A RF generator based on a FPGA system

BEIHANG University of Aeronautics and Astronautics.
北京航空航天大学。
Xueyuan Road 37#, Haidian District, Beijing.

Thesis for Industrial Engineering in Barcelona (ETSEIB)
Date: 2012-06-19

Eduard Puertas Molina
With professor Mr. Zhao Jingcheng and doctor Mr. Tian Jinjun.
With the collaboration of:

- Doctor Mr. Zhaojingcheng. Lecturer in Electromagnetic field and microwave technology, Supervisor of postgraduate.
  赵京城博士，电磁场与微波技术讲师，硕士生导师。

- Doctor Mr. Tian Jinjun. Doctor of Electromagnetic field and microwave technology, postdoc of Transportation Engineering.
  田进军博士，电磁场与微波技术专业博士，交通运输工程专业博士后。
Contents

Acknowledgments ........................................................................................................... 7
Abstract ............................................................................................................................ 8

1 Introduction .................................................................................................................. 9
  1.1 Motivation .................................................................................................................. 10
  1.2 Project Description .................................................................................................. 11
  1.3 Organization ............................................................................................................ 12

2 Background and previous work ............................................................................... 13
  2.1 FPGA-based Control Systems ............................................................................... 13
    2.1.1 Hardware Implementation Challenges .............................................................. 13
    2.1.2 Advantages of FPGA-based Control Systems ............................................... 14
    2.1.3 Applications of FPGA-based Control Systems .............................................. 16
  2.2 FPGA and CAD Flow ............................................................................................ 17

3 System hardware ......................................................................................................... 23
  3.1 Architecture of FPGAs .......................................................................................... 23
    3.1.1 Configurable Logic Block ............................................................................... 23
    3.1.2 FPGA routing .................................................................................................. 29
    3.1.3 I/O blocks ....................................................................................................... 31
  3.2 FPGA programming technologies .......................................................................... 33
  3.3 Main board ............................................................................................................. 36
    3.3.1 PCB design ...................................................................................................... 36
    3.3.2 Power Supply .................................................................................................. 37
    3.3.3 Safety Switch .................................................................................................. 43
    3.3.4 I/O interfaces .................................................................................................. 46

4 RF Implementation ...................................................................................................... 47
  4.1 System Architecture .............................................................................................. 47
  4.2 System Blocks ........................................................................................................ 53
    4.2.1 Por_gen ............................................................................................................ 54
    4.2.2 pll ..................................................................................................................... 54
    4.2.3 RF_gen ............................................................................................................ 56
    4.2.4 Mcu_param_reg ............................................................................................. 58
    4.2.5 Decoder_16 .................................................................................................... 61

4.3 Simulation Results .................................................................................................. 61
  4.4 System Architecture Synthesis ............................................................................. 65
  4.5 Summary ................................................................................................................ 67

5 Prototype Testing ........................................................................................................ 68

6 Conclusions .................................................................................................................. 72

7 Appendix: source code listing .................................................................................... 73
  7.1 Top.v ....................................................................................................................... 74
  7.2 top_tb.v .................................................................................................................. 77
  7.3 mcu_param_reg.v .................................................................................................. 82
  7.4 por_gen.v .............................................................................................................. 85
  7.5 read.v .................................................................................................................... 86
  7.6 RF_gen.v ................................................................................................................. 87
  7.7 trigger_gen.v ......................................................................................................... 88
  7.8 wr_rd.v .................................................................................................................. 90
  7.9 decoder_16.v ....................................................................................................... 91
7.10 decoder_32.v ................................................................. 92
7.11 decoder_64.v ................................................................. 93
7.12 frequency_sweep.v ....................................................... 94

8 References .............................................................................. 97
GLOSSARY

FPGA  Field Programmable Gate Array
SRAM  Static Random Access Memory
RF    Radio Frequency
SCC   System Control Complexity
CPU   Central Processing Unit
HM    Hardware Module
CAD   Computer-Aided Design
TTM   Time To Market
DoM   Degree of Modifiability
UCA   Universal Communication Abstraction
IP    Intellectual Property
I/O   Input/Output
FLC   Fuzzy Logic Control
CLB   Configurable Logic Blocks
LUT   Look-Up Table
BRAM  Block Random Access Memory
DPR   Dynamic Partial Reconfiguration
HDL   Hardware Description Language
PI    Packet-In
RAM   Random Access Memory
DC    Direct Current
DSP   Digital Signal Processor
INS   Inertial Navigation System
IP    Intellectual Property
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>MOSFET</td>
<td>Metal-Oxide-Semiconductor Field-Effect Transistor</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional-Integral-Derivative</td>
</tr>
<tr>
<td>POR</td>
<td>Power-on-Reset</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>LAB</td>
<td>Logic Array Block</td>
</tr>
<tr>
<td>LE</td>
<td>Logic Element</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>ALM</td>
<td>Adaptive Logic Module</td>
</tr>
<tr>
<td>ALUT</td>
<td>Adaptive Lookup table</td>
</tr>
<tr>
<td>PLL</td>
<td>Phase-Locked Loops</td>
</tr>
</tbody>
</table>
Acknowledgments

A good number of individuals have assisted me in a variety of ways in the preparation of this work. Firstly, I would like to thank the support of Professor Zhao Jingcheng and the assistance of Doctor Tian Jinjun in the improvement of this thesis, because without them I couldn’t have carried out this assignment. Secondly, I would like to appreciate the effort of some institutions as Beihang University of Aeronautics and Astronautics, or Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona (ETSETB) because they brought me the chance to work out in Beijing (China).

