8-2023

A Real-Time ANPC Inverter Digital Twin with Integrated Design-For-Trust

Paulo Vitor Do Amaral Custodio
University of Arkansas, Fayetteville

Citation
Do Amaral Custodio, P. V. (2023). A Real-Time ANPC Inverter Digital Twin with Integrated Design-For-Trust. *Graduate Theses and Dissertations* Retrieved from https://scholarworks.uark.edu/etd/5035

A Real-Time ANPC Inverter Digital Twin with Integrated Design-For-Trust

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

by

Paulo Vitor do Amaral Custodio
State University of Londrina
Bachelor of Science in Electrical Engineering, 2015

August 2023
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

H. Alan Mantooth, Ph.D.
Committee Member

Roy A. McCann, Ph.D.
Committee Member

Chis Farnell, Ph.D.
Committee Member

ABSTRACT

The demand for renewable energy has increased over the last few years, and so has the demand for greater expectations within the energy market. This increasing trend has been accompanied by more significant usage of internet-connected devices (IoT), leading to critical electrical infrastructure being connected to the internet. Implementing internet connectivity with such devices and systems provides benefits such as improving the system's performance, facilitating irregularity and anomaly mitigation, and providing additional situational awareness for enhanced decision-making. However, enhancing the connected system with IoT introduces a drawback – a greater vulnerability to cyber-attacks.

Cyber-attacks targeting critical infrastructure in the electrical sector have occurred in the United States and Ukraine. These cyber-attacks highlight and expose vulnerabilities that a system inherits when connecting to the internet. These attacks left thousands of customers without electricity for hours until operators could regain control of the electric utility grid.

Therefore, to address the vulnerabilities of an internet-connected power electronic device, this work focused on the hardware layer of the system. Implementing a cyber-control system inside the hardware layer can significantly reduce the possibility of an attacker patching malicious controller firmware into a photovoltaic grid-connected inverter, thus mitigating the likelihood that the inverter becomes inactive a cyber-attack scenario. With this mitigation technique, if a cyber-attack is successful and an attacker gains control of the network, a cyber-defense technique is in place to mitigate the impact of the cyber-attack.

This additional protection layer was developed based on an innovative concept known as Digital Twin (DT). A DT, in this case, replicates an Active-Neutral Point Clamped (ANPC) inverter and was designed using a hardware language known as VHDL (Very High-Speed

Integrated Circuit Hardware Description Language) and applied to Field-Programmable-Gate-Array (FPGA). The DT is embedded within the FPGA and contained in a controller board, the UCB (Unified Controller Board), developed by the University of Arkansas electrical engineering team. This UCB also contains two Digital Signal Processors (DSPs) responsible for generating associated signals to control an authentic physical inverter. These DSP signals are received and processed by the FPGA that implements the DT of an ANPC; in other words, it simulates in real-time the expected output of an actual ANPC inverter using the signals from the DSP.

When a new firmware is ready to be patched, the DT provides output signals simulating behavior that a real ANPC inverter would generate with the new firmware. The new firmware is tested to check if it meets all the operational requirements established using a Design-For-Trust technique (DFTr). If the new firmware fails in at least one of the DFT tests, it is considered malicious and must be rejected.

This work is divided into sections, such as Background, which explains the pieces that were used and the strategy behind this work; Process and Procedure, which explains the methodology that was adopted to prove the reliability and effectiveness of this work; Results and Discussion, where the simulations and results are described and explained; followed by Conclusion and Future work section, which concludes this work and adds possible future projects to continue this work further.

ACKNOWLEDGEMENTS

First, I would like to thank God for giving me good health, courage, wisdom, and perseverance to follow my dreams and to be able to pursue my master's degree. Second, I want to thank my family for their support and guidance and for always believing in myself and pushing me forward. I would also like to thank my wife, Ana Paula Garcia, who always stood by my side and gave me all the support I needed. I am also very grateful for the opportunity to work with Dr. H. Alan Mantooth, Dr. Chris Farnell, and Dr. Le-Vasicek. Without them, this project would not exist, and the knowledge and the experiences they provided to me during this time we worked together are unforgettable and priceless. Lastly, I would like to especially thank Brady McBride and Justin Jackson for all their support and knowledge during the creation of this work, their help and partnership made my master's degree journey much lighter and exciting.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AC	Alternative Current
CFM	Configuration Flash Memory
CLBs	Configurable Logic Blocks
DFTr	Design-For-Trust
DNS	Domain Name System
DC	Direct Current
DER	Distributed Energy Resource
DoS	Denial-of-Service
DSP	Digital Signal Processor
DT	Digital Twin
EBR	Embedded Block RAM
FFs	Flip-Flops
FPGA	Field-Programmable-Gate-Array
GUI	Graphic User Interface
IEA	International Energy Agency
IoT	Internet-of-Things
I²C	Inter-Integrated Circuit
JTAG	Joint Test Action Group
LUT	Look-up table
MitM	Man-in-the-Middle
MUX	Multiplexer
NERC	North American Electric Reliability Corporation

NVM	Non-volatile Memory
NPC	Neutral Point Clamped
PFUs	Programmable Function Units
PLL	Phase Lock Loop
PV	Photovoltaic
PWM	Pulse Width Modulation
RAM	Random Access Memory
SRAM	Static random-access memory
SPWM	Sinusoidal Pulse Width Modulation
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver and Transmitter
UCB	Unified Controller Board
UI	User Interface
VHDL	Very High-Speed Integrated Circuit Hardware Description Language

CHAPTER 1

INTRODUCTION

Advances in technology have led to significant improvements in computing power while reducing overall device size and increasing availability and speed of communication. These developments have led to a dramatic increase in internet-connected devices, including IoTs. While IoTs offer many benefits, such as increased situational awareness, they also create new vulnerabilities for cyber-attacks to exploit.

In 2015, Ukraine experienced a significant power outage when a cyber-attack resulted in the disconnection of twenty-three 35kV and seven 110kV substations for three hours. The attack was initiated using a phishing technique and resulted in power loss for 225,000 customers. Similarly, in 2016, part of the capital city of Ukraine, Kyiv, was left without electricity for over an hour due to a cyber-attack [1].

In response to these emerging threats, in 2017, the US president signed a bill to increase the cybersecurity of federal networks and critical infrastructure. This order highlighted the risks of "electricity disruption" caused by cyber-attacks. It is essential to address these risks and intensify efforts to improve cybersecurity and protect against cyber-attacks capable of causing significant disruption to critical infrastructure [2].

In addition, according to the International Energy Agency (IEA), the potential to produce energy via renewable technologies, such as wind and solar power, is expected to increase around 60% of the renewable electricity capacity by 2026, making renewables the primary source responsible for almost 95% of this increase, with more than half coming from solar photovoltaic (PV) by itself, consequently becoming “the powerhouse of growth in renewable electricity” [3].

Therefore, this work applied the modern idea of utilizing Digital Twin to lessen the vulnerabilities of internet-connected devices in light of the growing need for renewable energy and the rising application of internet-connected devices. As [4] presented a cyber protection system for grid-connected devices using embedded systems, this work proposed an improvement to the hardware layer of the cyber-physical devices - creating a Digital Twin of an ANPC inverter within a custom controller, which was the same controller used in [4].

The controller contains a Field Programmable Gate-Array (FPGA), which emulates a 3-level ANPC inverter, instead of a 2-level inverter, as was cited in [4]. Additionally, it has two Digital Signal Processors (DSPs): one is used to control an actual inverter – called "Active DSP"; while the other is used to create a Digital Twin and authenticate a new firmware before patching it – called "Stand By DSP." In this case, the authentication method is called Design-For-Trust (DFTr). The purpose of utilizing this technique is to prevent malicious firmware from being installed or updated inside grid-connected inverters used within solar distributed energy resources (DERs). Such malicious firmware can potentially carry out a cyber-physical attack on the DERs, which can have serious consequences, such as shutting the grid down. DFTr ensures that the firmware installed on these inverters is trustworthy and free of malicious code, thereby reducing the risk of cyber-physical attacks.

CHAPTER 2

BACKGROUND

2.1 Cyber-Attacks

Cyberattacks usually concentrate on revealing the weaknesses of the communications layer. Cyber attackers connect to the network using a variety of techniques, including phishing, Man-in-the-Middle (MitM) attacks, Denial of Service (DoS), SQL injection, Domain Name System (DNS) tunneling, and more, to obtain access to the communication layer. Attackers that take over the communications section can send the controller malicious commands or software, which could damage the power electronic device [4]. Several cybersecurity techniques are specific to the grid for determining the primary forms of attack vectors and performing risk evaluations. Since the communications layer is the first point of interaction with the system, most cybersecurity techniques concentrate on protecting it. However, new system vulnerabilities are continually being found, raising serious concerns about the grid's dependability [5].

Grid vulnerabilities are a serious threat because they allow cyberattacks to take down the power grid in an entire nation or city, as was the case with Ukraine strikes in 2015 and 2016 [6]. Recently, a ransomware attack shut down pipeline operations on the Colonial Pipeline in the southeast of the United States [7]. Ransomware programs have caused several cyberattacks that shut down physical activities in 2020, as discussed in [7], highlighting the significant need for cybersecurity. Cybersecurity must now be incorporated into the design of power electronics control systems to decrease the electric grid's vulnerability to cyberattacks that target the communication network [4]. If an attacker successfully takes control of the communications layer, they can manipulate the controller, hardware layer, and other layers. Fig. 1 below displays a graphical depiction of these layers.

For example, when malicious firmware is uploaded to the controller, it might force the grid-connected device to shut down or start operating with suboptimal settings. This attack might not shut down the entire grid but a portion of it, similar to what happened during the attacks targeting Ukraine in 2015 and 2016. This work presents a technique for further protecting grid-connected devices that use the Supervisory, Control, and Hardware layers.

This project was designed to address a scenario where an attacker had already taken control of the network. The objective was to enhance the security of the power electronics controller in order to safeguard the grid operation. In case of an attack, a compromised controller would issue a command to shut down the system. The suggested method does not allow unauthorized firmware updates that could compromise the controller board. Moreover, a validated backup firmware replaces the compromised firmware without disrupting the ongoing system control. This technique ensures that the system will not crash during an attack, enhancing its resilience and security.

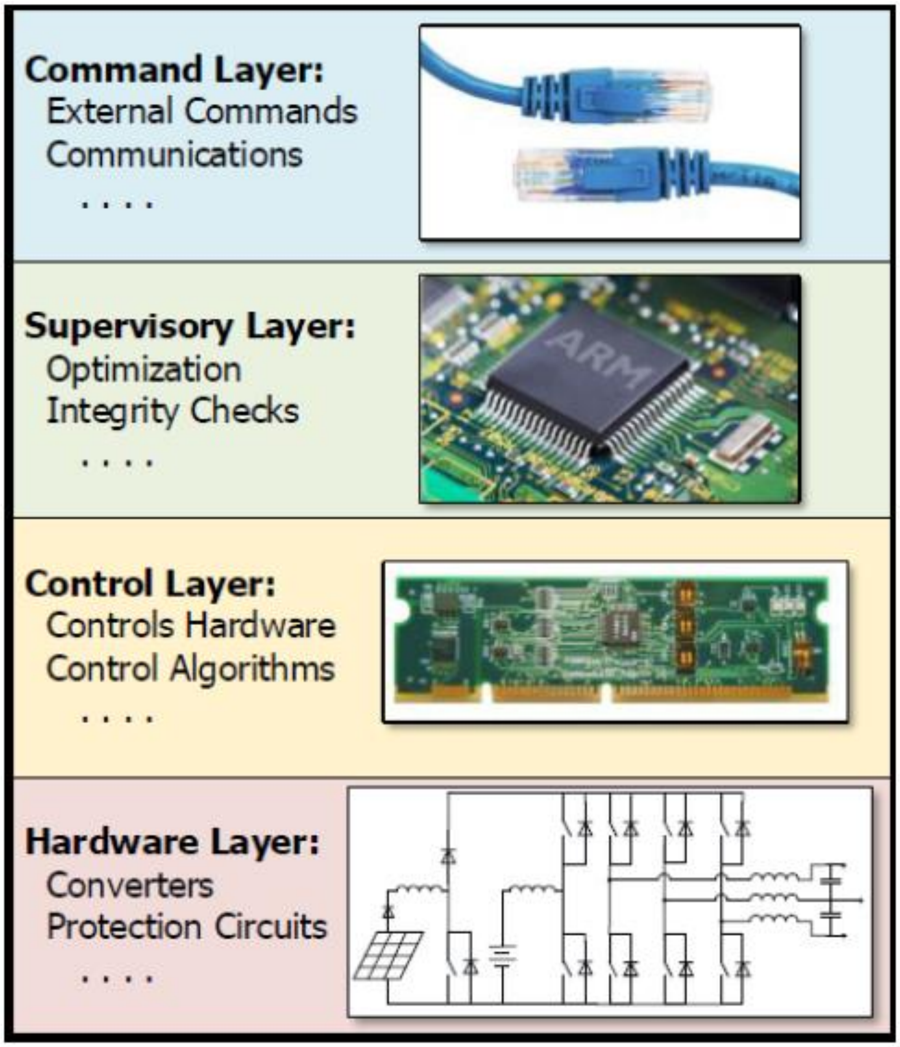


Fig. 1 Cyber-physical layer representation [2].

2.2 Photovoltaic Systems

The conversion process of light (photons) into electricity (voltage) is known as the photovoltaic effect. This effect gives the field of photovoltaics (often abbreviated as PV) its name. The significance of this effect was first demonstrated in 1954 by researchers at Bell Laboratories, who built a silicon solar cell capable of generating electric current upon exposure to light. Since then, the development of photovoltaic systems has progressed significantly. Due to their increasing economic viability, they are now widely installed and used on a large scale to help power electric grids [8].

In transmission and distribution networks, almost all power is provided as alternating current (AC), while the photovoltaic cells produce Direct Current (DC), the same type of current provided by batteries. To connect solar-power systems to the grid, inverters, and other components, shown in Fig. 2, necessary to connect a solar power plant to the grid, are utilized to convert DC to AC power [9].

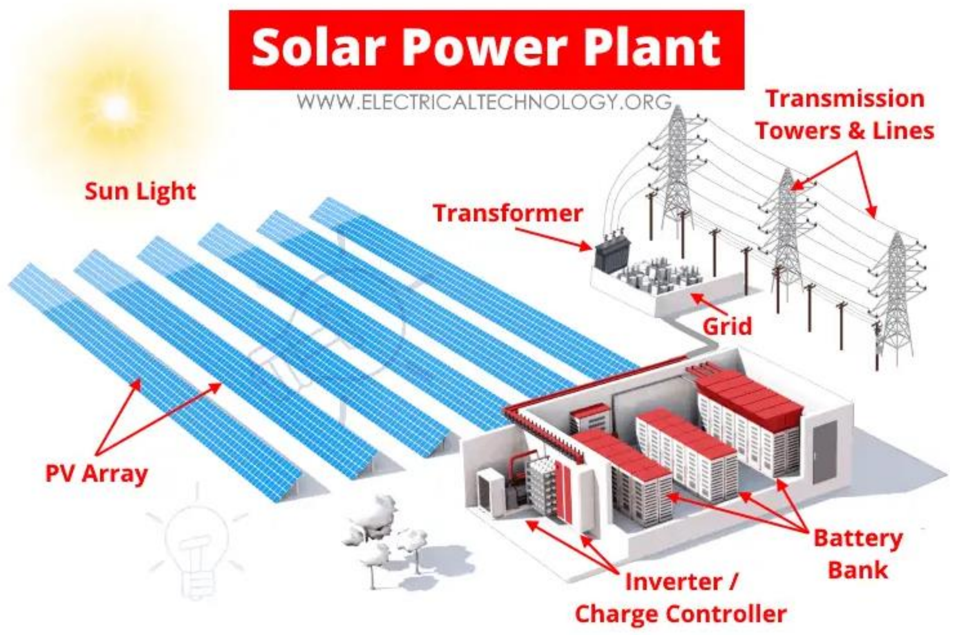

Solar Power Plant
WWW.ELECTRICALTECHNOLOGY.ORG
Transmission Towers & Lines
Sun Light
Transformer
Grid
PV Array
Battery Bank
Inverter / Charge Controller

Fig. 2. Solar Power Plant [10]

2.3 ANPC Inverter

Neutral-point-clamped (NPC), capacitor-clamped, and cascaded H-bridge inverters are just a few examples of inverters commonly used in PV systems [11]. These power converters, known as inverters, take a DC link supply as input. Using Pulse Width Modulation (PWM) signals, it controls its output to generate a three-phase sinusoidal AC with each phase offset by 120 degrees from the other. The whole system is depicted in Fig. 3 as having four sections:

- DC input voltage supply: Representing PV arrays from Fig. 2.
- An inverter: Considering a 3-level ANPC inverter in this case.
- A three-phase filter.
- Three-phase load.

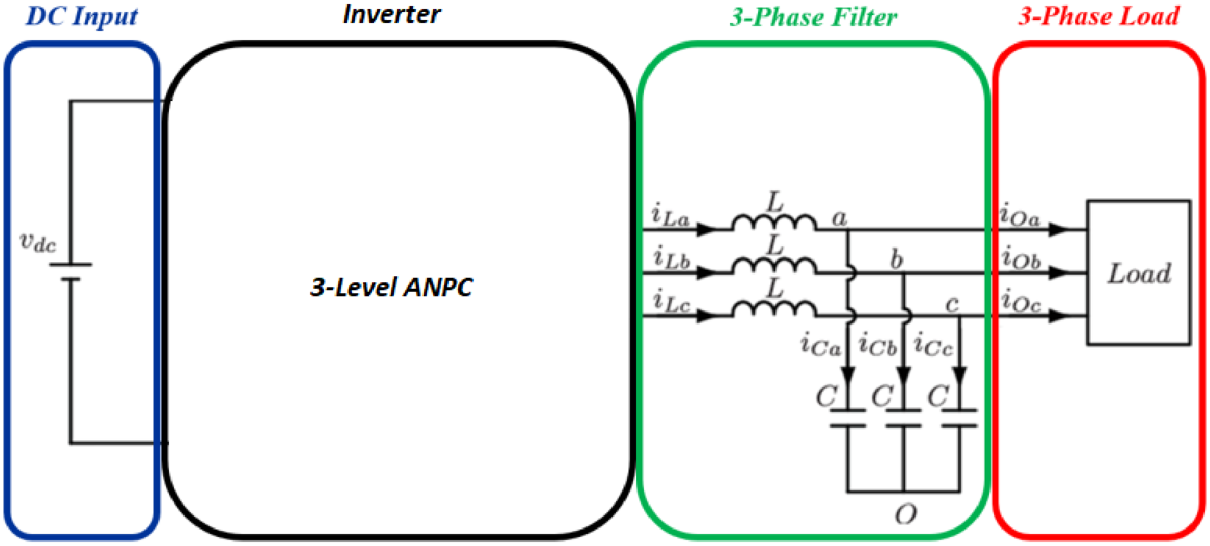
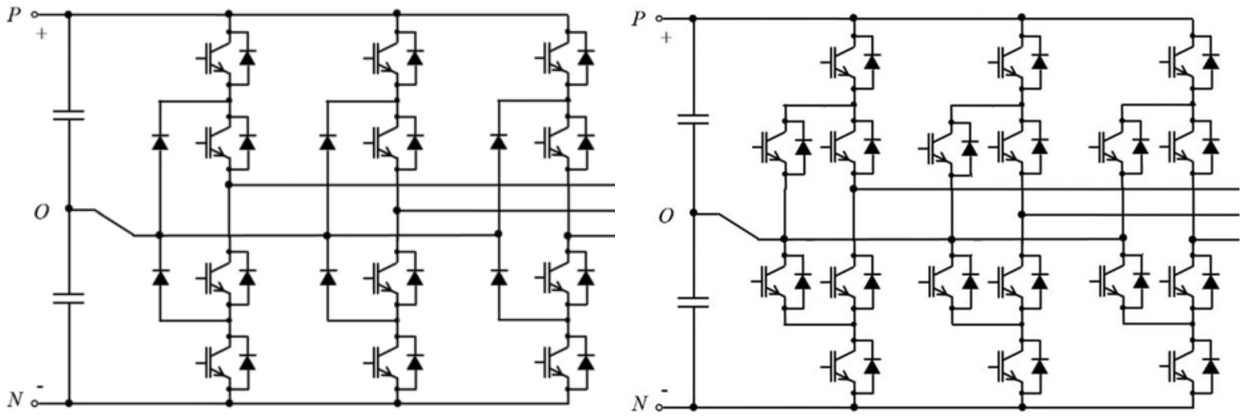


Fig. 3. Typical representation of a 3-level ANPC inverter hardware

The NPC inverter design can handle higher voltage levels using semiconductor components with lower voltage ratings while generating fewer harmonics in its output, and it is appealing for use in high-power applications. The semiconductor devices used in this inverter architecture have a rating of half the input DC bus voltage. Although, the unbalanced loss distribution across its semiconductor components is a drawback of the NPC inverter [12] [11].

Conversely, this drawback is resolved by the ANPC inverter design. Due to the two redundant neutral current pathways in this architecture, semiconductor device losses may be balanced regardless of the load power factor [13], [14]. Additionally, it needs low voltage-rated semiconductor components for high voltage applications, just like the three-level neutral-point clamped inverter structure. As a result, it is a highly appealing option for applications requiring high-power energy conversion [12] [11].

The NPC design (Fig. 4a) involves twelve switches (four for each phase) and six clamping diodes (two for each phase), while the ANPC design uses 18 switches, with six switches for each phase, as Fig. 5 illustrates. With these six transistors, it became possible to manipulate more switches involved in the design, increasing the number of possible modulation strategies that could be used to enhance the ANPC performance, as presented in work [12].



(a) (b)

Fig. 4. Inverters: (a) NPC; (b) ANPC

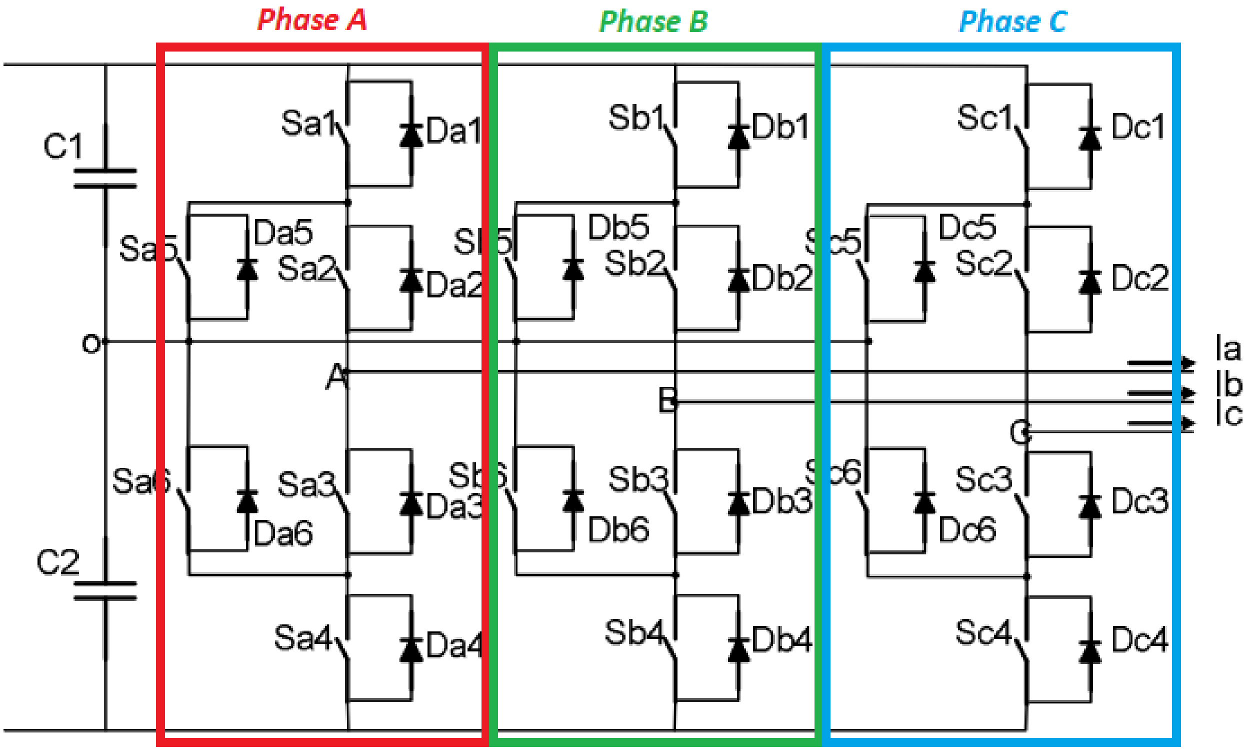


Fig. 5. Three-level ANPC inverter topology

This project is centered on implementing a Digital Twin of an ANPC (Active Neutral Point Clamped) inverter, as this inverter type was incorporated with the controller of a solar farm during testing. In light of the growing significance of photovoltaic energy in the renewable energy sector, three-level inverter topologies have gained prominence over two-level inverters, owing to their distinct advantages, such as lowered switching loss, diminished electromagnetic interference, and reduced harmonic content in the output current waveform. These benefits are characteristic of three-level inverter topologies, setting them apart from their two-level counterparts [12].

2.4 Modulation

Pulse-width modulation (PWM) is a crucial component of power electronic converters that was initially proposed to facilitate the production of sinusoidal AC voltage and current by inverters. Despite being suggested over 60 years ago, in 1964, PWM continues to be widely used with the rise of advanced power electronic converters and growing requirements for superior output voltage and current. PWM remains a significant subject of exploration in the realm of power electronics, captivating the curiosity and enthusiasm of researchers and scholars. The ongoing interest in PWM reflects its continued relevance and importance in enabling the efficient and effective use of power electronic converters in a range of applications [15].

The effectiveness and reliability of the inverter can be affected immediately by the switching frequency of the PWM technique. Increasing the switching frequency can lead to a lower distortion rate in the inverter's AC output current, as well as a decrease in the size and capacity of the filter inductor and capacitor. However, increasing the switching frequency also results in higher switching losses and greater performance demands on the switching device [15].

In order to maintain the output voltage of the single-phase inverter at a specific level, it is necessary to apply a control signal that will activate the inverter switches, and a PWM is commonly used for this purpose (Fig. 6). PWM signals have two main variables:

- Duty-Cycle: Also known as “On time,” it is the length for which the switch is in operation (On).
- Switching period: sum of the on-time and the off-time - duration time.

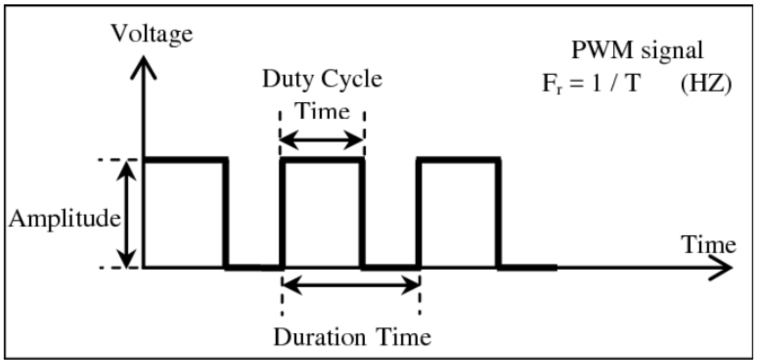


Fig. 6. Pulse Width Modulation [16]

The PWM generation is frequently based on comparing a low-frequency sine wave signal to a high-frequency carrier signal, which is usually a triangular method known as SPWM (Sinusoidal Pulse-Width Modulation). The fundamental concept behind natural sampling SPWM involves the comparison of a sinusoidal modulating voltage with a high-frequency triangular carrier wave. This comparison generates a rectangular pulse sequence whose width follows the sinusoidal law, as represented by Fig. 7 It is then power amplified and used to drive the inverter, ultimately producing a sinusoidal voltage or current output. [15].

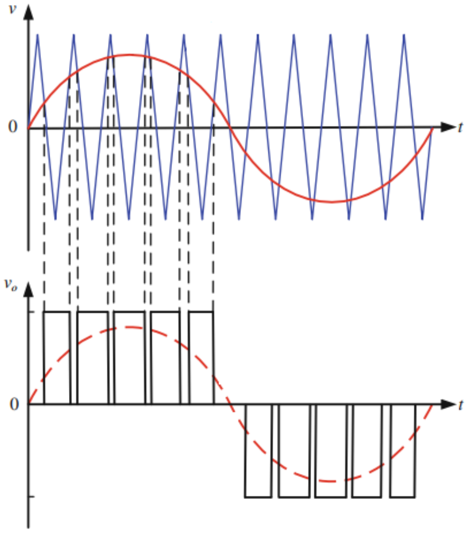


Fig. 7. SPWM generation principle

The voltage output of an unfiltered single-phase inverter is half of the DC input voltage during the on-time. However, the filtered output voltage is limited to a certain percentage of the DC input voltage. As the on-time approaches the maximum limit of 100%, the filtered output voltage increases proportionally. When the on-time is 100%, the filtered output voltage equals 50% of the DC input voltage, equivalent to half of the DC input voltage. The same line of principle is applied to the negative side, generating an output voltage as a sinusoidal waveform, which symbolizes an AC output, by raising and lowering the on-time, as Fig. 8 portrayed.

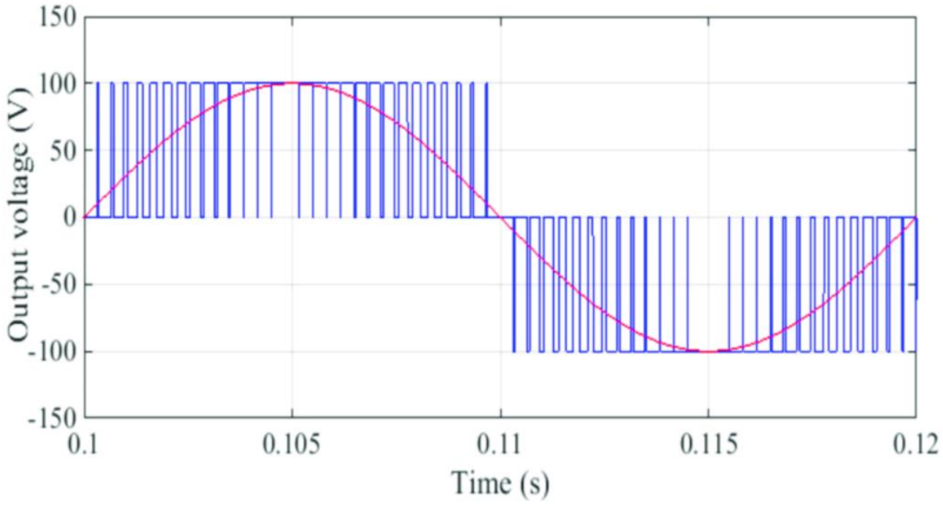


Fig. 8. Ideal PWM inverter output voltage [17]

2.4.1 Modulation: 3-level ANPC Inverter

A three-level ANPC inverter possesses eighteen transistors, allowing several strategies to improve the inverter's performance using different transistor types or modulation, as proposed in [12]. As this work was put into practice, an authentic ANPC inverter provided by SMA was used. As another group chose the modulation in the same project, modulation type two was selected from [12], where the external switching devices (Q2 and Q3) commutate at the carrier frequency.

In contrast, the inner transistors commutate at the US's fundamental line frequency - 60 Hz – as presented in Fig. 9.

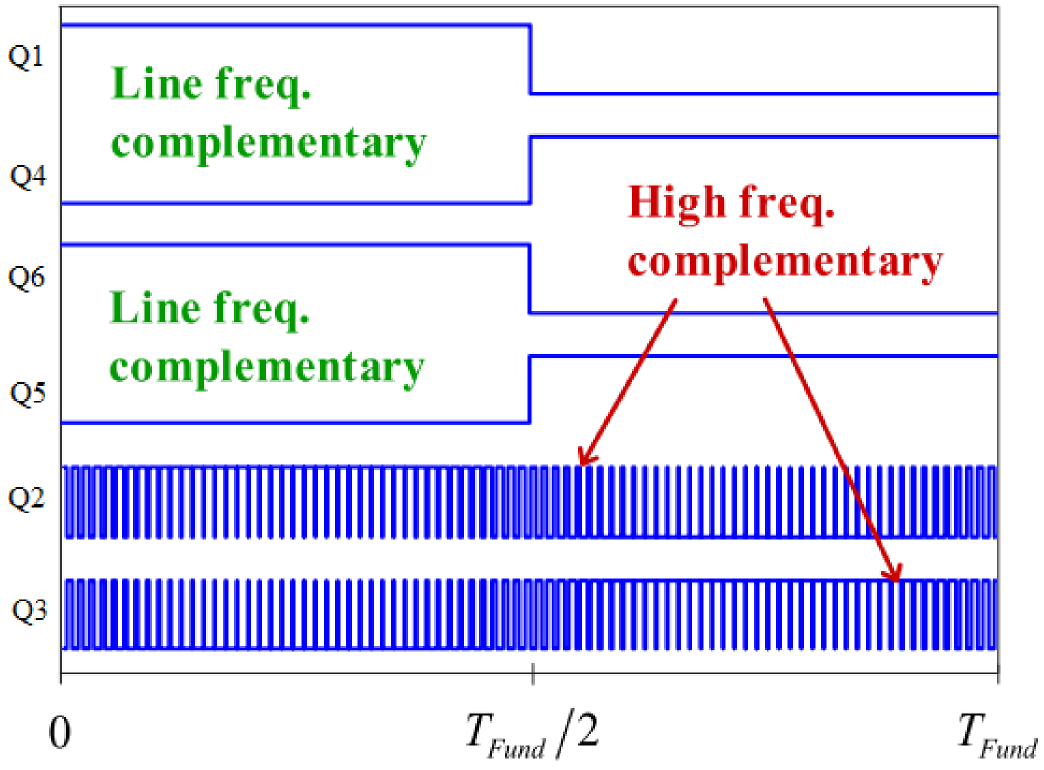


Fig. 9. Gate signals for modulation type II [12]

In order to simplify the explanation of how the ANPC inverter works, Fig. 10 illustrates one phase leg of the inverter. Each transistor can be considered a switch that can be turned on and off, and depending on the state of each switch, the output might change. Considering non-malicious states only, Table 1 represents each transistor's possible output and states, where Vdc is the DC input from the system (Fig. 3), while Fig. 11 illustrates the current path on each of these states.

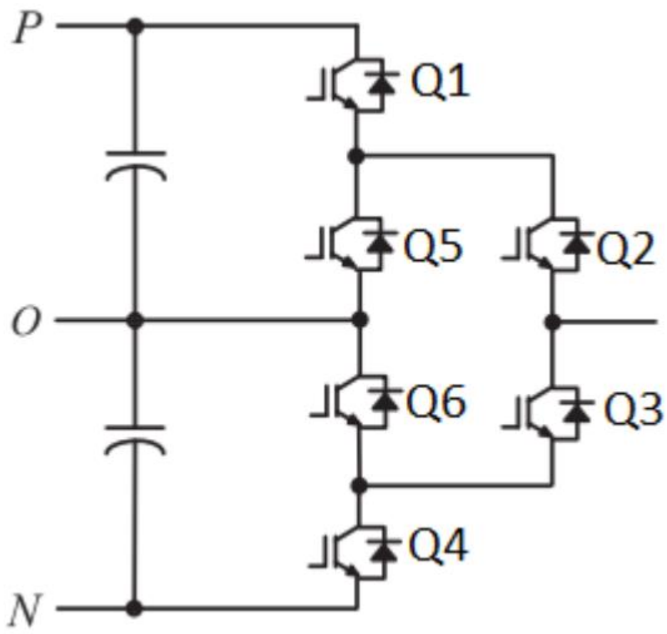
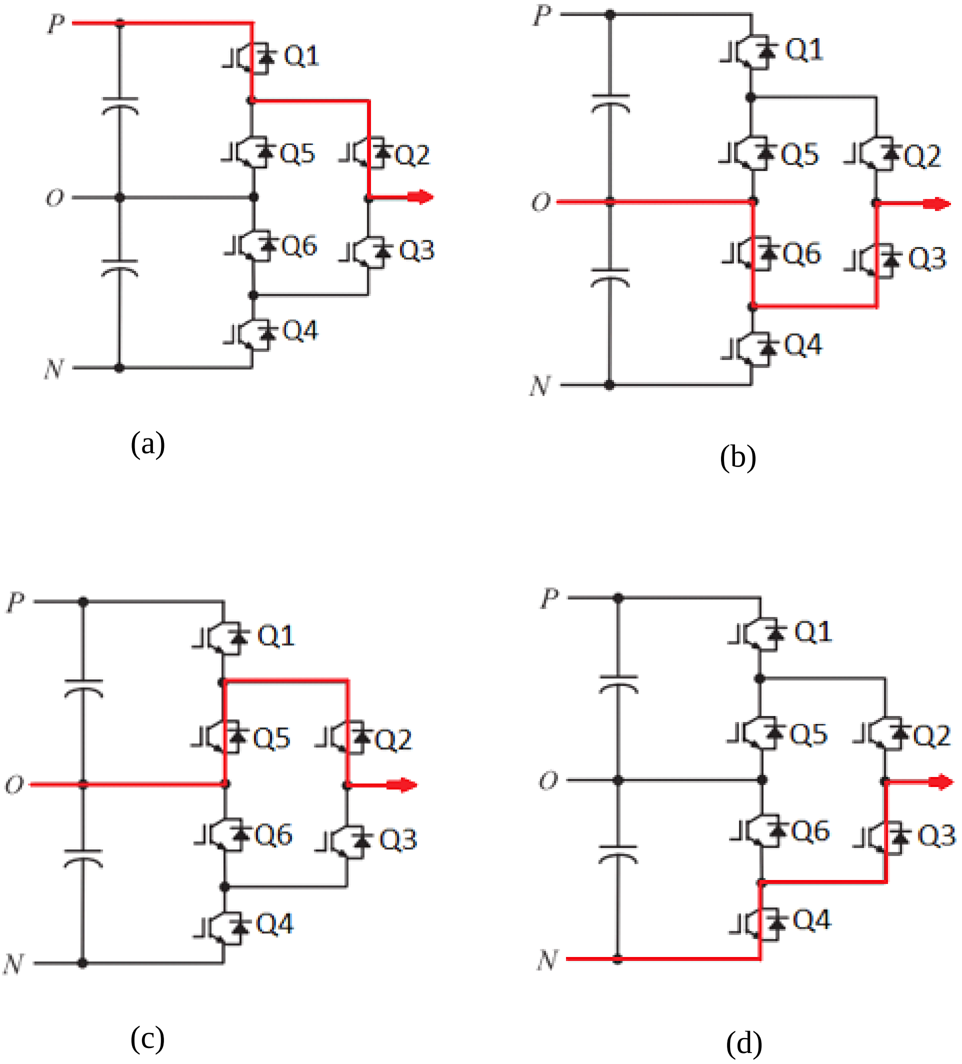


Fig. 10. Phase leg of a three-level ANPC inverter topology [2]

Table 1. Switch states modulation type II [12]

State	Output	Q1	Q2	Q3	Q4	Q5	Q6
P	0.5Vdc	1	1	0	0	0	1
O^+	0	1	0	1	0	0	1
O^-	0	0	1	0	1	1	0
N	-0.5Vdc	0	0	1	1	1	0



(a) (b) (c) (d)

Fig. 11. Switching states: (a) P state, (b) O+ state, (c) O- state, (d) N state

During the P state, transistors Q1 and Q2 are turned on, allowing the positive half of the DC input voltage to reach the output, generating $\frac{Vdc}{2}$ Volts on the output Fig. 11(a). Meanwhile, the O+ state, a zero state during the positive cycle, involves turning the transistors Q6 and Q3 On, allowing the zero voltage to reach the output Fig. 11(b). In the next state O-, represented by Fig. 11(c), the transistors Q2 and Q5 are on, allowing the zero voltage to reach the output. Lastly, the N state, illustrated in Fig. 11(d), affects the output when transistors Q3 and Q4 are turned on, creating a path for the negative half of the DC input to influence the output, generating $\frac{-Vdc}{2}$ Volts. The inverter's output voltage is shown in Fig. 12, where the on-time fluctuates. This voltage output goes through a filter and gives it a sinusoidal form, as presented in Fig. 13.

This sinusoidal output is produced by fluctuating on time with the constant filter settings. The output voltage increases as the on-time increases and lowers as the on-time decreases. Switch Q6 is set ON in the "P" state to ensure that Q3 and Q4 share the same amount of voltage, while switch Q5 is switched ON in the "N" state to ensure that Q1 and Q2 share the same amount of voltage [13]. Table 1 shows that Q1 and Q6 are linked because they share the same position for each state, and the same applies to Q4 and Q5.

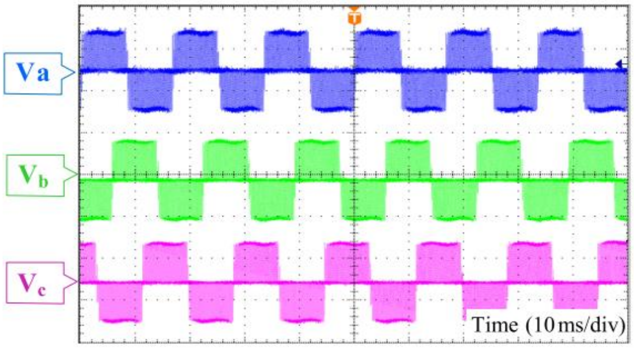


Fig. 12. Inverter voltage output waveforms [12]

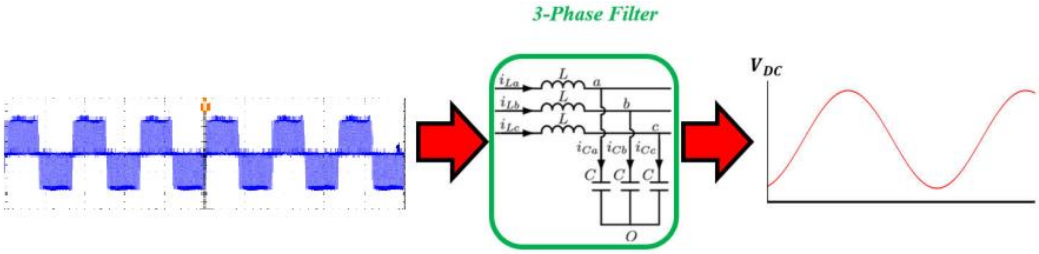


Fig. 13. Filtered Inverter Voltage Output

2.5 Controller board and architecture

The work described in [4] used a controller called UCB (Unified Controller Board), which contains two DSPs, one FPGA, one Xport gate, four expansion headers, a JTAG interface, IDC expansions, and ADC signal conditioners, as illustrated in Fig. 15. As this work is a continuation of the work developed in [4], so the same controller board was used.

The highlighted components displayed in Fig. 15 played a crucial role in the design of this project. The expansion headers were utilized primarily to establish a connection between the controller and the SMA inverter while providing a pathway to connect other peripheral components to the controller, which will be elaborated on in further detail in this work. The DSPs and FPGA were extensively integrated into the project since it was primarily developed in VHDL and embedded within the FPGA. Communication between these components relied on the Modbus RTU protocol, designed by the University of Arkansas and implemented using VHDL. Additionally, the Serial Communication Interface (SCI) was heavily utilized to facilitate communication between the FPGA and a computer. A picture of the physical board is presented in Fig. 15.

Although not depicted in Fig. 14, the external SPI Flash was another critical component. As the development progressed, the FPGA firmware grew to a point where the internal Flash memory within the FPGA was insufficient to store the FPGA firmware and the DSP firmware, which was initially stored in the same Flash memory. To overcome this issue, an external board with an SPI Flash chip was added as a solution, so the DSP firmware could be stored apart from the FPGA firmware. Micron's Micron Serial NOR Flash Memory was used for this project.

