus_Int1_WE <='1';
                        NS <= S13;

                when S13 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S13;
                        else

                                NS <=S14;
                        end if;
                        Temp_Emu_V_Type <= (others => '0');
                        LD_Emu_V_Type <= '1';


                -- Reset Emu Prescale RAM Reg
                when S14 =>
                        Bus_Int1_AddrIn <= Addr_Emu_Prescale; --Addr_Emu_Prescale
is a constant from Common file
                        Bus_Int1_DataIn <= X"0000";
                        Bus_Int1_WE <='1';
                        NS <= S15;

                when S15 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S15;
                        else

                                NS <=S16;
                        end if;
                        Temp_PreScale <= (others => '0');
                        LD_PreScale <= '1';

                -- Reset Emu Trigger RAM Reg
                when S16 =>
                        Bus_Int1_AddrIn <= Addr_Emu_Trigger; --Addr_Emu_Trigger is
a constant from Common file
                        Bus_Int1_DataIn <= X"0000";
                        Bus_Int1_WE <='1';
                        NS <= S17;

                when S17 =>
```

```
                        if(Bus_Int1_Busy = '1') then
                                NS <= S17;
                        else

                                NS <=S18;
                        end if;
                        Temp_Trigger <= (others => '0');
                        LD_Trigger <= '1';


                -- After Error is Cleared, Reset Error bit in Emu DL Stat RAM Reg
                when S18 =>
                        Bus_Int1_AddrIn <= Addr_Emu_DL_Status; --
Addr_Emu_DL_Status is a constant from Common file
                        Bus_Int1_DataIn <= X"0000"; -- Emu_DL_Stat = 0 (Ready/Done)
                        Bus_Int1_WE <='1';
                        NS <= S0;
        ---------------------------------- End ERROR Procedure ----------------------------------------


-----Sa
                when S19 =>
                        NS <= S20;


                -- Check if Emu_DL_Start is ON
                when S20 =>
                        if(Emu_DL_Start = '1') then
                                Bus_Int1_AddrIn <= Addr_Emu_DL_Status; --
Addr_Emu_DL_Status is a constant from Common file
                                Bus_Int1_DataIn <= X"0001"; -- Emu_DL_Stat = 1
(Busy/Waiting for Trigger)
                                Bus_Int1_WE <='1';
                                NS <= S21;
                        else
                                NS <= S0;
                        end if;


---------------------------------------- Start Emu Data Logging ----------------------------------------
```

```
                ---------------------------- Start pulling Emu DL data from RAM ----------------------
--------

                ---- If need to pull more Emu DL data from RAM, multiply this section below as
needed ----
                        ---------------------------------------------------------------------
                        --when S=>
                                --if(Bus_Int1_Busy = '1') then
                                        --NS <= Sloop;
                                --else

                                        --NS <=S;
                                --end if;

                        --when S=>
                                --Bus_Int1_AddrIn <= Addr_; --Addr_ is a constant from
Common file
                                --Bus_Int1_RE <='1';
                                --NS <= S;

                        --when S =>
                                --if(Bus_Int1_Busy = '1') then
                                        --NS <= Sloop;
                                --else

                                        --NS <=S;
                                --end if;
                                --Temp_Vrble_Data <= Bus_Int1_DataOut;
                                --LD_Vrble_Data <= '1';

                        --when S =>
                                --<= Vrble_Data();
                                --LD_ <= '1';

                        --- If needed more data saved to registers ---

                                --      <= Vrble_Data();
                                --LD_ <= '1';

                                --      <= Vrble_Data();
                                --LD_ <= '1';

                                --      <= Vrble_Data();
                                --LD_ <= '1';

                                --NS <= S;
```

```
                ----------------------------------------------------------------

                -- Pulling Emu DL PreScale data from RAM
                when S21=>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S21;
                        else

                                NS <=S22;
                        end if;

                when S22=>
                        --Bus_Int1_AddrIn <= Addr_Emu_Prescale; --
Addr_Emu_Prescale is a constant from Common file
                        --Bus_Int1_RE <='1';
                        Bus_Int1_AddrIn <= Addr_Emu_Prescale; --
Addr_Emu_DL_Status is a constant from Common file
                        Bus_Int1_DataIn <= Ref_Ton;--X"0001"; -- Emu_DL_Stat = 1
(Busy/Waiting for Trigger)
                        Bus_Int1_WE <='1';
                        NS <= S23;

                when S23 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S23;
                        else