In particular, I would like to thank Eduard Martinez and Juan Serra for their friendship and assistance over the past few years. Finally, I am especially grateful to my family for their understanding, support, and encouragement during the time it took to complete my graduate studies.
Abstract

This work is part of ongoing research conducted at Beihang University relating to signal generating. The primary objective of this thesis was to develop a flexible, high-performance Radio Frequency (RF) generator platform in order to facilitate research on channel response. A FPGA-based system architecture utilizing a stacked, multi-board design was created to meet this goal. A combination of a Liquid-Crystal Display (LCD), power supply, FPGA control and RF generator were employed in the new system.
1 Introduction

Field Programmable Gate Arrays (FPGA), using Static Random Access Memory (SRAM) technology provide designers with a programmable fabric that can be configured to implement a specific hardware circuit. This technology provides designers the ability to reconfigure the entire FPGA as an application’s hardware requirements change. We can see the utility of this technology in the fields of wired and wireless communication [1], image and signal processing [2][3], medical equipment [4], robotics [5], and automotive applications [6] to name a few. Also, FPGAs have been used to implement control system applications such as analog process control [7], power converters [8], and state-machine-based control systems [9].

![Diagram of a Simple MIMO control system.](image)

Figure 1 Simple MIMO control system.

In many respects, we may consider an FPGA with the functionality of a MIMO (multiple-input and multiple-output) control system as illustrated in Figure 1. The controller continuously samples the multiple input modules (e.g., sensors, push-buttons), and based on the information received, makes decisions to activate one or more of the output modules (e.g., actuators,
motors). The controller itself can be implemented with a CPU and embedded software or with dedicated hardware. Each implementation strategy offers advantages and disadvantages that must be carefully considered during the design phase of the project.

While a hardware implementation offers unique benefits over software such as higher data throughput, reliability, and concurrency, hardware systems are inherently more difficult to design and implement. Some of the issues facing hardware designers include:

- Challenges of spatial and concurrent programming models,
- Incremental changes that impact timing and spatial constraints,
- Prolonged synthesis time required by the Computer-Aided-Design (CAD) tools.

These issues become particularly prevalent when we look to utilize FPGAs in control system environments as opposed to traditional uses of FPGAs in high performance computing.

1.1 Motivation

Most previous work has focused on increasing the quality of FPGA designs, specifically, improving the CAD flow to optimize device architecture and increasing the logical resources available to the developers [10-13]. Advances made in these areas have allowed FPGAs to accommodate sophisticated applications, and even complete Systems-on-Chip (SoCs) with smaller footprints, operating at a higher frequency and consuming less power.
than their predecessors [14-16]. However, the increased capability of hardware has brought with it an exponential increase in complexity and engineering cost. There is also a steep learning curve for developers to implement their initial hardware design, or to update their previous designs for the next generation product.

The accessibility of hardware design has remained a critical issue for developers wanting to leverage the benefits of hardware implementation in computing platforms. The complexity and the prolonged design time often impede their use in development of the application. In these cases, an increase in productivity may outweigh the importance of an optimized circuit implementation. Certain applications, such as FPGA-based control systems, often do not require the underlying FPGA to operate at its maximum frequency. In many instances, the loop speed of the control system is limited by the input and output modules rather than the processing unit. Relaxing the timing constraint enables the CAD tools to spend less time optimizing the circuit, and thus, can generate the FPGA configuration bitstream in less time. Lanvin et al. [17] used hard-macros for rapid prototyping to greatly accelerate FPGA compilation time at the expense of reduced peak performance. The time saving can be significant given that the hardware design may be synthesized hundreds of times during its design cycle, which can lead to a faster Time-to-Market (TTM).

1.2 Project Description
To automate the measures have created a control RF generator. For this we have developed a platform based on an LCD, control of the entire system, based on a FPGA Cyclone, the power supply and the RF generator.

This project has worked with VHDL FPGA programming to create an interface between it and the rest of the system. Moreover a GUI to program the LCD. This project can serve to sweeps frequencies and study the impulse responses of the various channels of communication. To be able to study and then find the appropriate signals for maximum efficiency in terms of power, bit error, and performance.

1.3 Organization

This thesis is organized into 7 chapters. Chapter 2 provides the necessary background information for this thesis. Specifically, topics discussed include hardware implementation challenges, FPGA-based Control Systems and previous work to facilitate its development. Chapter 3 presents a detailed discussion of the physical hardware developed for the RF generator.

Chapter 4 describes the system architecture of the proposed RF controller and its FPGA implementation. Section 4.1 introduces the system architecture while section 4.2 describes the blocks of the architecture. Additional findings regarding the testing of the physical prototype used to implement the RF generator are in chapter 5. A quantitative analysis of the RF generator and it prototype testing is given in Chapter 6. Finally, Chapter 7 concludes this thesis by providing a discussion on the direction and impact of the research, and summarizes future works to be done.
2 Background and previous work

This thesis has a main area for contribution; a system framework for control a RF system. The following provides a background on FPGA control systems and previous work in system architecture.

2.1 FPGA-based Control Systems

This section provides background information on hardware implementation challenges. Also discussed are the advantages and applications of FPGA-based control systems.

2.1.1 Hardware Implementation Challenges

A hardware-based solution provides unique advantages that a processor-based solution cannot offer. However, hardware designs are inherently more complex and require much more engineering effort. Hardware design methodologies have not been able to keep pace with the increased hardware design complexity. Some of the challenges include:

System Control Complexity (SCC) – Issues such as concurrency and synchronicity between competing software components being for systems resources are usually coordinated through a central controller such as a CPU. However, hardware design offers the concurrency and therefore lacks a dedicated central controller that arbitrates competing requests. For these reasons, the system control complexity of hardware designs is much higher than of software designs.

Degree of Modifiability (DoM) - Incremental changes in software only result in
a simple shifting or reordering of binary words in memory that can be easily accommodated. An incremental change in hardware logic may result in new spatial planning of the previously placed HMs and rerouting of the nets that connect them. This could in turn lead to spatial and latency issues that are not present before the change, thus further complicating the design.

Universal Communication Abstraction (UCA) - In addition to being physically connected, modules must share complementing interfaces to facilitate their communication. Unlike software where there is a commonly used universal communication abstractions (i.e. stack), hardware developers are often required to implement unique interfaces to facilitate the communication between modules.