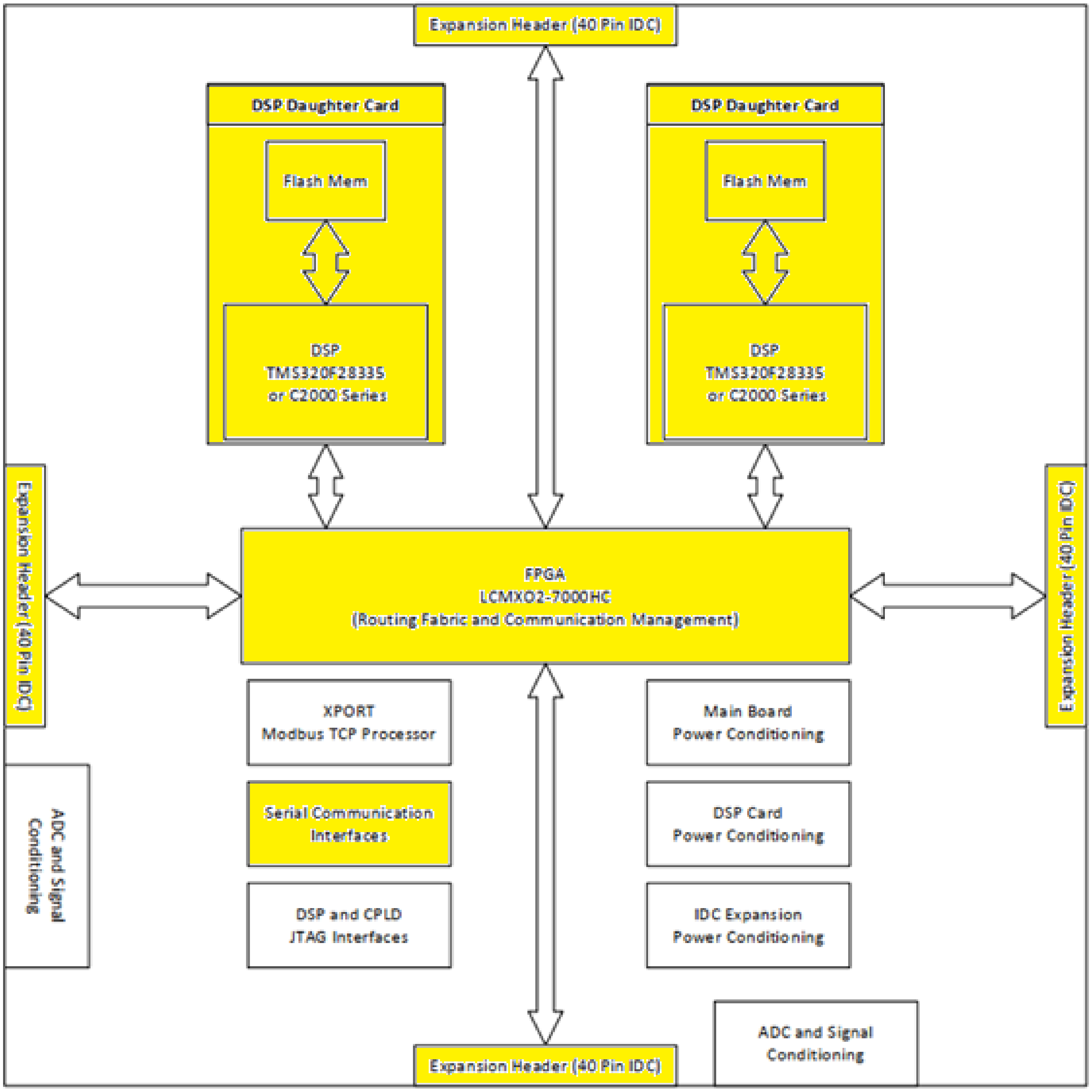


Fig. 14. Block diagram of UCB architecture showing significant components [2]

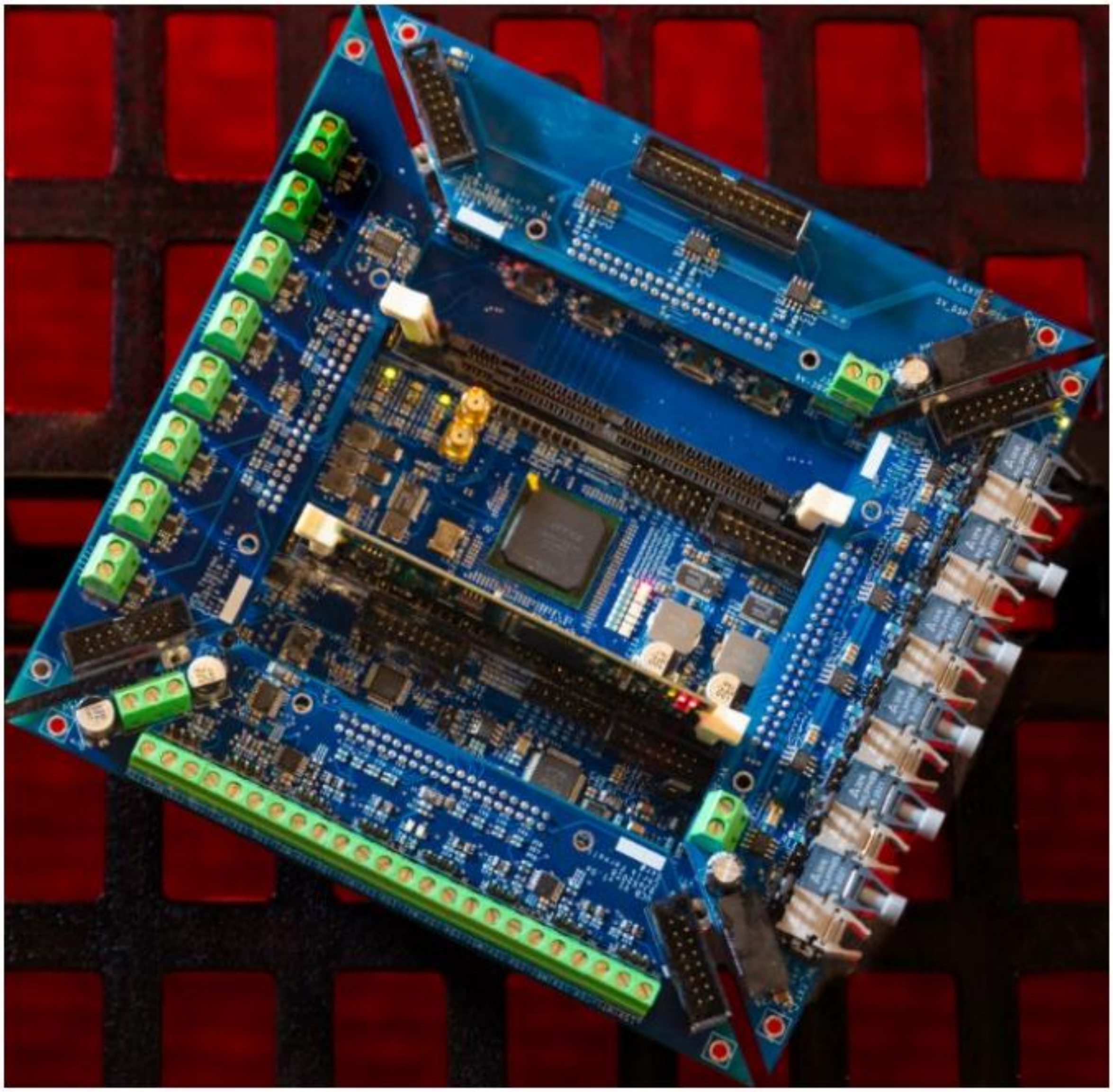


Fig. 15. UCB with auxiliary daughter boards installed [2]

2.5.1 Field-Programmable-Gate-Array (FPGA)

An array of configurable logic gates makes up a Field-Programmable Gate Array (FPGA), which may be programmed internally using either a special Joint Test Action Group (JTAG) or another type of serial/parallel non-volatile memory. Static random-access memory (SRAM), a volatile memory type where, once the board is shut down, the data stored in an FPGA's memory is erased, is widely used in the FPGA architecture, and an external non-volatile memory (EEPROM) is connected to the FPGA in order to configure the data.

The FPGA Architecture allows for the implementation of any design of digital hardware circuit, and it is based on three distinguished elements:

- Configurable Logic Blocks (CLBs): The CLBs are the blue boxes represented in Fig. 16. Each of these blocks consists of a large number of look-up tables (LUTs), multiplexers (MUXs), and Flip-Flops (FFs), as they can be used to implement logic functions.
- Input/Output Blocks (IOBs): Are external connection resources near the FPGA's edge. These programmable blocks carry signals "to" or "from" an FPGA device. IOBs are depicted in Fig. 16 as rectangular boxes bounded by the FPGA.
- Switch Matrix: A configuration of linking wires inside an FPGA that provides low-impedance and low-delay dedicated pathways for the CLBs.

The Lattice MachX02-7000HC FPGA device used in this project can be programmed using the IDE provided by Lattice named “Lattice Diamond.” This FPGA includes an embedded clock system providing a frequency not higher than 400MHz, including a Phase Lock Loop (PLL) that provides alternative frequency domains for different designs. It also includes Configuration Flash Memory, or CFM, where the developed firmware is stored, Embedded Block RAM (EBR), a component that can be used to store parameters, was used to store some variable’s values, I/O banks, Programmable Function Units (PFUs) that contains 6864 Look-Up-Tables (LUTs) are used in the design and have a voltage core of 2.5-3.3V.

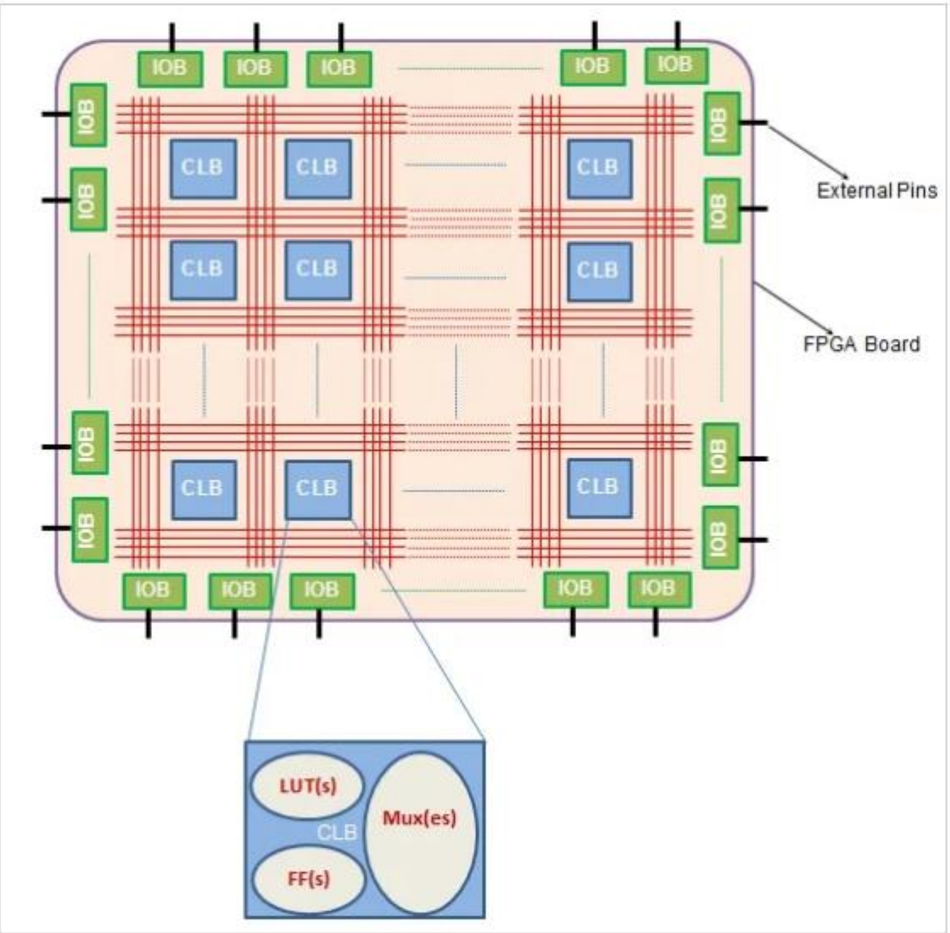


Fig. 16. Internal architecture of a typical FPGA [18]

For external communication, the FPGA can communicate through Inter-Integrated Circuit (I²C), Serial Peripheral Interface (SPI), and Universal Asynchronous Receiver and Transmitter (UART) protocols. In this design, the SPI communicated between the FPGA and an external Flash memory to increment the controller's memory capacity. At the same time, the UART was crucial to connect the board with the User Interface (UI) during development and testing. More information about the MachX02 can be found in [19].

2.5.2 Digital Signal Processor (DSP)

It is common to employ DSPs in regulating power inverters that transform DC power derived from solar panels or batteries into AC power suitable for utilization in electrical systems. DSPs can be used to execute complicated control algorithms that govern inverter voltage, frequency, and power production and monitor and fix problems. For this project, DSPs were utilized to generate the PWM signals.

PWM is a crucial feature of DSPs in inverter management. PWM is a method for controlling an inverter's output voltage by changing the width of its output pulses, in this case, to control the ANPC inverter output. The average voltage can be changed over time by changing the pulse width, providing precise output voltage control. The PWM impulses can be generated in real-time using DSP, allowing for fast and precise changes to working circumstances.

The PWM was generated based on the previously explained method of natural sampling SPWM, which compares a triangle carrier wave, and a sinusoidal modulating voltage with a fundamental frequency, ensuring the carrier has a much higher frequency than the fundamental. Then, a rectangular pulse sequence that varies its width is produced, and the pulse sequence drives the inverter to provide a sinusoidal voltage or current output, as presented in Fig. 7.

The two DSP cards utilized within the controller, model Delfino F28335, manufactured by Texas Instruments, use Code Composer Studio (CCS) as an interface to communicate and control the devices. This DSP was chosen due to its capabilities. As a C2000 real-time microcontroller, it was designed to increase closed-loop performance and was specifically manufactured for use in real-time control applications, such as solar inverters [20]. Fig. 17 represents the F28335 schematic.

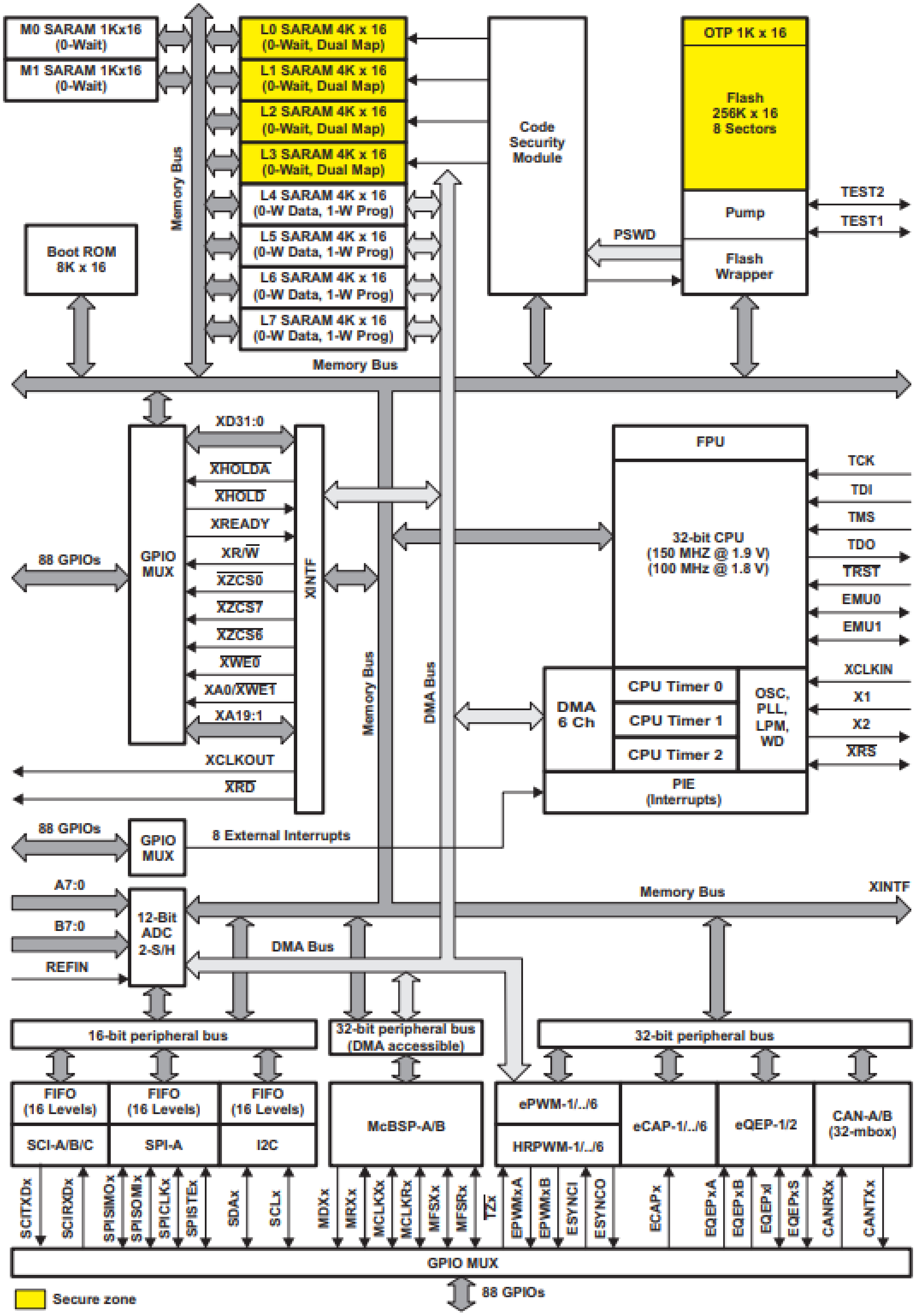


Fig. 17. DSP Block diagram

2.5.3 Serial Peripheral Interface Flash Memory (SPI Flash)

2.5.3.1 Flash Memory

One of the main differences between a volatile memory, like Random Access Memory (RAM), and a non-volatile memory (NVM) is that in a volatile memory, the data stored in it is lost when the power is switched off. However, in a non-volatile memory such as Flash Memory, this limitation does not occur. The Flash memory can retain the data through multiple power cycles, which means the program stored in the Flash is not lost even when switched off [21].

The Flash memory was since, during the firmware loading process, the FPGA needs to access the DSP firmware when the user requests. In addition, as it also needs to keep a genuinely known firmware as a backup, the controller must not lose the DSP firmware in case the system is turned off.

The memory component used as a solution to the lack of internal memory in the FPGA was the MT25QL128ABA manufactured by Micron. This chip was selected due to its memory size, 128Mb, its voltage application – 2.7 to 3.6V – and its versatility to read and program it with ease.

2.5.3.2 SPI Protocol

The communication within the Flash Memory is made through a protocol named Serial Peripheral Interface, or SPI. It is a widely used synchronous serial communication protocol developed by Motorola in the mid-1980s to facilitate data transfer between various electronic components [22]. This protocol uses a four-wire interface consisting of a clock line, a master-out-slave-in (MOSI) line, a master-in-slave-out (MISO) line, and a slave select (SS) line, as depicted in Fig. 18.

SPI utilizes a master-slave architecture, meaning it has one device (the master) that controls the communication and one or more devices (the slaves) that respond to the master's commands. This structure provides a straightforward and efficient method of communication between devices. It allows for full-duplex communication, meaning the master and slave devices can transmit and receive data simultaneously [22].

The SPI protocol employs two lines, one for transmitting data and the other for synchronization via clock pulses. Whenever the receiver detects a clock edge, it reads the bit from the data line. The entity that generates the clock signal is called the "master," while the other party is known as the "slave." Typically, there is only one master, which in this case was the FPGA, but there may be one or more slaves. To send data from the master to a slave, the master sends bits through the MOSI line, and the MISO is used by the slave to return the response. When multiple slaves are present, the SS line chooses the intended one and signals the slave to prepare for receiving or sending data. The SS line is usually held high, severs the slave's connection to the SPI bus [21].

For example, in Fig. 18, the master sends a binary command “01010011”, which corresponds to “53” in hexadecimal format. After a while, the slave replies to the master with a binary message "01000110", which correlates to "46" in hexadecimal. It is important to note that the SS is low during the entire communication process between the master and slave because that is the method the master uses to select the slave with whom it will communicate.

SPI can achieve high-speed data transfer rates of up to 400 Mbps, making it an ideal choice for applications that require fast and reliable communication between devices, such as sensors, displays, and memory chips. Additionally, SPI is commonly used in embedded systems and microcontroller-based projects because of its simplicity and low hardware requirements [22].

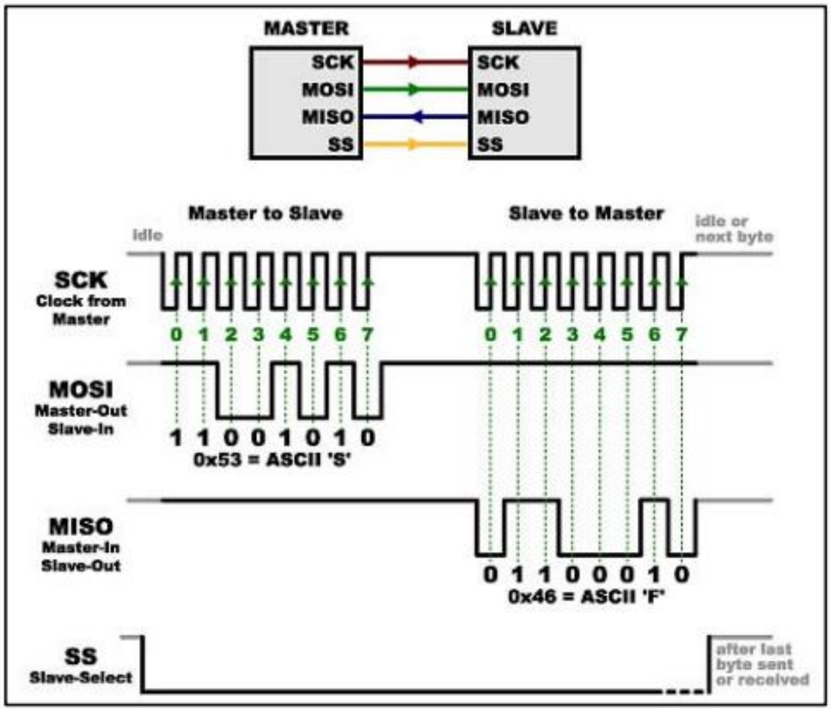


Fig. 18. SPI Connection [21]

Fig. 19 depicts the MT25Q128ABA pinout. The nomenclature provided earlier in the SPI protocol does not appear in this picture. However, the manufacturer provides the correlation in the component's datasheet, where the S# is the slave-select, also known as Chip Select, the C pin is the clock input, DQ0 is the input MOSI, DQ1 is the output MISO, Vcc is the power supply, and Vss is the ground [23]. Table 2 presents the correlation between the SPI protocol and the SPI Flash chip used in this work. The W# and DQ3/HOLD were not used in this work; they are extra protection pulled high to disable them.

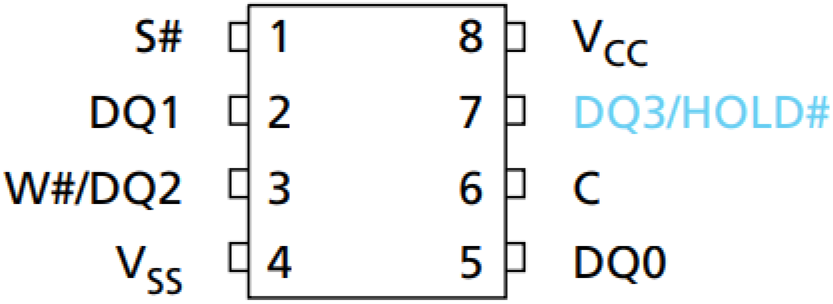


Fig. 19. MT25Q128ABA [23]

Table 2. Correlation SPI - MT25Q128ABA

SPI	MT25Q128ABA
MOSI	DQ0
MISO	DQ1
SS	S#
SCLK	C

An example of when the FPGA needs to read data from the SPI Flash, it must send the read command – “03” in hexadecimal – followed by the register address where the data is stored. As presented in Fig. 20, the command takes eight clock cycles, 0 to 7, since “03” in hex would be “0000 0011” in binary, and each bit takes one clock to be read. After sending the desired register, the slave, which is the flash memory itself, replies to the master with the data that was stored at that address. LSB and MSB presented in the pictures stand for Least Significant Bit and Most Significant Bit, respectively.

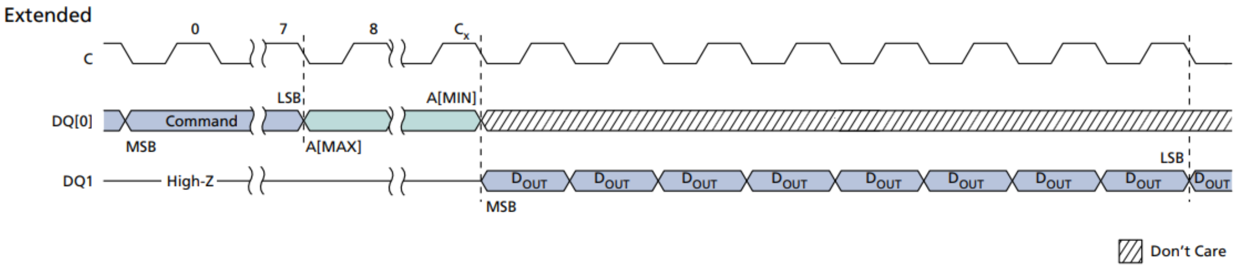


Fig. 20. SPI Flash reading procedure [23]

A VHDL-based method was developed to extract data from the MT25Q128ABA, essential to store the DSP firmware externally from the FPGA. The reason for this was that the MachXO2 device had limited resources, and it was no longer feasible to store the firmware within the FPGA memory. As the project progressed, the FPGA firmware size grew to a point where the internal Flash memory was inadequate to contain both the FPGA and DSP firmware.

2.6 Digital Twin

The Digital Twin (DT) was initially introduced by Professor Grieves at the University of Michigan in 2003 while teaching a product life cycle management course. Grieves defined DT as a virtual information structure representing a manufactured product [5]. He proposed that a DT model should have three dimensions: a physical entity, a virtual entity, and an interconnection between them [6].

In their research on the prediction of complex product/system behaviors through Digital Twins, Grieves highlighted the importance of using simulation predictions to minimize the complexity of such products/systems. The ultimate goal is to prevent unforeseen and unfavorable outcomes that could result in disastrous consequences. For instance, when launching a rocket, a virtual space is created to simulate the Digital Twin of an actual rocket. The Digital Twin allows for quick replacements and repairs in the event of failure, reducing the risk of catastrophic problems [24].

This technology is considered the leading force in changing the norms of aviation manufacturing in the years to come [25]. This technology is causing significant disruption in various industries by utilizing data feeds to map physical entities. The German Information Technology and New Media Association BITKOM predicts the manufacturing market will see immense value in digital twins, with estimates surpassing 78 billion euros by 2025. In 2016 and 2017, Gartner – a 5+ billion-dollar company that provides insights and guidance to other businesses - recognized DT as one of the top ten strategic technology development trends. In November 2017, the largest weapons manufacturer globally, Lockheed Martin, identified DT as one of the top six technologies in the future defense and aerospace industry [26].

Furthermore, according to [27], applying DT in automated industries is vital. They refer to the comprehensive simulations used to create a virtual replica of a physical system. By embracing digital twins, operators can oversee production, analyze deviations in a controlled virtual setting, and enhance the safety of process industries.

However, the meaning of DT may vary depending on the context in which it is used. For instance, aircraft or system orientation, optimal utilization of advanced physical models, sensors, historical operating data, integration of various multi-disciplinary and multi-scale probabilistic simulation processes and mapping the physical aircraft's corresponding state are all encompassed in NASA's definition of a digital twin.

Meanwhile, in the electrical engineering realm, more specifically in grid-connected IoT devices, some experts argue that DTs for cyber-secure grid-connected devices are real-time simulations that can be employed to monitor system health and event response, and overall efficiency during cyberattack scenarios [4].

Based on the concept of monitoring system health, creating an alternative to check system responses for new patches, and also offering the possibility to check system performance without putting it into jeopardy, an emulator that replicates a 3-level ANPC inverter behavior (DT) was designed in VHDL and embedded within the FPGA.

Fig. 21 illustrates the DT implementation for the FPGA subsystem, where the FPGA contains a hardware emulator that mimics the physical hardware of the grid-connected device, which in this case is an ANPC inverter, as shown in Fig. 5. The emulator employs the PWM signals generated by the DSP. Based on its status, the DT determines the corresponding output voltages. Once the 3-level ANPC output has been determined, the FPGA proceeds to collect 192 output samples, with a sampling interval of 100µs, and stores them in its internal RAM, as illustrated in Fig. 21 and Fig. 22. When a user requires access to the output generated by the new firmware in the standby DSP, the FPGA retrieves the relevant data from its RAM and transmits it to the user via the SCI interface. The output is then made available on LabVIEW, among other platforms.

The process of creating a DT for a 3-level ANPC inverter involves utilizing a DSP that is not currently in charge of controlling the inverter. As depicted in Fig. 21, the DSP1 is classified as the active DSP since it is responsible for routing the PWMs that control the inverter. In contrast, DSP2 is identified as the standby or non-active DSP since its PWM signals are directed not to the inverter but to the emulator, which generates the DT and performs firmware validation.

As an illustration, consider a scenario where a user intends to update the device's firmware. The new firmware is transmitted to the FPGA via the SCI interface and is stored in the Flash Memory using the SPI protocol. At this point, the user can load and test the firmware. If the user decides to proceed with testing, the firmware is extracted from the Flash Memory via SPI and transmitted to the standby DSP, which in this instance is DSP2, using the MODBUS RTU protocol. While the DSP1 continues to control the grid-connected device, the DSP2 undergoes an online validation process, which is integrated into the controller board. The validation feature evaluates a set of potential firmware flaws, ensuring that the firmware meets all pre-established requirements, as described in the subsequent section.

Additionally, it simulates the behavior of the 3-level ANPC inverter to verify the functionality of the firmware on the standby DSP. If the new firmware passes all the tests, it becomes available to take over control of the inverter, facilitating hot-patching and providing the DT of the new firmware. If the user opts to hot-patch, the signals that regulate the inverter are switched, with the DSP1 transitioning into the standby DSP role and the DSP2 becoming the active DSP, as illustrated in Figure 19.

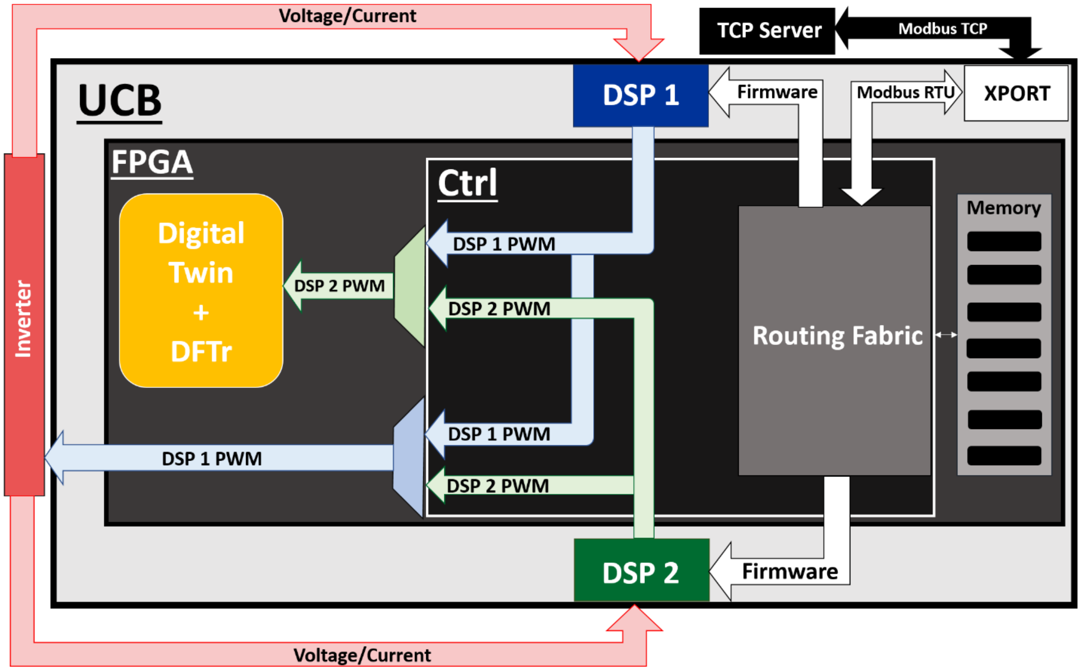


Fig. 21. Hardware Architecture (DSP1 as active)

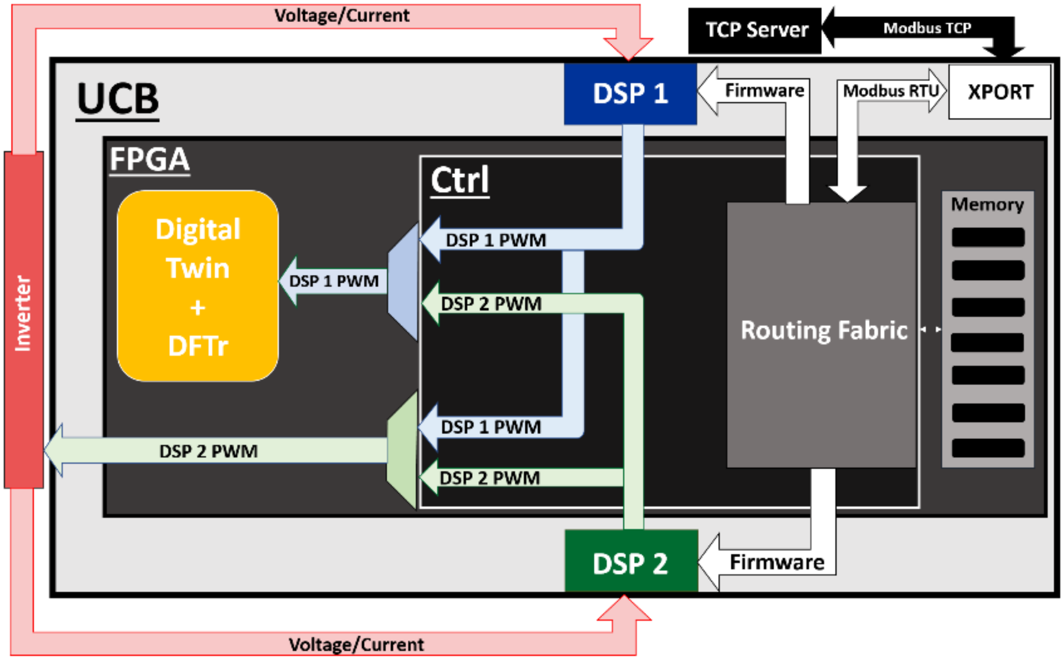


Fig. 22. DSP2 as active

2.7 Design-For-Trust

Considering a scenario where an attacker gained access to the inverter and attempted to update the DSP with malicious firmware, to harm the grid or the inverter, or to shut the inverter down, a couple of crucial tests were established and designed in VHDL and embedded in the FPGA. Thus, the DSP firmware must be trustworthy to be approved and allowed to control the inverter.

The DFTr technique was designed to prevent the system from entering situations that pose a potential risk to the ANPC inverter or the power grid. Before a new firmware is activated, tests are conducted to ensure its reliability. If any of these tests fail, the firmware is considered inherently harmful, and the system rejects it and prevents it from becoming active. These tests are processed simultaneously, enhancing the efficiency of the authentication process.

The considered tests were based on critical scenarios that could cause significant damage to the grid or the inverter. One of the considered tests was to detect short-circuit scenarios due to malicious DSP firmware, which could cause immense damage to the grid and the inverter. Another test was made to prevent new DSP firmware from lacking deadtime, generating short-circuits for a short period but very frequently, which could jeopardize the inverter and the grid. Another test checks if the new firmware is based on the fundamental frequency of 60Hz. The last test is a watchdog that ensures a new firmware is not blank, which would turn the inverter off without any visible changes in the inverter and impact the power delivered to the grid.

2.7.1 Short-Circuit

A low-resistance connection between two conductors that power a circuit is commonly referred to as a short-circuit. When electricity flows through a path with low resistance, it creates an electrical short circuit, causing an excessive current flow and voltage streaming in the power supply, leading to potentially dangerous consequences such as circuit overheating, fire, or explosion [28].

Considering a 3-level ANPC inverter, the scenario that creates a path between the positive, negative, and neutral that generates a short-circuit, known as a shoot-through, must be avoided. Considering one phase lag of the ANPC, Fig. 23 illustrates the scenario when Q1 and Q5 are on simultaneously. The short-circuit path is generated between P and O, which could harm the device. The same idea applies to the scenario where Q4 and Q6 are on simultaneously.

The purpose of the short-circuit tests is to assess the operational switching states of the new firmware and ensure that a short-circuit condition among the switches never occurs, as depicted in Fig. 23.

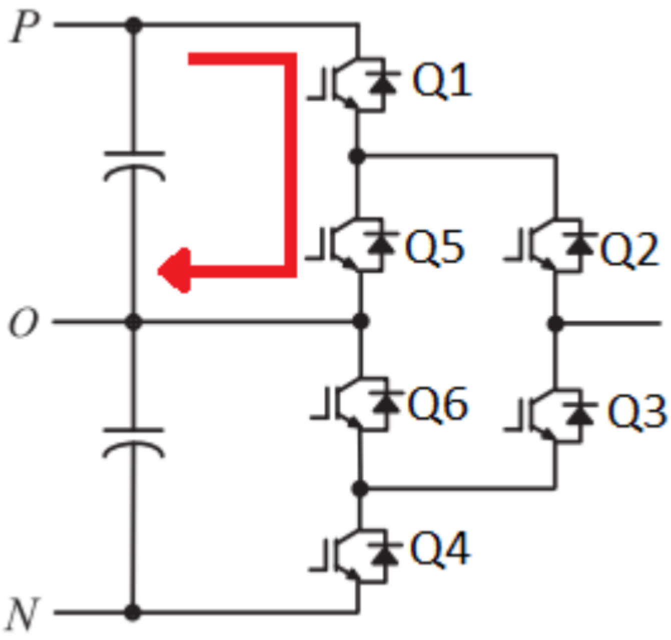


Fig. 23. Short Circuit scenario [2]

2.7.2 Deadtime

Actual transistors are not ideal, requiring a small amount of time to switch between the on and off states. Hence, a deadtime is necessary for modulation controls to prevent a shoot-through scenario in transistors that cannot be on at the same time. Deadtime refers to the interval between the first transistor turning off and the second turning on. Fig. 24 illustrates a deadtime between Q1 and Q4, as it was used in this project since Q1 and Q4 cannot be on simultaneously because the same PWM sent to Q1 is also forwarded to Q6, while Q4 and Q5 share the same PWM. Therefore, if Q1 and Q4 are on simultenously, it implies that Q1, Q4, Q5, and Q6 are all on together. This delay is generated by the control circuit, which is the DSP, and is essential because switching delays can cause cross-conduction. The gap is then necessary to prevent it [29].

The deadtime test aims to establish a sufficient delay between switching states to prevent the simultaneous conduction of switches that should not be conducted simultaneously. If this invalid switching configuration were to occur, it could result in a short-circuit for a short period but very frequently, putting the inverter and the grid in jeopardy.

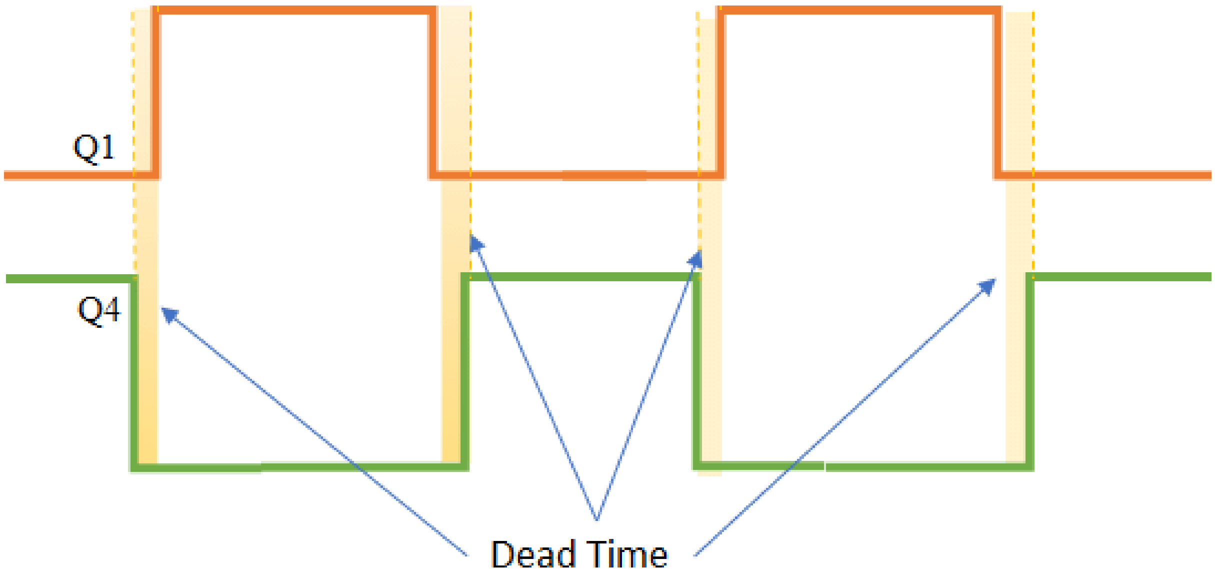


Fig. 24. Dead Time

2.7.3 Fundamental Frequency

According to the source [30], the electrical systems in the United States currently operate at a frequency of 60Hz. It is essential to maintain a high level of stability in frequency to ensure a reliable electric system. Various factors, such as generation loss and demand overload, can cause frequency variations, adversely affecting the grid. These variations can trigger protection relays involuntarily and lead to the grid reaching the lowest acceptable frequency, which can severely impact the system's stability, as stated in [31].

Furthermore, according to [31], the deployment of under-frequency load-shedding schemes varies across NERC (North American Electric Reliability Corporation) regions and subregions, with different frequency set points. In the United States, the highest initial blocks of load shedding have frequency set points ranging from 59.7 to 59.3 Hz.

Under-frequency load shedding is a process that involves disconnecting a significant number of predetermined customers from the power grid when the frequency drops to pre-set frequency thresholds.

Therefore, when dealing with grid-connected energy resources, it is essential to maintain a stable operational frequency to prevent unwanted harmonic distortions in the grid. The DFTr inside the controller ensures that the DSP firmware of the inverter generates a frequency of 60Hz, which is necessary for the grid's stability. [2].

2.7.4 Fast Frequency

This test checks if the new firmware has 42kHz as the frequency for the fast transistors (Q2 and Q3), as this frequency was defined during the progress of this work. Even though new firmware with different frequency values might not be dangerous to the inverter or grid, it can reduce the inverter’s performance and change the semiconductors’ response with the possibility of increasing losses or harmonics. Thus, during the tests on this work, the firmware needed to remain consistent with the defined characteristics.