                                NS <=S24;
                        end if;
                        --Temp_Vrble_Data <= Bus_Int1_DataOut;
                        --LD_Vrble_Data <= '1';

                when S24 =>
                        --Temp_PreScale<= Vrble_Data;
                        --LD_PreScale <= '1';



                        NS <= S25;



                -- Pulling Emu DL Trigger data from RAM
                when S25=>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S25;
```

```
                else

                        NS <=S26;
                end if;

        when S26=>
                --Bus_Int1_AddrIn <= Addr_Emu_Trigger; --Addr_Emu_Trigger
is a constant from Common file
                --Bus_Int1_RE <='1';
                Bus_Int1_AddrIn <= Addr_Emu_Trigger; --
Addr_Emu_DL_Status is a constant from Common file
                Bus_Int1_DataIn <= Tsw_reg_o;--X"0001"; -- Emu_DL_Stat = 1
(Busy/Waiting for Trigger)
                Bus_Int1_WE <='1';

                NS <= S27;

        when S27 =>
                if(Bus_Int1_Busy = '1') then
                        NS <= S27;
                else

                        NS <=S28;
                end if;
                --Temp_Vrble_Data <= Bus_Int1_DataOut;
                --LD_Vrble_Data <= '1';

        when S28 =>
                --Temp_Trigger<= Vrble_Data;
                --LD_Trigger <= '1';



                NS <= S29;



        -- Pulling Emu Voltage Option data from RAM
        when S29=>
                if(Bus_Int1_Busy = '1') then
                        NS <= S29;
                else

                        NS <=S30;
                end if;
```

```
                when S30=>
                        Bus_Int1_AddrIn <= Addr_Emu_V_OP; --Addr_Emu_V_OP is a
constant from Common file
                        Bus_Int1_RE <='1';
                        NS <= S31;

                when S31 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S31;
                        else

                                NS <=S32;
                        end if;
                        Temp_Vrble_Data <= Bus_Int1_DataOut;
                        LD_Vrble_Data <= '1';

                when S32 =>
                        Temp_Emu_V_Type<= Vrble_Data(7 downto 0);
                        LD_Emu_V_Type <= '1';

                --- If needed more data saved to registers ---

                        --      <= Vrble_Data();
                        --LD_ <= '1';

                        --      <= Vrble_Data();
                        --LD_ <= '1';

                        --      <= Vrble_Data();
                        --LD_ <= '1';

                        NS <= S33;

----Sback1
                when S33 =>
                                                        -- This EN is the signal that Enables
VA, VB, VC Duty Cycle processes.
                                                        -- So if it is on, this means that the
Fsw and 1%Ton process has finished.
                                                        -- Next checks for Error or HP_EN
to make sure that it did not miss the EN.
                        --if(EN = '0')then
                                NS <= S34;
                        --else
                                Cnt_LeadReg_Rst <= '0';
-----Sb
```

```
                                --NS <= S35 ;
                        --end if;

                when S34 =>
                        --if(Error = '1')then
                                --NS <= S7;
                        --elsif(HP_EN = '1')then
                                --NS <= S58;
                        --else
                                --NS <= S35;
                        --end if;

                        if((Error = '0') and (HP_EN = '0'))then
------Sback1
                                --NS <= S33;
                                NS <= S35;
                        else
-----SE
                                NS <= S7;
                        end if;
        --------------------------- End pulling Emu DL data from RAM ------------------------
-------


        --------------------------- Start Trigger, Scaling, Emu Data Saving ---------------------
-----------
        --*May need to adjust the formatting of PreScale in LabVIEW, or in more States
HERE
                --when S =>
----Sb
                when S35  =>                                            --Scale Ref is
used during the Scale count delay when sampling
        -- FOR DEMO THE SCALE REF WILL BE Tsw_reg_o, BUT WILL CHANGE
IN FUTURE DESIGN --
                        --if(PreScale < Tsw_reg_o)then
                                Temp_Scale_Ref <= X"07D0";--Tsw_reg_o;
                        --else
        --*If PreScale in LabVIEW is formatted weird change Scale_Ref to Tsw and not
PreScale for Demo
                                --Temp_Scale_Ref <= PreScale;
                        --end if;
                        LD_Scale_Ref <= '1';
                        NS <= S36;
----Sc
                when S36 =>
                        if (Cnt_LeadReg_Out < X"C0") then
```

```
                                Cnt_Scale_INC <= '1';
                                NS <= S37;
                        else
                                Cnt_Scale_Rst <= '0';
                                --Temp_Va_Samp <= (others => '0');
                                --Temp_Vb_Samp <= (others => '0');
                                --Temp_Vc_Samp <= (others => '0');
                                --LD_Va_Samp <= '1';
                                --LD_Vb_Samp <= '1';
                                --LD_Vc_Samp <= '1';
-----Sd
                                NS <= S40;
                        end if;

                -- when S => -- These (Trigger setup and wait) states may need to be
added later.