2.1.2 Advantages of FPGA-based Control Systems

FPGAs have grown in size from their initial applications as hardware glue logic and interface chips [18]. Using an FPGA as part of the implementation platform allows the control system to leverage its many benefits. In closed-loop control systems, the primary factor affecting overall system performance is often attributed to the speed of the control loop [19]. The loop speed can be simply defined as the total time needed to read sensor inputs, process the control algorithm and output the resulting values to the actuators. An FPGA-based solution has the advantage of hardware concurrency such that independent control loops can run at different rates without relying on shared resources that might slow down their responsiveness. On the other hand, a processor-based solution with multiple control loops must compete for processor bandwidth and relies on context switching. Different parts of the
application may generate unwanted delays and induce jitter into time-critical tasks. As system requirements become increasingly complex, FPGAs also provide the scalability to add functionality without having a significant impact on the remainder of the system. For example, a machine controlled application may require additional vibration sensors to monitor early-stage machine failures. This functionality can be straightforwardly added to a FPGA-based control system without affecting the machine control module. Available IP blocks for control systems can also be utilized to help developers facilitate their designs. Whether it is a simple PID controller or complex algorithm implementation (e.g., Model-Free Adaptive control), pre-verified IP blocks simplify the design process and can provide efficient means to easily implement the desired operation.

Typically, in a control application, the parts and requirements of the actual system being controlled are likely to change over time. The controller must adapt to these changes. An FPGA-based solution provides complete hardware re-configurability that could be used to adapt to system needs in the field. The entire FPGA can be reconfigured to accommodate new and improved algorithms, different types of I/Os, firmware upgrades, and bug fixes. Avoiding a complete hardware redesign can reduce long-term engineering costs and system downtime. The overall flexibility and performance of FPGAs, combined with an extensive set of pre-verified IPs, provides a spectrum of solutions for a wide range of industries such as embedded designs, applications, and certainly control systems.
2.1.3 Applications of FPGA-based Control Systems

FPGAs have been widely used in control systems mainly because of their consistent performance, while at the same time reducing the time and cost of implementation. In particular, FPGAs have been used in control applications such as pulse-width modulation inverters [20, 21], power-factor correction [22], multi-level and matrix converters [23, 24], and soft switching [25,26]. FPGAs have also been incorporated into electrical machine applications such as induction machine drive [27, 28], motion control [29], neural-network control of induction motors [30], speed measurements [31], and Fuzzy Logic Control (FLC) of power generators [32, 33].

Poorani et al. [34] outlined a novel approach to implement FLC for speed control of an electric vehicle using a FPGA. Parameters such as acceleration, braking, energy status, gear and terrain are all considered to estimate the variation of the motor speed. A FPGA-based FLC that combines fuzzy logic and sliding mode control is proposed in Lin et al. [35] to control the mover position of a linear induction motor drive to compensate for uncertainties such as friction. Kim [36] presented an implementation of an FLC on a reconfigurable FPGA system.

Sercos [37] reports new Ethernet-based FPGA controllers for motion control. The cores of the controllers are based on the low-cost Spartan-3 Xilinx FPGA platform. The controllers include all hardware functionality such as timing, synchronization, and processing of cyclic and non-cyclic data on the basis of two integrated Ethernet MACs. Chapuis et al. [38] proposed a FPGA-based...
quasi-analogue Digital Torque Controller, where the torque regulation can be updated in as little as 2μs. Fratta et al. [39] presented an ideal PWM ripple filter that was obtained by oversampling the measurements of the currents, then by computing the average value inside a time-sliding window, which reintroduced to the current loop. Significant improvements were made when compared to an alternative DSP implementation.

2.2 FPGA and CAD Flow

FPGAs are programmable semiconductor devices developed in the mid-1980s based around a matrix of Configurable Logic Blocks (CLBs) connected via programmable switch blocks [40]. Unlike ASICs where a device is custom built for the specific design, SRAM-based FPGAs can be configured to implement the desired circuits. Figure 2 shows the major components of a modern day FPGA.

![Figure 2 Major components of a modern FPGA.](image-url)
The CLB is the basic logic unit in an FPGA. The exact numbers and features can vary between FPGAs, but every CLB consists of a Look-Up Table (LUT) with multiple inputs, some selection circuitry and flip flops. The CLBs can be configured to implement combinatorial and sequential logic, shift registers, or RAM. Figure 3 shows a simplified 6-input CLB.

While the CLBs provide logical implementation, flexible interconnect provides the signaling between the CLBs and the I/Os. Part of this flexible interconnect is the routing matrix shown in Figure 2. To provide efficient signaling, different length routing wires are available to route signals to adjacent CLBs, or across the FPGA. Actual routing decisions are usually made by the CAD tools unless specified by the designer. Embedded blocks such as Block RAM (BRAM), Multipliers, and DSPs are available to the designers to provide specialized functionality such as convolution and digital filtering. These pre-placed IPs are efficient alternatives to implementing resource-intensive modules using generic CLBs.

FPGA CAD tools provide a process to describe hardware designs typically
using a Hardware Description Language (HDL) such as VHDL or Verilog. The source code is ultimately translated into a program bitstream. The bitstream contains logical information of the programmable fabric and is loaded onto the FPGA to configure the desired operations. The entire process can be divided into five major steps: high-level synthesis, technology mapping, placement, routing, and bitstream generation. Figure 4 provides an overview of the processes.

The high-level synthesis step outlined in Figure 4 looks for synthesizable constructs from the HDLs and converts them into equivalent circuits made of primitive hardware components such as flip-flops, combinatorial logic, or a combination of the two. The final product of high-level synthesis is a netlist file. It contains a list of all the instances of primitive components in the converted circuit and a description of how they are connected. The netlist effectively translates the description of the hardware circuit from HDLs into a form that other steps in the CAD flow can understand.