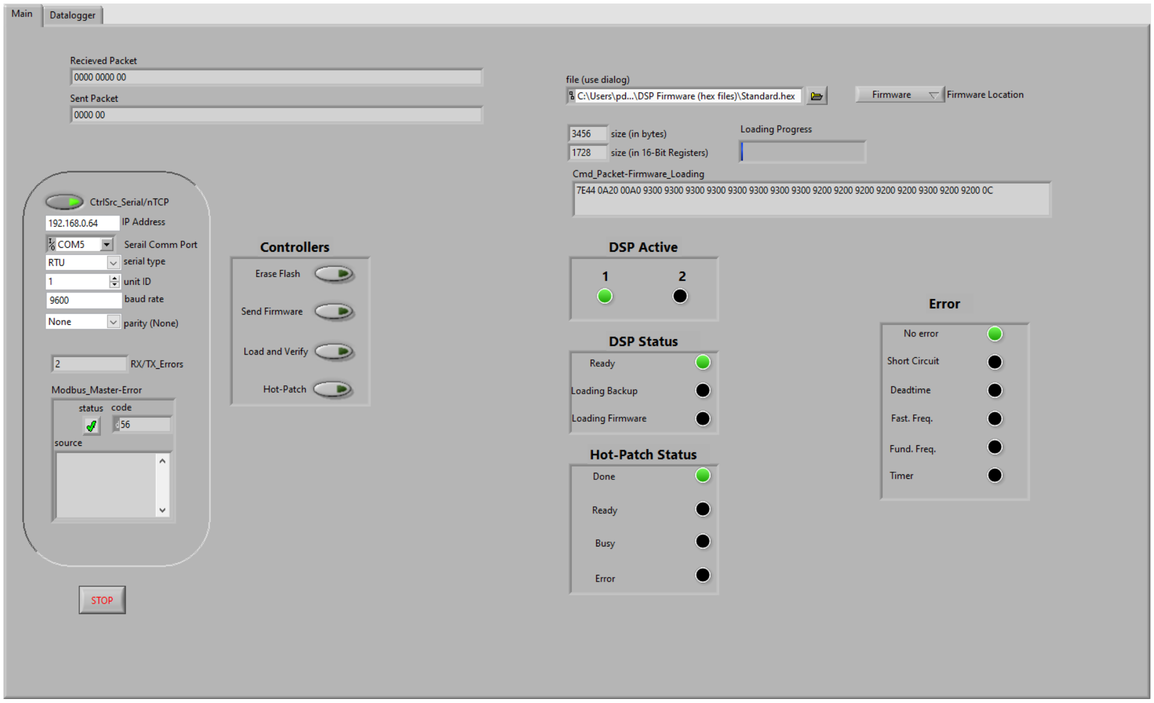
2.7.5 Watchdog (Timer)

The final step in validating the DFTr strategy involves verifying that the DSP firmware does not cause the controller to enter a stall state. In this context, a stall state refers to a situation where the FPGA is awaiting both rising and falling edges from the Pulse Width Modulation (PWM) signals but instead receives malicious firmware that lacks any oscillation in the control signals. During the firmware testing phase, the DSP firmware is subject to a maximum waiting time, and if the timer reaches this threshold, the new DSP firmware is rejected.

CHAPTER 3

METHODOLOGY

During the development of this work, the LabVIEW 2018 software was used as a Graphic User Interface (GUI) because it allowed the user to interface with the FPGA through Serial Communication Interface (SCI) to send and verify a new DSP firmware, check the DT signals and the active DSP (1 or 2), monitor the DSP status, and show the hot-patch status and possible errors, as presented in Fig. 25.



Recieved Packet: 0000 0000 00

Sent Packet: 0000 00

IP Address: 192.168.0.64

Serail Comm Port: COM5

serial type: RTU

unit ID: 1

baud rate: 9600

parity (None): None

RX/TX_Errors: 2

Modbus_Master-Error: status ✓ code 56

file (use dialog): C:\Users\pd...\DSP Firmware (hex files)\Standard.hex

Firmware Location: Firmware

size (in bytes): 3456

size (in 16-Bit Registers): 1728

Cmd_Packet-Firmware_Loading: 7E44 0A20 00A0 9300 9300 9300 9300 9300 9300 9300 9300 9200 9200 9200 9200 9200 9300 9200 9200 0C

DSP Active: 1 ● 2 ●

DSP Status: Ready ● Loading Backup ● Loading Firmware ●

Hot-Patch Status: Done ● Ready ● Busy ● Error ●

Error: No error ● Short Circuit ● Deadtime ● Fast. Freq. ● Fund. Freq. ● Timer ●

Fig. 25. LabVIEW interface

The FPGA is the primary reference in the UCB since it can process different tasks simultaneously. The FPGA regulates signal routing, data flow, security measures, and firmware patching. To do that, the FPGA was programmed using VHDL language, and some software components were designed and embedded into the FPGA to create a cyber-secured system. The main components responsible for initializing the procedure and system, performing the DFTr process, generating the DT, and hot-patching were named Bootloader, Firmware Validation, Emulation, and Hot-Patching, respectively. Each of these components was considered internal processes and was described in detail in the next section of this work.

3.1 PROCESSES

This section outlines the steps required for a user to replicate the tests conducted in this work. The instructions not only covered the process for obtaining the same results as in this work but also provided insight into the inner workings of the controller. The primary aim was to enhance the user's comprehension of the interaction between themselves and the controller.

3.1.1 EXTERNAL PROCESSES

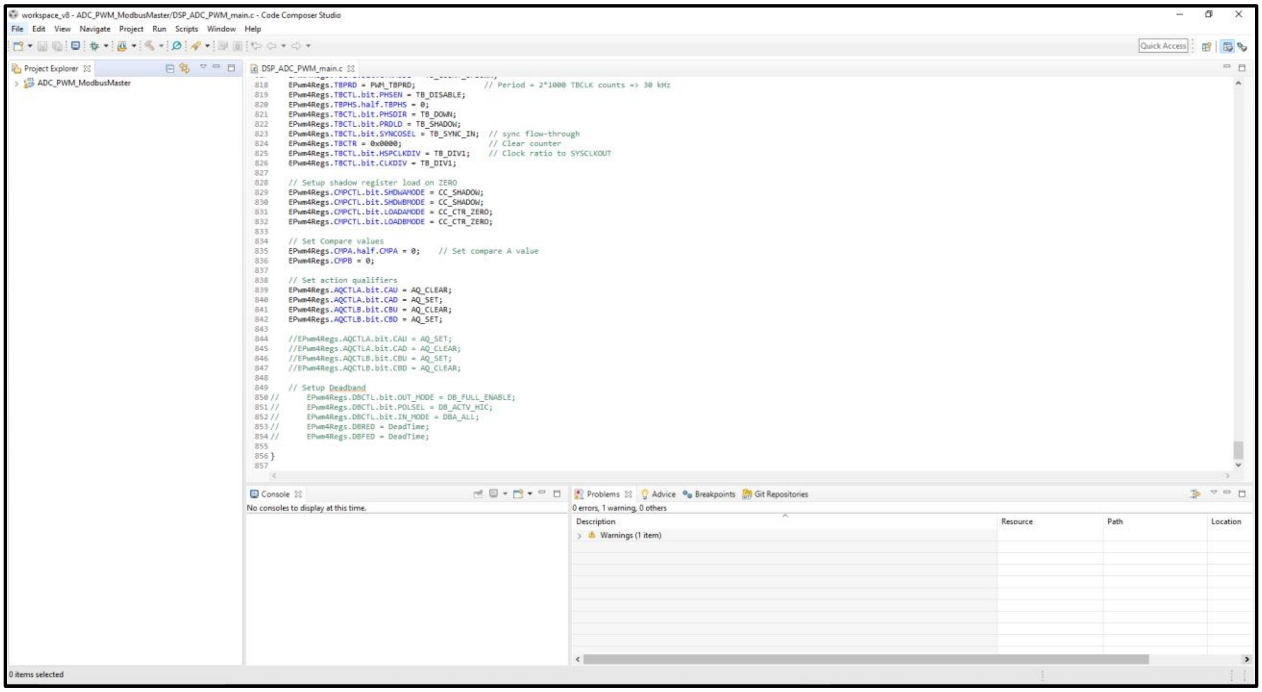
The procedures outlined in this section are deemed external, as they pertain to a protocol that occurs outside of the controller. These procedures involve the user, the IDE, and the controller. As established in this project, they are crucial for enabling the user to execute tests with the controller and arrive at their conclusion regarding its trustworthiness and dependability.

3.1.1.1 DSP Firmware Development

The first step in generating a DT and testing the DFTr is to design firmware to control the DSP PWMs. Using the CCS software, the user can create a DSP firmware using C language, as presented in Fig. 26. After developing the firmware to generate the PWMs using the GPIOs, as presented in Table 3, it was necessary to change the project's properties to generate a ". hex" file when the project is built. To make the changes in the properties, was requested to follow the instructions below:

1. Right-click on the desired project.
2. Select "Properties."

3. Select the item “C2000 Hex Utility”.
4. Click on the checkbox “Enable C2000 Hex Utility.”
5. On “Boot Table Options,” mark the “Specify table source as the SCI-A port, 8-bit mode (--sci8, -sci8)” checkbox.
6. On “Output Format Options,” select “Output Intel hex format (--intel, -i)” as Output format and check the “Binary output format (for DSKs) (--binary, -b)” checkbox.
7. Click on “Apply and Close”
8. Right-click on the desired project and select “Clean Project.”
9. Right-click on the desired project and select “Build Project.”



```
    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;

}
```

No consoles to display at this time.

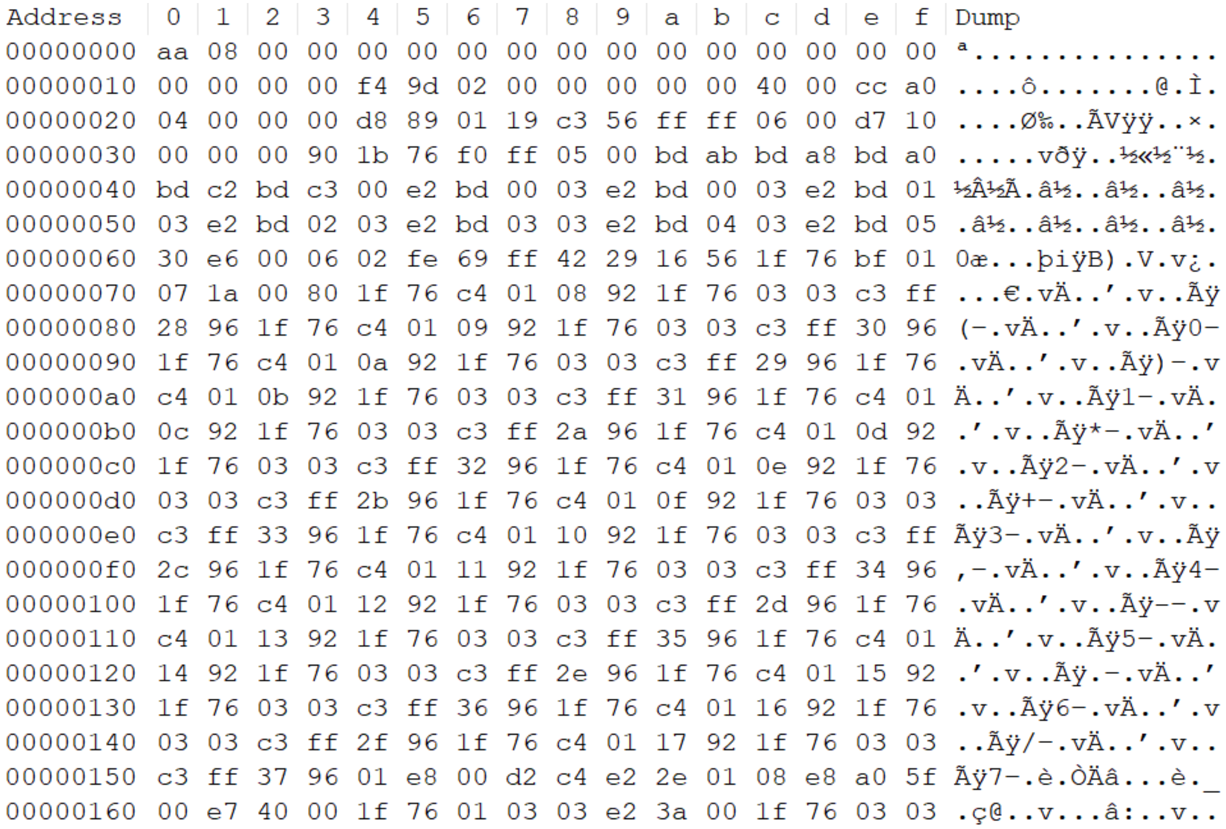
0 errors, 1 warning, 0 others

Fig. 26. CCS IDE

Table 3. Correlation between DSP and FPGA GPIOs

Phase	PWM	DSP Output (GPIO)	FPGA Input
A	Q1 & Q6	25	AB6
A	Q4 & Q5	12	U10
A	Q2	00	AA8
A	Q3	01	Y7
B	Q1 & Q6	26	Y4
B	Q4 & Q5	27	W11
B	Q2	02	T8
B	Q3	03	V8
C	Q1 & Q6	14	T10
C	Q4 & Q5	19	V11
C	Q2	04	U8
C	Q3	05	W9

After completing the procedure previously described, a “. hex” file was generated and stored in the “Debug” folder. An example of a “. hex” file is depicted in Fig. 27.



```
Address  | 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f | Dump
00000000  aa 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ª...............
00000010  00 00 00 00 f4 9d 02 00 00 00 00 40 00 cc a0     ....ô.......@.Ì.
00000020  04 00 00 00 d8 89 01 19 c3 56 ff ff 06 00 d7 10  ....Ø‰..ÃVÿÿ..×.
00000030  00 00 00 90 1b 76 f0 ff 05 00 bd ab bd a8 bd a0  .....vðÿ..½«½¨½.
00000040  bd c2 bd c3 00 e2 bd 00 03 e2 bd 00 03 e2 bd 01  ½Â½Ã.â½..â½..â½.
00000050  03 e2 bd 02 03 e2 bd 03 03 e2 bd 04 03 e2 bd 05  .â½..â½..â½..â½.
00000060  30 e6 00 06 02 fe 69 ff 42 29 16 56 1f 76 bf 01  0æ...þiÿB).V.v¿.
00000070  07 1a 00 80 1f 76 c4 01 08 92 1f 76 03 03 c3 ff  ...€.vÄ..’.v..Ãÿ
00000080  28 96 1f 76 c4 01 09 92 1f 76 03 03 c3 ff 30 96  (–.vÄ..’.v..Ãÿ0–
00000090  1f 76 c4 01 0a 92 1f 76 03 03 c3 ff 29 96 1f 76  .vÄ..’.v..Ãÿ)–.v
000000a0  c4 01 0b 92 1f 76 03 03 c3 ff 31 96 1f 76 c4 01  Ä..’.v..Ãÿ1–.vÄ.
000000b0  0c 92 1f 76 03 03 c3 ff 2a 96 1f 76 c4 01 0d 92  .’.v..Ãÿ*–.vÄ..’
000000c0  1f 76 03 03 c3 ff 32 96 1f 76 c4 01 0e 92 1f 76  .v..Ãÿ2–.vÄ..’.v
000000d0  03 03 c3 ff 2b 96 1f 76 c4 01 0f 92 1f 76 03 03  ..Ãÿ+–.vÄ..’.v..
000000e0  c3 ff 33 96 1f 76 c4 01 10 92 1f 76 03 03 c3 ff  Ãÿ3–.vÄ..’.v..Ãÿ
000000f0  2c 96 1f 76 c4 01 11 92 1f 76 03 03 c3 ff 34 96  ,–.vÄ..’.v..Ãÿ4–
00000100  1f 76 c4 01 12 92 1f 76 03 03 c3 ff 2d 96 1f 76  .vÄ..’.v..Ãÿ-–.v
00000110  c4 01 13 92 1f 76 03 03 c3 ff 35 96 1f 76 c4 01  Ä..’.v..Ãÿ5–.vÄ.
00000120  14 92 1f 76 03 03 c3 ff 2e 96 1f 76 c4 01 15 92  .’.v..Ãÿ.–.vÄ..’
00000130  1f 76 03 03 c3 ff 36 96 1f 76 c4 01 16 92 1f 76  .v..Ãÿ6–.vÄ..’.v
00000140  03 03 c3 ff 2f 96 1f 76 c4 01 17 92 1f 76 03 03  ..Ãÿ/–.vÄ..’.v..
00000150  c3 ff 37 96 01 e8 00 d2 c4 e2 2e 01 08 e8 a0 5f  Ãÿ7–.è.ÒÄâ...è._
00000160  00 e7 40 00 1f 76 01 03 03 e2 3a 00 1f 76 03 03  .ç@..v...â:..v..
```

Fig. 27. Example of a Hex File (firmware)

3.1.1.2 Firmware Upload – Firmware to be tested.

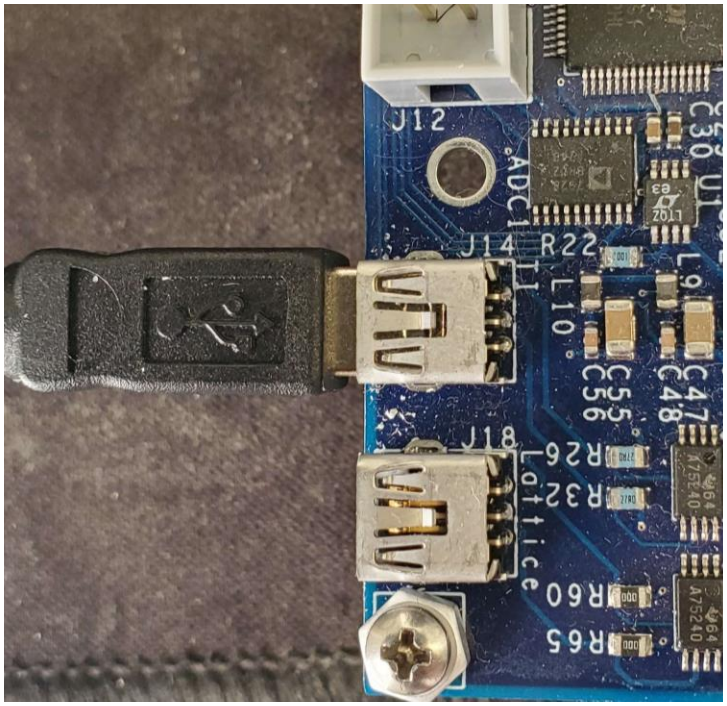


Fig. 28. USB cable connection

The first step in uploading firmware was to connect a USB cable to the correct (non-Lattice/FPGA) port on the UCB. Fig. 28 shows the USB connection made with J14 to load a DSP firmware. The pictured port, J18, is used to program the FPGA instead. Next, was selected the correct setting to connect the computer to the controller. Fig. 29 illustrates the LabVIEW interface with the GUI project designed for this work. On the left, a purple square highlights the configuration settings for the serial communication between LabVIEW and the UCB. For example, the user had to select the correct Serial Communication port, COM5, and make sure the "Serial Type" selected was RTU. The "Unit ID" had to be 1, the "baud rate" was 9600, and the "parity" field was None,

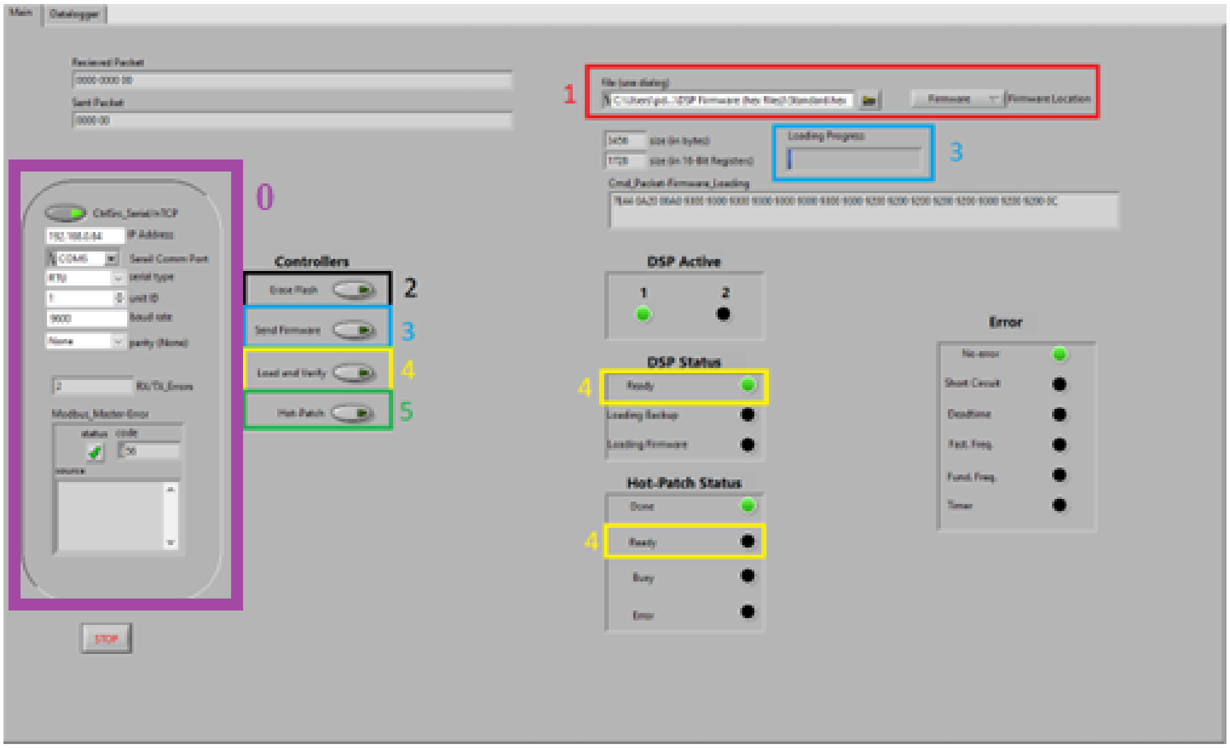


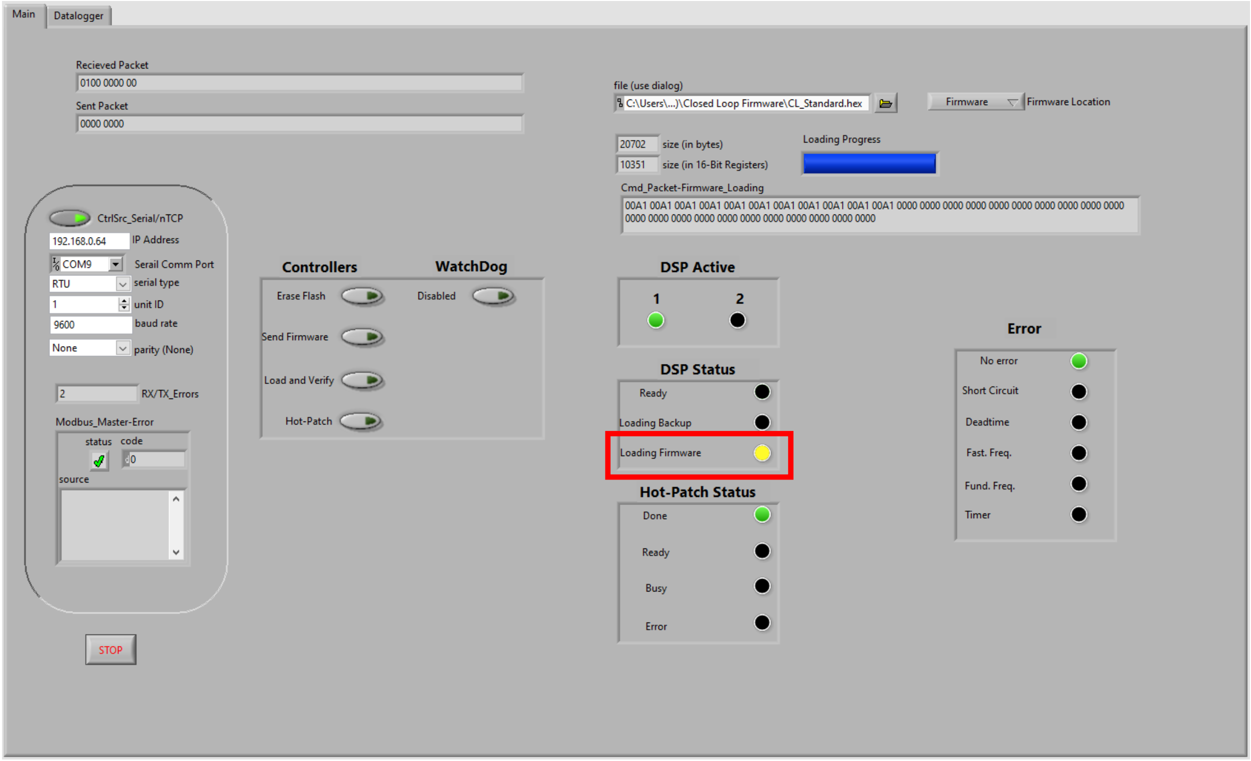
Fig. 29. LabVIEW project

The second step in this procedure was to send the DSP firmware to the controller. With the LabVIEW project, the user selected the “. hex” file generated in the previous subsection (3.1.1.1 DSP Firmware Development). The user had to click on the folder button highlighted in red in Fig. 29 and with the number “1” written in red as well.

Next, after selecting the “. hex” file correctly, it was necessary to erase the Flash memory. As this type of memory cannot be overwritten, it must first be erased. To achieve this, the user had to click the “Erase Flash” button, highlighted in black, with the number “2” by its side, in Fig. 29.

The button to perform the third step is displayed in blue in Fig. 29, with the number “3”. The user sends the “. hex” file to the controller in this step. The FPGA received the .hex file and stored it in the Flash Memory using the SPI protocol. On the top right of Fig. 29, there is also a blue square with the number “3” on its side that indicates the delivery progress of the firmware to the controller.

When the transmission was complete, the interface allowed the user to click on “Load and Verify,” highlighted by a yellow square in the “Controllers” buttons section. Here, the user sent a command to the FPGA to pull the firmware previously stored into the Flash memory, reset the DSP to enable it to receive a new firmware, and rerouted the data to the DSP. In this step, the “DSP Status” lights changed status. The “Ready” light went off, and the “Loading Firmware” lighted up, as depicted in Fig. 30. The “Loading Firmware” status not only indicates the firmware was being loaded into the DSP but also that the firmware validation was running as soon as the loading process ends. In this state, no other controller button affected the controller.



Recieved Packet: 0100 0000 00

Sent Packet: 0000 0000

file (use dialog): C:\Users\...)\Closed Loop Firmware\CL_Standard.hex

Firmware Location: Firmware

20702 size (in bytes)

10351 size (in 16-Bit Registers)