                --Start PreScale Delay Counting--
                when S37 =>
                        if(Cnt_Scale_Out < Scale_Ref)then
                                Cnt_Scale_INC <= '1';
                                NS <= S37;
                        else

                                Cnt_Scale_Rst <= '0';
                                Temp_Va_Samp <= Va_DC*Vdc;
                                Temp_Vb_Samp <= Vb_DC*Vdc;
                                Temp_Vc_Samp <= Vc_DC*Vdc;
                                LD_Va_Samp <= '1';
                                LD_Vb_Samp <= '1';
                                LD_Vc_Samp <= '1';
                                NS <= S38;
                        end if;

                when S38 =>
                        Cnt_Scale_INC <= '1';
                        Cnt_LeadReg_INC <= '1';
                        NS <= S39;

                -- Start Saving Emu Va, Vb, Vc Data in FIFO--
                when S39 =>
                        if (STD_FIFO_Va_Full = '0') then
                                STD_FIFO_Va_DataIn <= Va_Samp(15 downto 0);
        --16 bit FIFO. DATA_WIDTH in FIFO must be 16 and not 8.
                                STD_FIFO_Va_WriteEn <='1';
                        end if;
```

```
                        if (STD_FIFO_Vb_Full = '0') then
                                STD_FIFO_Vb_DataIn <= Vb_Samp(15 downto 0);
        --16 bit FIFO. DATA_WIDTH in FIFO must be 16 and not 8.
                                STD_FIFO_Vb_WriteEn <='1';
                        end if;
                        if (STD_FIFO_Vc_Full = '0') then
                                STD_FIFO_Vc_DataIn <= Vc_Samp(15 downto 0);
        --16 bit FIFO. DATA_WIDTH in FIFO must be 16 and not 8.
                                STD_FIFO_Vc_WriteEn <='1';
                        end if;
                        Cnt_Scale_INC <='1';
------Sc
                        NS <= S36 ;


                --Check if Bus is Busy--
-----Sd
                when S40 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S40;
                        else

                                NS <=S41;
                        end if;
                -- Update Emu_DL_Status and save in RAM--
                when S41 =>
                        Bus_Int1_AddrIn <= Addr_Emu_DL_Status; --
Addr_Emu_DL_Status is a constant from Common file
                        Bus_Int1_DataIn <= X"0002"; -- Emu_DL_Stat = 1 (Saving Data)
                        Bus_Int1_WE <='1';
                        NS <= S42;

                -- Start counting Registers until 192 have saved--
-----Sf
                when S42 =>
                        if(Cnt_FollowReg_Out < X"C0")then

                                NS <= S43;
                        else
                                Cnt_FollowReg_Rst <= '0';
-----Si
                                NS <= S58;
                        end if;


                ---- Start Saving Emu Va, Vb, Vc Data from FIFO to RAM-----
```

```
                -- Va FIFO to RAM
                when S43 =>
                        if(STD_FIFO_Va_Empty = '1') then
-----Sg
                                NS<=S48;
                        else
                                STD_FIFO_Va_ReadEn <= '1';
                                NS<=S44;
                        end if;

                when S44=>
                        Temp_Vrble_Data<=STD_FIFO_Va_DataOut;
                        LD_Vrble_Data <='1';
                        NS<=S45;
                when S45=>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S45;
                        else
                                NS <=S46;
                        end if;

                when S46=>
                        Bus_Int1_AddrIn <= Addr_Va_Start + Cnt_FollowReg_Out;
        --Send Va data to RAM Addr X"0800" + Counter[1:192]
                        Bus_Int1_DataIn <= Vrble_Data;
                        Bus_Int1_WE <='1';
                        NS<=S47;

                when S47=>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S47;
                        else
                                Temp_Vrble_Data<=(others => '0');
                                LD_Vrble_Data <='1';
                                NS <=S48;
                        end if;