Technology mapping [41] involves allocating the netlist of primitive logic gates onto the technology available on the target device. Families of FPGAs may contain different types of resources and thus the CAD tool must appropriately map the logic gates onto the resources available on the target FPGA. For example, the Xilinx Virtex5 FPGA is comprised of 6-inputs LUTs, while the Virtex4 FPGAs uses 4-input LUTs as the basis of implementation. This difference greatly affects a specific hardware circuit is to be implemented on different target devices. Another notable difference between FPGA
architectures that could impact technology mapping is the number and availability of hard cores such as BRAM and DSP blocks. Also, in most cases, constraints such as minimal delay, power, and/or area are used to guide the CAD tool to an acceptable solution.

```
entity multiplier is
  port( num1, num2: in std_logic_vector(1 downto 0);
        product: out std_logic_vector(3 downto 0));
end multiplier;
```

Figure 4 Simplified high-level synthesis of FPGA CAD tool flow. The placement step involves “placing” the set of technology mapped components at specific locations on the FPGA. This is inherently an NP-
complete (Non-deterministic Polynomial-time) problem. A NP-complete problem implies that the optimal answer cannot be guaranteed within a reasonable amount of time. Approximate solutions for NP-complete problems can be found using heuristic methods. Popular placement techniques are Partitioning-based [42], Quadratic [43], Hybrid and Hierarchical [44, 45], and Simulated Annealing (SA) [46].

The goal of the routing step is to efficiently connect the placed logic given a set of constraints such as timing. Routing consumes most of the chip area and is responsible for most of the circuit delay. Ideally, the shortest possible path between the placed logics is desired. However, this will cause congestion if many are placed close together and the switch block has insufficient resources. The switch block routes each incoming track to a number of outgoing tracks. A fully-connected (all inputs can be connected every output) switch box is ideal. However, since the majority of area on a FPGA is designated for routing, fully-connected switch blocks requiring too much area would have an adverse affect on the achievable logic density of the FPGA.

The bitstream generation step involves the translation of the placed and routed circuits into the necessary configuration bits to activate and deactivate the appropriate switches and LUT values that will implement this circuit. The bitstreams are device specific, and there is a one-to-one mapping between the final circuit and its bitstream. There are different methods to configure (i.e. download) the bitstream onto the target FPGA. JTAG cables
can connect from the PC’s parallel or USB interface to the FPGA development board. Xilinx uses the proprietary Platform Cable USB to download the bitstreams, while Altera uses their USB-Blaster. Alternative methods to configure the FPGA include loading the bitstreams from off-chip storage devices such as compact, linear, platform and SPI flash.
3 System hardware

This chapter will present a detailed discussion of the physical hardware developed for the RF generator, along with the relevant design constraints and considerations. This will include information on the specific FPGA-based processor modules and the custom multi-board architecture utilized in this design. Implementation details will be covered, including PCB layout, power regulation, signal conditioning, and the various control and measurement interfaces. Information on the specific sensors and communications hardware used will also be provided. Analysis of potential system failure modes and any built-in fail-safe measures will be discussed where applicable.

3.1 Architecture of FPGAs

FPGAs can differ in size of the chip, in the speed and in how much energy they consume. However, these devices always have three main blocks in common: configurable logic blocks, input/output block and programmable interconnect.

The following section is a more detailed explanation of these blocks. An FPGA produced by Altera was chosen for this thesis purposes; hence, the information below is based on Altera’s FPGAs architecture.

3.1.1 Configurable Logic Block

Logic Blocks are the main components in FPGA’s architecture. They determine the capacity of the device and also are able to obtain any Boolean function from its input data. In an FPGA, the Logic Blocks are arranged into a
grid to reduce the sizes of the chip, for this reason, these blocks are called Logic Array Block (LAB).

Depending on the performance of the FPGA, each LAB may consist from hundreds to thousands of Logic Elements (LEs). Each LE within the same LAB is able to work freely or they can be grouped in twos or fours in order to make a module much more complex.

3.1.1.1 Logic Elements

The LE block carries out most of FPGA’s functionalities. These blocks consist of three main components: lookup table (LUT), carry logic and a programmable register.

Figure 5 shows a diagram example of the three main blocks of an LE.

![Figure 5 Example of a simplified block diagram of a Logic element](image)

1) Lookup Table

LUT is a programmable element with one bit output responsible for carrying out logic. LUTs consist of Static Random Access Memory (SRAM) cells and cascaded multiplexers. Memory cells are used to perform a Truth table,
where, for each possible combination of input values, each memory cell can generate a determined logic value at the output, either a 0 or a 1. Therefore, in an nx1 LUT is possible to implement any logic function of n inputs.

However, it is not recommended to create LUTs with too many inputs because then the SRAM size also increases. Hence, the area of the chip occupied when an LUT is created with a high number of inputs must be taken into account. Relatively large LUT reduces the number of LEs within the FPGA.

Conversely, if LUTs have a small number of inputs, the FPGA would contain many LEs. However, it requires more connections causing high delays induced by the cabling between LEs.

For this reason there is a need to reach a compromise between the area and the speed. Usually, this tradeoff is the use of LUTs with 3 or 4 inputs [47].

As an example, assuming that we want to implement the following function of four inputs:

\[ f = A'B' + ABC'D' + ABCD \]

where A, B, C, D are function inputs

The Figure 6 shows the truth table for the aforementioned function and how a
LUT with 4 inputs operates.

![Figure 6 Example of 4-inputs Look-up table](image)

It is important to mention that the process of storing values on the LUTs is completely transparent for the digital system's designer, because it is realized directly by the FPGA manufacturer's software.

2) Programmable register

Programmable register is the synchronous part of an LE and it can be configured by a user as a flip-flop. As seen in Figure 7, the programmable register is controlled by a synchronous clock and it has also asynchronous control signals generated by other LE.

Moreover, the output signal may involve four possible routes. All four routes are highlighted in Figure 7:
- Brown route: Data is driven out of the LE
- Green route: Data is feed backed into the LUT
- Purple route: Data bypass the LUT. In this case the programmable register is used for data storage or for synchronization.
- Blue route: Data bypass the programmable register in order to perform a combinatorial logic function.