Cmd_Packet-Firmware_Loading

```
00A1 00A1 00A1 00A1 00A1 00A1 00A1 00A1 00A1 00A1 00A1 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

CtrlSrc_Serial/nTCP

192.168.0.64 IP Address

COM9 Serail Comm Port

RTU serial type

1 unit ID

9600 baud rate

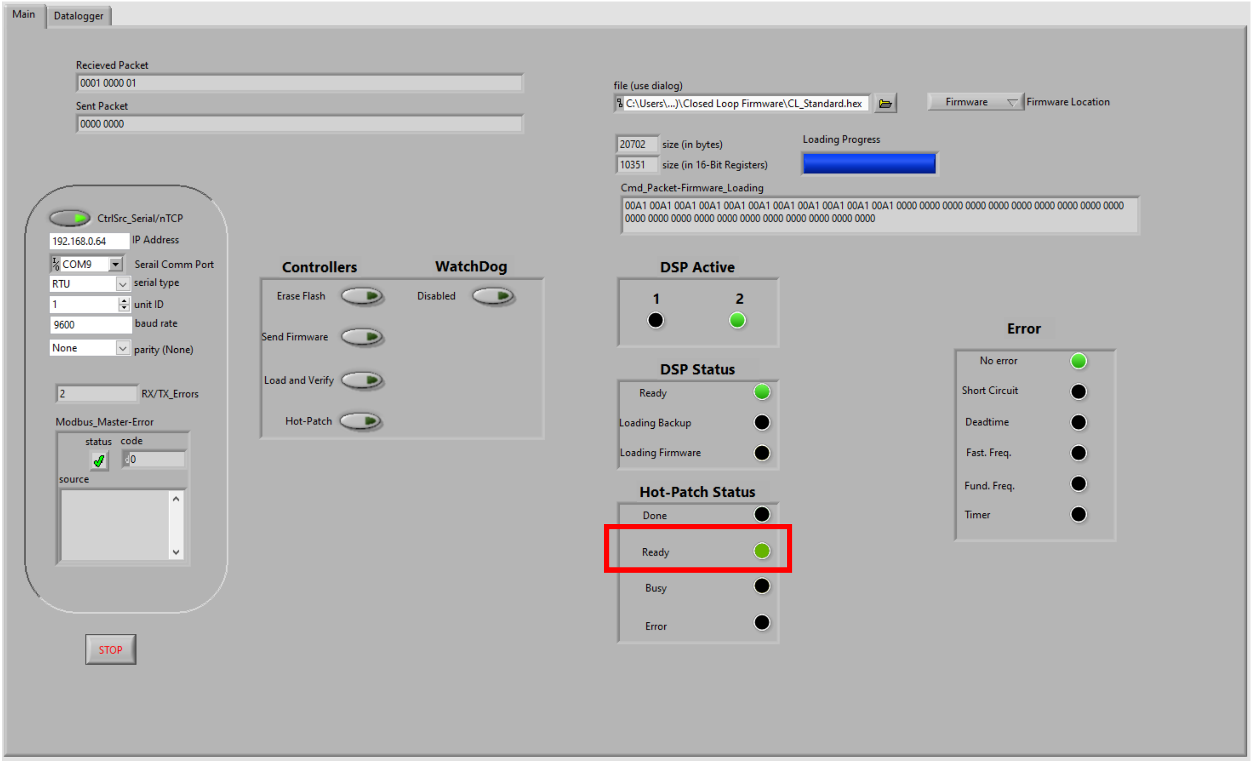
None parity (None)

2 RX/TX_Errors

Modbus_Master-Error: status ✓ code 0 source

Fig. 30. Loading Firmware

If the updated firmware met all the requirements, meaning that it passed the DFTr tests, the FPGA enabled the Hot-Patch button, granting the user the possibility of swapping the active DSP and then letting the new firmware control the inverter. When the hot-patch function was allowed, the “Hot-Patch Status” section turned all lights off except the “Ready,” as presented in Fig. 31, meaning the hot-patch was waiting for a command. When a user clicks on the “Hot-Patch” button, the status changes from “Ready” to “Done.”



Recieved Packet
0001 0000 01
Sent Packet
0000 0000

192.168.0.64 IP Address
COM9 Serail Comm Port
RTU serial type
1 unit ID
9600 baud rate
None parity (None)
2 RX/TX_Errors

Modbus_Master-Error
status ✓ code 0
source

file (use dialog)
C:\Users\...)\Closed Loop Firmware\CL_Standard.hex
Firmware Firmware Location
20702 size (in bytes)
10351 size (in 16-Bit Registers)

Cmd_Packet-Firmware_Loading
00A1 00A1 00A1 00A1 00A1 00A1 00A1 00A1 00A1 00A1 00A1 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

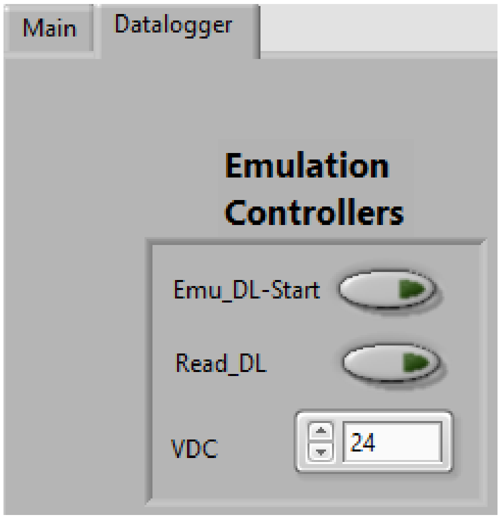
Fig. 31. Hot-Patch Ready

During the validation process, the Hot-Patch status might have changed. While the firmware was undergoing tests, the “Busy” light turned on briefly, indicating that the firmware was under evaluation. Because the process was too fast, when this light turned on was almost unnoticeable. On the other hand, the “Error” light was effortless to see since it glowed when the

DSP firmware was considered malicious; it stayed on until the backup firmware was loaded entirely into the standby DSP.

After Hot-Patching, to generate the DT of the new firmware, the procedure starting at the "Load and Verify" step had to be done again so that both DSPs would embed the same firmware. This way, one DSP controlled the inverter while the other generated the DT. With the DSP firmware loaded into the standby DSP, the FPGA received its output and replicated the output that a real ANPC inverter would provide, which had to be similar to what Fig. 12 exhibits.

To check the provided DT output, the user had to select the "Datalogger" tab at the top left of the LabVIEW project window. Then, the user could choose the Vdc the system used, as presented in Fig. 32. Next, click "Emu_DL-Start." This button sent a command to the FPGA, informing it to start sampling and storing the samples in the internal RAM. Lastly, the user clicked on "Read_DL," which commanded the FPGA to transmit the data previously stored in RAM to LabVIEW.



VDC 24

Fig. 32. Generate Digital Twin output.

The firmware was rejected if the loaded firmware did not meet the requirements defined by the DFTr. Fig. 33 represents the possible errors displayed in LabVIEW. The FPGA rejected the new firmware by resetting the standby DSP and sending the backup firmware. This procedure guaranteed that both DSPs could control the inverter if any fault happened to the active DSP.

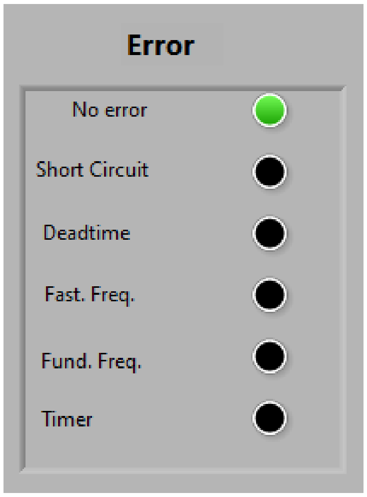


Fig. 33. Possible errors

3.1.1.3 Firmware Upload – Backup Firmware.

The Backup Firmware is a DSP firmware previously tested and approved by the controller. In other words, genuine firmware has all the requirements to control the inverter without entering a potentially detrimental state. This backup firmware was essential because when the firmware is tested and rejected due to not having all the requirements defined in the DFTr, the FPGA sends a command to reset the standby DSP where the malicious firmware was loaded. Then, the standby DSP is loaded with the backup firmware, guaranteeing that the standby DSP has a known non-malicious firmware loaded that can generate a DT of a 3-level ANPC inverter.

Assuming the LabVIEW configuration was correctly set up by the user using the specifications described in the previous subsection to send the backup firmware to the FPGA, it was necessary to select the ".hex" file with the genuine firmware and click on the drop-down menu, as depicted in Fig. 34, and selected "FW for Backup." Next, it was required to click the "Erase Flash" button, which commanded the FPGA to erase the Flash memory section designated to store the backup firmware. The last step was to push the “Send Firmware” button and wait until the progress was complete.

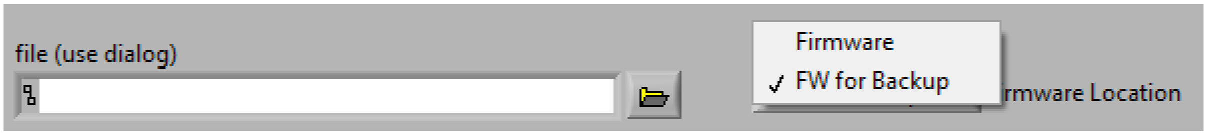


Fig. 34. Backup FW: Drop-down menu

The difference between sending the firmware to be tested and a backup firmware is that sending the backup firmware is unnecessary to do the other steps, as cited in the previous subsection since the backup firmware must be considered adequate to perform them.

3.1.2 INTERNAL PROCESSES

The steps described in this section were considered internal since they relate to a protocol executed within the controller. The subsequent subsections elaborate on the procedures within the controller that enabled the patching of new firmware and the creation of a DT for virtually monitoring the output of a 3-level ANPC inverter by the user.

3.1.2.1 Bootloader

The responsibility of the Bootloader was to retrieve the firmware data from the Flash memory and transmit it to the DSP. To prepare the DSP for the new firmware, the Bootloader clears the DSP's existing contents, resets it, and sends the autobaud configuration data. Subsequently, the Bootloader extracts the firmware data from the allocated registers in Flash memory, divides it into two 8-bit data sets, and combines them to form a 10-bit packet, including the beginning and end portions. To ensure the complete firmware file is transmitted to the DSP, the Bootloader utilizes the total number of firmware registers that are recorded in the first allocated register of the Flash, as indicated in Table 4. After transmitting the firmware file to the DSP, the Bootloader activates the firmware validation feature to ensure the entire firmware file has been successfully transmitted.

Table 4. Firmware Register Map

Name	**Ram Address** **(16-Bit Hex)**	**Data** **(16-Bit Hex)**	**Description**
FW Len	1000	16-bit	FW size register
FW Data	1001:2FFF	16-bit	Firmware

If the new firmware was valid, the emulator continued to operate, and the Hot-Patch procedure started. However, if the new firmware was not valid, then the validation identified the new firmware as malicious. The firmware validation interacted directly with the Bootloader, Hot-Patch, and Emulation, triggering the "error function" in these components. The emulation stopped and reset while the FPGA disabled the Hot-Patch and saved the error flag and error type into the registers in the DP-RAM (Dual Port – Random Access Memory), and informed the user of the error type. Meanwhile, the Bootloader started the backup procedure by loading the standby DSP with secure firmware and disabled user commands until the backup firmware was completely loaded.

The backup process was initiated if the new firmware was rejected during firmware validation. When in backup mode, the Bootloader retrieves backup firmware data from designated registers in Flash, which are enabled once the backup procedure has been activated. The Flash space assigned for the backup firmware data can be found in Table 5. It is worth noting that the backup Bootloader method is identical to the new firmware's Bootloader.

Table 5. Backup Firmware Register Map

Name	Ram Address (16-Bit Hex)	Data (16-Bit Hex)	Description
FW Len	3000	16-bit	Backup FW size register
FW Data	3001:4FFF	16-bit	Backup Firmware

3.1.2.2 Firmware Validation

The security reference design's firmware validation function focused on particular firmware instructions that might jeopardize the power electronic inverter or the grid it is connected to. This feature's architecture made it simple to detect malicious firmware, putting the firmware under the DFTr specifications.

The firmware validation feature watched the standby DSP's PWM signals and compared and checked against the requirements previously specified in the DFTr. To prevent malicious firmware from taking control of the inverter, the FPGA conducts processes described in this section by simultaneously comparing all switches in each phase leg's PWM signals using this function.

The firmware validation process takes up to three seconds, enough time to check the PWMs, primarily the 60Hz ones. Three seconds was equivalent to 180 cycles of a 60Hz PWM's frequency. The period for testing can be increased, if necessary, but for this work, the time chosen was sufficient to test and mitigate flaws in malicious firmware.

Considering that MachX02 can provide different clock frequencies, the clock frequency chosen for this work was 25MHz (or a period of 40ns). Thus, for all the following tests described in this work, the base measure of time considered for a single clock cycle was 40 ns.

3.1.2.2.1 Short-Circuit

In this test, the FPGA reads the PWMs received from the standby DSP and searches for any scenario where a short-circuit occurs between the slower frequency transistors, which is considered the fundamental frequency (60Hz). This test ensures that Q1, Q4, Q5, and Q6 are never active simultaneously.

To achieve that, the FPGA reads the pins routed to the DSP's GPIOs from Table 3. If a firmware provides any scenario where these transistors are switched on together during the test period, the FPGA automatically invalidates this firmware. The invalidation of firmware consists of raising a malicious firmware flag, stopping all other processes, and starting the backup firmware loading procedure.

The short-circuit test starts by waiting for a transition on phase A Q1. When a transition happens, it waits for three more transitions, a period necessary to ensure signal stability. After reaching signal stability in phase A, the process is repeated for phases B and C. After reaching a stable state, the FPGA keeps reading all three phases, at the same time, waiting to identify a scenario where Q1 and Q4 or Q5 and Q6 are on simultaneously in any phase. If this scenario happens at least once, the firmware is rejected, a malicious firmware flag is raised, and the backup firmware process is started.

3.1.2.2.2 Deadtime

The deadtime test is very similar to the short-circuit test, starting with waiting for the switching on Q1. The main difference between these tests is that each phase is a process that runs in parallel. After receiving the first state change in Q1, this process ignores the first three changes, waiting for signal stability.

After reaching stability, this test compares the identical transistors but waits specifically for a falling edge – a transition from the high state (“1”) to the low state (“0”) - of each of them, and, when the falling edge occurs, it starts to count the number of clock cycles between the falling edge of one transistor and the rising edge of a different transistor that is not allowed to be on simultaneously. If the number of clock cycles is less than specified, the malicious flag is raised, the firmware is rejected, and the backup process is started.

For example, when a falling edge of Q1 happens, a counter starts counting the clock cycles and waiting for Q4 (which is not allowed to be on at the same time as Q) to turn on. When Q4 is switched on, the counter stops, and the number of clock cycles is checked. If the number of clock cycles is less than 25, which implies a deadtime of 1μs, then the firmware is considered malicious; otherwise, the firmware passed this test.

$$\text{Deadtime} = \text{number of clock cycles} * \text{clock period}$$

$$\text{Deadtime} = 25 * 40\text{ns} = 1\mu\text{s}$$

3.1.2.2.3 Fundamental Frequency

The Fundamental Frequency test checks if the firmware uses a frequency of 60Hz to generate the slow PWMs. A frequency of 60Hz implies that Q1, Q4, Q5, and Q6 must have a period of approximately 1.667ms. Considering the frequency fluctuation, this work allowed a minimum frequency of 59.5Hz, staying above the minimum set point of 59.3Hz, as in [31]. The same difference was allowed above 60Hz. In other words, the range accepted for a slow PWM frequency is between 59.5 and 60.5Hz.

This test starts waiting for a transition in Q1, and, similarly to the other tests, it ignores the first three switches until it becomes stable. After that, it waits for a falling edge to start counting and continues counting until the next falling edge. When the second falling edge occurs, the counter stops, and the FPGA checks how many clock cycles were counted within the PWM period. If this value is greater than 420,000 (the number of clock cycles in a period of 59.5Hz, as demonstrated in the following equations) or less than 413,000 (the number of clock cycles in a period of 60.5Hz, as presented in the following equations) the firmware is considered malicious, and the backup firmware procedure starts.

$$\mathrm{Frequency} = \frac{1}{\mathrm{Period}}$$

$$\mathrm{Period\_59.5} = \frac{1}{59.5} \approx 16.80\mathrm{ms}$$

$$\mathrm{Period\_60.5} = \frac{1}{60.5} \approx 16.53\mathrm{ms}$$

$$\mathrm{Number\ of\ Clock\ Cycles}_{\mathrm{Minimum}} = \frac{\mathrm{Period}}{\mathrm{Clock\ Period}} = \frac{16.53\mathrm{ms}}{40\mathrm{ns}} \approx 413.000$$

$$\mathrm{Number\ of\ Clock\ Cycles}_{\mathrm{Maximum}} = \frac{\mathrm{Period}}{\mathrm{Clock\ Period}} = \frac{16.80\mathrm{ms}}{40\mathrm{ns}} \approx 420.000$$

3.1.2.2.4 Fast frequency

The fast frequency test is similar to the Fundamental Frequency, but instead of 60Hz, this test used the period in a 42 kHz frequency. At the beginning of this test, the FPGA waits for a transition in Q1 and ignores the first three switches. After that, it waits for a falling edge, and then the FPGA starts to count until the next falling edge. The counter stops as soon as the second falling edge is read, and the FPGA compares how many clock cycles were counted within the fast PWM period. If this value is outside the range between 581-609, the firmware is considered malicious, and the backup firmware procedure starts. The range values consider some oscillation in the PWMs, with a minimum of 41kHz and a maximum of 43kHz.

$$\text{Period\_42k} = \frac{1}{42000} \approx 23.80\mu\text{s}$$

$$\text{Number of Clock Cycles}_{42\text{k}} = \frac{\text{Period}}{\text{Clock Period}} = \frac{23.80\mu\text{s}}{40\text{ns}} \approx 595$$

3.1.2.2.5 Watchdog (Timer)

As mentioned, the watchdog test guarantees that a new DSP firmware patch is not blank firmware that could stall the inverter. This test starts a counter that begins with all the other tests and continues counting until the short-circuit test disables it. When the short-circuit test is complete, it tells the FPGA that the new firmware has PWMs and disables the watchdog.

However, if the firmware is blank, the short-circuit test keeps waiting for a falling edge that does not exist in blank firmware. When the counter reaches 71,000,000 (43B5FC0 in hexadecimal), or 2.84 seconds, the watchdog raises the malicious firmware flag invalidating the new firmware. After that, the FPGA stops all other running processes and loads the backup firmware.

3.1.2.3 Hot-Patch

The ability to patch a device's firmware without affecting the system's functionality is known as hot-patching [4]. Assuming the firmware passes the verification, the hot-patching procedure will commence. Otherwise, the backup firmware process will be activated. The Hot-Patch uses the FPGA as a "routing fabric" and may swap the control signals for the DSP output in the order of nanoseconds. The Hot-Patch causes the standby DSP to activate and operate the grid-connected device while the current active DSP enters standby mode as a backup.

On the other hand, if the firmware is malicious, then the standby DSP remains in standby mode, and the current active DSP stays active throughout the backup process. The standby DSP is patched with backup firmware saved in the Flash memory to complete this procedure. Redundancy and failsafe functionality are built into this system because if the active DSP malfunctions, the standby DSP, equipped with the same firmware, will take over the management of the grid-connected device. As this project continued [4], the same Hot-patch and Bootloader systems were used.

To achieve a seamless transition during hot-patching and avoid causing service disruptions, synchronization between the two DSPs is essential because they work simultaneously and have access to identical measurement data from the physical hardware, which enables them to coordinate their actions. Fig. 35 depicts the outcome of a DSP transition without synchronization following firmware patching and verification [4]. The waveform demonstrates the changeover, which, in a real-world scenario, might result in loss of power and/or damage to equipment. The synchronous transition event is depicted in Fig. 36, with the DSPs correctly coordinated and ensuring no service interruptions or equipment damage occurs[4].

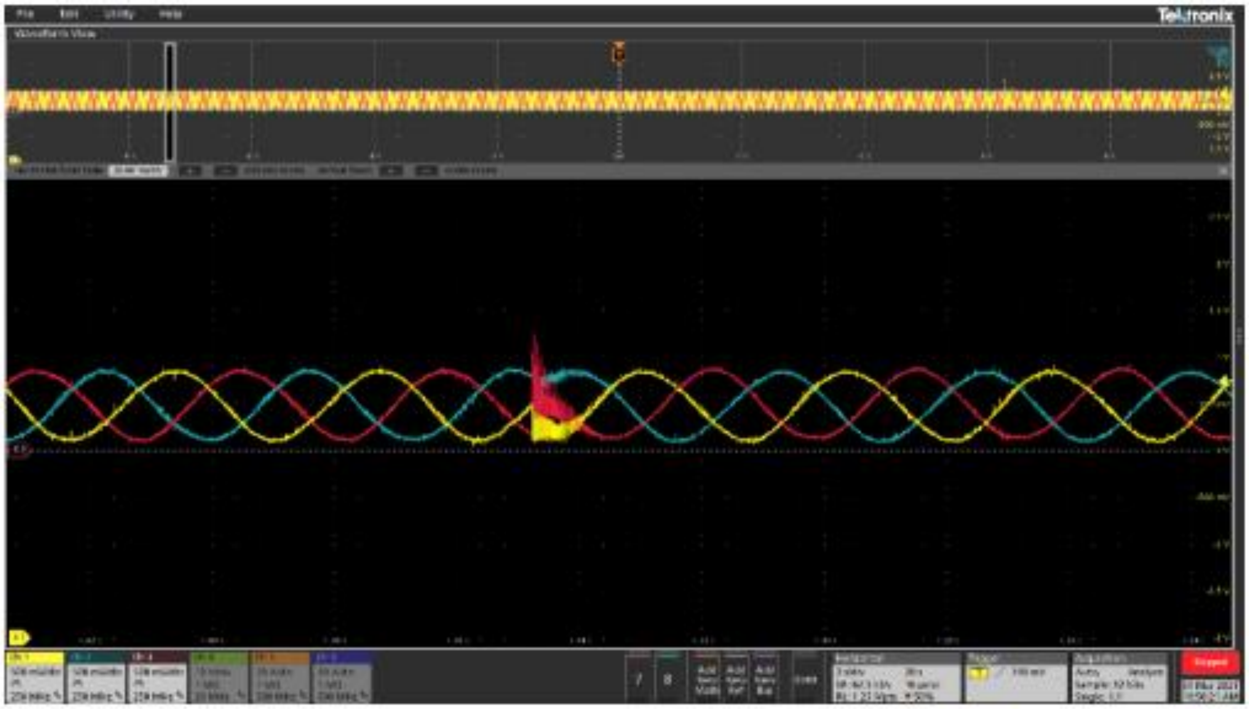


Fig. 35. Asynchronous Hot-Patch [4]

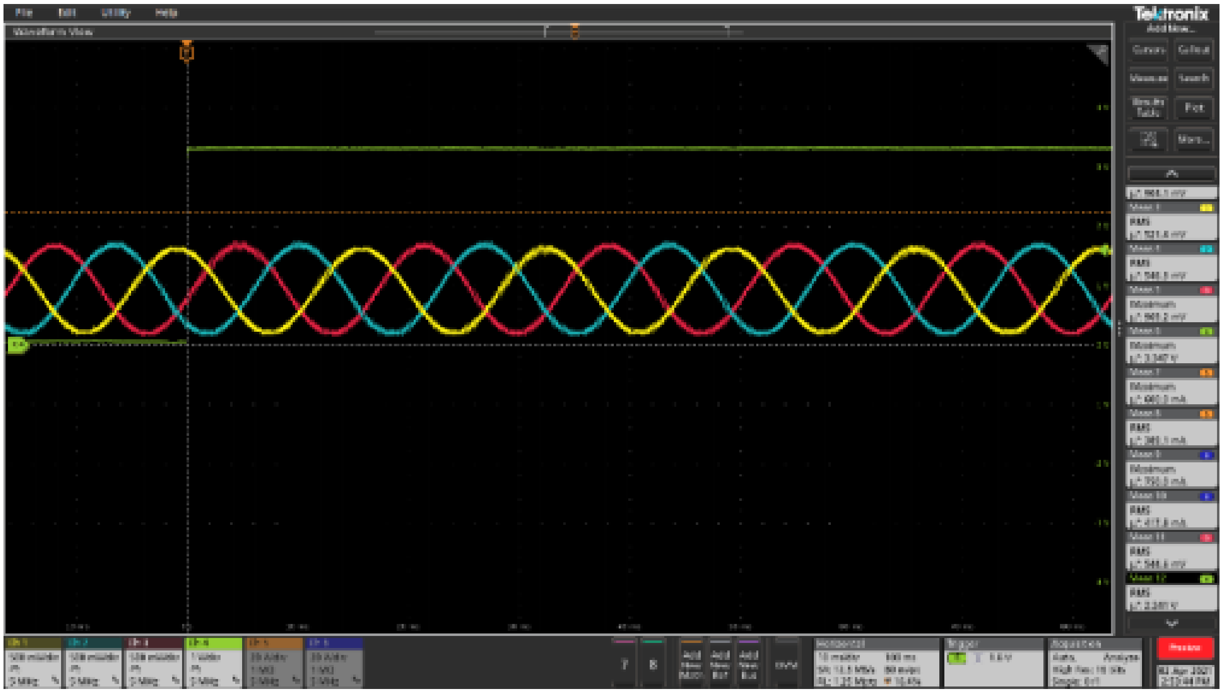


Fig. 36. Synchronous Hot-Patch [4]

3.1.2.4 Digital Twin

The Digital Twin in this security architecture serves to digitally replicate the 3-Phase 3-Level ANPC inverter, as mentioned in section 2.2. This inverter produces AC output after converting the DC input source using its switching mechanism. The DSP controller creates PWM signals to operate the switches on the inverter, which the emulator processes. The emulator also replicates an inverter output being managed by these PWM signals. After generating the corresponding output, the FPGA was programmed to gather 192 samples of these signals and store them in the DP-RAM, volatile memory in the FPGA, which can be accessed externally using LabVIEW.

To replicate an ANPC inverter, the FPGA reads the DSP output and applies each signal to the eighteen virtual transistors designed as switches, which only have two states: on and off. The virtual replica does not account for external factors influencing an actual transistor, such as noise or transient time (rise or fall time). In the virtual environment, the eighteen transistors are considered ideal, with zero delay response to the received signals from the DSP. This information is translated into switch states, as presented in Table 1.

Combining the digital transistors with the DSP output made it possible to replicate the behavior of an authentic ANPC inverter, as illustrated in Table 6. The ANPC inverter generates a percentage of the DC input using a duty cycle. For example, in Fig. 11, when the DSP output sends a high signal ("1") to transistors Q1 and Q2 simultaneously, the ANPC inverter recreates an output that is half of the DC input (state P). The same idea is applied to generate the negative half of the DC input (state N) when the DSP signals are on for transistors Q3 and Q4. This behavior was designed using VHDL and embedded within the FPGA.

From Fig. 32, , a user can fill out the VDC field, and depending on the inserted value, the DT values might change since it multiplies the VDC with the proportional output, as shown in Table 6. For example, if a user adds a value of 24 to VDC, the possible outputs will be 12, -12, and 0.

After translating the DSP state to an ANPC output, the FPGA collects and stores the samples in the DP-RAM, accessing them using the addresses presented in Table 7. Whenever a user requests a new sample from the Digital Twin, the FPGA overwrites the data in these registers with the new samples collected from the DT output and stores the new data in the DP-RAM. The data consists of real-time output values that can be accessed anytime.

Table 6. Digital Twin ANPC Output

Switch	**DSP Output State**			
Q1	1	1	0	0
Q2	1	0	1	0
Q3	0	1	0	1
Q4	0	0	1	1
Q5	0	0	1	1
Q6	1	1	0	0
Digital Twin ANPC Output	**50%**	**0V**	**0V**	**-50%**

Table 7. Digital Twin Registers

Name	Ram Address (16-Bit Hex)	Data (16-Bit Hex)	Description
Phase A	200	16-bit	Emulation Phase A
Phase B	300	16-bit	Emulation Phase B
Phase C	400	16-bit	Emulation Phase C

CHAPTER 4

RESULTS AND DISCUSSION

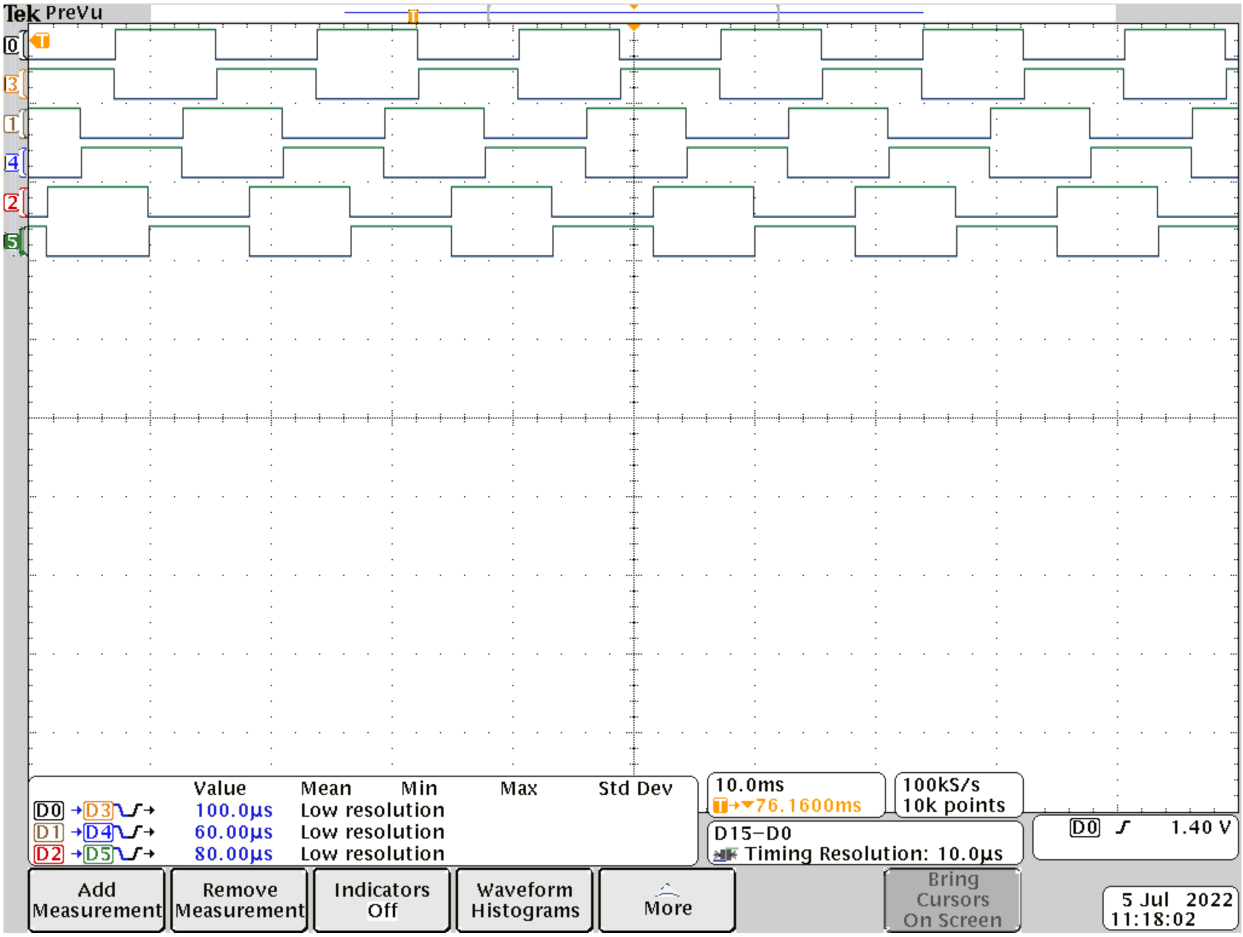
This section explains the creation of the DT and the validation process of newly received firmware by the system. To verify the firmware type and to check if the controller's behavior was correct, the oscilloscope model Tektronix MSO 4034 was employed to monitor the DSP and controller output signals.

In addition, this section discusses the DSP and controller's output signals of a known non-malicious firmware that has all the requirements to pass the DFTr test and would be capable of controlling a 3-level ANPC inverter. Next, each malicious firmware type that failed to pass each DFTr test is explained and illustrated.

4.1 Standard Firmware (Non-malicious)

After following the steps described in the previous section, the DSP firmware was generated and sent to the FPGA. Fig. 37 depicts the DSP signals generated, and each of the different channels represents the phases in the ANPC inverter. Phase A is represented by channels 0 and 3, where channel 0 controls Q1 and Q6, and channel 3 controls Q4 and Q5. Phase B is represented by channels 1 and 4, and phase C is represented by channels 2 and 5. Table 1 represents the correlation between each channel and the relevant transistors associated with them. The ANPC inverter's fast frequency transistor behavior was not monitored for this test.

The blue text values at the bottom-left of Fig. 37 indicate the deadtime between D0 and D3, interpreted as Channel 0 and 3, which is 100 µs. It also depicts a deadtime of 60 µs for Channels 1 and 4 and 80 µs between Channels 2 and 5.



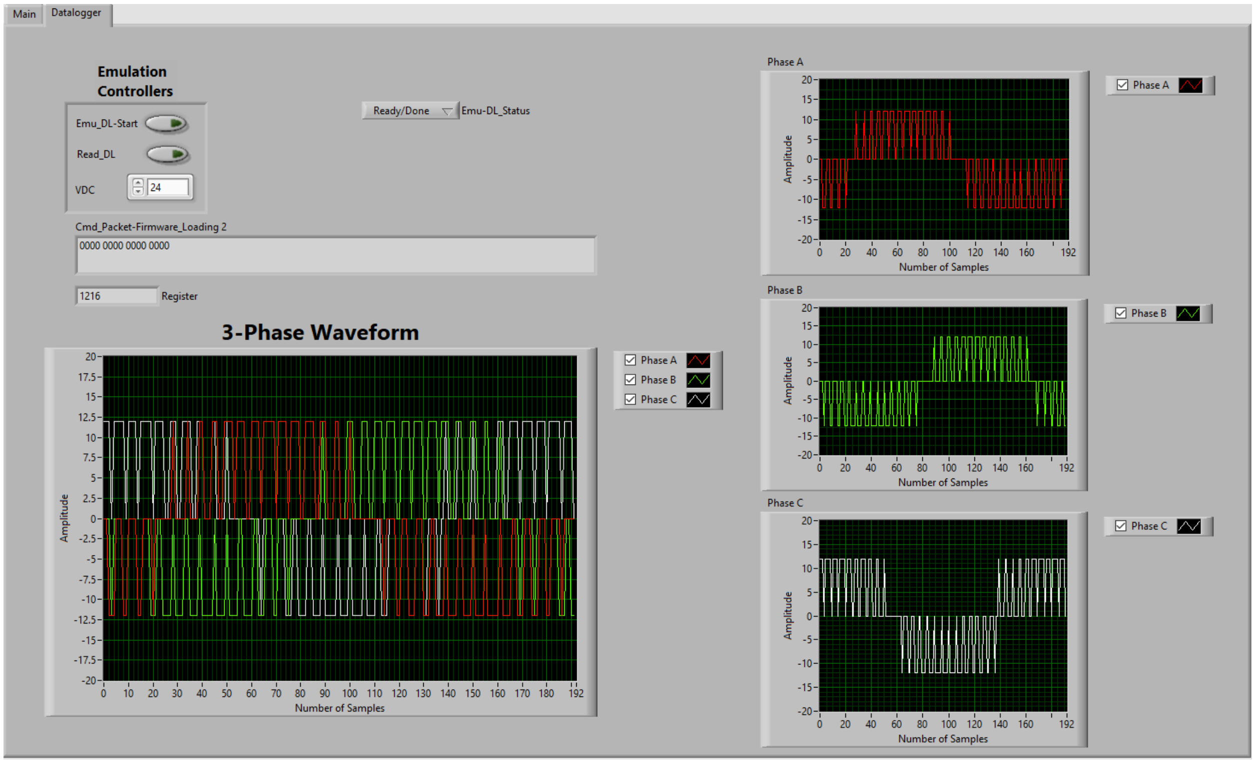
```
                 Value      Mean        Min      Max      Std Dev
D0 →D3 ⌐_↑→     100.0µs    Low resolution
D1 →D4 ⌐_↑→     60.00µs    Low resolution
D2 →D5 ⌐_↑→     80.00µs    Low resolution

10.0ms                 100kS/s
T→▼76.1600ms           10k points

D15–D0                                  D0  ↑   1.40 V
Timing Resolution: 10.0µs

                                        5 Jul  2022
                                        11:18:02
```

Fig. 37. DSP Standard Firmware

Table 8. Transistor/Channel relation

Phase A	
Q1 & Q6	Channel 0
Q4 & Q5	Channel 3
Phase B	
Q1 & Q6	Channel 1
Q4 & Q5	Channel 4
Phase C	
Q1 & Q6	Channel 2
Q4 & Q5	Channel 5

Upon conducting the necessary DFTr checks, the FPGA has successfully validated the authenticity of the DSP firmware. As a result, the FPGA could emulate the ANPC inverter's output and generate the DT. In Fig. 38, the emulated output provided by the FPGA is presented and displayed using LabVIEW. Despite the limited resolution of the waveforms depicted in Fig. 38, it is apparent that the signals produced through the emulation process resemble those typically produced by a genuine ANPC inverter as depicted in Fig. 12.



Main | Datalogger

Emulation Controllers

Emu_DL-Start

Read_DL

VDC: 24

Emu-DL_Status: Ready/Done

Cmd_Packet-Firmware_Loading 2: 0000 0000 0000 0000

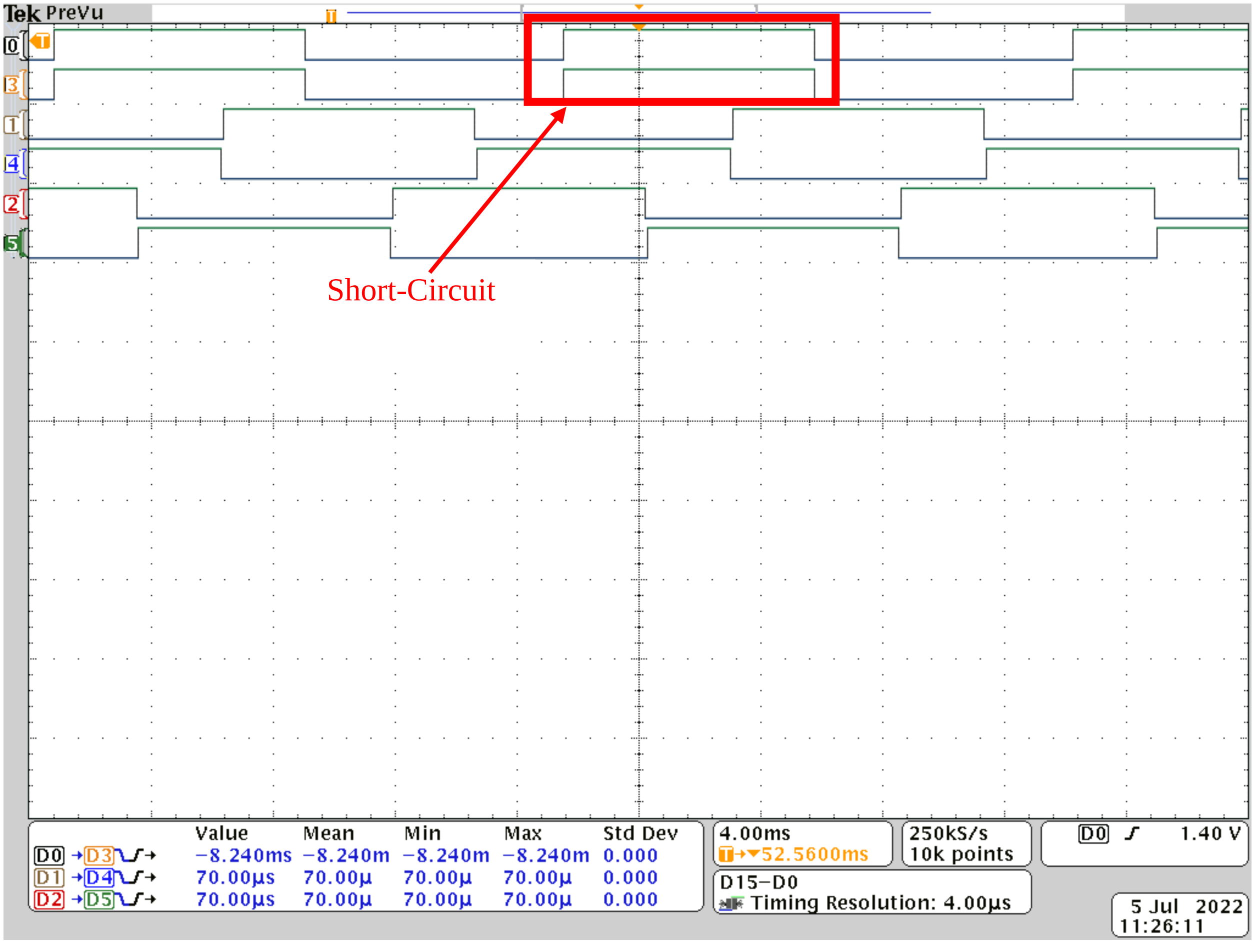
Register: 1216

Fig. 38. Three-Level ANPC Inverter Digital Twin

4.2 Short-circuit corrupted Firmware (Malicious)

If a malicious firmware attempts to take control of the inverter, the FPGA must decline it and revert to dependable firmware, which in this instance, is a backup firmware. Furthermore, the FPGA must prevent the activation of the hot-patching feature while maintaining the operation of the currently active DSP.

Initially, a malicious firmware was transmitted to the FPGA where Phase A had Q1, Q4, Q5, and Q6 turned on concurrently, as presented in Fig. 39. If firmware with the same characteristics as in Fig. 39 was loaded into the controller, it could cause a short-circuit in the inverter, damaging it and harming the grid which it was connected.



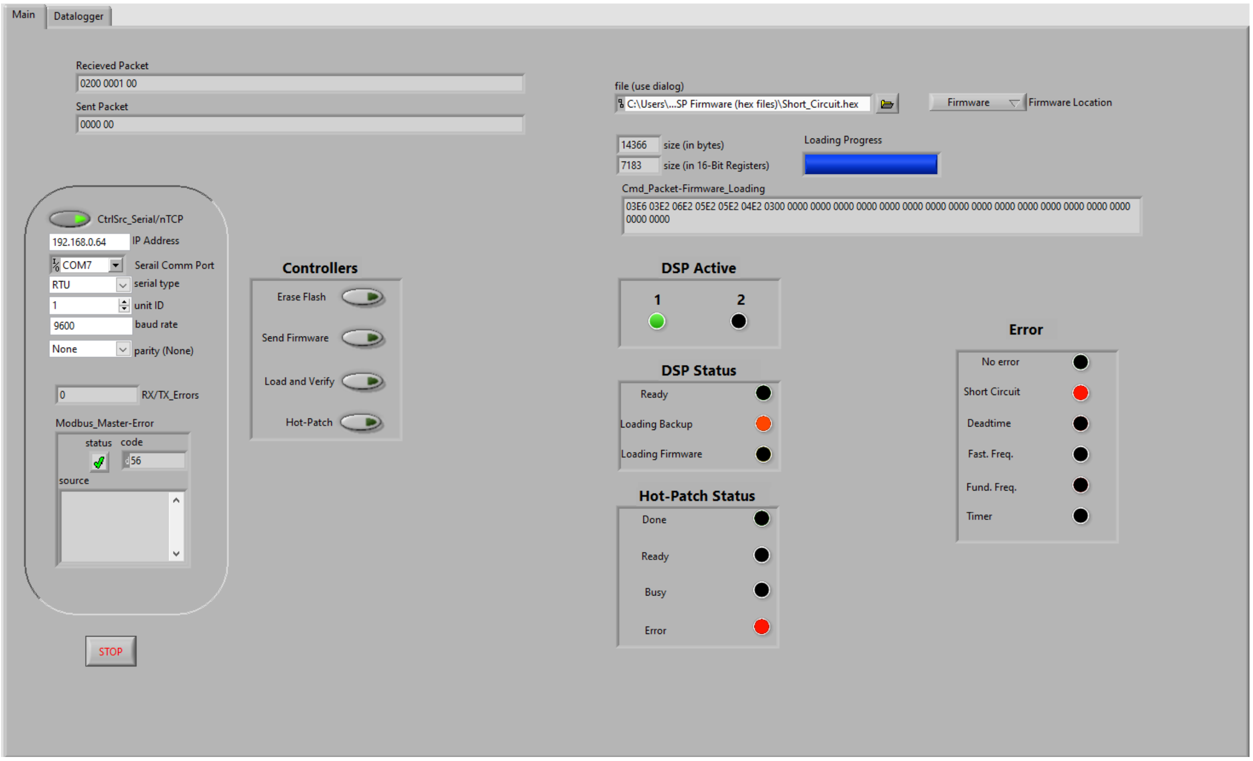
Short-Circuit

```
                Value     Mean     Min      Max      Std Dev
D0 →D3 ⌐_┘→   −8.240ms  −8.240m  −8.240m  −8.240m  0.000
D1 →D4 ⌐_┘→   70.00µs   70.00µ   70.00µ   70.00µ   0.000
D2 →D5 ⌐_┘→   70.00µs   70.00µ   70.00µ   70.00µ   0.000
```

```
4.00ms            250kS/s       D0  ⌐   1.40 V
T→▼52.5600ms      10k points
D15−D0
Timing Resolution: 4.00µs
                                5 Jul  2022
                                11:26:11
```

Fig. 39. Channel 0 and 3 on simultaneously

The controller's response to this type of firmware is depicted in Fig. 40. After testing this firmware and checking if a short-circuit scenario was present (short-circuit scenario is present in this figure), the FPGA correctly refused the firmware and successfully returned an error state. The reported error was a short-circuit condition, and the system reacted by loading the backup firmware, as shown on the graphical display.



Recieved Packet: 0200 0001 00

Sent Packet: 0000 00

CtrlSrc_Serial/nTCP

IP Address: 192.168.0.64

Serail Comm Port: COM7

serial type: RTU

unit ID: 1

baud rate: 9600

parity (None): None

RX/TX_Errors: 0

Modbus_Master-Error

status: ✓ code: 56

source

file (use dialog): C:\Users\...SP Firmware (hex files)\Short_Circuit.hex

Firmware Location: Firmware

size (in bytes): 14366

size (in 16-Bit Registers): 7183

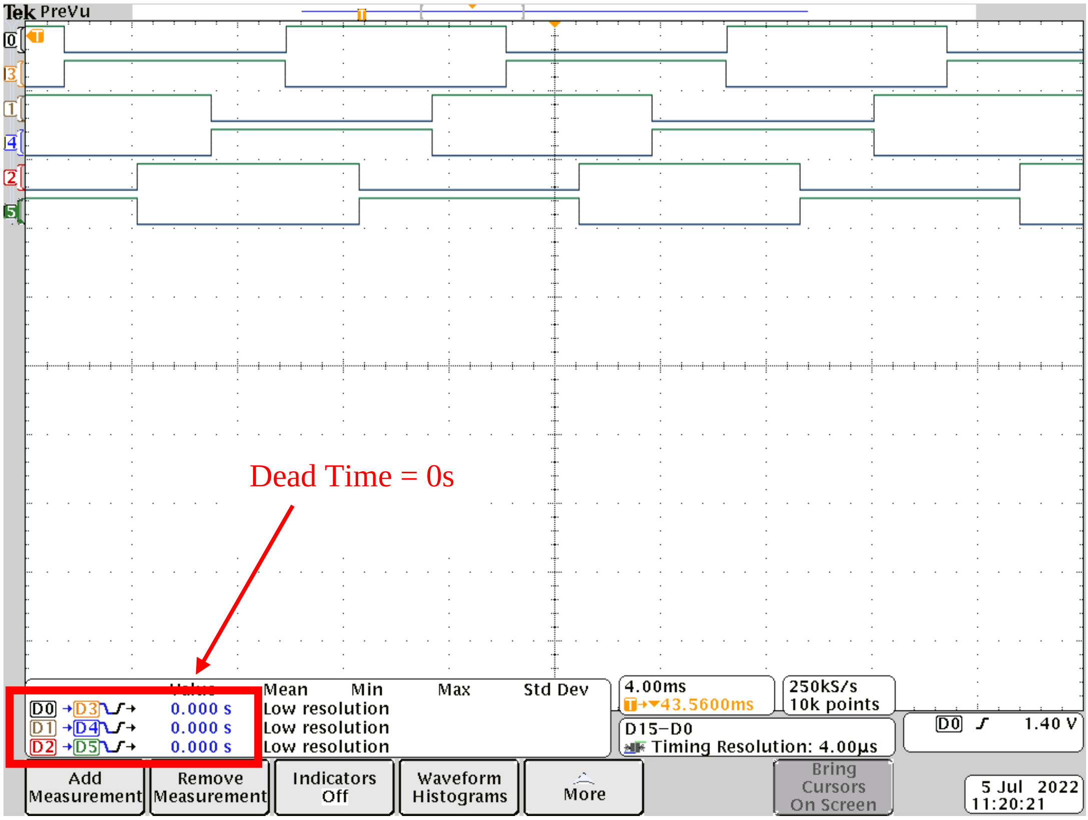
Loading Progress

Cmd_Packet-Firmware_Loading: 03E6 03E2 06E2 05E2 05E2 04E2 0300 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Fig. 40. Short-Circuit scenario detected.

4.3 Firmware with missing deadtime (Malicious)

Subsequently, malicious firmware devoid of any deadtime was transmitted to the FPGA. As depicted in Fig. 41, the firmware lacked deadtime, as denoted by the measurement indicators positioned at the lower-left corner, indicating the deadtime between channels 0 (D0) and 3 (D3) was zero seconds. The same can be observed between channels 1 (D1) and 4 (D4) and between channels 2 (D5) and 5 (D5). Firmware such as this (lacking deadtime) could harm the inverter's components and possibly jeopardize the grid by causing short-circuits for a short period. Upon detecting the deadtime error, the controller answered by rejecting the new malicious firmware and instead reverted to the backup firmware, as evidenced by the observation in Fig. 42.



Dead Time = 0s

```
            Value     Mean            Min       Max       Std Dev
D0 →D3    0.000 s    Low resolution
D1 →D4    0.000 s    Low resolution
D2 →D5    0.000 s    Low resolution
```

4.00ms
T→▼43.5600ms

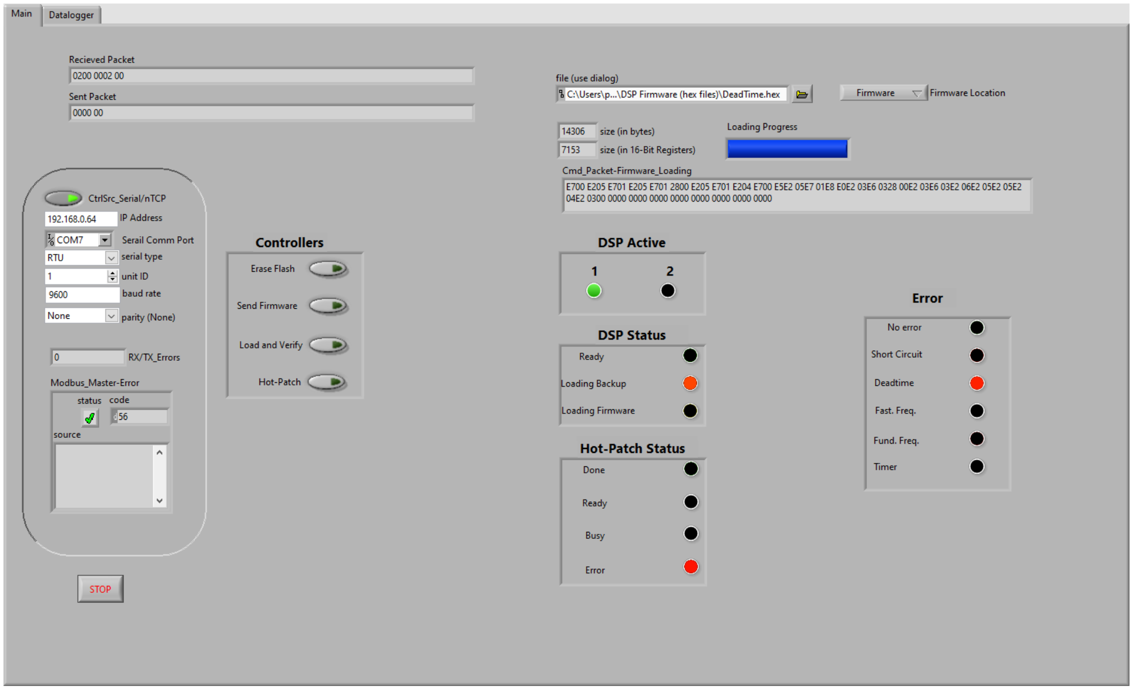
250kS/s
10k points

D15–D0
Timing Resolution: 4.00µs

D0 ⌠ 1.40 V

5 Jul 2022
11:20:21

Fig. 41. DSP firmware without deadtime



Recieved Packet
0200 0002 00

Sent Packet
0000 00

CtrlSrc_Serial/nTCP
192.168.0.64 IP Address
COM7 Serail Comm Port
RTU serial type
1 unit ID
9600 baud rate
None parity (None)

0 RX/TX_Errors

Modbus_Master-Error
status ✓ code 56
source

file (use dialog)
C:\Users\p...\DSP Firmware (hex files)\DeadTime.hex

Firmware Firmware Location

14306 size (in bytes)
7153 size (in 16-Bit Registers)

Loading Progress

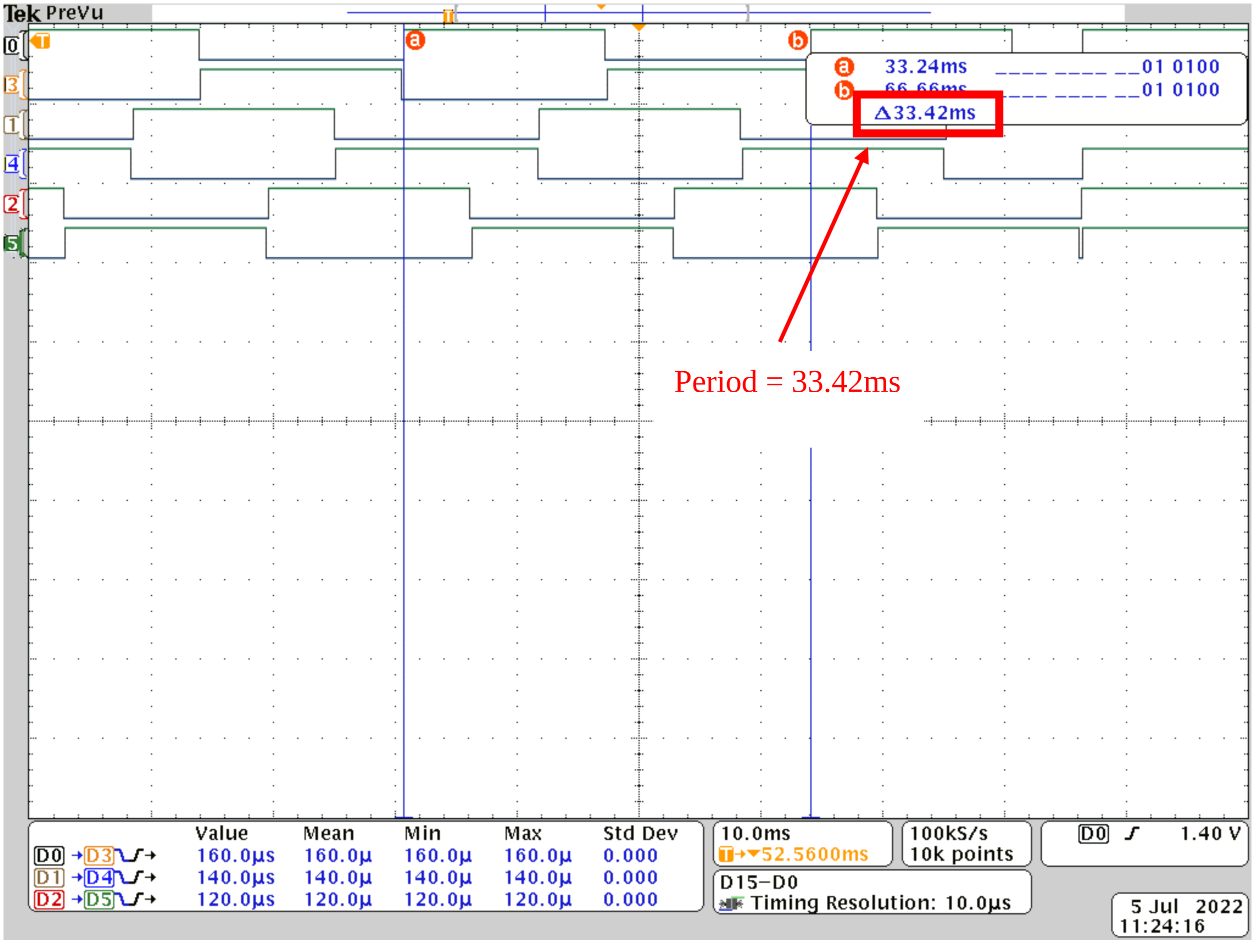
Cmd_Packet-Firmware_Loading

```
E700 E205 E701 E205 E701 2800 E205 E701 E204 E700 E5E2 05E7 01E8 E0E2 03E6 0328 00E2 03E6 03E2 06E2 05E2 05E2
04E2 0300 0000 0000 0000 0000 0000 0000 0000 0000
```

Fig. 42. Missing deadtime detected.

4.4 Firmware with a fundamental frequency different than 60Hz (Malicious)

If a firmware has a suitable deadtime and is devoid of short-circuit scenarios, it is still essential to confirm its fundamental frequency. Operating at an inappropriate fundamental frequency, such as 30 Hz, can have a detrimental effect on the power grid. Therefore, in such cases,\ the controller mechanism must reject the firmware to safeguard the inverter hardware and protect the power grid. Fig. 43 depicts a firmware with a fundamental frequency of 30 Hz that falls beyond the acceptable operational range.