                -- Vb FIFO to RAM
------Sg
                when S48 =>
                        if(STD_FIFO_Vb_Empty = '1') then
```

```
-----Sh
                                        NS<=S53;
                                else
                                        STD_FIFO_Vb_ReadEn <= '1';
                                        NS<=S49;
                                end if;

                        when S49=>
                                Temp_Vrble_Data<=STD_FIFO_Vb_DataOut;
                                LD_Vrble_Data <='1';
                                NS<=S50;
                        when S50=>
                                if(Bus_Int1_Busy = '1') then
                                        NS <= S50;
                                else
                                        NS <=S51;
                                end if;

                        when S51=>
                                Bus_Int1_AddrIn <= Addr_Vb_Start + Cnt_FollowReg_Out;
        --Send Va data to RAM Addr X"0800" + Counter[1:192]
                                Bus_Int1_DataIn <= Vrble_Data;
                                Bus_Int1_WE <='1';
                                NS<=S52;

                        when S52=>
                                if(Bus_Int1_Busy = '1') then
                                        NS <= S52;
                                else
                                        Temp_Vrble_Data<=(others => '0');
                                        LD_Vrble_Data <='1';
                                        NS <=S53;
                                end if;




                        -- Vc FIFO to RAM
------Sh
                        when S53 =>
                                if(STD_FIFO_Vc_Empty = '1') then
                                        Cnt_FollowReg_INC <= '1';
------Sf
                                        NS<=S42;
```

```
                        else
                                STD_FIFO_Vc_ReadEn <= '1';
                                NS<=S54;
                        end if;

                when S54=>
                        Temp_Vrble_Data<=STD_FIFO_Vc_DataOut;
                        LD_Vrble_Data <='1';
                        NS<=S55;
                when S55=>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S55;
                        else
                                NS <=S56;
                        end if;

                when S56=>
                        Bus_Int1_AddrIn <= Addr_Vc_Start + Cnt_FollowReg_Out;
        --Send Va data to RAM Addr X"0800" + Counter[1:192]
                        Bus_Int1_DataIn <= Vrble_Data;
                        Bus_Int1_WE <='1';
                        NS<=S57;

                when S57=>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S57;
                        else
                                Cnt_FollowReg_INC <= '1';
                                Temp_Vrble_Data<=(others => '0');
                                LD_Vrble_Data <='1';
-------Sf
                                NS <=S42;
                        end if;
                ---------------------------- End Trigger, Scaling, Emu Data Saving ----------
------------------




                ---------- Start Resetting RAM Regs for Trigger, Scaling, Start Emu DL,
Emu Vout OP --------------
```

```
------Si
                        when S58 =>

                                Bus_Int1_AddrIn <= Addr_Emu_DL_Status; --
Addr_Emu_DL_Status is a constant from Common file
                                Bus_Int1_DataIn <= X"0000"; -- Emu_DL_Stat = 0 (Ready/Done)
                                Bus_Int1_WE <='1';
                                NS <= S59;

                        when S59 =>
                                if(Bus_Int1_Busy = '1') then
                                        NS <= S59;
                                else

                                        NS <=S60;
                                end if;




                                -- Reset Emu DL Start RAM Reg
                        when S60 =>
                                Bus_Int1_AddrIn <= Addr_Emu_DL_Start; --
Addr_Emu_DL_Start is a constant from Common file
                                Bus_Int1_DataIn <= X"0000";
                                Bus_Int1_WE <='1';
                                NS <= S61;

                        when S61 =>
                                if(Bus_Int1_Busy = '1') then
                                        NS <= S61;
                                else

                                        NS <=S62;
                                end if;
                                Temp_Emu_DL_Start <= '0';
                                LD_Emu_DL_Start <= '1';


                        -- Reset Emu Vout OP RAM Reg
                        when S62 =>
                                Bus_Int1_AddrIn <= Addr_Emu_V_OP; --Addr_Emu_V_OP is a
constant from Common file
                                Bus_Int1_DataIn <= X"0000";
                                Bus_Int1_WE <='1';
                                NS <= S63;
```

```
                when S63 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S63;
                        else

                                NS <=S64;
                        end if;
                        Temp_Emu_V_Type <= (others => '0');
                        LD_Emu_V_Type <= '1';