Figure 7 Main signal paths within a Logic Element

3) Carry logic

LAB performs arithmetical sum in order to improve the performance of adders, comparators and logic functions.

The carry logic consists of a carry chain and a register chain. When carry bits reach a carry logic, the components of this block are responsible for providing shortcuts to those bits. The carry bits produced can be addressed to another
LE, or to the device interconnect (Red route in Figure 7). Besides, there is the possibility to bypass the LUT and the carry chain, thereby interconnecting all the registers output belonging to different LEs within the same LAB (Yellow route in Figure 7).

### 3.1.1.2 Adaptive Logic Module

More inputs than an LE can offer are needed to provide the option of programming more complex operations with an FPGA. For this reason, the commercially available boards with the highest performance have an adaptive logic module (ALM) instead LEs. ALMs are quite similar to LEs; however, they provide higher performance logic operations using fewer resources.

The basic structure of the Altera’s ALM is shown in Figure 8.

![Figure 8 Simplified block diagram of an Adaptive Logic Module](image)

The main differences between an ALM (Figure 8) and an LE (Figure 6) are:

1. **Number of output registers**
An LE contains a single output register while an ALM may contain 2 or 4 registers providing more options for chain logic and generate multiple functions.

2. In ALM module appears adders

The function of these adders is to do simple arithmetic operations. In LEs, this kind of operations is taking place in the LUTs; consequently thanks to adders is to considerably reduce processing time while simplifying LUTs architecture.

3. LUT becomes Adaptive LUTs

It can be said that an Adaptive LUT is an LUT that contains more than 4 inputs, nevertheless, beyond this similarity; they have a great advantage over the LUTs. ALUTs inputs can be split as efficiently as possible depending on which task has been entrusted. As an example, an ALUT split up in 2 LUTs of 4 inputs each is shown in Figure 8. Making this division it gets 2 blocks working at the same time at different logic functions.

3.1.2 FPGA routing

The interconnect components within an FPGA include all those elements available to the designer in order to program the routing in the device. All the resources in the chip are connected over interconnect routes to be able to communicate with each other.

Two types of routing channels can be found in an FPGA:
1. **Local interconnect routes**

Local interconnect routes are internal connections within the LABs that are required to ensure communication between LEs. This kind of connection is used for chaining the different logic elements and providing short cuts for the transfer data.

2. **Row and column interconnect**

The different LABs that are included in the FPGA are interconnected through the installation of a structured cabling and switches matrix. These structured cabling and switches matrix are used for rearranging the interconnection between LABs depending on the logic function required. These routing mechanisms are among the space between LAB blocks.

Within an FPGA there are three kinds of connections between LABs: row interconnections, column interconnections and connections that isolate a little number of LABs from the others. The longest interconnection runs along the entire length of the chip. This interconnection has the function of ensuring that the connections provide a minimal delay and distortion to the signals.

Figure 9 below shows all the interconnections that can be found within an FPGA.
3.1.3 I/O blocks

Input/Output blocks or IOB are found around the outer edge and they provide the interface between pins outside of the FPGA and internal logic of IC. That I/O block enables the signals to move between inside and outside of the device. Furthermore, every single IOB controls an external pin and it can be configured by the end user as output, input or bidirectional pin.

Figure 10 depicts a block diagram of the generic structure of one IOB.
The red block, called “input path”, comes into operation when the system finds new data at the input of the FPGA. The purpose of this block is to capture the input data from the device pin and store it in the input register or send the data to the logic device depending on how the I/O block has been programmed.

The green block, called “output path”, comes into operation when the system finds new data ready to go outside the FPGA from the logic array. This block consists of two output registers. These registers can be used also for data storage.
The blue block, called “output enable control”, comes into operation when a bidirectional pin is desired. Its function is the synchronization between inputs and outputs or it can be used for data storage.

Besides the main Logic Arrays Blocks explained further above, the new generation of FPGA devices also has embedded specialized hardware. This extra hardware may include high speed clock blocks such as PLLs (phase-locked loops), embedded multipliers, on-board memory structures of several megabits and high speed transceivers.

To summarize the FPGA architecture, in Figure 11 all the components the device consists of are depicted.

![Figure 11 Components that compose a FPGA](image)

3.2 FPGA programming technologies
Besides logic blocks, it is also very important to know the technology used to establish communication between channels, i.e. how LUT cells have been programmed and how the inputs and the outputs have been assigned.

The most important programming methods are shown below:

- **Antifuse:**
  An FPGA that uses this type of technology can only be programmed once and this method uses an antifuse. The antifuse causes a connection when they are programmed; hence, usually they are open.

  On one hand, it is evident that the devices that use this technology are not reprogrammable. On the other hand this method greatly reduces the size and cost of the device due to they do not have embedded memories.

- **SRAM (StaticRAM):**
  An FPGA that uses this type of technology keeps the configuration of the circuit. That means that the SRAM is used as function generator and to control interconnections between blocks.

  The FPGA uses an automatic programming process for storing the information needed to implement the design created by the user. Only when the circuit is powered-up it is possible store this information into the static memory. Then, these bits of information will be transferred to the logic blocks and interconnections in an FPGA in order to program them. Once the FPGA
is reset all the contents are deleted from the memory. Therefore, an FPGA that uses this type of technology can be programmed as many times as user likes, however their drawback is a large size of the device due to the volume occupied by the RAM.

- **Flash:**

An FPGA based on flash cells combine the main advantages of the Antifuse and SRAM programmed technologies. Its size is much more reduced than SRAM cell dimension but not as reduced as Antifuse size; they are reprogrammable, nevertheless, programming speed is slower than SRAM programming speed; and they are not volatile, consequently they keep their configuration after being powered-off.