```
a   33.24ms   ____ ____ __01 0100
b   66.66ms   ____ ____ __01 0100
    Δ33.42ms
```

Period = 33.42ms

	Value	Mean	Min	Max	Std Dev
D0 →D3	160.0µs	160.0µ	160.0µ	160.0µ	0.000
D1 →D4	140.0µs	140.0µ	140.0µ	140.0µ	0.000
D2 →D5	120.0µs	120.0µ	120.0µ	120.0µ	0.000

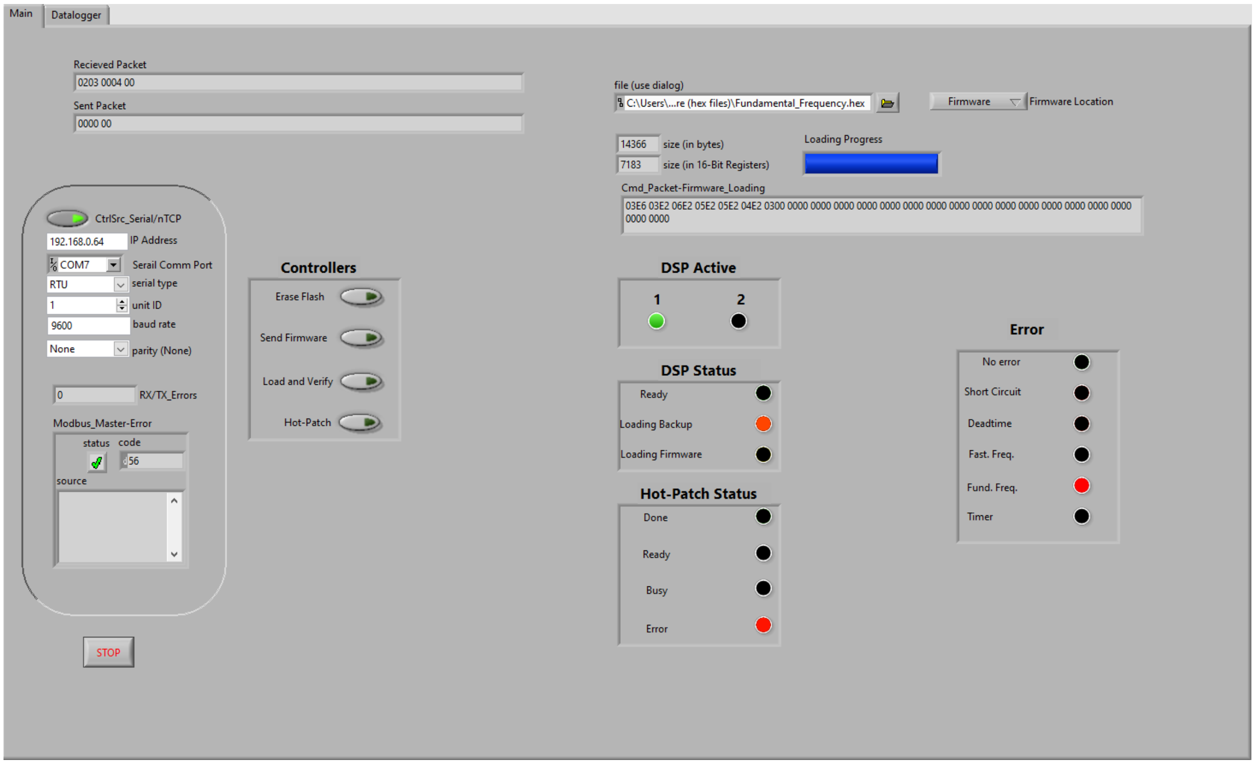
```
10.0ms           100kS/s        D0   1.40 V
T→▼52.5600ms     10k points
D15–D0
Timing Resolution: 10.0µs
                                5 Jul  2022
                                11:24:16
```

Fig. 43. DSP firmware with a fundamental frequency of 30Hz

The top right corner of Fig. 43 shows a value of 33.42ms, which corresponds to the period of the signal on channel 0 and 3. This period indicates a 30Hz signal rather than the intended frequency of 60Hz (16.66ms), as presented in the following equations. Therefore, the expected controller response is to reject the new firmware and load the backup firmware. Fig. 44 displays the FPGA's reaction to this malicious firmware, which includes rejecting it and loading the backup firmware as expected, as well as pointing that the firmware has an error with its fundamental frequency.

$$\mathrm{Period}_{60\mathrm{Hz}} = \frac{1}{60} \approx 16.67\mathrm{ms}$$

$$\mathrm{Period}_{30\mathrm{Hz}} = \frac{1}{30} \approx 33.33\mathrm{ms}$$



Recieved Packet
0203 0004 00

Sent Packet
0000 00

192.168.0.64 IP Address
COM7 Serail Comm Port
RTU serial type
1 unit ID
9600 baud rate
None parity (None)
0 RX/TX_Errors

Modbus_Master-Error
status ✓ code 56
source

file (use dialog)
C:\Users\...re (hex files)\Fundamental_Frequency.hex
Firmware Firmware Location

14366 size (in bytes)
7183 size (in 16-Bit Registers)

Cmd_Packet-Firmware_Loading
03E6 03E2 06E2 05E2 05E2 04E2 0300 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Fig. 44. Fundamental Frequency different than 60 Hz

4.5 Fast Frequency not matching 42 kHz firmware (Malicious)

If firmware with a different frequency than what was established is uploaded, it might not cause any harm to the inverter or the grid. However, it could still significantly impact the inverter's performance by changing the semiconductor's response and increasing the power loss or, in the worst case, reducing the semiconductors' life usage.

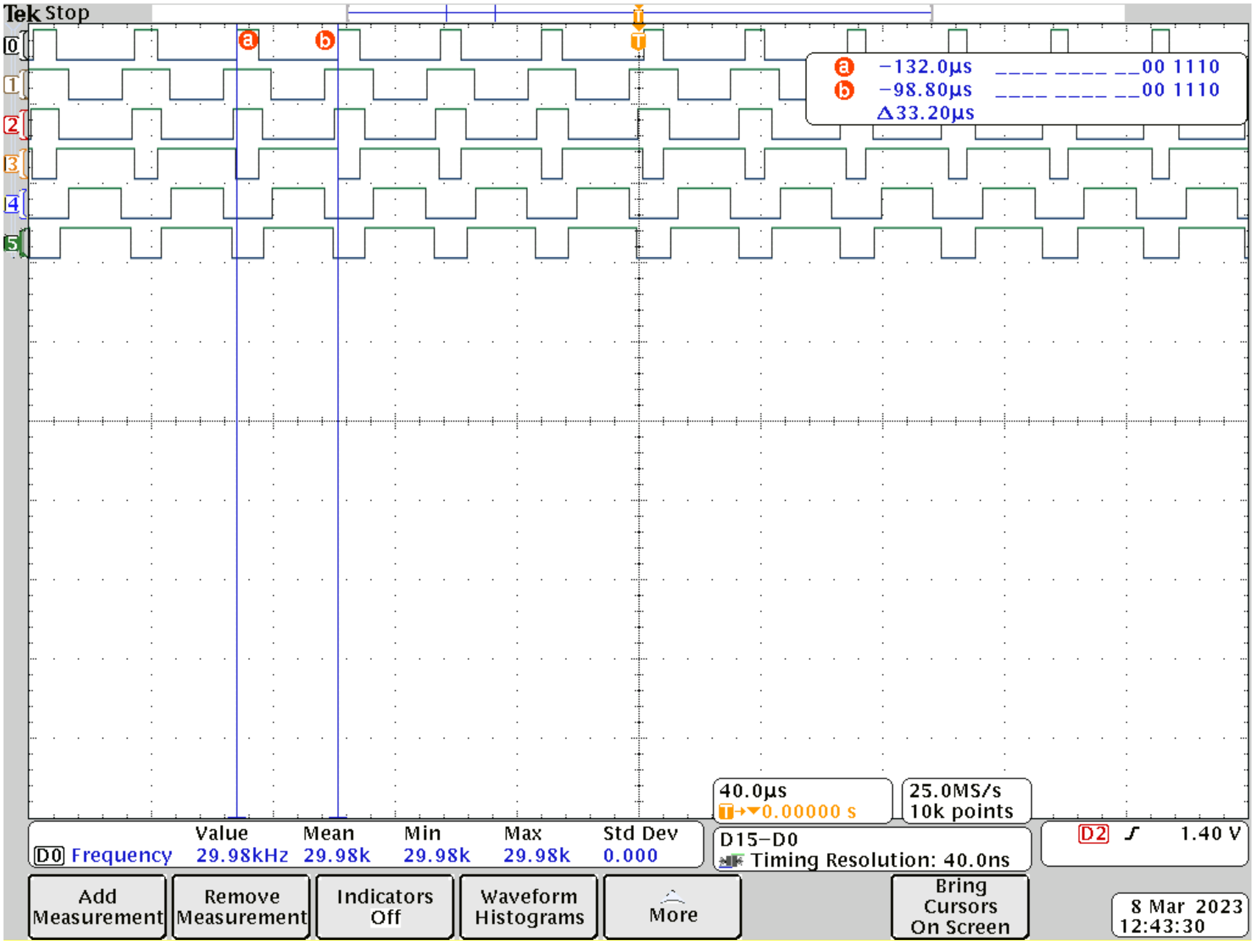
Fig. 45 shows an example of firmware using a high frequency of 30 kHz instead of 42 kHz. The frequency can be found in the bottom-left corner of Fig. 45 and is represented by the difference between cursors "a" and "b" in the top-right corner of the respected picture. The period represented by the difference between the cursors is 33.20 µs, representing a 30 kHz frequency, as illustrated by the following equation.

$$\text{Period}_{30\text{kHz}} = \frac{1}{30000} \approx 33.33\mu\text{s}$$

Table 9 represents the relationship between the channels depicted in Fig. 45 and the correspondent transistors.

Table 9. Fast transistors and channels relationship

Phase A	
Q2	Channel 0
Q3	Channel 1
Phase B	
Q2	Channel 2
Q3	Channel 3
Phase C	
Q2	Channel 4
Q3	Channel 5



a −132.0µs ____ ____ __00 1110
b −98.80µs ____ ____ __00 1110
Δ33.20µs

	Value	Mean	Min	Max	Std Dev
D0 Frequency	29.98kHz	29.98k	29.98k	29.98k	0.000

40.0µs
T→▼0.00000 s

25.0MS/s
10k points

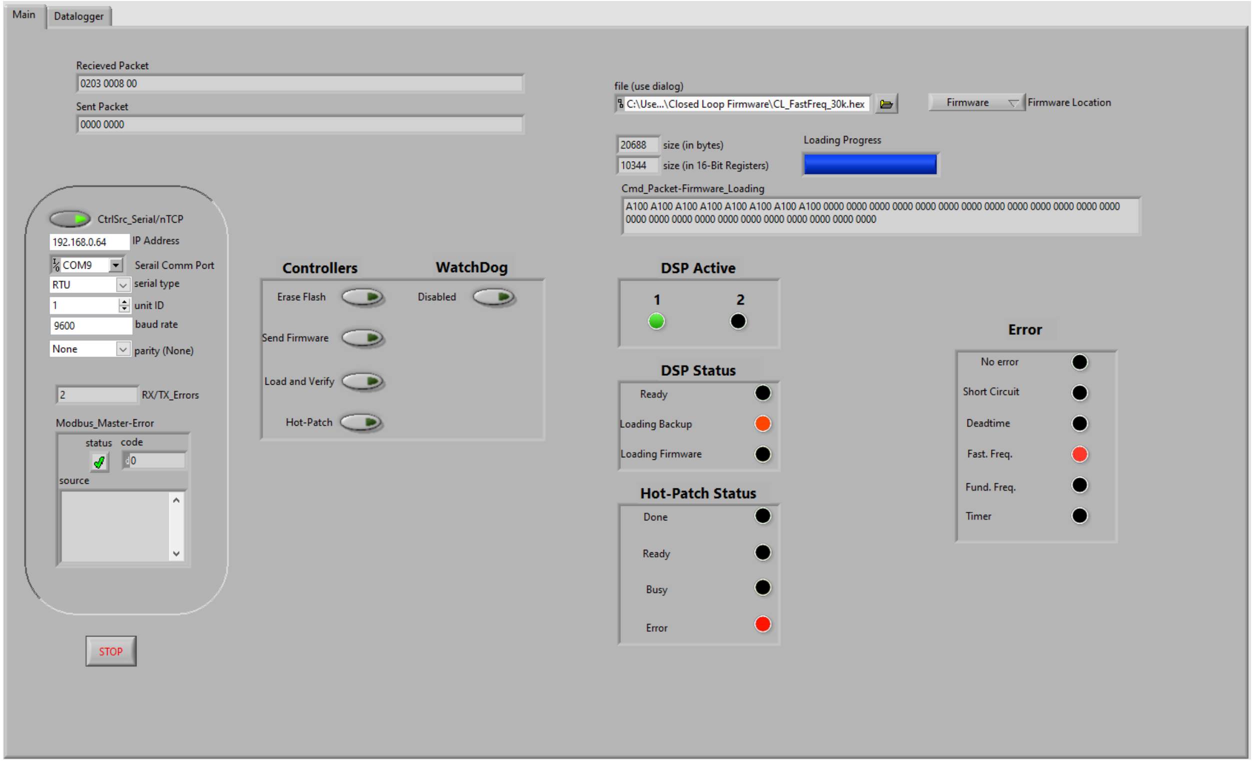
D15–D0
Timing Resolution: 40.0ns

D2 ⨍ 1.40 V

8 Mar 2023
12:43:30

Fig. 45. DSP firmware with a fast frequency of 42 kHz

As per the predetermined invalid firmware contingency plan, the controller was expected to reject the malfunctioning firmware and initiate the loading of the backup firmware. In Fig. 46, the FPGA's response to the detrimental firmware is seen, where it carries out the expected action of rejecting it and loading the backup firmware. This response further indicates that the firmware contains an error related to its fast frequency, as identified by the FPGA.



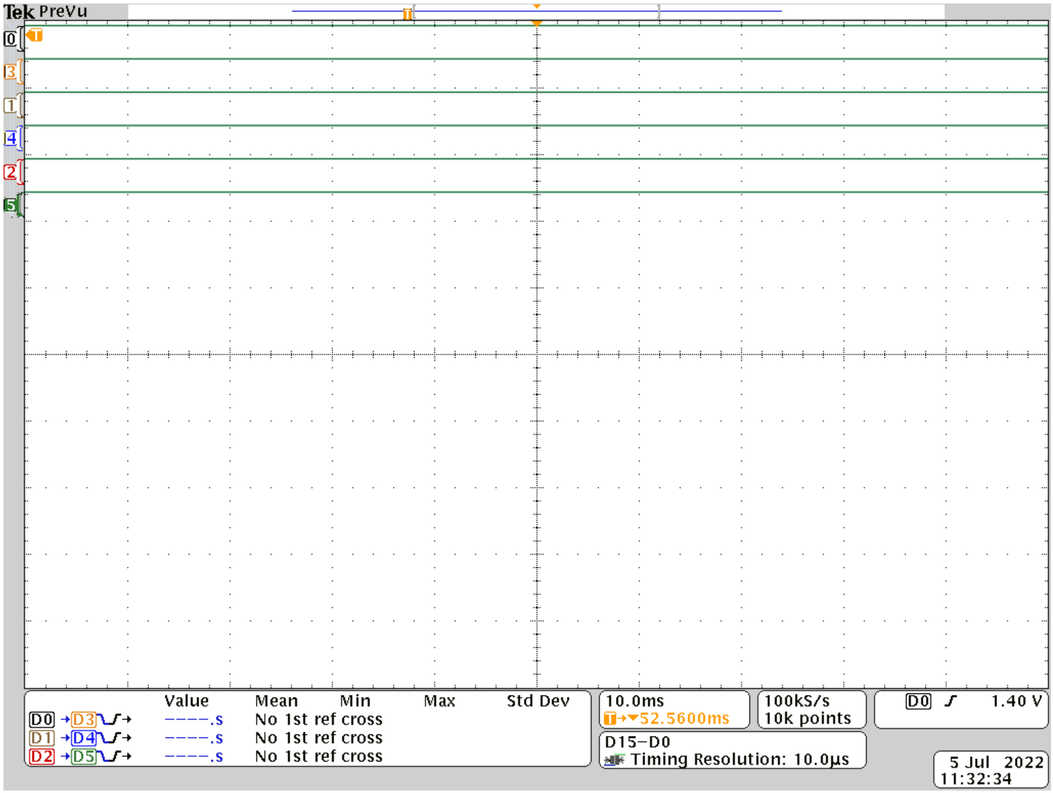
Recieved Packet: 0203 0008 00
Sent Packet: 0000 0000
CtrlSrc_Serial/nTCP
IP Address: 192.168.0.64
Serail Comm Port: COM9
serial type: RTU
unit ID: 1
baud rate: 9600
parity (None): None
RX/TX_Errors: 2
Modbus_Master-Error: status ✓ code 0 source
file (use dialog): C:\Use...\Closed Loop Firmware\CL_FastFreq_30k.hex
Firmware Location: Firmware
size (in bytes): 20688
size (in 16-Bit Registers): 10344
Loading Progress
Cmd_Packet-Firmware_Loading: A100 A100 A100 A100 A100 A100 A100 A100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Fig. 46. Fast Frequency different than 42 kHz

4.6 Stall state firmware (Malicious)

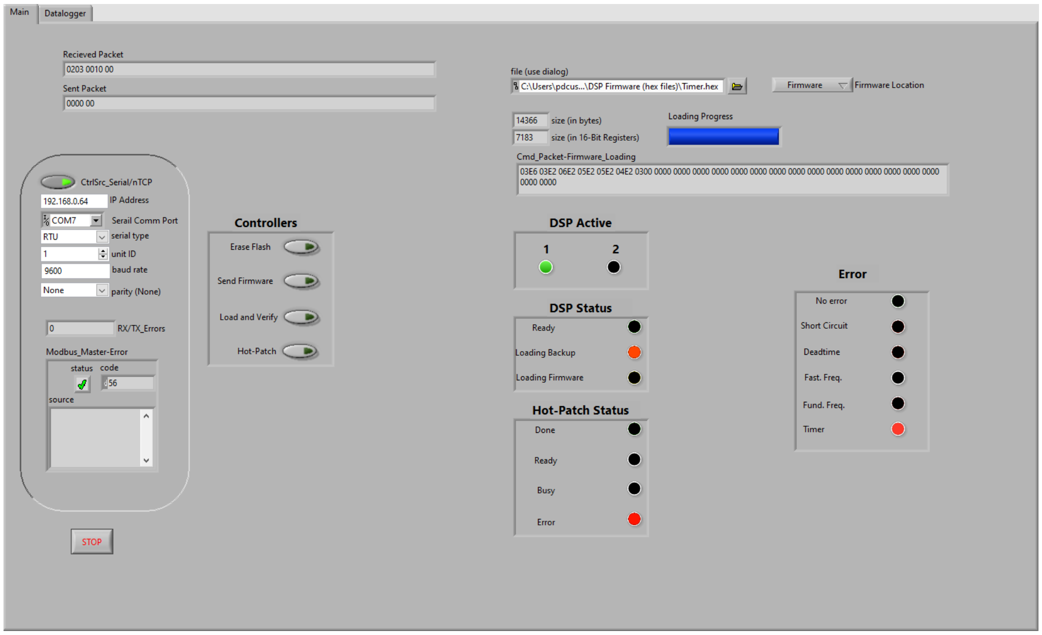
Lastly, firmware that has constant values instead of PWM signals was loaded into the controller. This type of firmware was illustrated in Fig. 47, where all DSP output signals remain constant, which could put the inverter in a stall position, thus disabling it and affecting the grid’s power availability.

The expected controller’s response for this kind of firmware is a rejection, followed by the loading of the backup firmware along with displaying the “Timer” error in LabVIEW. Fig. 48 shows the response of the DT confirming that the controller enhances the system’s security, is acting as expected, and is rejecting the firmware. Additionally, the system prevents the now-known malicious firmware from being active by tagging the firmware as malicious and indicating that the watchdog was not disabled, triggering the timer error.



```
                  Value     Mean       Min       Max      Std Dev     10.0ms           100kS/s        D0  ʃ    1.40 V
D0 →D3            ----.s    No 1st ref cross                          T→▾52.5600ms      10k points
D1 →D4            ----.s    No 1st ref cross                          D15–D0
D2 →D5            ----.s    No 1st ref cross                          Timing Resolution: 10.0µs    5 Jul  2022
                                                                                                   11:32:34
```

Fig. 47. Stall Firmware



Recieved Packet: 0203 0010 00

Sent Packet: 0000 00

IP Address: 192.168.0.64

Serail Comm Port: COM7

serial type: RTU

unit ID: 1

baud rate: 9600

parity (None): None

RX/TX_Errors: 0

Modbus_Master-Error: status ✓ code 56 source

file (use dialog): C:\Users\pdcus...\DSP Firmware (hex files)\Timer.hex

Firmware Location: Firmware

14366 size (in bytes)

7183 size (in 16-Bit Registers)

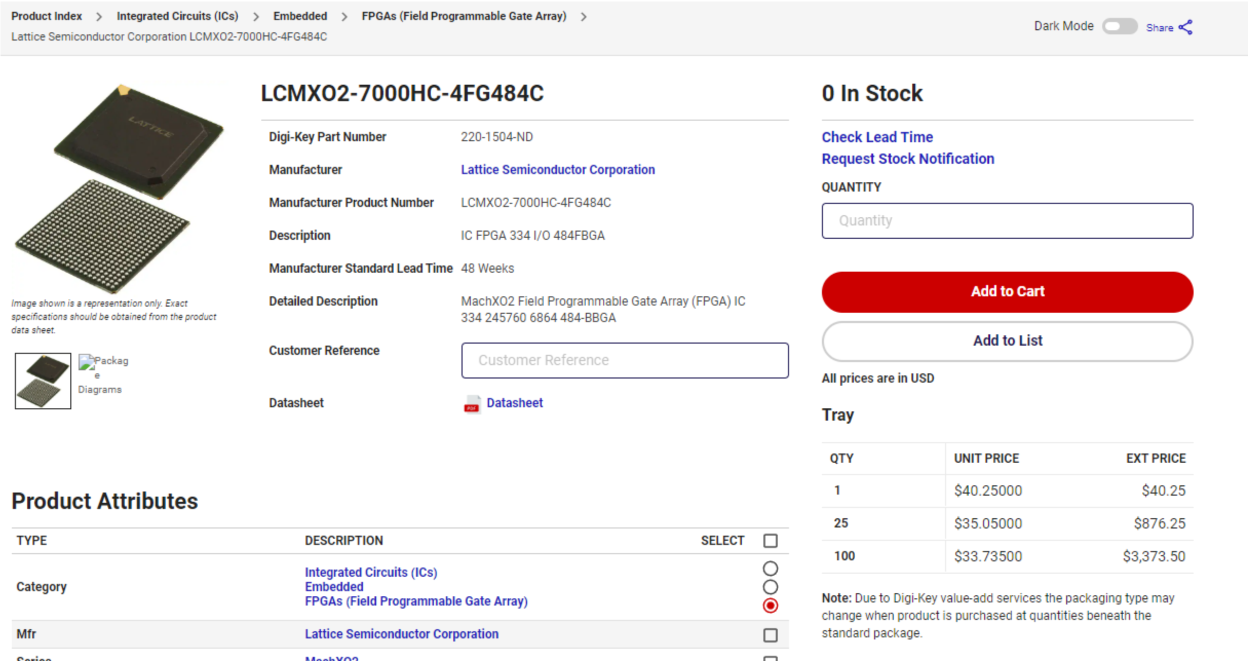
Cmd_Packet-Firmware_Loading:
03E6 03E2 06E2 05E2 05E2 04E2 0300 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Fig. 48. Timer Error

4.7 Cost Analysis

This work introduced a solution enabling hot-patching, reducing downtime, and incorporating a DT replicating an ANPC. This replication allows for testing new patches for errors before implementing them while also serving as a monitoring tool for system health and performance. The standby DSP receives signals from the real ANPC controlled by the active DSP, enhancing the system's resilience against cyber-attacks aimed at compromising the inverter and power grid with malicious firmware updates.

Furthermore, these advantages can be incorporated into an affordable inverter costing less than two hundred dollars. This cost estimation takes into account the expense of one FPGA and two DPS. Based on the prices provided in Fig. 49, and Fig. 50, the price range for a single unit or a hundred units falls between $178.25 and $171.73, respectively. These figures represent the lowest prices discovered during the preparation of this research.



Product Index > Integrated Circuits (ICs) > Embedded > FPGAs (Field Programmable Gate Array) >
Lattice Semiconductor Corporation LCMXO2-7000HC-4FG484C

Dark Mode Share

LCMXO2-7000HC-4FG484C

Digi-Key Part Number	220-1504-ND
Manufacturer	Lattice Semiconductor Corporation
Manufacturer Product Number	LCMXO2-7000HC-4FG484C
Description	IC FPGA 334 I/O 484FBGA
Manufacturer Standard Lead Time	48 Weeks
Detailed Description	MachXO2 Field Programmable Gate Array (FPGA) IC 334 245760 6864 484-BBGA
Customer Reference	Customer Reference
Datasheet	Datasheet

Image shown is a representation only. Exact specifications should be obtained from the product data sheet.

Package Diagrams

0 In Stock

Check Lead Time
Request Stock Notification

QUANTITY

Quantity

Add to Cart

Add to List

All prices are in USD

Tray

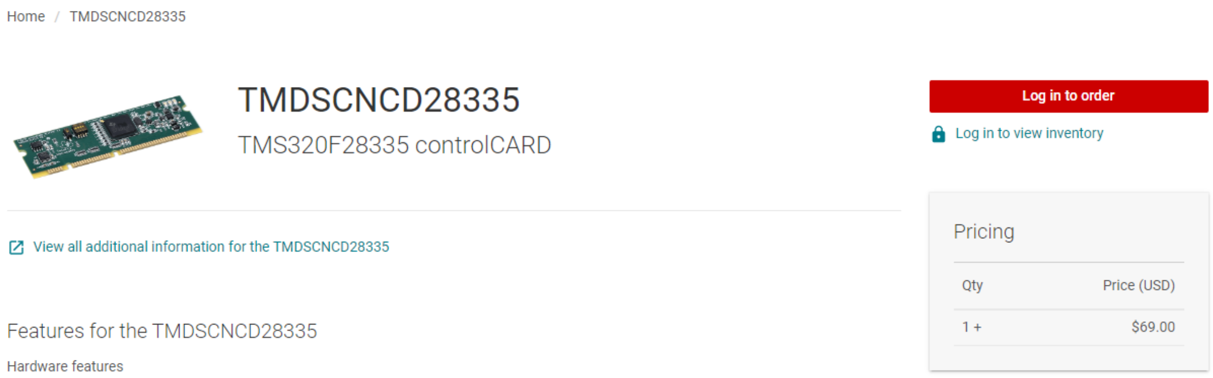
QTY	UNIT PRICE	EXT PRICE
1	$40.25000	$40.25
25	$35.05000	$876.25
100	$33.73500	$3,373.50

Note: Due to Digi-Key value-add services the packaging type may change when product is purchased at quantities beneath the standard package.

Product Attributes

TYPE	DESCRIPTION	SELECT □
Category	Integrated Circuits (ICs) Embedded FPGAs (Field Programmable Gate Array)	○ ○ ●
Mfr	Lattice Semiconductor Corporation	□
Series	MachXO2	□

Fig. 49. MachXO2 Price [32]



Home / TMDSCNCD28335

TMDSCNCD28335

TMS320F28335 controlCARD

Log in to order

Log in to view inventory

View all additional information for the TMDSCNCD28335

Features for the TMDSCNCD28335

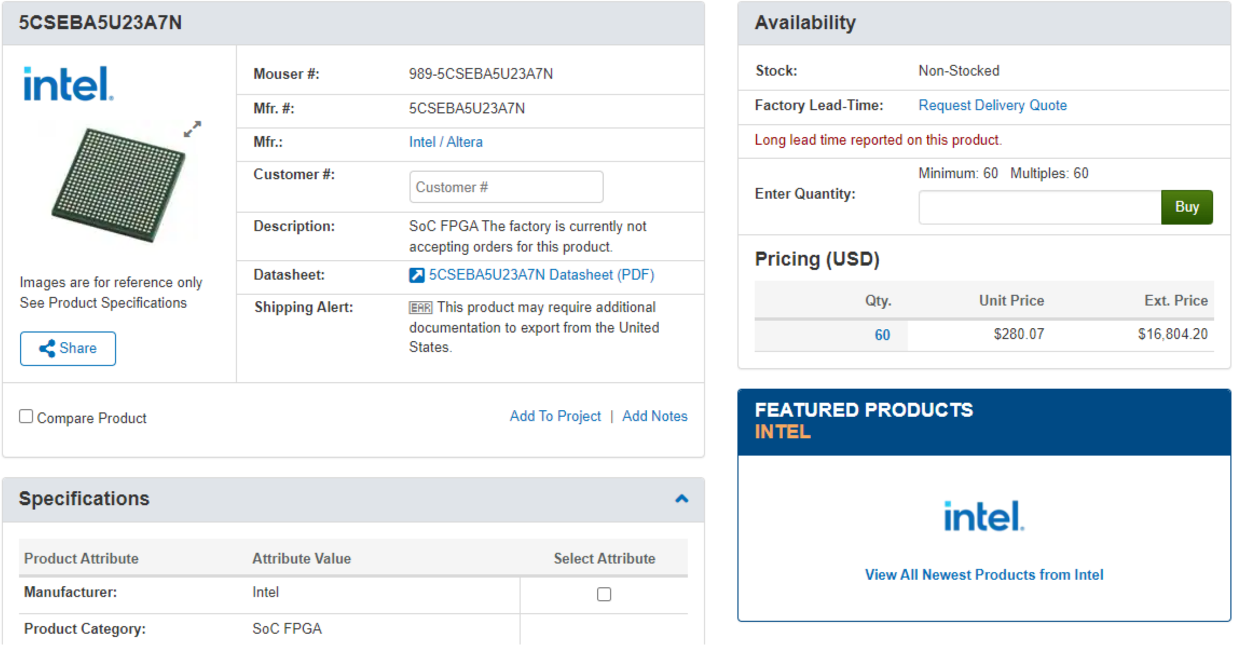
Hardware features

Pricing

Qty	Price (USD)
1 +	$69.00

Fig. 50. DSP controlCard Price [32]

Moreover, the inverter employed in this study, which had its control system replaced by the UCB discussed in this work, incorporated a distinct FPGA within its system architecture, specifically the Intel Altera 5CSEBA5U23A7N. The cost of this FPGA ranged from $280.07 to $308.07, as indicated in Fig. 51 and Fig. 52, respectively. Consequently, considering solely the components discussed, integrating the FPGA and the two DSPs described in this research proves to be more cost-effective than the FPGA currently utilized in commercial inverters.



5CSEBA5U23A7N

intel.

Images are for reference only
See Product Specifications

Share

Mouser #:	989-5CSEBA5U23A7N
Mfr. #:	5CSEBA5U23A7N
Mfr.:	Intel / Altera
Customer #:	Customer #
Description:	SoC FPGA The factory is currently not accepting orders for this product.
Datasheet:	5CSEBA5U23A7N Datasheet (PDF)
Shipping Alert:	EAR This product may require additional documentation to export from the United States.

□ Compare Product

Add To Project | Add Notes

Specifications

Product Attribute	Attribute Value	Select Attribute
Manufacturer:	Intel	□
Product Category:	SoC FPGA	

Availability

Stock:	Non-Stocked
Factory Lead-Time:	Request Delivery Quote

Long lead time reported on this product.

Enter Quantity: Minimum: 60 Multiples: 60

Buy

Pricing (USD)

Qty.	Unit Price	Ext. Price
60	$280.07	$16,804.20

FEATURED PRODUCTS
INTEL

intel.

View All Newest Products from Intel

Fig. 51. FPGA Intel Altera [23]

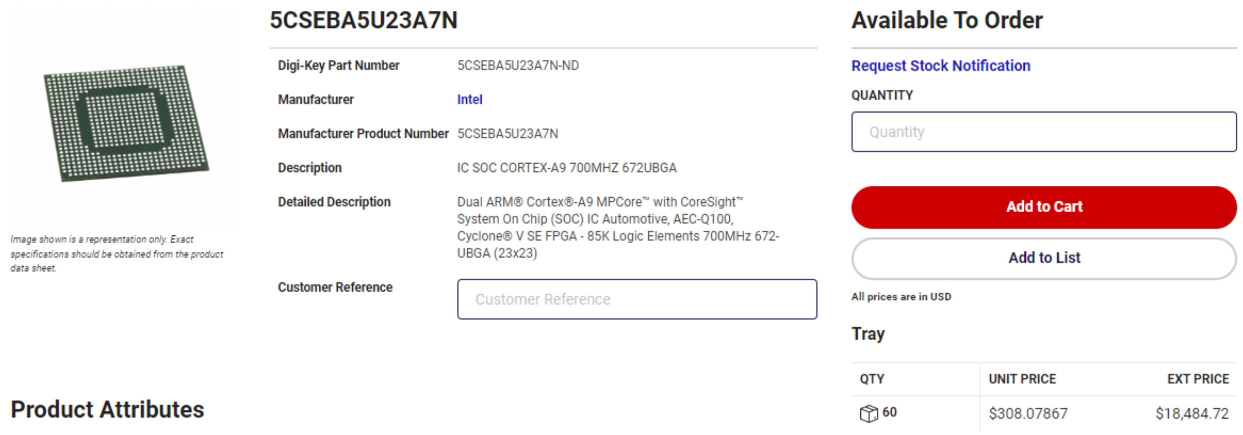


Image shown is a representation only. Exact specifications should be obtained from the product data sheet.

5CSEBA5U23A7N

Digi-Key Part Number	5CSEBA5U23A7N-ND
Manufacturer	Intel
Manufacturer Product Number	5CSEBA5U23A7N
Description	IC SOC CORTEX-A9 700MHZ 672UBGA
Detailed Description	Dual ARM® Cortex®-A9 MPCore™ with CoreSight™ System On Chip (SOC) IC Automotive, AEC-Q100, Cyclone® V SE FPGA - 85K Logic Elements 700MHz 672-UBGA (23x23)
Customer Reference	

Available To Order

QUANTITY

All prices are in USD

Tray

QTY	UNIT PRICE	EXT PRICE
60	$308.07867	$18,484.72

Product Attributes

Fig. 52. FPGA Intel Altera [32]

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

The proliferation of technology, the widespread accessibility of high-speed internet, and the growing connectivity of power grids to various networks have brought about numerous advantages in the field of distributed energy resource applications. These benefits include heightened awareness of the situation, multi-disciplinary event response strategies support, and sophisticated secondary and tertiary controls that enhance grid efficiency and resilience. Nevertheless, linking these devices to the internet allows for additional attack vectors, introducing vulnerabilities to the power grid. Therefore, it is vital to improve the cybersecurity of these devices to safeguard against cyber-attacks.

A new system was developed to safeguard the hardware layer of distributed energy resource controls and block many cyber-attacks that try to manipulate the inverter's behavior. This system builds upon the previous work presented in [4] and has demonstrated its ability to replicate and emulate a 3-level ANPC inverter output while ensuring the new DSP firmware meets all the firmware validation requirements of the DFTr system.

Furthermore, the advantages outlined in this study, including the ability to perform firmware hot-patching to minimize downtime, enhancing cyber-security robustness, and real-time monitoring of the inverter using the DT, can be seamlessly incorporated into a commercial inverter without any cost increase. In fact, the integration of the FPGA and two DSPs utilized in this research would cost less than two hundred dollars. In contrast, the lowest price range for the FPGA currently employed in commercial inverters is between two hundred and eighty to three hundred and eight dollars.

This project provided the foundation and system structure for future work to modify the emulation architecture to accommodate different inverter topologies. This system can also be improved by expanding the tests in the DFTr, which could guarantee the ANPC inverter does not suffer a performance downgrade by indicating a subpar system change when being updated to a new firmware with lower performance.

In addition, an improvement that could be added is to check the backup firmware for malicious firmware. This work assumed that the backup firmware is non-malicious, but this new test is important in increasing the system's cyber-robustness.

Finally, two other improvements can be added to the system: an authentication process that verifies who is sending commands to the inverter and rejects commands being sent from unknown sources; and an addition of a real-time system health monitor that would alert the user to an error in case of a fault within the ANPC inverter. To clarify, if a transistor within the inverter malfunctions, the ANPC inverter would change its output signals, and as this is a closed-loop system, the DSP firmware would change its response. After reading the DSP's response, the FPGA could identify a problem and alert the user that the ANPC inverter is malfunctioning. This implementation could be done by switching the user request for the DT response, as described in this work, to an automated request of the DT output – request every second, for example – and using the DT output to diagnose the ANPC inverter's health.

References

[1] H. Bing, M. Mohammad and B. Ross, "Case Study of Power System Cyber Attack Using Cascading Outage Analysis Model," in *2018 IEEE Power & Energy Society General Meeting (PESGM)*, Portland, OR, USA, 2018.

[2] P. Custodio, B. McBride, T. Le, J. Jackson, K. Haulmark, J. Di, C. Farnell and H. A. Mantooth, "Digital Twin of an ANPC inverter with integrated Design-For-Trust," in *IEEE Design Methodologies Conference (DMC)*, Bath, 2022, pp1-7, doi: 10.1109/DMC55175.2022.9906472.

[3] "Renewable electricity growth is accelerating faster than ever worldwide, supporting the emergence of the new global energy economy," International Energy Agency, 01 December 2021. [Online]. Available: https://www.iea.org/news/renewable-electricity-growth-is-accelerating-faster-than-ever-worldwide-supporting-the-emergence-of-the-new-global-energy-economy. [Accessed 17 January 2023].

[4] C. Farnell, E. Soria, J. Jackson and H. A. Mantooth, "Cyber Protection of Grid-Connected Devices Through Embedded Online Security," in *IEEE Design Methodologies Conference (DMC)*, Bath, 2021, pp. 1-6, doi: 10.1109/DMC51747.2021.9529935.

[5] B. Huang, M. Majidi and R. Baldick, "Case Study of Power System Cyber Attack Using Cascading Outage Analysis Model," in *IEEE Power & Energy Society General Meeting (PESGM)*, Portland, OR, USA, 2018, pp. 1-5, doi: 10.1109/PESGM.2018.8585921..

[6] Q. Wang, W. Tai, T. Yi and M. Ni, "National Vulnerability Database," *IET Cyber-Physical Systems: Theory & Applications*, vol. 4, no. 2, pp. 101-107, Jun 2019.

[7] "Colonial Pipeline Ransomware Attack Rattles Power Industry, Renews Vulnerability," Power Maganize, 2021. [Online]. Available: https://www.powermag.com/colonial-pipeline-ransomware-attack-rattles-power-industryrenews-vulnerability-concerns/. [Accessed 20 December 2022].

[8] "Solar Photovoltaic Technology Basics," NREL, [Online]. Available: https://www.nrel.gov/research/re-photovoltaics.html. [Accessed 19 February 2023].

[9] "Photovoltaics and electricity," U.S. Energy Information Administration, [Online]. Available: https://www.nrel.gov/research/re-photovoltaics.html. [Accessed 19 February 2023].

[10] "How a Photovoltaic Power Plant Works? Construction and Working of a Solar Power Plant," Electrical Technology, [Online]. Available: https://www.electricaltechnology.org/2021/07/solar-power-plant.html. [Accessed 19 February 2023].

[11] A. S. Andrade, J. H. Muniz and E. R. Silva, "Three-level Hybrid Flying Dc-Source ANPC inverter: Application as a photovoltaic AC source," in *2015 IEEE 24th International Symposium on Industrial Electronics (ISIE)*, Buzios, Brazil, 2015, pp. 1094-1099, doi: 10.1109, 2015.

[12] D. Woldegiorgis, Y. Wu, Y. Wei and H. A. Mantooth, "A High Efficiency," *IEEE Open Journal of Industry Applications,* vol. 2, pp. 154-167, 2021, doi: 10.1109/OJIA.2021.3091549.

[13] Y. Deng, J. Li, K. H. Shin, T. Viitanen, M. Saeedifard and R. G. Harley, "Improved Modulation Scheme for Loss Balancing of Three-Level Active NPC," *IEEE Transactions*

on Power Electronics, vol. 32, no. 4, pp. 2521-2532, 2017, doi: 10.1109/TPEL.2016.2573823.

[14] T. Bruckner, S. Bernet and P. K. Steimer, "Feedforward Loss Control of Three-Level Active NPC Converters," *IEEE Transactions on Industry Applications,* vol. 43, no. 6, pp. 1588-1596, 2007, doi: 10.1109/TIA.2007.908164.

[15] B. Zhang and D. Qiu, "Introduction," in *m-Mode SVPWM Technique for Power Converters,* Singapore, Springer, 2019, doi: 10.1007/978-981-13-1382-0_1, p. 5.

[16] M. Mohamed, A. Elmahalawy and H. Harb, "Developing the pulse width modulation tool (PWMT) for two timer mechanism technique in microcontrollers," in *Second International Japan-Egypt Conference on Electronics, Communications and Computers (JEC-ECC),* 2013, pp.148-153, doi: 10.1109/JEC-ECC.2013.6766403.

[17] M. Vujacic, M. Hammami, M. Srndovic and G. Grandi, "Theoretical and Experimental Investigation of Swiching Ripple in the DC-Link Voltage of Single-Phase H-Bridge PWM Inverters," *Energies,* vol. 10, no. 8, 2017, doi: 10.3390/en10081189.

[18] S. H. L., "All about circuits," 09 November 2017. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/purpose-and-internal-functionality-of-fpga-look-up-tables/. [Accessed 14 February 2023].

[19] L. Semiconductor, "MachX02 Family Data Sheet," February 2022. [Online]. Available: https://www.latticesemi.com/view_document?document_id=38834. [Accessed 23 December 2022].

[20] T. Instrument, "TMS320x2833x, TMS320x2823x Technical Reference Manual," March 2020. [Online]. Available: https://www.ti.com/lit/pdf/sprui07. [Accessed 23 December 2022].

[21] P. S. Mutha and Y. M. Vaidya, "FPGA reconfiguration using UART and SPI flash," in *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, Tirunelveli, India, 2017.

[22] D. Trivedi, A. Khade, K. Jain and R. Jadhav, "SPI to I2C Protocol Conversion Using Verilog," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, Pune, India, 2018.

[23] "mouser.com," [Online]. Available: https://www.mouser.com/datasheet/2/671/mict_s_a0006161798_1-2291050.pdf. [Accessed 01 March 2023].

[24] M. Grieves and J. Vickers, Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems, J. Kahlen, S. Flumerfelt and A. Alves, Eds., Springer, Cham. https://doi.org/10.1007/978-3-319-38756-7_4, 2016, pp. 85-113.

[25] T. Uhlemann, C. Schock, C. Lehmann, S. Freiberger and R. Steinhilper, "The Digital Twin: Demonstrating the Potential of Real Time Data Acquisition in Production Sytems," *Procedia Manufacturing*, no. 9, pp. 113-120, 2017.

[26] J. Wu, Y. Yang, X. Chegn, H. Zuo and Z. Cheng, "The Development of Digital Twin Technology Review," in *2020 Chinese Automation Congress (CAC)*, Shanghai, China, 2020.

[27] R. He, G. Chen, C. Dong, S. Sun and X. Shen, "Data-driven digital twin technology for optimized control process systems," in *ISA Transactions (2019)*, https://doi.org/10.1016/j.isatra.2019.05.011, 2019.

[28] A. Bhatia, "Introduction to Short Circuit Analysis," PDG Online, Fairfax, 2020.

[29] P. Skarolek and J. Letti, "Influence of Deadtime on Si, SiC and GaN Converters," in *2020 21st International Scientific Conference on Electric Power Engineering (EPE)*, Prague, Czech Republic, 2020, pp. 1-4, doi: 10.1109/EPE51172.2020.9269208.

[30] M. Folgueras, E. Wenger, A. Florita, K. Clark and V. Gevorgian, "Grid Frequency Extreme Event Analysis and Modeling," in *2017 International Workshop on Large-Scale Integration of Wind Power into Power Systems as well as on Transmission Networks for Offshore Wind Power Plants*, Berlin, Germany, 2017.

[31] J. H. Eto, J. Undrill, P. Mackin, R. Daschmans, B. Williams, B. Haney, R. Hunt, J. Ellis, I. Howard, C. Martinez, M. O'Malley, K. Coughlin and K. H. LaCommare, Use of frequency response metrics to assess the planning and operating requirements for reliable integration of variable renewable generation, Berkeley, CA: Lawrence Berkeley National Lab. (LBNL), 2010.

[32] "Digikey," Digikey, [Online]. Available: https://www.digikey.com/. [Accessed 15 May 2023].

APPENDICES

APPENDIX A: VHDL CODE

A-1: Top file

```
--------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Estefano Soria and Paulo Custodio
--
-- Create Date:          26/10/2021
-- Project Name:         Digital_Twin
-- Module Name:          Top
-- Design Name:          Digital_Twin_Top
-- Target Devices:       LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:        Lattice Diamond_x64 Build 3.11
-- Description:
-- This project has the purpose to create a Digital Twin (DT) able to emulate an Active-Neutral
Point Clamped (ANPC) inverter using the inactive/standby DSP outputs
-- to check if the new DSP firmware has all the requirements designed with the Design-For-Trust
(DFTr) technique.
-- The ANPC inverter has 6 transistors per phase, being two Fast Frequency Transistors(Q2 and
Q3) and four Slow Frequency Transistors (Q1,Q4,Q5,Q6).
-- The strategy used to control the ANPC inverter is considering the PWM 01, which controls the
transistor 1 (Q1) the same as Q6, since they must be on and off at the same time.
```

```
-- The same concept was applied to transistors Q4 and Q5, because they also have the same behavior.
-- Slow transistors must have a switching frequency equivalent to the fundamental frequency 60Hz, while the Fast transistors (Q2 and Q3) must have a switching frequency of 42kHz.
--
-- PinOut:
-- ------Inputs------
-- ----DSP 1 (DIMM-B)----
-- --Phase A--
-- F20 -> Q1|Q6
-- M16 -> Q2
-- C22 -> Q3
-- K20 -> Q4|Q5
-- --Phase B--
-- G18 -> Q1|Q6
-- M19 -> Q2
-- C21 -> Q3
-- K18 -> Q4|Q5
-- --Phase C--
-- K22 -> Q1|Q6
-- L22 -> Q2
-- B22 -> Q3
-- J17 -> Q4|Q5
-- ---- END DSP 1 (DIMM-B)----

-- ----DSP 2 (DIMM-C)----

-- --Phase A--

-- AB6 -> Q1|Q6

-- Y7  -> Q2

-- T8  -> Q3

-- U10 -> Q4|Q5

-- --Phase B--

-- Y4  -> Q1|Q6

-- V8  -> Q2

-- U8  -> Q3

-- W11 -> Q4|Q5

-- --Phase C--

-- T10  -> Q1|Q6

-- W9   -> Q2

-- AA8  -> Q3

-- V11  -> Q4|Q5

-- ---- END DSP 2 (DIMM-C)----

-- ----Others----

-- C2  -> Flash SPI Slave Output

-- Y1  -> SCI RX

-- V13 -> SCI RX DSP

-- R3  -> SCI RX WEBSERVER
-- G13 -> Push Button (SW1) -> Erase Flash Memory manually
-- ---- End Others----
-- ------End Inputs------
-- ------Outputs------
-- --Phase A--
-- A21 -> Q1
-- C19 -> Q2
-- A20 -> Q3
-- D18 -> Q4
-- B19 -> Q5
-- C18 -> Q6
-- --Phase B--
-- F17 -> Q1
-- A18 -> Q2
-- D17 -> Q3
-- E17 -> Q4
-- A17 -> Q5
-- C18 -> Q6
-- --Phase C--
-- F16 -> Q1
-- E16 -> Q2
-- D16 -> Q3
-- B15 -> Q4
-- C16 -> Q5