                -- Reset Emu Prescale RAM Reg
                when S64 =>
                        --Bus_Int1_AddrIn <= Addr_Emu_Prescale; --
Addr_Emu_Prescale is a constant from Common file
                        --Bus_Int1_DataIn <= X"0000";
                        --Bus_Int1_WE <='1';
                        NS <= S65;

                when S65 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S65;
                        else

                                NS <=S66;
                        end if;
                        Temp_PreScale <= (others => '0');
                        LD_PreScale <= '1';


                -- Reset Emu Trigger RAM Reg
                when S66 =>
                        --Bus_Int1_AddrIn <= Addr_Emu_Trigger; --Addr_Emu_Trigger
is a constant from Common file
                        --Bus_Int1_DataIn <= X"0000";
                        --Bus_Int1_WE <='1';
                        NS <= S67;

                when S67 =>
                        if(Bus_Int1_Busy = '1') then
                                NS <= S67;
                        else

                                NS <=S0;
                        end if;
                        Temp_Trigger <= (others => '0');
```

```
                        LD_Trigger <= '1';

                    ---------- End Resetting RAM Regs for Trigger, Scaling, Start Emu DL,
Emu Vout OP ----------


---------------------------------------- End Emu Data Logging ----------------------------------------

                    when others =>
                        NS <= S0;
            end case;
        end process;

------------ ** EMU DATA DISTRIBUTION GOES HERE, BUT WILL BE EDITED IN
SEPARATE FILE FOR SIMPLICITY AND THEN COPED HERE **









        ---- Sync SW signal Inputs before applying to any process ----
        SW_Signal_Sync: process
        begin
            wait until clk'event and clk = '1';
            SW01_t <= Emu_SW01;
            SW01   <=   SW01_t;

            SW03_t <= Emu_SW03;
            SW03   <=   SW03_t;

            SW05_t <= Emu_SW05;
            SW05   <=   SW05_t;
```

```
	end process;



	----State Sync
	sync_States: process
	begin
		wait until clk'event and clk = '1';
		if rst = '0' then
			CS		<= S0;
			CSA	<= S0;
			CSB	<= S0;
			CSC	<= S0;
			CS_Fsw	<= S0;
			CSab	<= S0;
			CSbc	<= S0;
			CSca	<= S0;
		else
			CS		<= NS;
			CSA	<= NSA;
			CSB	<= NSB;
			CSC	<= NSC;
			CS_Fsw	<= NS_Fsw;
			CSab	<= NSab;
			CSbc	<= NSbc;
			CSca	<= NSca;
		end if;
	end process;
	----End State Sync


end Behavioral;
```

A-2-6 Hot-Patch

```
----------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Estefano Soria
--
-- Create Date:		26Dec2020
-- Design Name:		DSP_Hot_Patch
-- Module Name:		DSP_Hot_Patch - Behavioral
-- Project Name:		DSP_Hot_Patch
-- Target Devices:		LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:		Lattice Diamond_x64 Build 3.11
-- Description:
```

```
--
---- PinOut:
--
-- Revision
--
--
--
-- Additional Comments:
--
--
----------------------------------------------------------------------------


Library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library machxo2;
use machxo2.all;

library work;
use work.Test2_DT_Common.all;

entity Test2_DT_HP_Ctrl is
   Port (

                                clk : in std_logic;
                                rst : in std_logic;
                                EN : in std_logic;
                                DSP1_Act_Out : out std_logic;
                                DSP_Sync : out std_logic;
                                Done : out std_logic



                                        );
end Test2_DT_HP_Ctrl;

architecture Behavioral of Test2_DT_HP_Ctrl is

        type state_type is (S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16);
        signal CS, NS : state_type;

        signal DSP1_Act_Temp : std_logic;
        signal DSP1_Act : std_logic;
```

```
	signal LD_Done : std_logic;
	signal Done_Temp : std_logic;
	signal DSP1_Act_Out_Temp : std_logic;
	signal LD_DSP1_Act_Out : std_logic;
	signal LD_DSP1_Act : std_logic;
	signal LD_DSP_Sync     : std_logic;
	signal Temp_DSP_Sync   : std_logic;