The following Table 1 summarizes the main features of each method and allows a quick comparison between them:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Antifuse</th>
<th>SRAM</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reprogrammable</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Reprogramming Speed</td>
<td>--------</td>
<td>Fast</td>
<td>3x Slower than SRAM</td>
</tr>
<tr>
<td>Volatile</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Instant-On</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Security</td>
<td>Very good</td>
<td>Acceptable</td>
<td>Very good</td>
</tr>
<tr>
<td>Size</td>
<td>Very small</td>
<td>Large</td>
<td>Medium</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Cost</td>
<td>The cheapest</td>
<td>The most expensive</td>
<td>Expensive</td>
</tr>
</tbody>
</table>

Table 1 Comparisons between Antifuse, SRAM and Flash programming technologies [48]
3.3 Main board

As mentioned previously, the main board contains the primary components necessary for the RF generator. This includes the main power supply and associated circuitry, safety switch hardware and servo connectors, non-volatile storage, RS-232 serial and digital I/O interfaces, and supporting components such as buffers and level-shifters.

3.3.1 PCB design

The main board PCB is a four layer design measuring 3.8 inches by 2.9 inches. Dedicated ground and power planes are used to offer increased noise immunity and minimize voltage drop due to parasitic impedance. All high speed signals are routed on internal layers to reduce radiated EMI and limit susceptibility to external interference. Trace lengths are also kept as short as possible to prevent signal reflection issues. All bypass capacitors are placed close to the corresponding IC power pins and via-in-pad techniques are used when possible. This minimizes parasitic inductance, increasing decoupling performance at higher frequencies [49]. The actual layer stack-up and board layout are shown in Table 2 and Figure 12 respectively.

<table>
<thead>
<tr>
<th>#</th>
<th>Layer Type</th>
<th>Primary Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Top Layer</td>
<td>Low-Speed Signals, Components &amp; Connectors</td>
</tr>
<tr>
<td>2</td>
<td>Inner Layer</td>
<td>Dedicated Ground Plane</td>
</tr>
<tr>
<td>3</td>
<td>Inner Layer</td>
<td>Dedicated +3.3V Power Plane</td>
</tr>
<tr>
<td>4</td>
<td>Bottom Layer</td>
<td>Low-Speed Signals, Components &amp; Connectors</td>
</tr>
</tbody>
</table>

Tabla 2 Main Board Layer Stack-Up
3.3.2 Power Supply

A simplified diagram of the main board power supply is shown in Figure 13 below. It consists of three main stages: input protection, voltage regulation, and voltage monitoring. The input protection stage serves to prevent power faults from damaging the system. The voltage regulation stage provides the supply voltages needed by the FPGA modules and I/O interfaces on the main board, as well as providing power to the auxiliary board. The voltage monitoring stage ensures that the FPGAs are only active when the supply voltages are stable and within tolerance. These three stages will be covered in the following subsections.
3.3.2.1 Input Protection

The input protection stage provides the system with transient voltage suppression (TVS), current limiting and reverse voltage protection. These features help prevent damage to the system by blocking or limiting the negative effects of external power disturbances. The actual protection circuit is shown below in Figure 14. It consists of a TVS diode (D1), a resettable fuse (F1), and a power MOSFET (Q1).

The TVS diode (D1) used is a bidirectional avalanche diode with a breakdown voltage of 14.4V. Under normal operating conditions, the diode
will exhibit negligible leakage current on the order of a few microamps. However, if the input voltage exceeds ~14.4V, the diode will begin conducting due to avalanche breakdown and shunt the transient current to ground. This will effectively protect the system from momentary voltage spikes and surges by clamping the voltage that appears at the input [50]. Such disturbances can result due to load transients on the main battery or power supply. This includes possible overvoltage surges at power-on due to the combination of parasitic inductance in the power leads and the large low-ESR capacitance at the input of the voltage regulation stage [51].

The resettable fuse (F1) is a polymeric positive temperature coefficient (PPTC) device, commonly referred to as a “PolySwitch”. It is essentially a temperature controlled resistor with a non-linear response. During normal operation the PPTC will exhibit a low on-resistance on the order of 0.01 to 0.1 ohms. If the current through the PPTC exceeds the trip current, the device will heat up and increase in resistance several orders of magnitude, thus limiting the current. It will remain in this state until the fault is removed [52]. When combined with the TVS diode, it can potentially protect the system from a prolonged overvoltage fault. It will also prevent the drawing of excessive current from the battery in the event that the system fails in a low-impedance state.

The power MOSFET (Q1) is a p-channel MOSFET with a low on-resistance of only 0.05 ohms. As shown in Figure 14, the gate is connected to ground, the drain is connected to the FCS battery (via the PolySwitch), and the
source is connected to the input of the voltage regulation stage. This orients the intrinsic body diode in the direction of normal current flow. When the battery is installed correctly, a voltage of one diode drop below the battery voltage will initially appear at the source. This gate-source voltage will quickly enhance the MOSFET, minimizing the drain-source voltage drop. However, if the battery is installed backwards then the body diode will be reverse-biased and block the flow of current. This effectively prevents a reverse-voltage at the input from damaging the system [53].

3.3.2.2 Voltage Regulation

The voltage regulator portion of the power supply consists of four switching regulators, as shown previously in Figure 13. A PTR08060W switching regulator is used at the input to step the unregulated battery voltage down to 5.5V. This 5.5V power rail is then distributed to both the auxiliary board and three PTH04070W switching regulators on the main board. These secondary regulators are used to generate the 1.2V, 2.5V, and 3.3V digital supply voltages needed by the FPGA modules and other main board devices. The unregulated battery voltage and the 3.3 volt supply are also distributed to the auxiliary board.

The PTR08060W and PTH04070W are highly integrated, configurable switching regulator modules from Texas Instruments. Both devices allow the output voltage to be set using a single resistor. They require few external capacitors, which can simplify board layout and minimize the amount of area consumed. The PTR08060W can provide up to 6A of output current, while
the PTH04070W can supply up to 3A. Both regulators feature undervoltage lockout, output short-circuit protection with automatic reset, and achieve typical efficiencies of 80 to 90 percent.