-- E15 -> Q6

-- --LEDs--

-- R17 -> LED A

-- U17 -> LED B

-- T18 -> LED C

-- R16 -> LED D

-- T17 -> LED E

-- Y21 -> LED F

-- Y20 -> LED G

-- U18 -> LED H

-- --Flash Memory--

-- C3  -> Chip-Select (CSSPIN)

-- E4  -> Hold

-- D3  -> SPI Clock (MCLK)

-- F5  -> Slave Input SPI (SISPI)

-- F6  -> Write enable (WPn)

-- --DSPs--

-- G22 -> DIMM_B_GPIO30

-- H16 -> DIMM_B_SCI_RX

-- Y3  -> DIMM_C_GPIO30

-- AB2 -> DIMM_C_SCI_RX

-- F6-> IDC_D_GPIO_02 (DSP1 Reset)
-- F5 -> IDC_D_GPIO_03 (DSP2 Reset)
-- -- Others --
-- AA1 -> SCI_TX
-- V12 -> SCI_TX_DSP
-- R2 -> SCI_TX_Webserver
--
-- Revision:
--      v2.15.22 - Top file without the deadtime component (deadtime should be called by the firmware validation only)
--      v3.24.22 - Added the emulation control and debug signals. Adapted to ANPC inverter
--      v5.23.22 - Polishment and comments to make it easier to comprehend the code for future work.
-- 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.Digital_Twin_Common.all;


entity Digital_Twin_and_Hot_Patching is
  Port
        (
                  ------------------ Communication pins between CPLD and UI         ------------------
-
                  SCI_RX : in std_logic;         --UART RX pin for Serial Comm with UI: UI(TX)
-> CPLD (RX) -> Y1
                  SCI_TX : out std_logic;        --UART TX pin for Serial Comm with UI: UI(RX)
-> CPLD (TX) -> AA1


                  SCI_RX_Webserver : in std_logic;   --UART RX pin for Serial Comm with UI:
UI(TX) -> CPLD (RX) -> R2
    SCI_TX_Webserver : out std_logic;   --UART TX pin for Serial Comm with UI: UI(RX) -
> CPLD (TX) -> R3


                  IDC_D_GPIO_02 : out STD_LOGIC; -- Reset Pin of the DSP DIMM-B -> F6
                  IDC_D_GPIO_03 : out STD_LOGIC; -- Reset Pin of the DSP DIMM-C -> F5
```

```
DIMM_B_SCI_RX : out STD_LOGIC; -- This pin is used to send the firmware to the DSP and the bootloader. -> DIMM-B_GPIO-28 -> H16
DIMM_C_SCI_RX : out STD_LOGIC; -- This pin is used to send the firmware to the DSP and the bootloader. -> DIMM-C_GPIO-28 -> AB2
DIMM_B_SCI_TX : in STD_LOGIC;  -- This pin is used to send the firmware to the DSP and the bootloader. -> DIMM-B_GPIO-29 -> J19
DIMM_C_SCI_TX : in STD_LOGIC;  -- This pin is used to send the firmware to the DSP and the bootloader. -> DIMM-C_GPIO-29 -> U12

DIMM_B_GPIO30 : out STD_LOGIC; -- G22
DIMM_C_GPIO30 : out STD_LOGIC; -- Y3

Btna : in STD_LOGIC;

-------------------------------------- INPUTS --------------------------------------
-- SW : in STD_LOGIC;
------------------- DIMM_B -------------------
-- Phase A
DSP1_01_A : in  STD_LOGIC; -- Q1 -> F20 -> DIMM-B_GPIO-12
DSP1_02_A:  in  STD_LOGIC; -- Q2 -> B22 -> DIMM-B_GPIO-00
DSP1_03_A : in  STD_LOGIC; -- Q3 -> M16 -> DIMM-B_GPIO-01
DSP1_04_A : in  STD_LOGIC; -- Q4 -> K20 -> DIMM-B_GPIO-25
```

```
-- Phase B
DSP1_01_B : IN  STD_LOGIC; -- Q1 -> G18 -> DIMM-B_GPIO-26
DSP1_02_B : IN  STD_LOGIC; -- Q2 -> C22 -> DIMM-B_GPIO-02
DSP1_03_B : IN  STD_LOGIC; -- Q3 -> M19 -> DIMM-B_GPIO-03
DSP1_04_B : IN  STD_LOGIC; -- Q4 -> K18 -> DIMM-B_GPIO-27

-- Phase C
DSP1_01_C : IN  STD_LOGIC; -- Q1 -> K22 -> DIMM-B_GPIO-14
DSP1_02_C : IN  STD_LOGIC; -- Q2 -> C21 -> DIMM-B_GPIO-04
DSP1_03_C : IN  STD_LOGIC; -- Q3 -> L22 -> DIMM-B_GPIO-05
DSP1_04_C : IN  STD_LOGIC; -- Q4 -> J17 -> DIMM-B_GPIO-19

------------------- DIMM_C -------------------
-- Phase A
DSP2_01_A : in  STD_LOGIC; -- Q1 -> AB6 -> DIMM-C_GPIO-12
DSP2_02_A:  in  STD_LOGIC; -- Q2 -> AA8 ->  DIMM-C_GPIO-00
DSP2_03_A : in  STD_LOGIC; -- Q3 -> Y7 ->  DIMM-C_GPIO-01
DSP2_04_A : in  STD_LOGIC; -- Q4 -> U10 -> DIMM-C_GPIO-25

-- Phase B
DSP2_01_B : IN  STD_LOGIC; -- Q1 -> Y4 ->  DIMM-C_GPIO-26
DSP2_02_B : IN  STD_LOGIC; -- Q2 -> T8 ->  DIMM-C_GPIO-02
```

```
DSP2_03_B : IN  STD_LOGIC; -- Q3 -> V8 ->  DIMM-C_GPIO-03

DSP2_04_B : IN  STD_LOGIC; -- Q4 -> W11 -> DIMM-C_GPIO-27


-- Phase C

DSP2_01_C : IN  STD_LOGIC; -- Q1 -> T10 -> DIMM-C_GPIO-14

DSP2_02_C : IN  STD_LOGIC; -- Q2 -> U8 ->  DIMM-C_GPIO-04

DSP2_03_C : IN  STD_LOGIC; -- Q3 -> W9 -> DIMM-C_GPIO-05

DSP2_04_C : IN  STD_LOGIC; -- Q4 -> V11 -> DIMM-C_GPIO-19


------------------- Relays -------------------

DIMM_B_GPIO_32 : IN STD_LOGIC;       -- H17

DIMM_B_GPIO_33 : IN STD_LOGIC;       -- H21

DIMM_C_GPIO_32 : IN STD_LOGIC;       -- V6

DIMM_C_GPIO_33 : IN STD_LOGIC;       -- AA14



--------------------------------------- OUTPUTS -------------------------------------

------------------- Enable DSPs -------------------

-- Enable the DSP to generate the signals

-- DSP 1 (DIMM-B)

DSP1_DSPEnable      : out std_logic; -- M20  -> DIMM-B_GPIO-62

-- DSP 2 (DIMM-C)

DSP2_DSPEnable      : out std_logic; -- AA12 -> DIMM-B_GPIO-62
```

```
------------------- Leds -------------------
LED_A : out STD_LOGIC; -- R17 -> DSP01 is active (D1 from UCB)
LED_B : out STD_LOGIC; -- U17 -> DSP02 is active (D2 from UCB)
--LEDs not being used
LED_C : out STD_LOGIC; -- T18
LED_D : out STD_LOGIC; -- R16
LED_E : out STD_LOGIC; -- T17
LED_F : out STD_LOGIC; -- Y21
LED_G : out STD_LOGIC; -- Y20
LED_H : out STD_LOGIC; -- U18

-- Outputs to control the inverter
-- Phase A

SW01_A : out std_logic; -- Q1 -> V19 -> IDC-B_GPIO-05 -> Pin
SW02_A : out std_logic; -- Q2 -> W20 -> IDC-B_GPIO-04 -> Pin
SW03_A : out std_logic; -- Q3 -> W22 -> IDC-B_GPIO-03 -> Pin
SW04_A : out std_logic; -- Q4 -> Y22 -> IDC-B_GPIO-02 -> Pin
SW05_A : out std_logic; -- Q5 -> T19 -> IDC-B_GPIO-01 -> Pin
SW06_A : out std_logic; -- Q6 -> AA22 -> IDC-B_GPIO-00 -> Pin
-- Phase B
SW01_B : out std_logic; -- Q1 -> Y16 -> IDC-C_GPIO-12 -> Pin
```

```
SW02_B : out std_logic; -- Q2 -> AB17-> IDC-C_GPIO-13 -> Pin
SW03_B : out std_logic; -- Q3 -> W14 -> IDC-C_GPIO-14 -> Pin
SW04_B : out std_logic; -- Q4 -> V14 -> IDC-C_GPIO-15 -> Pin
SW05_B : out std_logic; -- Q5 -> Y17 -> IDC-C_GPIO-16 -> Pin
SW06_B : out std_logic; -- Q6 -> AB18-> IDC-C_GPIO-17 -> Pin
-- Phase C
SW01_C : out std_logic; -- Q1 -> Y14  -> IDC-C_GPIO-00 -> Pin
SW02_C : out std_logic; -- Q2 -> AB15 -> IDC-C_GPIO-01 -> Pin
SW03_C : out std_logic; -- Q3 -> W12  -> IDC-C_GPIO-02 -> Pin
SW04_C : out std_logic; -- Q4 -> V12  -> IDC-C_GPIO-03 -> Pin
SW05_C : out std_logic; -- Q5 -> Y12  -> IDC-C_GPIO-04 -> Pin
SW06_C : out std_logic; -- Q6 -> V13  -> IDC-C_GPIO-05 -> Pin


-- Debug outputs
--debug_emu_Q1_Q6_A : out std_logic; --   E4 -> IDC-D_GPIO-04 -> Pin 5 -> (Channel 0)
--debug_emu_Q4_Q5_A : out std_logic; --   D3 -> IDC-D_GPIO-05 -> Pin 6 -> (Channel 1)
--debug_emu_Q1_Q6_B : out std_logic; --   G6 -> IDC-D_GPIO-06 -> Pin 7 -> (Channel 2)
--debug_emu_Q4_Q5_B : out std_logic; --   H7 -> IDC-D_GPIO-07 -> Pin 8 -> (Channel 3)
```

```
          --debug_emu_Q1_Q6_C : out std_logic; --   B1 -> IDC-D_GPIO-08 -> Pin 9
     -> (Channel 4)
          --debug_emu_Q4_Q5_C : out std_logic; --   C1 -> IDC-D_GPIO-09 -> Pin 10
     -> (Channel 5)
          --debug_error             : out std_logic; --   H6 -> IDC-D_GPIO-10 -> Pin 11
     -> (Channel 6)
          --debug_FW_Val_E1   : out std_logic; --      G5 -> IDC-D_GPIO-11 -> Pin 12
     -> (Channel 7)  -- Short-Circuit
          --debug_FW_Val_E2   : out std_logic; --      E2 -> IDC-D_GPIO-12 -> Pin 13
     -> (Channel 8)  -- DeadTime
          --debug_FW_Val_E3   : out std_logic; --      D1 -> IDC-D_GPIO-13 -> Pin 14
     -> (Channel 9)  -- Fund Freq
          --debug_FW_Val_E4   : out std_logic; --      F4 -> IDC-D_GPIO-14 -> Pin 21
     -> (Channel 10) -- Fast. Freq
          --debug_FW_Val_E5   : out std_logic; --      G4 -> IDC-D_GPIO-15 -> Pin 22
     -> (Channel 11) -- Timer
          --debug_FW_Val_EN  : out std_logic; --      F1 -> IDC-D_GPIO-16 -> Pin 23
     -> (Channel 12)


          Flash_CSSPIN:         inout  std_logic;          -- A21 -> CS   -> IDC-A_GPIO-00 -
> Pin 1 (Channel 0)
          Flash_SPISO: in  std_logic;              -- C19 -> SPO -> IDC-A_GPIO-01 -> Pin 2
(Channel 1)
```

```
		Flash_WPn:		inout  std_logic;		-- A20 -> WP  -> IDC-A_GPIO-02 -
> Pin 3 (Channel 2 - Always high)
		Flash_SISPI:	inout  std_logic;		-- D18 -> SPI  -> IDC-A_GPIO-03 -> Pin 4
(Channel 3)
		Flash_HOLDn:		inout  std_logic;		-- B19 -> Hold-> IDC-A_GPIO-04 -
> Pin 5 (Channel 4 - Always high)
		Flash_MCLK:		inout  std_logic;		-- C18 -> clk   -> IDC-A_GPIO-05 -
> Pin 6 (Channel 5)


		---------------- SMA Hard wired inputs ----------------
		-- Relays --
		IDC_D_GPIO_00 : out std_logic; -- Relay#1		-	C3
		IDC_D_GPIO_01 : out std_logic; -- Relay#2		-	C2
		---- IDC_B ----
		-- Constants
		IDC_B_GPIO_06 : out std_logic;	--	V21
		IDC_B_GPIO_07 : out std_logic;	--	V22
		IDC_B_GPIO_08 : out std_logic;	--	U22
		IDC_B_GPIO_09 : out std_logic;	--	U19
		IDC_B_GPIO_10 : out std_logic;	--	T21
		IDC_B_GPIO_11 : out std_logic;	--	R19
		IDC_B_GPIO_12 : out std_logic;	--	U20
		IDC_B_GPIO_13 : out std_logic;	--	T22
```

```
IDC_B_GPIO_14 : out std_logic;      --        R20
IDC_B_GPIO_15 : out std_logic;      --        R18
IDC_B_GPIO_16 : out std_logic;      --        R21
IDC_B_GPIO_17 : out std_logic;      --        P19


---- IDC_C ----
-- Constants
IDC_C_GPIO_06 : out std_logic;      --        AB15
IDC_C_GPIO_07 : out std_logic;      --        W12
IDC_C_GPIO_08 : out std_logic;      --        V12
IDC_C_GPIO_09 : out std_logic;      --        Y12
IDC_C_GPIO_10 : out std_logic;      --        V13
IDC_C_GPIO_11 : out std_logic;      --        U13


---- Security ----
DIMM_B_GPIO_60 : in std_logic;
DIMM_B_GPIO_61 : in std_logic;
DIMM_C_GPIO_60 : in std_logic;
DIMM_C_GPIO_61 : in std_logic;


---- Others ----
-- Jinan --
-- DSP INPUTS --
```

```
                DIMM_B_GPIO_48 : in std_logic; -- E20

                DIMM_B_GPIO_84 : in std_logic; -- D22

                DIMM_B_GPIO_86 : in std_logic; -- F19

                DIMM_C_GPIO_48 : in std_logic; -- AA7

                DIMM_C_GPIO_84 : in std_logic; -- V7

                DIMM_C_GPIO_86 : in std_logic; -- Y6

                -- Output --

                Jinan_01 : out std_logic; -- H6 - IDC-D_GPIO-10 - Pin 11

                Jinan_02 : out std_logic; -- G5 - IDC-D_GPIO-11 - Pin 12

                Jinan_03 : out std_logic  -- E2 - IDC-D_GPIO-12 - Pin 13

                -- End of Jinan --


        );

END Digital_Twin_and_Hot_Patching;


ARCHITECTURE Behavioral OF Digital_Twin_and_Hot_Patching 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 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');


-- Bootloader
SIGNAL  Bootload_EN : std_logic := '1';
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  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');


SIGNAL       DT_EN : std_logic := '0';

SIGNAL       DT_Rst : std_logic := '0';


------------------- DSPs -------------------

SIGNAL DSPEnable  : std_logic := '0';

--Phase A Inputs
```

```vhdl
SIGNAL Emu_SW01_A        : std_logic := '0';
SIGNAL Emu_SW02_A        : std_logic := '0';
SIGNAL Emu_SW03_A        : std_logic := '0';
SIGNAL Emu_SW04_A        : std_logic := '0';
SIGNAL Emu_SW05_A        : std_logic := '0';
SIGNAL Emu_SW06_A        : std_logic := '0';


--Phase B Inputs
SIGNAL Emu_SW01_B        : std_logic := '0';
SIGNAL Emu_SW02_B        : std_logic := '0';
SIGNAL Emu_SW03_B        : std_logic := '0';
SIGNAL Emu_SW04_B        : std_logic := '0';
SIGNAL Emu_SW05_B        : std_logic := '0';
SIGNAL Emu_SW06_B        : std_logic := '0';


--Phase C Inputs
SIGNAL Emu_SW01_C        : std_logic := '0';
SIGNAL Emu_SW02_C        : std_logic := '0';
SIGNAL Emu_SW03_C        : std_logic := '0';
SIGNAL Emu_SW04_C        : std_logic := '0';
SIGNAL Emu_SW05_C        : std_logic := '0';
SIGNAL Emu_SW06_C        : std_logic := '0';
```

```
        SIGNAL SCI_RX_DSP : std_logic;   --UART RX pin for Serial Comm with UI: UI(TX)
-> CPLD (RX)
        SIGNAL SCI_TX_DSP : std_logic;   --UART TX pin for Serial Comm with UI: UI(RX)
-> CPLD (TX)



        ------ Inverted signals ------
        -- Phase A
        SIGNAL Q1_A         : std_logic := '1';
        SIGNAL Q2_A         : std_logic := '1';
        SIGNAL Q3_A         : std_logic := '1';
        SIGNAL Q4_A         : std_logic := '1';

        -- Phase B
        SIGNAL Q1_B         : std_logic := '1';
        SIGNAL Q2_B         : std_logic := '1';
        SIGNAL Q3_B         : std_logic := '1';
        SIGNAL Q4_B         : std_logic := '1';

        -- Phase C
        SIGNAL Q1_C         : std_logic := '1';
        SIGNAL Q2_C         : std_logic := '1';
        SIGNAL Q3_C         : std_logic := '1';
```

```
    SIGNAL Q4_C         : std_logic := '1';

    ---------------------------------------- Module Declaration ----------------------------------------
    ------------------- 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;

    ------------------- PLL -------------------
  COMPONENT PLL_Clk
        PORT
        (
                ClkI: in  std_logic;

                ClkOP: out std_logic;

                Lock: out std_logic

          );

END COMPONENT;


    ------------------- Bus_Master -------------------

COMPONENT Digital_Twin_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);

                Flash_CSSPIN: out  std_logic;

                Flash_MCLK: out  std_logic;

                Flash_SISPI: out  std_logic;

                Flash_SPISO: in  std_logic;
                Flash_WPn: out  std_logic;

                Flash_HOLDn: out  std_logic;

                Reset_Flash_Button: in std_logic

          );

END COMPONENT;


    ------------------- 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;

    ------------------- Test1_DT_Boot_Ctrl -------------------
    component Digital_Twin_Bootloader_Control
    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;
            DT_EN           :out STD_LOGIC;
            DT_Rst          :out STD_LOGIC



         );
    END component;

    ------------------- DT_Bootloader_Test Component -------------------
    component Digital_Twin_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_Rst : 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;

------------------- 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 Digital_Twin_Hot_Patch_Control

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;


      ------------------- Digital_Twin_Emulation_Control -------------------

      COMPONENT Digital_Twin_Emulation_Control

              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;


--Phase A Inputs
Emu_SW01_A              : in std_logic;
Emu_SW02_A              : in std_logic;
Emu_SW03_A              : in std_logic;
Emu_SW04_A              : in std_logic;
Emu_SW05_A              : in std_logic;
Emu_SW06_A              : in std_logic;


--Phase B Inputs
Emu_SW01_B              : in std_logic;
Emu_SW02_B              : in std_logic;
Emu_SW03_B              : in std_logic;
Emu_SW04_B              : in std_logic;
```

```
        Emu_SW05_B      : in std_logic;

        Emu_SW06_B      : in std_logic;


        --Phase C Inputs

        Emu_SW01_C      : in std_logic;

        Emu_SW02_C      : in std_logic;

        Emu_SW03_C      : in std_logic;

        Emu_SW04_C      : in std_logic;

        Emu_SW05_C      : in std_logic;

        Emu_SW06_C      : in std_logic;


        Error       : in STD_LOGIC;

        HP_EN           : in STD_LOGIC

    );

END component;


    ------------------- Test1_DT_Firmware_Validation -------------------

COMPONENT Digital_Twin_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;

--Phase A Inputs
Emu_SW01_A          : in std_logic;
Emu_SW02_A          : in std_logic;
Emu_SW03_A          : in std_logic;
Emu_SW04_A          : in std_logic;
Emu_SW05_A          : in std_logic;
Emu_SW06_A          : in std_logic;

--Phase B Inputs
Emu_SW01_B          : in std_logic;
Emu_SW02_B          : in std_logic;
Emu_SW03_B          : in std_logic;
Emu_SW04_B          : in std_logic;
Emu_SW05_B          : in std_logic;
Emu_SW06_B          : in std_logic;
```

```
               --Phase C Inputs
               Emu_SW01_C       : in std_logic;
               Emu_SW02_C       : in std_logic;
               Emu_SW03_C       : in std_logic;
               Emu_SW04_C       : in std_logic;
               Emu_SW05_C       : in std_logic;
               Emu_SW06_C       : in std_logic;

               HP_Done          : in std_logic;  -- Signal coming from DSP_Hot-
Patch module saying that hot-patch is completed
               Boot_Done   : in std_logic; -- Signal to inform that the boot loading is
done
               Boot_Wrkn   : in std_logic; -- Signal to inform that the boot loading is
working
               Emu_EN           : out std_logic; -- Start the emulation
               HP_EN            : out std_logic; -- Signal sent to DSP_Hot-Patch
module to enable HP PROCESS
               DSPEnable   : out std_logic; -- Enable the DSP to generate the signals

               Error       : out std_logic; -- Signal error to stop all other PROCESSes
               DSP1_Act    : in std_logic
```

```
                    --debug_FW_Val_E1: out std_logic;
                    --debug_FW_Val_E2: out std_logic;
                    --debug_FW_Val_E3: out std_logic;
                    --debug_FW_Val_E4: out std_logic;
                    --debug_FW_Val_E5: out std_logic;
                    --debug_FW_Val_EN: out std_logic
                );
        END COMPONENT;

BEGIN --------------------------------------------------------- BEGIN ---------------------------------------
------------------


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



        ------------------- Instantiate PLL -------------------
        PLL_1: PLL_Clk PORT MAP (
                ClkI => OSC_Out,
                ClkOP => clk,
                Lock =>Pll_Lock
        );

        ------------------- Instantiate Bus_Master -------------------
        BM: Digital_Twin_Bus_Master PORT MAP (
                clk                 => clk,
                rst                 => System_rst,
                Data                => Data,
                Addr                => Addr,
                Xrqst               => Xrqst,
                XDat                => XDat,
                YDat                => YDat,
                BusRqst             => BusRqst,
                BusCtrl             => BusCtrl,
                Flash_CSSPIN        => Flash_CSSPIN,
                Flash_MCLK          => Flash_MCLK,
                Flash_SISPI         => Flash_SISPI,
                Flash_SPISO         => Flash_SPISO,
                Flash_WPn           => Flash_WPn,
                Flash_HOLDn         => Flash_HOLDn,
                Reset_Flash_Button => Btna
        );
```

```
      ------------------- 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,
    Data => Data,
    Addr => Addr,
    Xrqst => Xrqst,
    XDat => XDat,
    YDat => YDat,
    BusRqst => BusRqst(1), -- Was 3
    BusCtrl => BusCtrl(1) -- Was 3
);


       --DSP
```

```
     RS232_Usr_DSP: 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_DSP,
   rs232_xmt => SCI_TX_DSP,
   Data => Data,
   Addr => Addr,
   Xrqst => Xrqst,
   XDat => XDat,
   YDat => YDat,
   BusRqst => BusRqst(2), -- Was 3
   BusCtrl => BusCtrl(2) -- Was 3
);

      --Webserver
     RS232_Usr_Webserver: 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_Webserver,
  rs232_xmt => SCI_TX_Webserver,
  Data => Data,
  Addr => Addr,
  Xrqst => Xrqst,
  XDat => XDat,
  YDat => YDat,
  BusRqst => BusRqst(5), -- Was 3
  BusCtrl => BusCtrl(5) -- Was 3
);

------------------- Instantiate Boot_Ctrl -------------------
Boot_Ctrl: Digital_Twin_Bootloader_Control
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,
    DT_EN                 => DT_EN,
    DT_Rst                => DT_Rst
);

------------------- Instantiate Bootloader -------------------
Bootload: Digital_Twin_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_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 ----
                Data                        => Data,
                Addr                        => Addr,
                Xrqst                       => Xrqst,
        XDat                => XDat,
        YDat                => YDat,
        BusRqst             => BusRqst(4),
        BusCtrl             => BusCtrl(4)
);

------------------- 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: Digital_Twin_Hot_Patch_Control
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
);



------------------- Instantiate Emu_Ctrl -------------------
Emu_Ctrl: Digital_Twin_Emulation_Control
PORT MAP (
        clk      => clk,
        rst      => System_rst,

        Emu_EN       => Emu_EN,

        Data     => Data,
        Addr     => Addr,
        Xrqst    => Xrqst,
        XDat     => XDat,
        YDat     => YDat,
        BusRqst  => BusRqst(3),
        BusCtrl  => BusCtrl(3),
        -- Phase A
        Emu_SW01_A => Emu_SW01_A,
        Emu_SW02_A => Emu_SW02_A,
        Emu_SW03_A => Emu_SW03_A,
        Emu_SW04_A => Emu_SW04_A,
        Emu_SW05_A => Emu_SW05_A,
        Emu_SW06_A => Emu_SW06_A,
        -- Phase B
        Emu_SW01_B => Emu_SW01_B,
        Emu_SW02_B => Emu_SW02_B,
        Emu_SW03_B => Emu_SW03_B,
        Emu_SW04_B => Emu_SW04_B,
        Emu_SW05_B => Emu_SW05_B,
        Emu_SW06_B => Emu_SW06_B,
        -- Phase C
        Emu_SW01_C => Emu_SW01_C,
        Emu_SW02_C => Emu_SW02_C,
        Emu_SW03_C => Emu_SW03_C,
        Emu_SW04_C => Emu_SW04_C,
        Emu_SW05_C => Emu_SW05_C,
        Emu_SW06_C => Emu_SW06_C,
        Error     => Error,
                HP_EN           => HP_EN
        );


        ------------------- Instantiate Firmware Validation_EN -------------------
        FW_Valid: Digital_Twin_Firmware_Validation
        PORT MAP
        (
                clk             => clk,
                rst             => System_rst,
                Data            => Data,
                Addr            => Addr,
                Xrqst           => Xrqst,
                XDat            => XDat,
                YDat            => YDat,
                BusRqst         => BusRqst(6),
                BusCtrl         => BusCtrl(6),
                -- Phase A
                Emu_SW01_A => Emu_SW01_A,
                Emu_SW02_A => Emu_SW02_A,
                Emu_SW03_A => Emu_SW03_A,
                Emu_SW04_A => Emu_SW04_A,
                Emu_SW05_A => Emu_SW05_A,
                Emu_SW06_A => Emu_SW06_A,
                 -- Phase B
                 Emu_SW01_B => Emu_SW01_B,
                 Emu_SW02_B => Emu_SW02_B,
                 Emu_SW03_B => Emu_SW03_B,
                 Emu_SW04_B => Emu_SW04_B,
                 Emu_SW05_B => Emu_SW05_B,
                 Emu_SW06_B => Emu_SW06_B,
                 -- Phase C
                 Emu_SW01_C => Emu_SW01_C,
                 Emu_SW02_C => Emu_SW02_C,
                 Emu_SW03_C => Emu_SW03_C,
                 Emu_SW04_C => Emu_SW04_C,
                 Emu_SW05_C => Emu_SW05_C,
                 Emu_SW06_C => Emu_SW06_C,

                 HP_Done             => HP_Done,
                 Boot_Done    => Boot_Done,
        Boot_Wrkn      => Boot_Wrkn,
        Emu_EN         => Emu_EN,
        HP_EN          => HP_EN,
                 DSPEnable    => DSPEnable,
        Error          => Error,
                 DSP1_Act     => DSP1_Act
```

```
        --debug_FW_Val_E1 => Debug_FW_Val_E1,

        --debug_FW_Val_E2 => Debug_FW_Val_E2,

        --Debug_FW_Val_E3 => Debug_FW_Val_E3,

        --Debug_FW_Val_E4 => Debug_FW_Val_E4,

        --Debug_FW_Val_E5 => Debug_FW_Val_E5,

        --Debug_FW_Val_EN => Debug_FW_Val_EN

);


-------------------  Oscillator -------------------

OSC_Stdby <= '0';


------------------- Tie unused ports to '0'-------------------

BusRqst(9 downto 7) <= (others => '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 --7F = 127
                System_rst <= '0';
                Reset_Cnt_INC <='1';
            else
                System_rst <= '1';
                Reset_Cnt_INC <='0';
            END IF;
END PROCESS;

------------------- Setting DSP1 assignment and debug signals -------------------
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;
            --debug_emu_Q1_Q6_A <= Emu_SW01_A;
            --debug_emu_Q4_Q5_A <= Emu_SW04_A;
            --debug_emu_Q1_Q6_B <= Emu_SW01_B;
            --debug_emu_Q4_Q5_B <= Emu_SW04_B;
            --debug_emu_Q1_Q6_C <= Emu_SW01_C;
            --debug_emu_Q4_Q5_C <= Emu_SW04_C;
            --debug_error            <= Error;
    END PROCESS;


    ------------------- Main Routing PROCESS (Combinatorial) -------------------


    PROCESS (SCI_RX, DSP_Rst, DSP_rcv, DSP1_Act)
        BEGIN
            IF (DSP1_Act = '1') THEN
                DSP1_DSPEnable <= '1'; -- Enable the DSP1 to generate the
PWMs
                IF (DIMM_B_GPIO_60 = '1' AND DIMM_B_GPIO_61 = '0')
THEN -- Consider the PWMs only if the DSP outputs are enabled
                    ------------------- DSP 1 Active -------------------
                    -- Invert signals --
                    -- Phase A
                    Q1_A <= NOT(DSP1_01_A);
```

```
Q2_A <= NOT(DSP1_02_A);

Q3_A <= NOT(DSP1_03_A);

Q4_A <= NOT(DSP1_04_A);

-- Phase B

Q1_B <= NOT(DSP1_01_B);

Q2_B <= NOT(DSP1_02_B);

Q3_B <= NOT(DSP1_03_B);

Q4_B <= NOT(DSP1_04_B);

-- Phase C

Q1_C <= NOT(DSP1_01_C);

Q2_C <= NOT(DSP1_02_C);

Q3_C <= NOT(DSP1_03_C);

Q4_C <= NOT(DSP1_04_C);


-- Phase A

SW01_A <= Q1_A;

SW02_A <= Q2_A;

SW03_A <= Q3_A;

SW04_A <= Q4_A;

SW05_A <= Q4_A; -- Same as PWM 04

SW06_A <= Q1_A; -- Same as PWM 01

-- Phase B

SW01_B <= Q1_B;
```

```
        SW02_B <= Q2_B;
        SW03_B <= Q3_B;
        SW04_B <= Q4_B;
        SW05_B <= Q4_B; -- Same as PWM 04
        SW06_B <= Q1_B; -- Same as PWM 01
        -- Phase C
        SW01_C <= Q1_C;
        SW02_C <= Q2_C;
        SW03_C <= Q3_C;
        SW04_C <= Q4_C;
        SW05_C <= Q4_C; -- Same as PWM 04
        SW06_C <= Q1_C; -- Same as PWM 01
ELSE
        -- Phase A
        SW01_A <= '1';
        SW02_A <= '1';
        SW03_A <= '1';
        SW04_A <= '1';
        SW05_A <= '1';
        SW06_A <= '1';
        -- Phase B
        SW01_B <= '1';
        SW02_B <= '1';
```

```
    SW03_B <= '1';
    SW04_B <= '1';
    SW05_B <= '1';
    SW06_B <= '1';
    -- Phase C
    SW01_C <= '1';
    SW02_C <= '1';
    SW03_C <= '1';
    SW04_C <= '1';
    SW05_C <= '1';
    SW06_C <= '1';
END IF;

------------------- DSP 2 Emulation -------------------
--DSP 2 Enable DSP
DSP2_DSPEnable <= DSPEnable;
-- Phase A
Emu_SW01_A <= DSP2_01_A;
Emu_SW02_A <= DSP2_02_A;
Emu_SW03_A <= DSP2_03_A;
Emu_SW04_A <= DSP2_04_A;
Emu_SW05_A <= DSP2_04_A; -- Same as PWM 04
Emu_SW06_A <= DSP2_01_A; -- Same as PWM 01
```

```
-- Phase B
Emu_SW01_B <= DSP2_01_B;
Emu_SW02_B <= DSP2_02_B;
Emu_SW03_B <= DSP2_03_B;
Emu_SW04_B <= DSP2_04_B;
Emu_SW05_B <= DSP2_04_B; -- Same as PWM 04
Emu_SW06_B <= DSP2_01_B; -- Same as PWM 01
-- Phase C
Emu_SW01_C <= DSP2_01_C;
Emu_SW02_C <= DSP2_02_C;
Emu_SW03_C <= DSP2_03_C;
Emu_SW04_C <= DSP2_04_C;
Emu_SW05_C <= DSP2_04_C; -- Same as PWM 04
Emu_SW06_C <= DSP2_01_C; -- Same as PWM 01

-----------------------------------------------------------
DIMM_C_GPIO30 <= DSP_Sync_HP;
DIMM_B_GPIO30 <= DSP_Sync_HP;

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

LED_C <= '1';
```

```
                        LED_D <= '1';
                        LED_E <= '1';
                        LED_F <= '1';
                        LED_G <= '1';
                        LED_H <= '1';

                        IDC_D_GPIO_02 <= '1';                   ---- Reset is active
low, and 1(NO Reset) is routed to pin 00 of IDC B (DSP1 is Active)
                        IDC_D_GPIO_03 <= DSP_Rst;               ---- DSP_Rst
signal(Bootloader) routed to pin 00 of IDC C (DSP2 is Stand-By)

                        -- DSP 1 Active (DIMM_B), communicate through MODBUS,
while DSP 2 is able to bootload
                        -- DSP 1 (Modbus)
                        DIMM_B_SCI_RX <= SCI_TX_DSP;            ---- Stop bit is high,
and is sent to the serial receiver of DIMM B (DSP1 is Active)
                        SCI_RX_DSP <= DIMM_B_SCI_TX;

                        -- DSP 2 (Bootloading)
                        DIMM_C_SCI_RX <= DSP_rcv;       ---- DSP_rsv
signal(Bootloader) is routed to the serial receiver of DIMM C (DSP2 is Stand-By)
                        DSP_xmt <= DIMM_C_SCI_TX;
```

```
                IDC_D_GPIO_00 <= DIMM_B_GPIO_32;   -- Relay #1
                IDC_D_GPIO_01 <= DIMM_B_GPIO_33;   -- Relay #2

                Jinan_01 <= DIMM_B_GPIO_48;
                Jinan_02 <= DIMM_B_GPIO_84;
                Jinan_03 <= DIMM_B_GPIO_86;

            ELSE
                ------------------- DSP 2 Active -------------------
                DSP2_DSPEnable <= '1'; -- Enable the DSP2 to generate the
PWMs
                IF (DIMM_C_GPIO_60 = '1' AND DIMM_C_GPIO_61 = '0')
THEN -- Consider the PWMs only if the DSP outputs are enabled
                    -- Invert signals --
                    -- Phase A
                    Q1_A <= NOT(DSP2_01_A);
                    Q2_A <= NOT(DSP2_02_A);
                    Q3_A <= NOT(DSP2_03_A);
                    Q4_A <= NOT(DSP2_04_A);
                    -- Phase B
                    Q1_B <= NOT(DSP2_01_B);
                    Q2_B <= NOT(DSP2_02_B);
```

```
Q3_B <= NOT(DSP2_03_B);

Q4_B <= NOT(DSP2_04_B);

-- Phase C

Q1_C <= NOT(DSP2_01_C);

Q2_C <= NOT(DSP2_02_C);

Q3_C <= NOT(DSP2_03_C);

Q4_C <= NOT(DSP2_04_C);


-- Phase A

SW01_A <= Q1_A;

SW02_A <= Q2_A;

SW03_A <= Q3_A;

SW04_A <= Q4_A;

SW05_A <= Q4_A; -- Same as PWM 04

SW06_A <= Q1_A; -- Same as PWM 01

-- Phase B

SW01_B <= Q1_B;

SW02_B <= Q2_B;

SW03_B <= Q3_B;

SW04_B <= Q4_B;

SW05_B <= Q4_B; -- Same as PWM 04

SW06_B <= Q1_B; -- Same as PWM 01

-- Phase C
```

```
        SW01_C <= Q1_C;
        SW02_C <= Q2_C;
        SW03_C <= Q3_C;
        SW04_C <= Q4_C;
        SW05_C <= Q4_C; -- Same as PWM 04
        SW06_C <= Q1_C; -- Same as PWM 01
ELSE
        -- Phase A
        SW01_A <= '1';
        SW02_A <= '1';
        SW03_A <= '1';
        SW04_A <= '1';
        SW05_A <= '1';
        SW06_A <= '1';
        -- Phase B
        SW01_B <= '1';
        SW02_B <= '1';
        SW03_B <= '1';
        SW04_B <= '1';
        SW05_B <= '1';
        SW06_B <= '1';
        -- Phase C
        SW01_C <= '1';
        SW02_C <= '1';
        SW03_C <= '1';
        SW04_C <= '1';
        SW05_C <= '1';
        SW06_C <= '1';
END IF;
------------------- DSP 1 Emulation -------------------
--DSP 1 Enable DSP
DSP1_DSPEnable <= DSPEnable;
-- Phase A
Emu_SW01_A <= DSP1_01_A;
Emu_SW02_A <= DSP1_02_A;
Emu_SW03_A <= DSP1_03_A;
Emu_SW04_A <= DSP1_04_A;
Emu_SW05_A <= DSP1_04_A;
Emu_SW06_A <= DSP1_01_A;
-- Phase B
Emu_SW01_B <= DSP1_01_B;
Emu_SW02_B <= DSP1_02_B;
Emu_SW03_B <= DSP1_03_B;
Emu_SW04_B <= DSP1_04_B;
Emu_SW05_B <= DSP1_04_B;
Emu_SW06_B <= DSP1_01_B;
```

```
-- Phase C
Emu_SW01_C <= DSP1_01_C;
Emu_SW02_C <= DSP1_02_C;
Emu_SW03_C <= DSP1_03_C;
Emu_SW04_C <= DSP1_04_C;
Emu_SW05_C <= DSP1_04_C;
Emu_SW06_C <= DSP1_01_C;

---------------------------------------------------------
DIMM_C_GPIO30 <= DSP_Sync_HP;
DIMM_B_GPIO30 <= DSP_Sync_HP;

LED_A <= '1';
LED_B <= '0';
LED_C <= '1';
LED_D <= '1';
LED_E <= '1';
LED_F <= '1';
LED_G <= '1';
LED_H <= '1';

IDC_D_GPIO_02 <= DSP_Rst;                ---- DSP_Rst
signal(Bootloader) routed to pin 00 of IDC B (DSP1 is Stand-By)
```

```
                IDC_D_GPIO_03 <= '1';        ---- Reset is active low, and 1(NO
Reset) is routed to pin 00 of IDC C (DSP2 is Active)

                -- DSP 2 Active (DIMM_C), communicate through MODBUS,
while DSP 1 is able to bootload
                -- DSP 1 (Bootloading)
                DIMM_B_SCI_RX <= DSP_rcv;                ---- DSP_rsv
signal(Bootloader) is routed to the serial receiver of DIMM B (DSP1 is Stand-By)
                DSP_xmt <= DIMM_B_SCI_TX;

                -- DSP 2 (Modbus)
                DIMM_C_SCI_RX <= SCI_TX_DSP;         ---- Stop bit is high,
and is sent to the serial receiver of DIMM C (DSP2 is Active)
                SCI_RX_DSP <= DIMM_C_SCI_TX;

                IDC_D_GPIO_00 <= DIMM_C_GPIO_32;   -- Relay #1
                IDC_D_GPIO_01 <= DIMM_C_GPIO_33;   -- Relay #2

                Jinan_01 <= DIMM_C_GPIO_48;
                Jinan_02 <= DIMM_C_GPIO_84;
                Jinan_03 <= DIMM_C_GPIO_86;
            END IF;
            IDC_B_GPIO_06 <= '1';
            IDC_B_GPIO_07 <= '1';
            IDC_B_GPIO_08 <= '1';
            IDC_B_GPIO_09 <= '1';
            IDC_B_GPIO_10 <= '1';
            IDC_B_GPIO_11 <= '1';

            IDC_B_GPIO_12 <= '1';
            IDC_B_GPIO_13 <= '1';
            IDC_B_GPIO_14 <= '1';
            IDC_B_GPIO_15 <= '1';
            IDC_B_GPIO_16 <= '1';
            IDC_B_GPIO_17 <= '1';

            IDC_C_GPIO_06 <= '1';
            IDC_C_GPIO_07 <= '1';
            IDC_C_GPIO_08 <= '1';
            IDC_C_GPIO_09 <= '1';
            IDC_C_GPIO_10 <= '1';
            IDC_C_GPIO_11 <= '1';
    END PROCESS;
END Behavioral;
```

A-2: Firmware Validation

```
--------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Estefano Soria and Paulo Custodio
--
-- Create Date:            11/18/2021
-- Project Name:           Digital_Twin
-- Module Name:            Firmware_Validation
-- Project Name:           Digital_Twin_Firmware_Validation
-- Target Devices:         LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:          Lattice Diamond_x64 Build 3.11
-- Description:
-- This module uses different components to test the firmware and integrates them to generate an
error in case
-- one or more components detects an issue.
--
---- PinOut:
--
-- Revision: V1.1
-- v3.26.22 - Components added: Deadtime, Timer, Fundamental Frequency Detector,
-- v5.26.22 - Polish and comments removed/added
--
```

```
-- Additional Comments:
--
----------------------------------------------------------------------------------


Library IEEE;
Library STD;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use STD.textio.all;
use IEEE.std_logic_textio.all;

library machxo2;
use machxo2.all;

library work;
use work.Digital_Twin_Common.all;


entity Digital_Twin_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;

--Phase A Inputs
Emu_SW01_A        : in std_logic;
Emu_SW02_A        : in std_logic;
Emu_SW03_A        : in std_logic;
Emu_SW04_A        : in std_logic;
Emu_SW05_A        : in std_logic;
Emu_SW06_A        : in std_logic;

--Phase B Inputs
Emu_SW01_B        : in std_logic;
Emu_SW02_B        : in std_logic;
Emu_SW03_B        : in std_logic;
```

```
        Emu_SW04_B          : in std_logic;
        Emu_SW05_B          : in std_logic;
        Emu_SW06_B          : in std_logic;

        --Phase C Inputs
        Emu_SW01_C          : in std_logic;
        Emu_SW02_C          : in std_logic;
        Emu_SW03_C          : in std_logic;
        Emu_SW04_C          : in std_logic;
        Emu_SW05_C          : in std_logic;
        Emu_SW06_C          : in std_logic;

        HP_Done             : in std_logic;  -- Signal coming from DSP_Hot-Patch
module saying that hot-patch is completed
        Boot_Done     : in std_logic; -- Signal to inform that the boot loading is done
        Boot_Wrkn     : in std_logic; -- Signal to inform that the boot loading is working
        Emu_EN              : out std_logic; -- Start the emulation
        HP_EN               : out std_logic; -- Signal sent to DSP_Hot-Patch module to
enable HP PROCESS
        DSPEnable     : out std_logic; -- Enable the DSP to generate the signals to
emulate

        --Debug_FW_Val: out std_logic;
```

```
		Error		: out std_logic; -- Signal error to stop all other processes
		DSP1_Act	: in std_logic


		--debug_FW_Val_E1   : out std_logic;
		--debug_FW_Val_E2   : out std_logic;
		--debug_FW_Val_E3   : out std_logic;
		--debug_FW_Val_E4   : out std_logic;
		--debug_FW_Val_E5   : out std_logic;
		--debug_FW_Val_EN   : out std_logic
	);
END Digital_Twin_Firmware_Validation;


ARCHITECTURE Behavioral of Digital_Twin_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,S45,S46,S47,S48,S49,
		S50,S51,S52,S53,S54,S55,S100,S101,S102,S103,S104,S_error
	);
```

```
signal CS, NS, CS_Chk, NS_Chk, CS_ShCrk, NS_ShCrk, CS_DeadT, NS_DeadT : state_type;

------------------------------- Bad Firmware -------------------------------
signal  Bad_Firmware : std_logic := '0'; -- IF all Bad_FW are OFF THEN Bad_Firmware is OFF. IF it is ever ON, THEN backup FW is EN.