	--signal                    DSP_Sync   : std_logic;
	signal Cnt_Sync_INC : std_logic := '0';
	signal Cnt_Sync_Rst : std_logic := '0';
	signal Cnt_Sync_Out : std_logic_vector(31 downto 0) := (others => '0');
	-- Declare Std_Counter Component
	component Std_Counter is
	generic
	(
		Width : integer                    -- width of counter
	);
	port(INC,rst,clk: in std_logic;
		 Count: out STD_LOGIC_VECTOR(Width-1 downto 0));
	end component;


begin

	-- Instantiate Reset_Cnt_8
	Sync_Cnt: Std_Counter
	generic map
	(
		Width => 32
	)
	port map(
		clk => clk,
		rst=> Cnt_Sync_Rst,
		INC=> Cnt_Sync_INC,
		Count=> Cnt_Sync_Out
	);

	----Registers
	Reg_Proc: process
	begin
		wait until clk'event and clk = '1';
		if (rst = '0') then

			DSP1_Act <= '1';
			DSP1_Act_Out <= '1'; --When rst is pressed, DSP1 is active by default
```

```
                Done <= '0';


        else
                if (LD_Done = '1') then Done <= Done_Temp; end if;
                if (LD_DSP1_Act = '1') then DSP1_Act <= DSP1_Act_Temp; end if;
                if (LD_DSP1_Act_Out = '1') then DSP1_Act_Out <=
DSP1_Act_Out_Temp; end if;
                if (LD_DSP_Sync = '1') then DSP_Sync <= Temp_DSP_Sync; end if;

        end if;
    end process;
    ----End Registers



    --Next State Logic for Hot-Patch
    NSL_HP:process(CS, EN)--, DSP1_Act)
    begin
        Done_Temp <= '0';

        LD_Done <= '0';


        DSP1_Act_Out_Temp <= '0';
        LD_DSP1_Act_Out <= '0';

        Temp_DSP_Sync <= '1';--Active Low
        LD_DSP_Sync <= '0';

        DSP1_Act_Temp <= '0';
        LD_DSP1_Act <= '0';

        Cnt_Sync_INC <= '0';
        Cnt_Sync_Rst <= '1';


        case CS is

            when S0 =>
                if (EN = '0') then
                    NS <= S0;
                else
                    NS <= S1;
                end if;
                Done_Temp <= '0';
```

```
                LD_Done <= '1';
                Temp_DSP_Sync <= '1';--Active Low
                LD_DSP_Sync <= '1';
                Cnt_Sync_Rst <= '0';
            when S1 =>

                DSP1_Act_Temp <= not DSP1_Act;
                LD_DSP1_Act <= '1';

                --Temp_DSP_Sync <= '0';--Active Low
                --LD_DSP_Sync <= '1';
                NS <= S2;

            when S2 =>
                if(Cnt_Sync_Out < X"0000067E")then -- Counter for 2 Switching
cycles (30kHz)
                    Cnt_Sync_INC <= '1';
                    NS <= S2;
                else
                    Cnt_Sync_Rst <= '0';
                    NS <= S3;
                end if;
                Temp_DSP_Sync <= '0';--Active Low
                LD_DSP_Sync <= '1';

                Done_Temp <= '0';
                LD_Done <= '1';
                --Temp_DSP_Sync <= '1';--Active Low
                --LD_DSP_Sync <= '1';
                --NS <= S3;
            when S3 =>
                Temp_DSP_Sync <= '1';--Active Low
                LD_DSP_Sync <= '1';
                NS <= S4;

            when S4 =>
                if(Cnt_Sync_Out < X"000CAE18")then -- Counter for 2
fundamental cycles (60Hz) CAE18
                    Cnt_Sync_INC <= '1';
                    NS <= S4;
                else
                    Cnt_Sync_Rst <= '0';
                    NS <= S5;
                end if;
                Temp_DSP_Sync <= '1';--Active Low
                LD_DSP_Sync <= '1';
```

```
                when S5 =>
                        DSP1_Act_Out_Temp <= DSP1_Act;
                        LD_DSP1_Act_Out <= '1';

                        NS <= S6;
                when S6 =>
                        Done_Temp <= '1';
                        LD_Done <= '1';
                        NS <= S7;

                when S7 =>
                        if (EN = '1')then
                                Done_Temp <= '1';
                                LD_Done <= '1';
                                NS <= S7;
                        else
                                Done_Temp <= '0';
                                LD_Done <= '1';
                                NS <= S0;
                        end if;

                when others =>
                        NS <= S0;
        end case;
    end process;



    ----State Sync
    sync_States: process
    begin
        wait until clk'event and clk = '1';
        if rst = '0' then
                CS <= S0;
        else
                CS <= NS;
        end if;
    end process;
    ----End State Sync


end Behavioral;
```