The PTR08060W and PTH04070W devices, like all switching regulators, exhibit two forms of noise at their inputs and outputs: ripple voltage and switching spikes. Ripple voltage occurs at the fundamental switching frequency of the regulator and can generally be attenuated by increasing the input/output capacitance or decreasing capacitor ESR. The switching spikes are much higher frequency in nature (over 100 MHz) and cannot be reduced as easily due to the parasitic inductance (ESL) of the input/output capacitors [54]. This high frequency noise is problematic since it contributes to the EMI generated by the system and can interfere with the sensitive analog circuitry on the auxiliary board. In order to decrease this noise, “PI Filters” were added to the input and output of the voltage regulation stage. These filters are composed of a series ferrite bead surrounded by two ceramic bypass capacitors. Because ferrite beads exhibit increasing impedance at higher frequencies, while presenting minimal resistance at DC, this configuration has significantly improved spike attenuation compared to a simple capacitor-only filter [54, 55]. The PTR08060W regulator and associated PI filters are shown in Figure 15. Similar filters were used at the outputs of the PTH04070W regulators.
3.3.2.3 Voltage monitoring

The voltage monitoring stage provides undervoltage (brownout) detection and power-on-reset (POR) functionality for the two FPGA modules. This is accomplished using the ISL88021IU8FCZ Triple Voltage Monitor from Intersil. This IC monitors the three secondary power rails (1.2V, 2.5V & 3.3V) and triggers a reset condition whenever any of the three power rails drops below the limits specified in Table 3. Reset is only deasserted after all power rails remain above their respective thresholds for at least 200 ms.

<table>
<thead>
<tr>
<th>Power Rail</th>
<th>Trip Voltage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2 V</td>
<td>1.10 V</td>
</tr>
<tr>
<td>2.5 V</td>
<td>2.32 V</td>
</tr>
<tr>
<td>3.3 V</td>
<td>3.09 V</td>
</tr>
</tbody>
</table>

Tabla 3 Undervoltage Limits
This functionality ensures that both FPGAs will always be correctly initialized at startup and will be reprogrammed in the event of a momentary power failure. Without this capability, an undervoltage condition could corrupt an FPGA’s RAM contents and result in a lockup or other undesired behavior. This is especially important since the ICM FPGA controls the safety switch’s mode select signal during normal operation. During reset the safety switch will also be held in the manual control state, ensuring that recovery of the aircraft is always possible. More details will be provided in section 3.4.3 below.

### 3.3.3 Safety Switch

The main board features an integrated safety switch that allows for changing between manual and autonomous control of the UAV during flight. This is a critical safety feature that enables the human safety pilot to take direct control of the aircraft at any time, overriding the autopilot software. This is necessary in the event of an autopilot malfunction or failure, as well as during manual takeoff and landing.

The main board safety switch provides nine pulse-width modulation (PWM) input channels and eight PWM output channels. Eight of the input channels are used for servo control signals, while the ninth channel is used by the safety pilot to indicate manual or autonomous operation. When in manual mode, the safety switch will buffer the input signals from the RC receiver and pass them directly to the servo outputs. In autonomous mode, the autopilot control signals will be buffered and passed to the servo outputs. In both
cases, all nine input channels will be buffered and connected to the ICM FPGA. All safety switch connector leads are soldered directly to the main board, eliminating one potential source of failure during flight.

A logic diagram of the safety switch is provided below, in Figure 16. Red signals and devices are powered from the RC flight battery, while blue signals and devices are powered by the FCS power supply. As shown, the output multiplexer is powered off of the RC flight battery. This is done so that the safety switch can remain functional even if the autopilot hardware loses power. It also has the effect of increasing the output signal amplitude to that of the RC battery, which has a nominal voltage of 4.8V. This increases the noise margin of the control signals compared to the typical output voltage of most RC receivers. It should be noted that the output multiplexer can remain functional at battery voltages as low as 3.3V, which is significantly less than the servos and RC receiver are capable of functioning at.
During normal operation, custom control logic in the ICM FPGA is used to monitor the PWM input signals and control the multiplexer at the output of the safety switch. If the ICM fails or loses power, the safety switch will be forced into manual mode. This is accomplished using the three-input AND gate and pull-down resistor shown in Figure 16. The AND gate takes in the FPGA Reset, FPGA Done, and Mode Select signals. The power supply voltage monitor will ensure that the Reset line is held low at startup and during undervoltage conditions. After reset, the FPGA Done signal will not go high until the FPGA is fully programmed and operational. Only when both signals are high will the ICM be able to put the safety switch in autonomous mode using the Mode Select signal. If the FCS completely loses power, then the pull-down resistor will ensure that the multiplexer stays in manual mode.
3.3.4 I/O interfaces

In addition to the PWM signals used by the safety switch, the main board features several other digital interfaces that allow for communication with external hardware. A total of six dedicated RS-232 serial ports are available. Two of these are connected to the FCM and four are connected to the ICM. These ports are typically used to interface with external IMU/INS devices, such as the Microbotics MIDG II or the Microstrain 3DM-GX3. A total of 24 general purpose I/O (GPIO) pins are also provided. Sixteen of these pins are directly connected to the FCM FPGA, while the remaining eight are connected to the ICM FPGA. These GPIO pins can be controlled through software or via dedicated logic in the FPGAs. This allows them to be customized to handle nearly any interface function compatible with 3.3V TTL signal levels.

Several GPIO and serial interface lines also exist between the main board and auxiliary board. These signal lines allow the ICM and FCM FPGAs to control and communicate with the application specific devices on the auxiliary board.
4 RF Implementation

This chapter describes the system architecture of the proposed RF controller and its FPGA implementation. Section 4.1 introduces the system architecture while section 4.2 describes the blocks of the architecture. Section 4.3 presents the simulation results of the Verilog models of the architectural blocks while section 4.4 explains the synthesis results by mapping the architectural blocks onto an Altera Cyclone EP1C12Q240I7 FPGA chip. Section 4.6 summarizes the chapter.