------------------------------- 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 -------------------------------
-- Hot Patch
signal  LD_HP_EN            : std_logic := '0'; -- Enable
signal  Temp_HP_EN                  : std_logic := '0'; -- Enable

signal  LD_HP_Done          : std_logic := '0'; -- Done
signal  HP_Done_reg_o       : std_logic := '0'; -- Done
```

```
-- Bad Firmware: Short Circuit

signal  Bad_FW1                         : std_logic := '0';


-- Bad Firmware: Dead Time

signal  Bad_FW2                         : std_logic := '0';


-- Bad Firmware: Fundamental Frequency

signal  Bad_FW3                         : std_logic := '0';


-- Bad Firmware: Fast Frequency

signal  Bad_FW4                         : std_logic := '0';


-- Bad Firmware: Timer Error

signal  Bad_FW5                         : std_logic := '0';


-- Check

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';

	-- Boot
	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';

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

	-- Hot Patch Command
	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');

	-- Error
	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');


-- Variable Data (used to collect data from the Bus)
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');


-- Validation Start
signal  LD_Val_Start         : std_logic := '0';
signal  Temp_Val_Start              : std_logic := '0';
signal  Val_Start                   : std_logic := '0';


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


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

```
        -- PreChk is used to count the fundamental period, to make sure that the erro checkings keep running for this period
        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');
        -------------------------------- End of counters --------------------------------

        -- Fundamental Frequency Error flags
        signal FF_error_SW01_A : std_logic := '0';
        signal FF_error_SW04_A : std_logic := '0';
        signal FF_error_SW01_B : std_logic := '0';
        signal FF_error_SW04_B : std_logic := '0';
        signal FF_error_SW01_C : std_logic := '0';
        signal FF_error_SW04_C : std_logic := '0';
        -- Fundamental Frequency Debug
        --signal debug_FF_detector : std_logic;

        -- Fast Frequency Error flags
        signal FastFrequency_error_SW02_A : std_logic := '0';
        signal FastFrequency_error_SW03_A : std_logic := '0';
        signal FastFrequency_error_SW02_B : std_logic := '0';
        signal FastFrequency_error_SW03_B : std_logic := '0';
```

```
	signal FastFrequency_error_SW02_C : std_logic := '0';
	signal FastFrequency_error_SW03_C : std_logic := '0';


	-- Timer Done Flag
	signal DisableTimer : std_logic := '0';


	-- DeadTime Error flags
	-- Phase A
	signal Dead_Time_SW_16_45_A	: std_logic := '0';			-- Deadtime between Q1/Q6 and Q4/Q5
	signal Dead_Time_SW_45_16_A	: std_logic := '0';			-- Deadtime between Q4/Q5 and Q1/Q6
	-- Phase B
	signal Dead_Time_SW_16_45_B	: std_logic := '0';			-- Deadtime between Q1/Q6 and Q4/Q5
	signal Dead_Time_SW_45_16_B	: std_logic := '0';			-- Deadtime between Q4/Q5 and Q1/Q6
	-- Phase C
	signal Dead_Time_SW_16_45_C	: std_logic := '0';			-- Deadtime between Q1/Q6 and Q4/Q5
	signal Dead_Time_SW_45_16_C	: std_logic := '0';			-- Deadtime between Q4/Q5 and Q1/Q6
```

```
-- Watchdog signals

signal LD_DisableWatchdog             : std_logic := '0';

signal DisableWatchdogReg             : std_logic := '0';

signal Temp_DisableWatchdog                     : std_logic := '0';


------------------------------- Components -------------------------------

-- Declare Counter

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;

    -- Declare Deadtime
    COMPONENT Digital_Twin_DeadTime
    PORT
     (
		clk				: in std_logic;
		rst				: in std_logic;
		DeadTime_Enable			: in std_logic;
		DeadTimeError			: out std_logic;

		Emu_SW01	: in std_logic;
		Emu_SW06	: in std_logic;
		Emu_SW04	: in std_logic;
		Emu_SW05	: in std_logic
	);
	END COMPONENT;

	-- Declare Fundamental Frequency Detector
	COMPONENT FF_detector is
		generic (
			maxValue : std_logic_vector(19 downto 0) := X"67C28";	-- 668A0h = 59.5Hz = 420,000 clock cycles + 5,000 margin
			minValue : std_logic_vector(19 downto 0) := X"64D48"	-- 64D48h = 60.5Hz = 413,000 clock cycles
		);
		port
		(
			--debug_FF_detector : out std_logic;
```

```
                SW                                      : in std_logic;
                enable_ff_check                 : in std_logic;
                stop                            : in std_logic;
                clk                                     : in std_logic;
                rst                                     : in std_logic;
                FF_det_error            : out std_logic
        );
END COMPONENT;

COMPONENT FastFrequency_detector is
        port
        (
                --debug_FF_detector : out std_logic;
                SW                                      : in std_logic;
                enable_ff_check                 : in std_logic;
                stop                            : in std_logic;
                clk                                     : in std_logic;
                rst                                     : in std_logic;
                FF_det_error            : out std_logic
        );
END COMPONENT;

COMPONENT timer_detector is
```

```
        PORT
        (
                enable  : in std_logic;
                done    : in std_logic;
                clk             : in std_logic;
                rst             : in std_logic;
                timer_error     : out std_logic
        );
END COMPONENT;

COMPONENT Digital_Twin_ShortCircuit is
        PORT
        (
                -- Inputs
                clk                     : in std_logic;
                rst                     : in std_logic;
                ShCrkEnable     : in std_logic;
                Cnt_PreChk_Out          : in std_logic_vector(31 downto 0);

                Emu_SW01_A                      : in std_logic;
                Emu_SW04_A                      : in std_logic;
                Emu_SW05_A                      : in std_logic;
                Emu_SW06_A                      : in std_logic;
```

```
            Emu_SW01_B          : in std_logic;

            Emu_SW04_B          : in std_logic;

            Emu_SW05_B          : in std_logic;

            Emu_SW06_B          : in std_logic;


            Emu_SW01_C          : in std_logic;

            Emu_SW04_C          : in std_logic;

            Emu_SW05_C          : in std_logic;

            Emu_SW06_C          : in std_logic;


            -- Outputs

            DisableTimer  : out std_logic;

            Bad_FW1             : out 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 PreChk counter
   Cnt_PreChk: Std_Counter
   generic map
   (
            Width => 32
   )
   port map(
            clk => clk,
            rst=> Cnt_PreChk_rst,
            INC=> Cnt_PreChk_INC,
            Count=> Cnt_PreChk_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
);

      TimerDetector: timer_detector
      PORT MAP(
              enable  => EN_Chk_reg_o,
              done    => DisableTimer,
              clk             => clk,
              rst             => rst,
              timer_error     => Bad_FW5
      );

      -- Instantiate DeadTime betweem Q1/Q6 and Q4/Q5 for Phase A
```

```
DeadTime_16_45_A: Digital_Twin_DeadTime
PORT MAP
(
        clk                 => clk,
        rst                 => rst,
        DeadTime_Enable     => EN_Chk_reg_o,
        DeadTimeError       => Dead_Time_SW_16_45_A,

        Emu_SW01 => Emu_SW01_A,
        Emu_SW06 => Emu_SW06_A,
        Emu_SW04 => Emu_SW04_A,
        Emu_SW05 => Emu_SW05_A
);

-- Instantiate DeadTime betweem Q4/Q5 and Q1/Q6 for Phase A
DeadTime_45_16_A: Digital_Twin_DeadTime
PORT MAP
(
        clk                 => clk,
        rst                 => rst,
        DeadTime_Enable     => EN_Chk_reg_o,
        DeadTimeError       => Dead_Time_SW_45_16_A,
        Emu_SW01 => Emu_SW04_A,

        Emu_SW06 => Emu_SW05_A,

        Emu_SW04 => Emu_SW01_A,

        Emu_SW05 => Emu_SW06_A

);



-- Instantiate DeadTime betweem Q1/Q6 and Q4/Q5 for Phase B

DeadTime_16_45_B: Digital_Twin_DeadTime

PORT MAP

(

        clk                 => clk,

        rst                 => rst,

        DeadTime_Enable     => EN_Chk_reg_o,

        DeadTimeError       => Dead_Time_SW_16_45_B,


        Emu_SW01 => Emu_SW01_B,

        Emu_SW06 => Emu_SW06_B,

        Emu_SW04 => Emu_SW04_B,

        Emu_SW05 => Emu_SW05_B

);
```

```
-- Instantiate DeadTime betweem Q4/Q5 and Q1/Q6 for Phase B

DeadTime_45_16_B: Digital_Twin_DeadTime

PORT MAP

(

        clk                 => clk,

        rst                 => rst,

        DeadTime_Enable     => EN_Chk_reg_o,

        DeadTimeError       => Dead_Time_SW_45_16_B,


        Emu_SW01 => Emu_SW04_B,

        Emu_SW06 => Emu_SW05_B,

        Emu_SW04 => Emu_SW01_B,

        Emu_SW05 => Emu_SW06_B

);



-- Instantiate DeadTime betweem Q1/Q6 and Q4/Q5 for Phase C

DeadTime_16_45_C: Digital_Twin_DeadTime

PORT MAP

(

        clk                 => clk,

        rst                 => rst,

        DeadTime_Enable     => EN_Chk_reg_o,
        DeadTimeError      => Dead_Time_SW_16_45_C,

        Emu_SW01 => Emu_SW01_C,
        Emu_SW06 => Emu_SW06_C,
        Emu_SW04 => Emu_SW04_C,
        Emu_SW05 => Emu_SW05_C
);

-- Instantiate DeadTime betweem Q4/Q5 and Q1/Q6 for Phase C
DeadTime_45_16_C: Digital_Twin_DeadTime
PORT MAP
(
        clk                => clk,
        rst                => rst,
        DeadTime_Enable    => EN_Chk_reg_o,
        DeadTimeError      => Dead_Time_SW_45_16_C,

        Emu_SW01 => Emu_SW04_C,
        Emu_SW06 => Emu_SW05_C,
        Emu_SW04 => Emu_SW01_C,
        Emu_SW05 => Emu_SW06_C
);
-- Instantiate Fundamental Frequency Detector for Q1/Q6 (Phase A)
```

```
Fundamental_Frequency_Detector_SW01_SW06_A: FF_detector
PORT MAP
(
        --debug_FF_detector => open,
        SW                              => Emu_SW01_A,
        enable_ff_check         => EN_Chk_reg_o,
        stop                    => Stop_Chk,
        clk                             => clk,
        rst                             => rst,
        FF_det_error    => FF_error_SW01_A
);

-- Instantiate Fundamental Frequency Detector for Q4/Q5 (Phase A)
Fundamental_Frequency_Detector_SW04_SW05_A: FF_detector
PORT MAP
(
        --debug_FF_detector => open,
        SW                              => Emu_SW04_A,
        enable_ff_check         => EN_Chk_reg_o,
        stop                    => Stop_Chk,
        clk                             => clk,
        rst                             => rst,
        FF_det_error    => FF_error_SW04_A
```

```
);

-- Instantiate Fundamental Frequency Detector for Q1/Q6 (Phase B)
Fundamental_Frequency_Detector_SW01_SW06_B: FF_detector
PORT MAP
(
        --debug_FF_detector => open,
        SW                              => Emu_SW01_B,
        enable_ff_check         => EN_Chk_reg_o,
        stop                    => Stop_Chk,
        clk                             => clk,
        rst                             => rst,
        FF_det_error    => FF_error_SW01_B
);

-- Instantiate Fundamental Frequency Detector for Q4/Q5 (Phase B)
Fundamental_Frequency_Detector_SW04_SW05_B: FF_detector
PORT MAP
(
        --debug_FF_detector => open,
        SW                              => Emu_SW04_B,
        enable_ff_check         => EN_Chk_reg_o,
        stop                    => Stop_Chk,
        clk                     => clk,
        rst                     => rst,
        FF_det_error    => FF_error_SW04_B
);

-- Instantiate Fundamental Frequency Detector for Q1/Q6 (Phase C)
Fundamental_Frequency_Detector_SW01_SW06_C: FF_detector
PORT MAP
(
        --debug_FF_detector => open,
        SW                      => Emu_SW01_C,
        enable_ff_check     => EN_Chk_reg_o,
        stop                => Stop_Chk,
        clk                     => clk,
        rst                     => rst,
        FF_det_error    => FF_error_SW01_C
);

-- Instantiate Fundamental Frequency Detector for Q4/Q5 (Phase C)
Fundamental_Frequency_Detector_SW04_SW05_C: FF_detector
PORT MAP
(
        --debug_FF_detector => open,
        SW                              => Emu_SW04_C,
        enable_ff_check         => EN_Chk_reg_o,
        stop                    => Stop_Chk,
        clk                             => clk,
        rst                             => rst,
        FF_det_error    => FF_error_SW04_C
);



-- Instantiate Fast Frequency Detector for Q2 (Phase A)
Fast_Frequency_Detector_SW02_A: FastFrequency_detector
PORT MAP
(
        --debug_FF_detector => debug_FF_detector,
        SW                              => Emu_SW02_A,
        enable_ff_check         => EN_Chk_reg_o,
        stop                    => Stop_Chk,
        clk                             => clk,
        rst                             => rst,
        FF_det_error    => FastFrequency_error_SW02_A
);


-- Instantiate Fast Frequency Detector for Q3 (Phase A)
```

```
Fast_Frequency_Detector_SW03_A: FastFrequency_detector
PORT MAP
(
        --debug_FF_detector => open,
        SW                              => Emu_SW03_A,
        enable_ff_check         => EN_Chk_reg_o,
        stop                            => Stop_Chk,
        clk                             => clk,
        rst                             => rst,
        FF_det_error    => FastFrequency_error_SW03_A
);

-- Instantiate Fast Frequency Detector for Q2 (Phase B)
Fast_Frequency_Detector_SW02_B: FastFrequency_detector
PORT MAP
(
        --debug_FF_detector => open,
        SW                              => Emu_SW02_B,
        enable_ff_check         => EN_Chk_reg_o,
        stop                            => Stop_Chk,
        clk                             => clk,
        rst                             => rst,
        FF_det_error    => FastFrequency_error_SW02_B
```

```
);


-- Instantiate Fast Frequency Detector for Q3 (Phase B)

Fast_Frequency_Detector_SW03_B: FastFrequency_detector

PORT MAP

(

        --debug_FF_detector => open,

        SW                         => Emu_SW03_B,

        enable_ff_check     => EN_Chk_reg_o,

        stop                => Stop_Chk,

        clk                        => clk,

        rst                        => rst,

        FF_det_error   => FastFrequency_error_SW03_B

);


-- Instantiate Fast Frequency Detector for Q2 (Phase C)

Fast_Frequency_Detector_SW02_C: FastFrequency_detector

PORT MAP

(

        --debug_FF_detector => open,

        SW                         => Emu_SW02_C,

        enable_ff_check     => EN_Chk_reg_o,

        stop                => Stop_Chk,
        clk                             => clk,
        rst                             => rst,
        FF_det_error    => FastFrequency_error_SW02_C
);

-- Instantiate Fast Frequency Detector for Q3 (Phase C)
Fast_Frequency_Detector_SW03_C: FastFrequency_detector
PORT MAP
(
        --debug_FF_detector => open,
        SW                              => Emu_SW03_C,
        enable_ff_check         => EN_Chk_reg_o,
        stop                    => Stop_Chk,
        clk                             => clk,
        rst                             => rst,
        FF_det_error    => FastFrequency_error_SW03_C
);

ShortCircuit : Digital_Twin_ShortCircuit
PORT MAP
(
        clk => clk,
        rst => rst,
        ShCrkEnable => EN_Chk_reg_o,
        Cnt_PreChk_Out => Cnt_PreChk_Out,

        Emu_SW01_A          => Emu_SW01_A,
        Emu_SW04_A          => Emu_SW04_A,
        Emu_SW05_A          => Emu_SW05_A,
        Emu_SW06_A          => Emu_SW06_A,

        Emu_SW01_B => Emu_SW01_B,
        Emu_SW04_B => Emu_SW04_B,
        Emu_SW05_B => Emu_SW05_B,
        Emu_SW06_B => Emu_SW06_B,

        Emu_SW01_C => Emu_SW01_C,
        Emu_SW04_C => Emu_SW04_C,
        Emu_SW05_C => Emu_SW05_C,
        Emu_SW06_C => Emu_SW06_C,

            -- Outputs
        DisableTimer => DisableTimer,
        Bad_FW1 => Bad_FW1
    );
```

```
------------------------------- 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_Chk_reg_o <= '0';

                Stop_Chk <= '0';

                Boot_Done_reg_o <= '0';

                Boot_Wrkn_reg_o <= '0';

                Emu_EN <= '0';

                Error <= '0';

                Val_Start <= '0';

                DisableWatchdogReg <= '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_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_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;
```

```
        IF (LD_DisableWatchdog  = '1') THEN         DisableWatchdogReg
<= Temp_DisableWatchdog;   END IF;
    END IF;
END PROCESS;


-------------------------------- Deadtime Check --------------------------------
Deadtime_Check :  PROCESS (
    EN_Chk_reg_o,
    Dead_Time_SW_16_45_A,
    Dead_Time_SW_45_16_A,
    Dead_Time_SW_16_45_B,
    Dead_Time_SW_45_16_B,
    Dead_Time_SW_16_45_C,
    Dead_Time_SW_45_16_C
)
BEGIN
    IF (EN_Chk_reg_o = '1') THEN
        IF ((Dead_Time_SW_16_45_A OR
            Dead_Time_SW_45_16_A OR
            Dead_Time_SW_16_45_B OR
            Dead_Time_SW_45_16_B OR
            Dead_Time_SW_16_45_C OR
            Dead_Time_SW_45_16_C
            ) = '1') THEN
            Bad_FW2 <= '1';
        ELSE
            Bad_FW2 <= '0';
        END IF;
    ELSE
        Bad_FW2 <= '0';
    END IF;
END PROCESS;
-------------------------------- Fast Frequency Check --------------------------------
FastFrequency_Check :  PROCESS (
    EN_Chk_reg_o,
    FastFrequency_error_SW02_A,
    FastFrequency_error_SW03_A,
    FastFrequency_error_SW02_B,
    FastFrequency_error_SW03_B,
    FastFrequency_error_SW02_C,
    FastFrequency_error_SW03_C
)
BEGIN
    IF (EN_Chk_reg_o = '1') THEN
        IF ((FastFrequency_error_SW02_A OR
            FastFrequency_error_SW03_A OR
            FastFrequency_error_SW02_B OR
            FastFrequency_error_SW03_B OR
            FastFrequency_error_SW02_C OR
            FastFrequency_error_SW03_C
            ) = '1') THEN
            Bad_FW4 <= '1';
        ELSE
            Bad_FW4 <= '0';
        END IF;
    ELSE
        Bad_FW4 <= '0';
    END IF;
END PROCESS;
-------------------------------- Fundamental Frequency Check --------------------------------
FF_Check :  PROCESS (
    EN_Chk_reg_o,
    FF_error_SW01_A,
    FF_error_SW04_A,
    FF_error_SW01_B,
    FF_error_SW04_B,
    FF_error_SW01_C,
    FF_error_SW04_C
)
```

```
BEGIN
        IF (EN_Chk_reg_o = '1') THEN
                IF ((FF_error_SW01_A OR
                        FF_error_SW04_A OR
                        FF_error_SW01_B OR
                        FF_error_SW04_B OR
                        FF_error_SW01_C OR
                        FF_error_SW04_C) = '1') THEN
                        Bad_FW3 <= '1';
                ELSE
                        Bad_FW3 <= '0';
                END IF;
        ELSE
                Bad_FW3 <= '0';
        END IF;
END PROCESS;
------------------------------- Bad Firmware Check -------------------------------
Bad_FW_Check :  PROCESS (
        EN_Chk_reg_o,
        Bad_FW1,
        Bad_FW2,
        Bad_FW3,
        Bad_FW4,
        Bad_FW5
    )
    BEGIN
        IF (EN_Chk_reg_o = '1') THEN
            IF ((Bad_FW1 OR Bad_FW2 OR Bad_FW3 OR Bad_FW4 OR
Bad_FW5) = '1') THEN
                Bad_Firmware <= '1';
            ELSE
                Bad_Firmware <= '0';
            END IF;
        ELSE
            Bad_Firmware <= '0';
        END IF;
    END PROCESS;



    Main: PROCESS (
        CS,
        Bus_Int1_Busy,
        Bus_Int1_DataOut,
        CntDelay_Out,
        CntBus_Out,
        Cnt_PreChk_Out,
        Vrble_Data,
        Val_Start,
        Bad_Firmware,
        Boot_Wrkn_reg_o,
        Boot_Done_reg_o,
        Bad_FW1,
        Bad_FW2,
        Bad_FW3,
        Bad_FW4,
        Bad_FW5,
        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';
LD_HP_EN <= '0';

LD_HP_Done <= '0';

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

LD_EN_Chk <= '0';

LD_Boot_Done <= '0';

LD_Boot_Wrkn <= '0';

Temp_Stop_Chk <= '0';
LD_Stop_Chk <= '0';
```

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

LD_HP_Cmd <= '0';


Temp_Error <= '0';

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';


Temp_DisableWatchdog <= '0';

LD_DisableWatchdog <= '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';

Temp_Stop_Chk <= '0';
LD_Stop_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;
        CntDelay_INC <= '1';
    ELSE
        NS<=S2;
    END IF;


WHEN S2=>                                                   -- Wait
    IF(CntBus_Out < 128) THEN
        NS<=S2;
        CntBus_INC<='1';
    ELSE
```

```
                    NS<=S3;
                END IF;

            WHEN S3 =>                              -- Wait for Bus
Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S3;
                    CntBus_Rst <='0';               -- Reset Bus Counter
                    CntDelay_Rst <='0';
                ELSE
                    NS <=S4;
                END IF;

            WHEN S4 =>                              -- Request if the
Validation Start button was pressed (Load & Verify)
                Bus_Int1_AddrIn <= Addr_Validation_Start; --
Addr_Validation_Start is a constant from Common file: = X0B08 = 2824
                Bus_Int1_RE <='1';
                NS <= S5;

            WHEN S5 =>                              -- Wait for Bus
Control
                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 =>                          -- Store the data
collected from the BUS to the Validation Start variable
                Temp_Val_Start <= Vrble_Data(0);
                LD_Val_Start <= '1';
                NS <= S7;

            WHEN S7=>                           -- Reset the Hot-Patch
status
                Bus_Int1_AddrIn <= Addr_HP_Status; -- Addr_HP_Status is a
constant from Common file
                Bus_Int1_DataIn <= X"0000"; -- HP_Stat = 0 (Done/Disabled)
                Bus_Int1_WE <='1';
                NS <= S8;
            WHEN S8 =>                              -- Check if the
Validation Start button was pressed, if not, roll back to S0
                IF (Val_Start = '1') THEN
                    NS <= S100;
                ELSE
                    NS <= S0;
                END IF;

            WHEN S100 =>                                -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S100;
                    CntBus_Rst <='0';           -- Reset Bus Counter
                    CntDelay_Rst <='0';
                ELSE
                    NS <=S101;
                END IF;

            WHEN S101 =>                                -- Request if
watchdog is disabled
                Bus_Int1_AddrIn <= Addr_DisableWatchdog; --
Addr_DisableWatchdog is a constant from Common file: = x0043
                Bus_Int1_RE <='1';
```

```
                                NS <= S102;


                        WHEN S102 =>                                                    -- Wait for
Bus Control
                                IF(Bus_Int1_Busy = '1') THEN
                                        NS <= S102;
                                ELSE
                                        NS <=S103;
                                END IF;
                                Temp_Vrble_Data <= Bus_Int1_DataOut;
                                LD_Vrble_Data <= '1';


                        WHEN S103 =>                                                    -- Store the
data collected from the BUS to the Validation Start variable
                                Temp_DisableWatchdog <= Vrble_Data(0);
                                LD_DisableWatchdog <= '1';
                                NS <= S104;


                        WHEN S104 =>
                                IF (DisableWatchdogReg = '1') THEN
                                        NS <= S30;
                                ELSE
                                        NS <= S9;
```

```
                END IF;

    ------------------------------------ Start the verification process --------------------------------------
------
            WHEN S9=>                                       -- If the Validation
Start (Load & Verify) button was pressed, start the firmware checking and emulation process
                Temp_HP_EN <= '0';                      -- Hot-Patching not enabled
                LD_HP_EN <= '1';

                Temp_Emu_EN <= '1';                         -- Enable emulation
                LD_Emu_EN <= '1';

                EN_Chk <= '1';                              -- Enable verification
processes
                LD_EN_Chk <= '1';

                NS <= S10;

            WHEN S10 =>                                 -- Set HP Status to busy

                Bus_Int1_AddrIn <= Addr_HP_Status;
                Bus_Int1_DataIn <= x"0002";
                Bus_Int1_WE <='1';
                NS <= S11;

            WHEN S11 =>                     -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S11;
                ELSE
                    NS <=S12;
                END IF;

            WHEN S12 =>
                NS <= S13;

            WHEN S13=>                  -- Wait for
fundamental period (60Hz) to have enough time for all the validation processes
                IF (Cnt_PreChk_Out < X"47868C0") THEN
                    Cnt_PreChk_INC <= '1';
                    NS <= S13;
                ELSE
                    Cnt_PreChk_INC <= '0';
                    NS <= S14;
                END IF;
                Temp_HP_EN <= '0';
                LD_HP_EN <= '1';
                Cnt_PreChk_Rst <= '1';

            WHEN S14=>                                  -- Check for Error. Error signal goes to all Modules
                IF (Bad_Firmware = '1' ) THEN
                    Temp_Error <= '1';
                    LD_Error <= '1';
                    NS <= S15;
                ELSE
                    Temp_Error <= '0';
                    LD_Error <= '1';
                    NS <= S30;
                END IF;

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


            -- Start Bootload Backup IF needed. Bootloader control may be able to handle the situation IF it receives the Error signal
            -- Set the Error Type from Bad_FW# into Err_Type
            WHEN S15 =>
```

```
IF(Boot_Wrkn_reg_o = '0')THEN
        NS <= S15;
ELSE
        NS <= S16;

END IF;
Temp_Error <= '1';
LD_Error <= '1';
Temp_HP_EN <= '0';
LD_HP_EN <= '1';

-- Error type
Temp_Err_Type(0) <= Bad_FW1; -- Short-Circuit
Temp_Err_Type(1) <= Bad_FW2; -- Deadtime
Temp_Err_Type(2) <= Bad_FW3; -- Fundamental Frequency
Temp_Err_Type(3) <= Bad_FW4; -- Fast Frequency (MOSFET 2 and 3)
Temp_Err_Type(4) <= Bad_FW5; -- Timer

LD_Err_Type <= '1';
LD_Boot_Wrkn <= '1';
```

```
            WHEN S16 =>                     -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S16;
                ELSE
                    NS <=S17;
                END IF;

            WHEN S17 =>                     -- Set the
Error Type RAM Reg
                Bus_Int1_AddrIn <= Addr_ERROR;
                Bus_Int1_DataIn <= Err_Type;
                Bus_Int1_WE <='1';
                NS <= S18;

            WHEN S18 =>                     -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S18;
                ELSE
                    NS <=S19;
                END IF;
            WHEN S19=>                              -- Set the Error bit on
HP RAM Reg
                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 <= S20;

            WHEN S20 =>                                 -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S20;
                ELSE
                    NS <=S21;
                END IF;
                LD_Boot_Done <= '1';

            WHEN S21 =>                                 -- Wait until
Bootload is done with Backup
                IF(Boot_Done_reg_o = '0')THEN
                    NS <= S21;
                ELSE
                    NS <= S22;
                END IF;

                Temp_HP_EN <= '0';
                LD_HP_EN <= '1';
                LD_Boot_Done <= '1';
                EN_Chk <= '0';
                LD_EN_Chk <= '1';
                Temp_Emu_EN <= '0';
                LD_Emu_EN <= '1';
                Cnt_PreChk_Rst <= '0';

            WHEN S22 =>                                     -- Reset Error
to all Modules, Reset Err_Type, and SEND Stop_Chk to all Checks
                Temp_Error <= '0';
                LD_Error <= '1';
                Temp_Err_Type <= (others => '0');
                LD_Err_Type <= '1';
                Temp_Stop_Chk <= '1';
                LD_Stop_Chk <= '1';
                NS <= S23;

            WHEN S23=>                                  -- Reset the validation
start register
```

```
                Bus_Int1_AddrIn <= Addr_Validation_Start;
                Bus_Int1_DataIn <= X"0000";
                Bus_Int1_WE <='1';
                NS <= S24;

            WHEN S24 =>                                     -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S24;
                ELSE
                    NS <=S25;
                END IF;
                Temp_Val_Start <= '0';
                LD_Val_Start <= '1';

            WHEN S25=>                                 -- Disable the Hot-
Patch command
                Bus_Int1_AddrIn <= Addr_HP_Cmd;
                Bus_Int1_DataIn <= X"0000"; -- Do not hot-patch: command =
0000
                Bus_Int1_WE <='1';
                NS <= S26;
            WHEN S26 =>                                        -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S26;
                ELSE
                    NS <=S27;
                END IF;
                Temp_HP_Cmd <= (others => '0');
                LD_HP_Cmd <= '0';


            WHEN S27=>                                     -- Reset Hot-Patch
status
                Bus_Int1_AddrIn <= Addr_HP_Status;
                Bus_Int1_DataIn <= X"0000"; -- HP_Status = 0 (Done/Disabled)
                Bus_Int1_WE <='1';
                NS <= S28;

            WHEN S28 =>                                        -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S28;
                ELSE
```

```
                    NS <=S29;

                END IF;


            WHEN S29 =>                                 -- Wait until

the error process is done

                IF(Bad_Firmware = '1')THEN

                    Temp_Stop_Chk <= '1';      -- Stop_Chk = 1 will tell

Check modules to restart and reset their Bad_FW signal to 0.

                    LD_Stop_Chk <= '1';

                    NS <= S29;

                ELSE

                    Temp_Stop_Chk <= '0';

                    LD_Stop_Chk <= '1';

                    NS <= S0;

                END IF;


    ----------------------------------- END ERROR Procedure ---------------------------------------

----

            WHEN S30=>                              -- Wait for Bus

Control

                IF(Bus_Int1_Busy = '1') THEN

                    NS <= S30;

                ELSE
```

```
                    NS <=S31;
                END IF;

            WHEN S31=>                              -- If there is no error,
set the Hot-Patch status to Ready
                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 <= S32;

            WHEN S32 =>                                 -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S32;
                ELSE
                    NS <=S33;
                END IF;

            WHEN S33 =>                                 -- Read the
Hot-Patch command, waiting for the user to press the Hot-Patch button
                Bus_Int1_AddrIn <= Addr_HP_Cmd;
                Bus_Int1_RE <='1';
                NS <= S34;


            WHEN S34 =>                                 -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S34;
                ELSE
                    NS <=S35;
                END IF;
                Temp_Vrble_Data <= Bus_Int1_DataOut;
                LD_Vrble_Data <= '1';



            WHEN S35=>                              -- Check for errors
one more time
                IF (Bad_Firmware = '1') THEN
                    Temp_Error <= '1';
                    LD_Error <= '1';

                    NS <= S15;
                ELSE
                    Temp_Error <= '0';
                    LD_Error <= '1';
                    NS <= S36;

                END IF;


            WHEN S36 =>                                 -- Store the

data collected from bus to the Hot-Patch command register

                Temp_HP_Cmd <= Vrble_Data;

                LD_HP_Cmd <= '1';

                Temp_Emu_EN <= '0';

                LD_Emu_EN <= '1';


                EN_Chk <= '0';

                LD_EN_Chk <= '1';

                NS <= S37;


            WHEN S37 =>                                 -- Wait for the

Hot-Patch button to be pressed

                IF (HP_Cmd > X"0000") THEN

                    NS <= S38;

                ELSE

                    NS <= S30;

                END IF;

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



            WHEN S38 =>                                     -- Check for
errors once more
                IF (Bad_Firmware = '1' ) THEN
                    Temp_Error <= '1';
                    LD_Error <= '1';
                    NS <= S15;
                ELSE
                    Temp_Error <= '0';
                    LD_Error <= '1';
                    NS <= S39;
                END IF;
                Temp_HP_EN <= '0';
                LD_HP_EN <= '1';

            WHEN S39 =>                                     -- Turn
everything off, prepare to hot-patch
                Temp_HP_EN <= '0';
                LD_HP_EN <= '1';
                Cnt_PreChk_Rst <= '0';
                Temp_Stop_Chk <= '1';
                LD_Stop_Chk <= '1';

                NS <= S40;

            WHEN S40 =>                                     -- Enable Hot-
Patch
                Temp_HP_EN <= '1';
                LD_HP_EN <= '1';
                LD_HP_Done <= '1';

                NS <= S41;

            WHEN S41 =>
                LD_HP_Done <= '1';

                NS <= S42;

            WHEN S42 =>                                     -- Wait until
the Hot-Patch is done
                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';

            WHEN S43 =>                                 -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S43;
                ELSE
                    NS <=S44;
                END IF;

            WHEN S44 =>                                 -- Set Hot-
Patch status to "Done"
                Bus_Int1_AddrIn <= Addr_HP_Status;
                Bus_Int1_DataIn <= X"0000";
                Bus_Int1_WE <='1';

                NS <= S45;
            WHEN S45 =>                                 -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S45;
                ELSE
                    NS <= S46;
                END IF;

            WHEN S46 =>                                 -- Reset
Validation Start register
                Bus_Int1_AddrIn <= Addr_Validation_Start;
                Bus_Int1_DataIn <= X"0000";
                Bus_Int1_WE <='1';
                NS <= S47;

            WHEN S47 =>                                 -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S47;
                ELSE
                    NS <=S48;
                END IF;
                Temp_Val_Start <= '0';
                LD_Val_Start <= '1';

            WHEN S48=>                              -- Reset the Hot-Patch
command register
                Bus_Int1_AddrIn <= Addr_HP_Cmd;
                Bus_Int1_DataIn <= X"0000";
                Bus_Int1_WE <='1';

                NS <= S49;

            WHEN S49 =>                                 -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S49;
                ELSE
                    NS <=S50;
                END IF;
                Temp_HP_Cmd <= (others => '0');
                LD_HP_Cmd <= '0';

            WHEN S50 =>                                 -- Reset Error
Type vector
                Temp_Err_Type <= (others => '0');
                LD_Err_Type <= '1';

                NS <= S51;

            WHEN S51 =>                                 -- Wait for
Bus Control
                IF(Bus_Int1_Busy = '1') THEN
                    NS <= S51;
                ELSE
                    NS <=S52;
                END IF;

            WHEN S52 =>                                 -- Set Error
Type register to zeros
                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 =>                                 -- Wait for
Bus Control
```

```
                                IF(Bus_Int1_Busy = '1') THEN
                                        NS <= S53;
                                ELSE
                                        NS <=S54;
                                END IF;

                        WHEN S54 =>                                              -- Swtich the
Active DSP register
                                Bus_Int1_AddrIn <= Addr_DSP_Active;
                                if (DSP1_Act = '1') THEN
                                        Bus_Int1_DataIn <= X"0000";
                                ELSE
                                        Bus_Int1_DataIn <= X"0001";
                                END IF;
                                Bus_Int1_WE <='1';

                                NS <= S55;

                        WHEN S55 =>                                              -- Wait for
Bus Control
                                IF(Bus_Int1_Busy = '1') THEN
                                        NS <= S55;
                                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;
        DSPEnable <= '0';
    ELSE
        CS <= NS;
        CS_Chk <= NS_Chk;
        CS_ShCrk <= NS_ShCrk;
        DSPEnable <= EN_Chk_reg_o;
        END IF;


        --debug_FW_Val_E1 <= Bad_FW1;
        --debug_FW_Val_E2 <= Bad_FW2;
        --debug_FW_Val_E3 <= Bad_FW3;
        --debug_FW_Val_E4 <= Bad_FW4;
        --debug_FW_Val_E5 <= Bad_FW5;
        --debug_FW_Val_EN <= EN_Chk_reg_o;
    END PROCESS;
    ----END State Sync
END Behavioral;
```

A-3: Short-circuit

```
--------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Paulo Custodio
--
-- Create Date:            11/18/2021
-- Project Name:           Digital_Twin
-- Module Name:            Dead Time
-- Project Name:           Digital_Twin_DeadTime
-- Target Devices:         LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:                    Lattice Diamond_x64 Build 3.11
-- Description:
-- This project was created to detect a Deadtime error, if the DSP firmware does not have enough
deadtime.
-- To check if the deadtime is sufficient, this project waits for Q1/Q6 to change from 1 to 0, and
start counting until Q4/Q5 change from 0 to 1.
-- After Q4/Q5 became "1", then the counter is compared with the minimum number of clock
cycles (deadtime). If the counter is greater than the minimum number of clock cycles,
-- it means that the deadtime is enough, otherwise it must set the error flag to "1" and stop all
other processes.
-- The delay of 12ms(300,000 clock cycles) on the first state is necessary to ignore random
outputs from the DSP while it's being bootloaded.
---- PinOut:
--
-- Revision
--        v2.15.22 - Debug signal added; Starts with 0 and when the deadtime is enable, should
change to 1.
--        v3.24.22 - Deadtime created as a component to check two different PWMs. Delay added
to ignore the first 12ms of DSP signals
--
-- Additional Comments:
--
--
```

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

Library IEEE;
Library STD;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use STD.textio.all;
use IEEE.std_logic_textio.all;

library machxo2;
use machxo2.all;

library work;
use work.Digital_Twin_Common.all;


entity Digital_Twin_ShortCircuit is
   Port (
                  -- Inputs
         clk                    : in std_logic;
         rst                    : in std_logic;
         ShCrkEnable            : in std_logic;
```

```
    Cnt_PreChk_Out    : in std_logic_vector(31 downto 0);

    Emu_SW01_A            : in std_logic;
    Emu_SW04_A            : in std_logic;
    Emu_SW05_A            : in std_logic;
    Emu_SW06_A            : in std_logic;

    Emu_SW01_B            : in std_logic;
    Emu_SW04_B            : in std_logic;
    Emu_SW05_B            : in std_logic;
    Emu_SW06_B            : in std_logic;

    Emu_SW01_C            : in std_logic;
    Emu_SW04_C            : in std_logic;
    Emu_SW05_C            : in std_logic;
    Emu_SW06_C            : in std_logic;

        -- Outputs
    DisableTimer  : out std_logic;
    Bad_FW1               : out std_logic
    );
end Digital_Twin_ShortCircuit;
```

```
architecture Behavioral of Digital_Twin_ShortCircuit is


        type state_type is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S_error);
        signal CS_ShCrk, NS_ShCrk : state_type;


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




BEGIN --------------------------------------------------------- BEGIN ---------------------------------------
------------------


        -- Error register
        Error: process(clk)
        BEGIN
                if (rising_edge(clk)) then
                        if rst = '0' then
                                Bad_FW1 <= '0';
                        else
                                IF (LD_Bad_FW1      = '1') THEN  Bad_FW1 <=
Temp_Bad_FW1;           END IF;
                        end if;
```

```
          end if;
end process;



-------------------------------- Short-Circuit Check --------------------------------
Short_Circuit_Check : PROCESS
BEGIN
          LD_Bad_FW1 <= '0';
          Temp_Bad_FW1 <= '0';
          DisableTimer <= '0';
          case CS_ShCrk is
                    WHEN S0 =>
                              IF (ShCrkEnable = '0') THEN
                                        NS_ShCrk <= S0;
                              ELSE
                                        NS_ShCrk <= S1;
                              END IF;
                              Temp_Bad_FW1 <= '0';
                              LD_Bad_FW1 <= '1';
                              DisableTimer <= '0';

                    when S1 => -- Wait for the positive Cycle: Phase A Q1 ON
                              if (Emu_SW01_A = '1') then
```

```
                NS_ShCrk <= S1;
        ELSE
                NS_ShCrk <= S2;
        end if;
        IF (ShCrkEnable = '0') THEN
                NS_ShCrk <= S0;
        END IF;

when S2 => -- Wait for Q1 and Q6 to be off (Phase A)
        IF (Emu_SW01_A = '0') THEN
                NS_ShCrk <= S2;
        ELSE
                NS_ShCrk <= S3;
        END IF;
        IF (ShCrkEnable = '0') THEN
                NS_ShCrk <= S0;
        END IF;

when S3 => -- Wait for the positive Cycle: Phase A Q1 ON
        if (Emu_SW01_A = '1') then
                NS_ShCrk <= S3;
        ELSE
                NS_ShCrk <= S4;
        end if;
        IF (ShCrkEnable = '0') THEN
                NS_ShCrk <= S0;
        END IF;

when S4 => -- Wait for Q1 and Q6 to be off (Phase A)
        IF (Emu_SW01_A = '0') THEN
                NS_ShCrk <= S4;
        ELSE
                NS_ShCrk <= S5;
        END IF;
        IF (ShCrkEnable = '0') THEN
                NS_ShCrk <= S0;
        END IF;

when S5 => -- Wait for the positive Cycle: Phase B Q1 ON
        if (Emu_SW01_B = '1') then
                NS_ShCrk <= S5;
        ELSE
                NS_ShCrk <= S6;
        end if;
        IF (ShCrkEnable = '0') THEN
                NS_ShCrk <= S0;
        END IF;

    when S6 => -- Wait for Q1 and Q6 to be off
        IF (Emu_SW01_B = '0') THEN
            NS_ShCrk <= S6;
        ELSE
            NS_ShCrk <= S7;
        END IF;
        IF (ShCrkEnable = '0') THEN
            NS_ShCrk <= S0;
        END IF;

    when S7 => -- Wait for the second positive Cycle: Q1 ON
        if (Emu_SW01_C = '1') then
            NS_ShCrk <= S7;
        ELSE
            NS_ShCrk <= S8;
        end if;
        IF (ShCrkEnable = '0') THEN
            NS_ShCrk <= S0;
        END IF;

    when S8 => -- Wait for Q1 and Q6 to be off
```

```
        IF (Emu_SW01_C = '0') THEN
            NS_ShCrk <= S8;
        ELSE
            NS_ShCrk <= S9;
        END IF;
        IF (ShCrkEnable = '0') THEN
            NS_ShCrk <= S0;
        END IF;

    WHEN S9 => -- Short circuit test
        ------ Phase A ------
        IF ((Emu_SW04_A AND Emu_SW06_A) = '1')then
            NS_ShCrk <= S_error;
        elsif ((Emu_SW01_A AND Emu_SW05_A) = '1')then
            NS_ShCrk <= S_error;
        -------- Phase B ------
        elsif((Emu_SW04_B AND Emu_SW06_B) = '1')then
            NS_ShCrk <= S_error;
        elsif((Emu_SW01_B AND Emu_SW05_B) = '1')then
            NS_ShCrk <= S_error;
        -------- Phase C ------
        elsif((Emu_SW04_C AND Emu_SW06_C) = '1')then
            NS_ShCrk <= S_error;
```

```
                elsif((Emu_SW01_C AND Emu_SW05_C) = '1')then
                    NS_ShCrk <= S_error;
                else
                    NS_ShCrk <= S10;
                END IF;
                IF (ShCrkEnable = '0') THEN
                    NS_ShCrk <= S0;
                END IF;

            WHEN S10 =>
                IF (Cnt_PreChk_Out < X"42C1D80") THEN       -- Do not stop
checking
                    NS_ShCrk <= S9;
                ELSE                                         -- Stop
checking
                    NS_ShCrk <= S11;
                END IF;
                IF (ShCrkEnable = '0') THEN
                    NS_ShCrk <= S0;
                END IF;

            WHEN S11 => -- Sit and wait
```

```
                IF (ShCrkEnable = '1') THEN
                    NS_ShCrk <= S11;
                ELSE                                            -- Stop
checking
                    NS_ShCrk <= S0;
                END IF;
                DisableTimer <= '1';

            WHEN S_error =>
                IF (ShCrkEnable = '1') THEN-- Flag erro, sit and wait
                    Temp_Bad_FW1 <= '1';
                    LD_Bad_FW1 <= '1';
                    NS_ShCrk <= S_error;
                ELSE                                            -- Stop
checking
                    NS_ShCrk <= S0;
                END IF;
                DisableTimer <= '1';

            WHEN others =>
                NS_ShCrk <= S0;

        END case;
```

```
    END PROCESS;

    ------------------- State Sync -------------------

    sync_States: process

    begin

        wait until clk'event and clk = '1';

        if rst = '0' then

            CS_ShCrk <= S0;

        else

            CS_ShCrk <= NS_ShCrk;

        end if;

    end process;


end Behavioral;
```

A-4: Deadtime

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

-- Company:  University of Arkansas (NCREPT)

-- Engineer: Estefano Soria and Paulo Custodio

--

-- Create Date:                11/18/2021

-- Project Name:               Digital_Twin

-- Module Name:                Dead Time

-- Project Name:               Digital_Twin_DeadTime
-- Target Devices:            LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:             Lattice Diamond_x64 Build 3.11
-- Description:
-- This project was created to detect a Deadtime error, if the DSP firmware does not have enough
deadtime.
-- To check if the deadtime is sufficient, this project waits for Q1/Q6 to change from 1 to 0, and
start counting until Q4/Q5 change from 0 to 1.
-- After Q4/Q5 became "1", then the counter is compared with the minimum number of clock
cycles (deadtime). If the counter is greater than the minimum number of clock cycles,
-- it means that the deadtime is enough, otherwise it must set the error flag to "1" and stop all
other processes.
-- The delay of 12ms(300,000 clock cycles) on the first state is necessary to ignore random
outputs from the DSP while it's being bootloaded.
---- PinOut:
--
-- Revision
--       v2.15.22 - Debug signal added; Starts with 0 and when the deadtime is enable, should
change to 1.
--       v3.24.22 - Deadtime created as a component to check two different PWMs. Delay added
to ignore the first 12ms of DSP signals
--
-- Additional Comments:
--
```

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


Library IEEE;

Library STD;

use IEEE.std_logic_1164.all;

use ieee.std_logic_unsigned.all;

use ieee.numeric_std.all;

use STD.textio.all;

use IEEE.std_logic_textio.all;


library machxo2;

use machxo2.all;


library work;

use work.Digital_Twin_Common.all;



entity Digital_Twin_DeadTime is

   Port (

         clk                      : in std_logic;

         rst                      : in std_logic;

         DeadTime_Enable          : in std_logic;
```

```
    DeadTimeError           : out std_logic;


    Emu_SW01    : in std_logic;
    Emu_SW06    : in std_logic;
    Emu_SW04    : in std_logic;
    Emu_SW05    : in std_logic
    );
end Digital_Twin_DeadTime;


architecture Behavioral of Digital_Twin_DeadTime is


    type state_type is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S_error, delay);
    signal CS_DeadT, NS_DeadT : state_type;


    --------------------- Signals for Phase A ---------------------
    ----------      Counter     -----------
    signal Cnt_DeadT_INC : std_logic := '0';
    signal Cnt_DeadT_Rst : std_logic := '0';
    signal Cnt_DeadT_Out : std_logic_vector(31 downto 0) := (others => '0');
    ----------      Error   -----------
    signal Temp_Error       :       std_logic := '0';
    signal LD_Error         :       std_logic := '0';
    constant numberOfClockCycles :std_logic_vector(7 downto 0) := X"19"; -- 25MHz -> 40ns period

    --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 --------------------------------------------------------- BEGIN ---------------------------------------------------------

    -- Counter to check the deadtime
    Det_Cnt: Std_Counter
    generic map
    (
        Width => 32
```

```
)
port map(
        clk => clk,
        rst=> Cnt_DeadT_Rst,
        INC=> Cnt_DeadT_INC,
        Count=> Cnt_DeadT_Out
);


-- Error register
Error: process(clk)
BEGIN
        if (rising_edge(clk)) then
                if rst = '0' then
                        DeadTimeError <= '0';
                else
                        if (LD_Error = '1') then        DeadTimeError <= Temp_Error;
end if;
                end if;
        end if;
end process;


-- Main Process
Dead_Time : process(
        Cnt_DeadT_Out,
        --_Counter_out,
        CS_DeadT,
        Emu_SW01,
        Emu_SW06,
        Emu_SW04,
        Emu_SW05,
        DeadTime_Enable
)
BEGIN
        LD_Error <= '0';
        Cnt_DeadT_Rst <= '1';
        Cnt_DeadT_INC <= '0';

        case CS_DeadT is
                when S0 =>    -- Wait until Enable is High

                        if(DeadTime_Enable = '0')then
                                NS_DeadT <= S0;
                        else
                                NS_DeadT <= S1;
                        end if;
                        Cnt_DeadT_Rst <= '0';
        Temp_Error <= '0';

        LD_Error <= '1';


---- Ignore first cycle ----

when S1 => -- Wait for the positive Cycle: Q1 ON

        if (Emu_SW01 = '0') then

                NS_DeadT <= S1;

        ELSE

                NS_DeadT <= S2;

        end if;

        IF (DeadTime_Enable = '0') THEN

                NS_DeadT <= S0;

        END IF;


when S2 => -- Wait for Q1 and Q6 to be off

        IF (Emu_SW01 = '1') THEN

                NS_DeadT <= S2;

        ELSE

                NS_DeadT <= S3;

        END IF;

        IF (DeadTime_Enable = '0') THEN

                NS_DeadT <= S0;

        END IF;
```

```
---- Ignore second cycle ----

when S3 => -- Wait for the positive Cycle: Q1 ON

        if (Emu_SW01 = '0') then

                NS_DeadT <= S3;

        ELSE

                NS_DeadT <= S4;

        end if;

        IF (DeadTime_Enable = '0') THEN

                NS_DeadT <= S0;

        END IF;


when S4 => -- Wait for Q1 and Q6 to be off

        IF (Emu_SW01 = '1') THEN

                NS_DeadT <= S4;

        ELSE

                NS_DeadT <= S5;

        END IF;

        IF (DeadTime_Enable = '0') THEN

                NS_DeadT <= S0;

        END IF;

---- Prepare to validate ----

when S5 => -- Wait for the positive Cycle: Q1 ON
```

```
        if (Emu_SW01 = '0') then
                NS_DeadT <= S5;
        ELSE
                NS_DeadT <= S6;
        end if;
        IF (DeadTime_Enable = '0') THEN
                NS_DeadT <= S0;
        END IF;
when S6 => -- Wait for Q1 and Q6 to be off
        IF (Emu_SW01 = '1') THEN
                NS_DeadT <= S6;
        ELSE
                NS_DeadT <= S7;
        END IF;
        IF (DeadTime_Enable = '0') THEN
                NS_DeadT <= S0;
        END IF;

---- Validate ----
when S7 => -- While Q1, Q4, Q5 and Q6 are off, count
        IF (Emu_SW04 = '1') THEN
                NS_DeadT <= S8;
        ELSE
        NS_DeadT <= S7;
    END IF;
    Cnt_DeadT_INC <= '1';
    IF (DeadTime_Enable = '0') THEN
        NS_DeadT <= S0;
    END IF;

when S8 => -- Check if the counter > deadtime
    IF (Cnt_DeadT_Out > numberOfClockCycles) THEN
        NS_DeadT <= S9; -- No errors
    ELSE
        NS_DeadT <= S_error;
    END IF;
    IF (DeadTime_Enable = '0') THEN
        NS_DeadT <= S0;
    END IF;

WHEN S9 => -- Sit and wait
    if (DeadTime_Enable = '1') then
        NS_DeadT <= S9;
    else
        NS_DeadT <= S0;
    end if;
```

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


        when S_error => -- Wait until reset or stay in this state holding the error
            IF (DeadTime_Enable = '1') then
                NS_DeadT <= S_error;
            else
                NS_DeadT <= S0;
            end if;
            Temp_Error <= '1';
            LD_Error <= '1';


        when others =>
            NS_DeadT <= S0;
        end case;
end process;



------------------- State Sync -------------------
sync_States: process
begin
    wait until clk'event and clk = '1';
    if rst = '0' then
```

```
                        CS_DeadT <= S0;
                else
                        CS_DeadT <= NS_DeadT;
                end if;
        end process;

end Behavioral;
```

A-5: Fast Frequency

```
----------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Paulo Custodio
--
-- Create Date:             01/17/2023
-- Project Name:            Digital_Twin
-- Module Name:             Fundamental Frequency
-- Project Name:            Digital_Twin_Fast_Frequency
-- Target Devices:          LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:           Lattice Diamond_x64 Build 3.11
-- Description:
-- This project goal is to detect the frequency of the fast frequency transistors and indicate an
error in case the frequency is not close to 42kHz.
```

```
---- PinOut:
--
-- Revision
--
-- Additional Comments:
--
--
----------------------------------------------------------------------------------
Library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;


entity FastFrequency_detector is
        generic (
                maxValue : std_logic_vector(19 downto 0) := X"00300"; -- Ideal value is 595,
which is 253h
                minValue : std_logic_vector(19 downto 0) := X"00200"
        );
        port
        (
                --debug_FF_detector  : out std_logic;
        SW                          : in std_logic;
        enable_ff_check             : in std_logic;
        stop                        : in std_logic;
        clk                         : in std_logic;
        rst                         : in std_logic;
        FF_det_error                : out std_logic
    );
end;



architecture BEHAVIOR of FastFrequency_detector 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;


    -- State signals
    type state_type is (S0, S1, S2, S3, S4, S5, S6, S7, S_error);
    signal CS, NS : state_type;

    -- Counter signals
    signal DC_INC : STD_LOGIC := '0';
    signal DC_cnt_out : STD_LOGIC_VECTOR(19 downto 0);
    signal DC_counter_rst : STD_LOGIC := '0';

    signal TimerDelay_INC : STD_LOGIC := '0';
    signal TimerDelay_OUT : STD_LOGIC_VECTOR(31 downto 0);
    signal TimerDelay_RST : STD_LOGIC := '0';

    -- Error signals
    signal FF_det_sig : STD_LOGIC := '0';
    signal LD_FF_det : STD_LOGIC := '0';
    signal det_overflow : STD_LOGIC := '0';

BEGIN

    -- instantiate DC counter
    DC_Cnt: Std_Counter
    generic map
    (
            Width => 20
    )
    port map(
            clk => clk,
            rst=> DC_counter_rst,
            INC=> DC_INC,
            Count=> DC_cnt_out
    );

    TimerDelay: Std_Counter
    generic map
    (
            Width => 32
    )
    port map(
            clk => clk,
            rst=> TimerDelay_RST,
            INC=> TimerDelay_INC,
            Count=> TimerDelay_OUT
    );

    ----Registers
    Reg_Proc: process
```

```
    begin
        wait until clk'event and clk = '1';
        if rst = '0' then
            FF_det_error <= '0';
        else
            if (LD_FF_det = '1') then  FF_det_error <= FF_det_sig; end if;
        end if;
    end process;
    ----End Registers


    process
    begin
        DC_INC <= '0';
        FF_det_sig <= '0';
        LD_FF_det <= '0';
        DC_counter_rst <= '0';
        TimerDelay_RST <= '0';
        TimerDelay_INC <= '0';
        case CS is
            when S0 =>                          -- Wait for the enable signal from the
Firmware Validation process
                if (enable_ff_check = '0') then
                    NS <= S0;
                else
                    NS <= S1;
                end if;



            when S1 => -- Ignore the first second to ignore transion values
                if (TimerDelay_OUT < X"17D7840") then --17D 7840 =
25,000,000 = 1s
                    NS <= S1;
                else
                    NS <= S2;
                end if;
                TimerDelay_RST <= '1';
                TimerDelay_INC <= '1';
                IF (enable_ff_check = '0') THEN
                    NS <= S0;
                END IF;

            ------------ Synchonization ------------
            when S2 =>                          -- Wait for SW to go high
                if (SW = '0') then
                    NS <= S2;
                else
                    NS <= S3;
                end if;
                DC_counter_rst <= '1';
                IF (enable_ff_check = '0') THEN
                    NS <= S0;
                END IF;

            when S3 =>                          -- Wait for SW to go low
                if (SW = '1') then
                    NS <= S3;
                else
                    NS <= S4;
                end if;
                DC_counter_rst <= '1';
                IF (enable_ff_check = '0') THEN
                    NS <= S0;
                END IF;
            ------------ Start ------------
            -- Negatie cycle of the new period
            when S4 =>                          -- Count while is low
                if (SW = '0') then
                    NS <= S4;
                else
```

```
                    NS <= S5;
                end if;
                DC_counter_rst <= '1';
                DC_INC <= '1'; -- Count while is low
                IF (enable_ff_check = '0') THEN
                    NS <= S0;
                END IF;

            -- Positive cycle of the new period
            when S5 =>
                if (SW = '1') then
                    NS <= S5;
                else
                    NS <= S6;
                end if;
                DC_counter_rst <= '1'; -- Keep counting while is high
                DC_INC <= '1';
                IF (enable_ff_check = '0') THEN
                    NS <= S0;
                END IF;

            -- Counting is over, check if the number of clock cycles are in the
acceptable range
```

```
            when S6 =>
                if ((minValue < DC_cnt_out) AND (DC_cnt_out < maxValue))
then
                    NS <= S7; -- No error
                else
                    NS <= S_error; -- Error
                end if;
                DC_counter_rst <= '1';
                IF (enable_ff_check = '0') THEN
                    NS <= S0;
                END IF;

            -- Firmware is valid. Wait for the enable to be turned off
            when S7 =>
                if (enable_ff_check = '0') then
                    NS <= S0; -- stop checking
                else
                    FF_det_sig <= '0';
                    LD_FF_det <= '1';
                    NS <= S7; -- Wait
                end if;
            -- Error detected. Wait for the stop checking from Firmware Validation process, before going back to S0
            when S_error =>
                if (enable_ff_check = '1') then
                    FF_det_sig <= '1'; --Error
                    LD_FF_det <= '1';
                    NS <= S_error;
                else
                    FF_det_sig <= '0';
                    LD_FF_det <= '1';
                    NS <= S0;
                end if;

        end case;
    end process;

    ----State Sync
    sync_States: process
    begin
        wait until clk'event and clk = '1';
        if rst = '0' then
            CS <= S0;
            --debug_FF_detector <= '0';
        else
            CS <= NS;
            --debug_FF_detector <= DC_INC;
        end if;
    end process;
    ----End State Sync
END BEHAVIOR;
```

A-6: Fundamental Frequency

```
---------------------------------Fundamental Frequency detector----------------------
--------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Paulo Custodio
--
-- Create Date:          03/16/2022
-- Project Name:         Digital_Twin
-- Module Name:          Fundamental Frequency
-- Project Name:         Digital_Twin_Fundamental_Frequency
-- Target Devices:       LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:        Lattice Diamond_x64 Build 3.11
-- Description:
```

```
-- This project goal is to detect the frequency of the low frequency transistors and indicate an
error in case the frequency is not close to 60Hz.
---- PinOut:
--
-- Revision
--        v2.15.22 - Debug signal added; Starts with 0 and when the deadtime is enable, should
change to 1.
--        v5.27.22 - Comments and Polish.
--
-- Additional Comments:
--
--
--------------------------------------------------------------------------------
Library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;



entity FF_detector is
        generic (
                maxValue : std_logic_vector(19 downto 0) := X"67C28";    -- 668A0h = 59.5Hz =
420,000 clock cycles + 5,000 margin
```

```
        minValue : std_logic_vector(19 downto 0) := X"64D48"     -- 64D48h = 60.5Hz =
413,000 clock cycles
    );
    port
    (
        --debug_FF_detector  : out std_logic;
        SW                                 : in std_logic;
        enable_ff_check             : in std_logic;
        stop                               : in std_logic;
        clk                                  : in std_logic;
        rst                                  : in std_logic;
        FF_det_error            : out std_logic
    );
end;


architecture BEHAVIOR of FF_detector 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;


    -- State signals
    type state_type is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S_error);
    signal CS, NS : state_type;


    -- Counter signals
    signal DC_INC : STD_LOGIC := '0';
    signal DC_cnt_out : STD_LOGIC_VECTOR(19 downto 0);
    signal DC_counter_rst : STD_LOGIC := '0';


    -- Error signals
    signal FF_det_sig : STD_LOGIC := '0';
    signal LD_FF_det : STD_LOGIC := '0';
    signal det_overflow : STD_LOGIC := '0';

BEGIN

    -- instantiate DC counter
    DC_Cnt: Std_Counter
```

```
generic map
(
        Width => 20
)
port map(
        clk => clk,
        rst=> DC_counter_rst,
        INC=> DC_INC,
        Count=> DC_cnt_out
);

----Registers
Reg_Proc: process
begin
        wait until clk'event and clk = '1';
        if rst = '0' then
                FF_det_error <= '0';
        else
                if (LD_FF_det = '1') then  FF_det_error <= FF_det_sig; end if;
        end if;
end process;
----End Registers
```

```
    process
    begin
        DC_INC <= '0';
        FF_det_sig <= '0';
        LD_FF_det <= '0';
        DC_counter_rst <= '0';
        case CS is
            when S0 =>                  -- Wait for the enable signal from the
Firmware Validation process
                if (enable_ff_check = '0') then
                    NS <= S0;
                else
                    NS <= S1;
                end if;

    ------------ First ignore cicle ------------
            when S1 => -- Wait for the positive Cycle: Q1 ON
                if (SW = '1') then
                    NS <= S1;
                ELSE
                    NS <= S2;
                end if;
                IF (enable_ff_check = '0') THEN
        NS <= S0;
    END IF;

when S2 => -- Wait for Q1 and Q6 to be off
    IF (SW = '0') THEN
        NS <= S2;
    ELSE
        NS <= S3;
    END IF;
    IF (enable_ff_check = '0') THEN
        NS <= S0;
    END IF;

when S3 => -- Wait for the positive Cycle: Q1 ON
    if (SW = '1') then
        NS <= S3;
    ELSE
        NS <= S4;
    end if;
    IF (enable_ff_check = '0') THEN
        NS <= S0;
    END IF;
```

```
when S4 => -- Wait for Q1 and Q6 to be off

        IF (SW = '0') THEN

                NS <= S4;

        ELSE

                NS <= S5;

        END IF;

        IF (enable_ff_check = '0') THEN

                NS <= S0;

        END IF;


when S5 =>                          -- Wait for SW to go low

        if (SW = '1') then

                NS <= S5;

        else

                NS <= S6;

        end if;

        IF (enable_ff_check = '0') THEN

                NS <= S0;

        END IF;


when S6 =>                          -- Wait for SW to go high

        if (SW = '0') then

                NS <= S6;
        else
                NS <= S7;
        end if;
        DC_counter_rst <= '1';
        IF (enable_ff_check = '0') THEN
                NS <= S0;
        END IF;
        ------------ Start ------------
-- Wait for a new period to start counting
-- Positive cycle of the new period
when S7 =>                              -- Count while is high
        if (SW = '1') then
                NS <= S7;
        else
                NS <= S8;
        end if;
        DC_counter_rst <= '1';
        DC_INC <= '1'; -- Count while is high
        IF (enable_ff_check = '0') THEN
                NS <= S0;
        END IF;


-- Negative cycle of the new period
```

```
            when S8 =>
                if (SW = '0') then
                    NS <= S8;
                else
                    NS <= S9;
                end if;
                DC_counter_rst <= '1'; -- Keep counting while is low
                DC_INC <= '1';
                IF (enable_ff_check = '0') THEN
                    NS <= S0;
                END IF;

            -- Counting is over, check if the number of clock cycles are in the
acceptable range
            when S9 =>
                if ((minValue < DC_cnt_out) AND (DC_cnt_out < maxValue))
then
                    NS <= S10; -- No error
                else
                    NS <= S_error; -- Error
                end if;
                DC_counter_rst <= '1';
                IF (enable_ff_check = '0') THEN
```

```
                    NS <= S0;
                END IF;

            -- Firmware is valid. Wait for the enable to be turned off
            when S10 =>
                if (enable_ff_check = '0') then
                    NS <= S0; -- stop checking
                else
                    FF_det_sig <= '0';
                    LD_FF_det <= '1';
                    NS <= S10; -- Wait
                end if;

            -- Error detected. Wait for the stop checking from Firmware Validation
process, before going back to S0
            when S_error =>
                if (enable_ff_check = '1') then
                    FF_det_sig <= '1'; --Error
                    LD_FF_det <= '1';
                    NS <= S_error;
                else
                    FF_det_sig <= '0';
                    LD_FF_det <= '1';
```

```
                                        NS <= S0;
                              end if;
                    end case;
          end process;


          ----State Sync
          sync_States: process
          begin
                    wait until clk'event and clk = '1';
                    if rst = '0' then
                              CS <= S0;
                              --debug_FF_detector <= '0';
                    else
                              CS <= NS;
                              --debug_FF_detector <= DC_cnt_out(0);
                    end if;
          end process;
          ----End State Sync
END BEHAVIOR;
```

A-7: Watchdog

```
--------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Paulo Custodio and Kelby Haulmark
--
-- Create Date:           03/24/2021
-- Project Name:          Digital_Twin
-- Module Name:           Timer
-- Project Name:          Digital_Twin_Timer
-- Target Devices:        LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:         Lattice Diamond_x64 Build 3.11
-- Description: This project has the purpose to add a timer to deadtime tests, to limit the firmware
validation test
-- to a certain period of time.
-- The counter should start when the deadtime is enabled, and the timer should stop when the
done signal is received or
-- if it overflows, flagging the error 5.
---- 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 work;

use work.Digital_Twin_Common.all;



entity timer_detector is

	PORT(

		enable	: in std_logic;

		done	: in std_logic;

		clk		: in std_logic;

		rst		: in std_logic;

		timer_error	: out std_logic

	);

end;
```

```
architecture BEHAVIOR of timer_detector 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;

--constant requirement : integer := 20; -- Number of clock cycles needed to meet freq

type state_type is (S0, S1, S2, S3, S4);
signal CS, NS : state_type;

signal det_INC : STD_LOGIC := '0';
signal det_cnt_out : STD_LOGIC_VECTOR(31 downto 0);
signal counter_rst : STD_LOGIC := '0';
signal hp_det_sig : STD_LOGIC := '0';
signal LD_hp_det : STD_LOGIC := '0';
signal det_overflow : STD_LOGIC := '0';


begin

        Det_Cnt: Std_Counter

        generic map

        (

                Width => 32

        )

        port map(

                clk => clk,

                rst=> counter_rst,

                INC=> det_INC,

                Count=> det_cnt_out

        );




        ----Registers

        Reg_Proc: process

        begin

                wait until clk'event and clk = '1';

                if rst = '0' then

                        timer_error <= '0';

                else
                        if (LD_hp_det = '1') then  timer_error <= hp_det_sig; end if;


        end if;

        end process;

        ----End Registers


        Main: process (CS, enable, det_overflow, done)

        begin

                counter_rst <= '1';

                hp_det_sig <= '0';

                LD_hp_det <= '0';

                det_INC <= '0';

                case CS is

                        when S0 =>

                                if (enable = '0') then

                                        NS <= S0;

                                else

                                        NS <= S1; -- When enable is 1, start to count.

                                end if;

                                counter_rst <= '0';

                                LD_hp_det <= '1';


                        when S1 =>
```

```
                                    if (done OR det_overflow) = '1' then -- Wait for the done signal or
the overflow
                                          NS <= S2;
                                    else
                                          NS <= S1;
                                    end if;
                                    det_INC <= '1';
                                    IF (enable = '0') THEN
                                          NS <= S0;
                                    END IF;

                              when S2 =>
                                    if (det_overflow = '1') then
                                          hp_det_sig <= '1';
                                          NS <= S3; --Error
                                    else -- Done without overflow
                                          hp_det_sig <= '0';
                                          NS <= S4; -- Ok
                                    end if;
                                    LD_hp_det <= '1';
                                    IF (enable = '0') THEN
                                          NS <= S0;
                                    END IF;
```

```
        when S3 => --error
            if (enable = '1') then
                NS <= S3;
            else
                NS <= S0;
            end if;
            hp_det_sig <= '1';
            LD_hp_det <= '1';

        when S4 => -- Sit and wait.
            if (enable = '1') then
                NS <= S4;
            else
                NS <= S0;
            end if;
            hp_det_sig <= '0';
            LD_hp_det <= '1';

    end case;
end process;

freq_overflow : process(det_cnt_out)
```

```
        begin
                -- Counter becomes bigger than freq range so throw flag
                if (det_cnt_out > X"43B5FC0") then -- 2FAF080 = 2s
                        det_overflow <= '1';
                else
                        det_overflow <= '0';
                end if;
        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 BEHAVIOR;
```

A-8: Emulation (Digital Twin)

```
----------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Estefano Soria and Paulo Custodio
--
-- Create Date:              11/18/2021
-- Project Name:             Digital_Twin
-- Module Name:              Emulation_Control
-- Design Name:              Digital_Twin_Emulation_Control
-- Target Devices:           LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:            Lattice Diamond_x64 Build 3.11
-- Description:
-- This project was first design to emulate the phase-to-phase voltage of a two-level inverter.
-- Then, it was modified to emulate an ANPC inverter that is used on solar farms.
-- It must capture 192 samples, catching each sample every 2500 clock cyles (resolution) and
data will be stored into RAM memory, so the LabVIEW is able to read
-- the RAM memory and display the data, showing the ANPC inverter output.
---- PinOut:
--
-- Revision
--
--
```

```
--
-- Additional Comments:
-- v5.25.22 - Modified the whole project to an ANPC inverter.
--
--------------------------------------------------------------------------------


Library IEEE;
Library STD;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use STD.textio.all;
use IEEE.std_logic_textio.all;

library machxo2;
use machxo2.all;

library work;
use work.Digital_Twin_Common.all;

ENTITY Digital_Twin_Emulation_Control 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;

--Phase A Inputs
Emu_SW01_A          : in std_logic;
Emu_SW02_A          : in std_logic;
Emu_SW03_A          : in std_logic;
Emu_SW04_A          : in std_logic;
Emu_SW05_A          : in std_logic;
Emu_SW06_A          : in std_logic;

--Phase B Inputs
Emu_SW01_B          : in std_logic;
```

```
                Emu_SW02_B          : in std_logic;

                Emu_SW03_B          : in std_logic;

                Emu_SW04_B          : in std_logic;

                Emu_SW05_B          : in std_logic;

                Emu_SW06_B          : in std_logic;


                --Phase C Inputs

                Emu_SW01_C          : in std_logic;

                Emu_SW02_C          : in std_logic;

                Emu_SW03_C          : in std_logic;

                Emu_SW04_C          : in std_logic;

                Emu_SW05_C          : in std_logic;

                Emu_SW06_C          : in std_logic;


                Error : in STD_LOGIC;

                HP_EN : in STD_LOGIC

        );

END Digital_Twin_Emulation_Control;


ARCHITECTURE Behavioral OF Digital_Twin_Emulation_Control 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,S45,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, NS_Fsw, CSab, NSab, CSbc, NSbc, CSca, NSca : state_type;

------------------- Other Signals -------------------
signal EN : std_logic := '0'; -- Enable

--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');


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

-----------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');

        ---------------------------- 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';


        -- Scale Ref Register Used for Scale Counter (Latched from PreScale Reg)

        constant           Scale_Ref                   : std_logic_vector (15 downto 0) := X"09C4"; --
Scale: 09C4h = 2500 clock cycles. Every 2500 clock cycles, one sample will be collected from
the DSP signals

        constant           numberOfSamples      : std_logic_vector (7 downto 0)          := X"C0";
-- From each phase, 192(C0h) samples will be collected



        -- 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 (15 downto 0) := (others => '0');

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


        signal   LD_Vb_Samp                       : std_logic := '0';

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

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

```
        signal   LD_Vc_Samp                 : std_logic := '0';
        signal Temp_Vc_Samp                 : std_logic_vector (15 downto 0) := (others => '0');
        signal          Vc_Samp                     : std_logic_vector (15 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';



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


    -- 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;


-- 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 --------------------------------------------------------- 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
);

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

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

```
-- 192 FIFO Reg Counter to Save Emu Data
Counter_LeadReg: Std_Counter
generic map
(
        Width => 8
)
port map(
        INC     => Cnt_LeadReg_INC,
        rst     => Cnt_LeadReg_Rst,
        clk     => clk,
        Count   => Cnt_LeadReg_Out
        );

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


-- 192 Reg Counter to Save Emu Data from FIFO to RAM
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

            Van_DC <= (others => '0');

            Vbn_DC <= (others => '0');

            Vcn_DC <= (others => '0');


            --Data Distribution

            Vrble_Data<= (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';


            EN <= '0';


        ELSE

            -- Load the data every rising edge.

            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;


                        --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_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;

        END IF;
    END PROCESS;
    ------------------- END Registers -------------------



    Va_Duty_Cycle: PROCESS(CSA, EN, Emu_SW01_A, Emu_SW02_A, Emu_SW03_A,
Emu_SW04_A)
    BEGIN
        LD_Van_DC <= '0';
        Temp_Van_DC <= (others => '0');



        case CSA is
            when S0 =>

                IF (EN <= '0')THEN
                    NSA<=S0;
                ELSE
                    NSA<=S1;
                END IF;
```

```
when S1 =>     -- 12V Offset (24<->0) instead of 12<->-12 range

        IF (Emu_SW01_A = '1') THEN -- Positive cycle

                IF (Emu_SW02_A = '1') THEN

                        Temp_Van_DC <= X"64"; -- 24V

                ELSE

                        Temp_Van_DC <= X"32"; -- 12v

                END IF;


        ELSIF (Emu_SW04_A = '1') THEN -- Negative cycle

                IF (Emu_SW03_A = '1') THEN

                        Temp_Van_DC <= X"00"; -- 0V


                ELSE

                        Temp_Van_DC <= x"32"; -- 12V


                END IF;

        ELSE

                Temp_Van_DC <= X"32"; -- 12v

        END IF;

        NSA <= S2;

        LD_Van_DC <= '1';
```

```
            when S2 =>
                NSA <= S1;

            when others=>
                NSA <= S0;

        END case;
    END PROCESS;



    ------------------- Calculate Duty Cycle % of Phase B -------------------
    Vb_Duty_Cycle: PROCESS(CSB, EN, Emu_SW01_B, Emu_SW02_B, Emu_SW03_B, Emu_SW04_B)
    BEGIN
        LD_Vbn_DC <= '0';
        Temp_Vbn_DC <= (others => '0');


        case CSB is
            when S0 =>
    IF (EN <= '0')THEN
        NSB<=S0;
    ELSE
        NSB<=S1;
    END IF;

when S1 =>    -- 12V Offset (24<->0) instead of 12<->-12 range
    IF (Emu_SW01_B = '1') THEN -- Positive cycle
        IF (Emu_SW02_B = '1') THEN
            Temp_Vbn_DC <= X"64"; -- 24V
        ELSE
            Temp_Vbn_DC <= X"32"; -- 12v
        END IF;

    ELSIF (Emu_SW04_B = '1') THEN -- Negative cycle
        IF (Emu_SW03_B = '1') THEN
            Temp_Vbn_DC <= X"00"; -- 0V

        ELSE
            Temp_Vbn_DC <= x"32"; -- 12V

        END IF;
    ELSE
```

```
                    Temp_Vbn_DC <= X"32"; -- 12v
                END IF;
                NSB <= S2;
                LD_Vbn_DC <= '1';


            when S2 => -- Refresh
                NSB <= S1;

            when others=>
                NSB <= S0;

        END case;
    END PROCESS;

    ------------------- Phase C -------------------
    Vc_Duty_Cycle: PROCESS(CSC, EN, Emu_SW01_C, Emu_SW02_C, Emu_SW03_C,
Emu_SW04_C)
    BEGIN
        LD_Vcn_DC <= '0';
        Temp_Vcn_DC <= (others => '0');

        case CSC is
```

```
when S0 =>

        IF (EN <= '0')THEN
                NSC<=S0;
        ELSE
                NSC<=S1;
        END IF;

when S1 =>      -- 12V Offset (24<->0) instead of 12<->-12 range
        IF (Emu_SW01_C = '1') THEN -- Positive cycle
                IF (Emu_SW02_C = '1') THEN
                        Temp_Vcn_DC <= X"64"; -- 24V
                ELSE
                        Temp_Vcn_DC <= X"32"; -- 12v
                END IF;

        ELSIF (Emu_SW04_C = '1') THEN -- Negative cycle
                IF (Emu_SW03_C = '1') THEN
                        Temp_Vcn_DC <= X"00"; -- 0V

                ELSE
                        Temp_Vcn_DC <= x"32"; -- 12V
                        END IF;

                    ELSE

                        Temp_Vcn_DC <= X"32"; -- 12v

                    END IF;

                    NSC <= S2;

                    LD_Vcn_DC <= '1';


                when S2 => -- Refresh

                    NSC <= S1;


                when others=>

                    NSC <= S0;


            END case;

        END PROCESS;


        ------------------------------------- Emulation Data Traffic -------------------------------------


        Emu_Data_Traffic : PROCESS(CS, CntDelay_Out, CntBus_Out, Bus_Int1_Busy,

    Bus_Int1_DataOut, Vrble_Data, Error, Emu_DL_Start, HP_EN, EN, Cnt_LeadReg_Out,

    Cnt_Scale_Out, Van_DC, Vbn_DC, Vcn_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';

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';         -- Reset Number of Samples
                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';
                Temp_EN <= '0';
                LD_EN <= '1';
```

```
          NS <= S1;


     when S1=>                                   -- Delay

          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 =>                                  -- Wait for Bus

Control

          if(Bus_Int1_Busy = '1') THEN

               NS <= S3;

          else
```

```
                    NS <=S4;
                END if;
                CntBus_Rst <='0';                      -- Reset Bus Counter

            when S4 =>                                     -- Request if the
Emulation button was pressed
                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 =>                                     -- Wait for Bus
Control
                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 =>                                     -- Store the register
value into Emulation Datalogger Start variable
```

```
                Temp_Emu_DL_Start <= Vrble_Data(0);

                LD_Emu_DL_Start <= '1';

                NS <= S7;


            when S7 =>                          -- Check if EMU Start

is pressed

                if(Emu_DL_Start = '1') THEN

                    NS <= S8;

                else

                    NS <= S0;                   -- If not, go back to

S0

                end if;


            when S8 =>  -- Check errors

                if(Error = '1') THEN

                    Temp_Emu_DL_Start <= '0';

                    LD_Emu_DL_Start <= '1';

                    NS <= S9;

                ELSE

                    NS <= S16;

                END if;
```

```
    ------------------------------------ Start ERROR Procedure --------------------------------------------
-
            when S9 => -- Wait bus
                if(Bus_Int1_Busy = '1') THEN
                    NS <= S9;
                else
                    NS <=S10;
                END if;

            when S10 =>  -- Set DL status to Error
                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 <= S11;

            when S11 => -- Wait bus
                if(Bus_Int1_Busy = '1') THEN
                    NS <= S11;
                else
                    NS <=S12;
                END if;
            when S12 =>             -- Overwrite the DL Start command
                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 <= S13;

            when S13 =>   -- Check if Error is still ON. Wait until the error is off
                IF (Error = '1') THEN
                    NS <= S13;
                else
                    NS <= S14;
                END if;

            when S14 => -- Wait bus
                if(Bus_Int1_Busy = '1') THEN
                    NS <= S14;
                else
                    NS <=S15;
                END if;

            when S15 => -- Set DL status to done and go back to S0
```

```
                                        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 ----------------------------------------
--
------------------------------------------ Start Emu Data Logging -------------------------------------------
                              when S16 =>                                             -- Wait for Bus
Control
                                        if(Bus_Int1_Busy = '1') THEN
                                                  NS <= S16;
                                        else
                                                  NS <=S17;
                                        END if;
                              when S17 =>                                             -- Set DL Status to
Busy and Enable emulation
                                        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
                                        Bus_Int1_WE <='1';
                                        Temp_EN <= '1';                               -- Enable Emulation
                                        LD_EN <= '1';
```

```
                        NS <= S18;

                when S18=>                                  -- Check if there's no
error and if HP is enabled
                        if((Error = '0') and (HP_EN = '0'))THEN
                                NS <= S19;
                        else
                                NS <= S9;                   -- If there's an error,
go back to error process (S9)
                        END if;

        -------------------------- Collect sample and save into FIFO loop ------------------------------
-
                when S19 =>
                        IF (Cnt_LeadReg_Out < numberOfSamples) THEN -- Check if all
samples were collected
                                NS <= S20;
                        else
                                Cnt_Scale_Rst <= '0';
                                NS <= S24;
                        END if;

                when S20 =>
                if(Cnt_Scale_Out < Scale_Ref)THEN -- Wait for resolution (2500
clock cycles)
                    Cnt_Scale_INC <= '1';
                    NS <= S20;
                else
                    NS <= S21;

                END if;

            when S21 =>
                Cnt_Scale_Rst <= '0';                    -- Reset resolution counter
                Temp_Va_Samp <= X"00" & Van_DC;      -- Load values of each
phase
                Temp_Vb_Samp <= X"00" & Vbn_DC;
                Temp_Vc_Samp <= X"00" & Vcn_DC;
                LD_Va_Samp <= '1';
                LD_Vb_Samp <= '1';
                LD_Vc_Samp <= '1';
                NS <= S22;

            when S22 =>
                Cnt_LeadReg_INC <= '1'; -- Count 1 sample collected
                NS <= S23;
            -- Start Saving Emu Va, Vb, Vc Data in FIFO--

            when S23 =>

                IF (STD_FIFO_Va_Full = '0') THEN

                    STD_FIFO_Va_DataIn <= Va_Samp;              --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;              --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;              --16 bit
FIFO. DATA_WIDTH in FIFO must be 16 and not 8.

                    STD_FIFO_Vc_WriteEn <='1';

                END if;

                NS <= S19 ;
            when S24 =>                             -- Wait for Bus
Control
                if(Bus_Int1_Busy = '1') THEN
                    NS <= S24;
                else
                    NS <=S25;
                END if;

            when S25 =>                             -- Update
Emu_DL_Status
                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 <= S26;

            when S26 =>
                if(Cnt_FollowReg_Out < numberOfSamples)THEN -- X"C0" =
192
                    NS <= S27;
                else
                    Cnt_FollowReg_Rst <= '0';
                    NS <= S40;
                END if;

    --------------------------- Saving Emu Va, Vb, Vc Data from FIFO to RAM -------------------
--------

            -- Va FIFO to RAM
            when S27 =>
                if(STD_FIFO_Va_Empty = '1') THEN -- Check if FIFO is empty.
if true, check Vb FIFO
                    NS<=S31;
                else
                    STD_FIFO_Va_ReadEn <= '1';          -- Read data from
FIFO
                    NS<=S28;
                END if;

            when S28=>                                          -- Load FIFO data

                Temp_Vrble_Data <= STD_FIFO_Va_DataOut;
                LD_Vrble_Data <='1';
                NS<=S29;
            when S29=>                                      --Wait for Bus
Control
                if(Bus_Int1_Busy = '1') THEN
                    NS <= S29;
                else
                    NS <=S30;
                END if;

            when S30=>                                      -- Send data to RAM
                Bus_Int1_AddrIn <= Addr_Va_Start + Cnt_FollowReg_Out;
    --SEND Va data to RAM Addr X"0200" + Counter[1:192]
                Bus_Int1_DataIn <= Vrble_Data;
                Bus_Int1_WE <='1';
                NS<=S31;

            -- Vb FIFO to RAM
            when S31 =>
                if(STD_FIFO_Vb_Empty = '1') THEN  -- Check if FIFO is empty.
if true, check Vc FIFO
                    NS <= S35;
                else
                    STD_FIFO_Vb_ReadEn <= '1';      -- Read data from
FIFO
```

```
                    NS <= S32;
                END if;

            when S32 =>                                     -- Load data from
FIFO
                Temp_Vrble_Data <= STD_FIFO_Vb_DataOut;
                LD_Vrble_Data <='1';
                NS <= S33;

            when S33 =>                                     -- Wait for Bus
Control
                if(Bus_Int1_Busy = '1') THEN
                    NS <= S33;
                else
                    NS <= S34;
                END if;

            when S34 =>                                     -- Send data to RAM
                Bus_Int1_AddrIn <= Addr_Vb_Start + Cnt_FollowReg_Out;
    --Send Vb data to RAM Addr X"0300" + Counter[1:192]
                Bus_Int1_DataIn <= Vrble_Data;
                Bus_Int1_WE <='1';
                NS <= S35;
            -- Vc FIFO to RAM
            when S35 =>
                if(STD_FIFO_Vc_Empty = '1') THEN -- Check if FIFO is empty.
If true go back to S26
                    NS <= S26;
                else
                    STD_FIFO_Vc_ReadEn <= '1';          -- Read data from
FIFO
                    NS <= S36;
                END if;

            when S36=>                              -- Load data from
FIFO
                Temp_Vrble_Data <= STD_FIFO_Vc_DataOut;
                LD_Vrble_Data <= '1';
                NS <= S37;

            when S37 =>                             -- Wait for Bus
Control
                if(Bus_Int1_Busy = '1') THEN
                    NS <= S37;
                else
```

```
                    NS <= S38;
                END if;

            when S38=>                          -- Send data to RAM
                Bus_Int1_AddrIn <= Addr_Vc_Start + Cnt_FollowReg_Out;
    --Send Vc data to RAM Addr X"0300" + Counter[1:192]
                Bus_Int1_DataIn <= Vrble_Data;
                Bus_Int1_WE <='1';
                NS <= S39;

            when S39 =>
                Cnt_FollowReg_INC <= '1';   -- Count 1 sample of each phase and
go back to S26
                NS <= S26;

            --------------------------- End Saving Emu Va, Vb, Vc Data from FIFO to
RAM --------------------------------

            --------------------------- Finalizing --------------------------------
            when S40 =>                         -- Wait for Bus
Control
                if(Bus_Int1_Busy = '1') THEN
                    NS <= S40;
                else
                    NS <= S41;
                END if;

            when S41 =>                                 -- Change register
status to Ready/Done
                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 <= S42;

            when S42 =>                                 -- Wait for Bus
Control
                if(Bus_Int1_Busy = '1') THEN
                    NS <= S42;
                else
                    NS <=S43;
                END if;

            when S43 =>                                 -- Reset the DL start
register and go back to S0
```

```
                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';
                Temp_Emu_DL_Start <= '0';
                LD_Emu_DL_Start <= '1';
                NS <= S0;



            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;
            CSA     <= S0;
            CSB     <= S0;
            CSC     <= S0;
```

```
                CSab   <= S0;

                CSbc   <= S0;

                CSca   <= S0;

            else

                CS                  <= NS;

                CSA    <= NSA;

                CSB    <= NSB;

                CSC    <= NSC;

                CSab   <= NSab;

                CSbc   <= NSbc;

                CSca   <= NSca;

            END if;

        END PROCESS;

        ----END State Sync

END Behavioral;
```