4.1 System Architecture

Is need to design a programmable RF resource. The resource includes a LCD panel, a controller and a RF generator, as shown in figure 17:

![Figure 17 System Architecture](image)

The LCD provides a GUI for user to set the parameters like start frequency, end frequency, frequency step, and the period of internal trigger signal. The LCD sends the parameters through a RS232 interface to the MCU on the CONTROLLER PCB, which includes a MCU and a FPGA. The MCU write the parameters to the proper registers in the FPGA, and then the FPGA generates control signals as expected and sends the signals with DB25 interface to the RG generator.
The controller is implemented with a PCB we designed. The PCB is as shown in figure 18.

![PCB board](image)

**Figure 18 PCB board.**

There is a MAX232 on the PCB to connect the input RS232 signal from LCD panel to the TXD/RXD pins of an AT89C51 MCU. The MCU’s data bus is connected to the FPGA, the MCU gets the parameters set by the user and write them to the proper register in the FPGA. The connection of the MCU pins is in FIG3.
The interface between FPGA and MCU has been designed according to datasheet AT89C51. The interface in the FPGA is shown in figure 19. For MCU, the registers are 64 groups with 8 bit width, but for other modules in FPGA registers are 1 group with 512 bit width. That is \( \text{PARAM } [(k*8+7):(k*8)] \) (\( k=0,1,2,..63 \)) is the \( k \)th group registers.

![Figure 19 Connections of MCU pins.](image)

**Figure 19 Connections of MCU pins.**

![Figure 20 Interface between FPGA and MCU.](image)

**Figure 20 Interface between FPGA and MCU.**
The GENERATOR has a DB25 interface with 25 pins, which 21 of these 25 pins, as frequency code pins (F[20:0]), set the output frequency of the generator. The frequency code is binary coded, with 10KHz step and 1.9GHz offset. That is, F[20:0]=21'D0 is 1.9GHz, F[20:0]=21'D1 is 1.9GHz+10KHz, F[20:0]=21'D1610000 is 18GHz. F[20:0] must not be more than 21'D1610000.

There is also a latch enable signal (LE) to lock the frequency code, when LE is “1”, the output frequency changes as the frequency code changes; while if LE is “0”, the output frequency will not change. The remaining 3 pins are 2 GNDs and 1 NC.

![Timing of the controller.](image)

The TRIGGER is an internal generated periodic signal, it’s period T can be set in the PARAM registers, with 8bit width and 1us step. That is, T[7:0]=120 is 120us. The period must not be less than 100us.

When the start frequency, the stop frequency and the frequency step are set, and an external “START” signal is effective (the negative edge of “START” is detected), F[20:0] changes at the positive edge of the TRIGGER to control the GENERATOR stepping from the start frequency f0 to the stop frequency, with desired step Δ. Once the frequency reaches to the stop frequency, the
GENERATOR locked its frequency to f0 until a new effective “START” signal is detected.

The connections between MCU and FPGA are wired directly on PCB, the FPGA pins locked to interface with MCU are:

<table>
<thead>
<tr>
<th>NAME</th>
<th>DIRECTION</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR_H[15]</td>
<td>IN</td>
<td>PIN_78</td>
</tr>
<tr>
<td>ADDR_H[14]</td>
<td>IN</td>
<td>PIN_79</td>
</tr>
<tr>
<td>ADDR_H[13]</td>
<td>IN</td>
<td>PIN_82</td>
</tr>
<tr>
<td>ADDR_H[12]</td>
<td>IN</td>
<td>PIN_83</td>
</tr>
<tr>
<td>AD[7]</td>
<td>IN/OUT</td>
<td>PIN_104</td>
</tr>
<tr>
<td>AD[6]</td>
<td>IN/OUT</td>
<td>PIN_105</td>
</tr>
<tr>
<td>AD[5]</td>
<td>IN/OUT</td>
<td>PIN_106</td>
</tr>
<tr>
<td>AD[4]</td>
<td>IN/OUT</td>
<td>PIN_107</td>
</tr>
<tr>
<td>AD[3]</td>
<td>IN/OUT</td>
<td>PIN_113</td>
</tr>
<tr>
<td>AD[2]</td>
<td>IN/OUT</td>
<td>PIN_114</td>
</tr>
<tr>
<td>AD[1]</td>
<td>IN/OUT</td>
<td>PIN_115</td>
</tr>
<tr>
<td>AD[0]</td>
<td>IN/OUT</td>
<td>PIN_116</td>
</tr>
<tr>
<td>ALE</td>
<td>IN</td>
<td>PIN_86</td>
</tr>
<tr>
<td>P34</td>
<td>IN</td>
<td>PIN_84</td>
</tr>
</tbody>
</table>
LAN_A[4]    OUT    PIN_95
LAN_A[1]    OUT    PIN_100
LAN_A[0]    OUT    PIN_101
nRD    IN    PIN_77
nWR    IN    PIN_76
MCU_RST    OUT    PIN_132
INT_CLK    IN    PIN_153

NOTE:
1. LAN_A[7:0] are similar to ADDR_L[7:0], these pins are connected to the Ethernet interface.
2. INT_CLK is a 50MHz CLK output of the Oscillator on board, this CLK is used as the global clock signal of FPGA.
3. MCU_RST is an POR signal generated by FPGA to reset MCU when power up.

The outputs pins that control the generator are assigned at the followings IOs on FPGA:

<table>
<thead>
<tr>
<th>NAME</th>
<th>DIRECTION</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>F[0]</td>
<td>OUT</td>
<td>PIN_214</td>
</tr>
<tr>
<td>F[1]</td>
<td>OUT</td>
<td>PIN_208</td>
</tr>
<tr>
<td>PIN</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>F[2]</td>
<td>OUT PIN_206</td>
<td></td>
</tr>
<tr>
<td>F[3]</td>
<td>OUT PIN_202</td>
<td></td>
</tr>
<tr>
<td>F[4]</td>
<td>OUT PIN_200</td>
<td></td>
</tr>
<tr>
<td>F[5]</td>
<td>OUT PIN_196</td>
<td></td>
</tr>
<tr>
<td>F[6]</td>
<td>OUT PIN_194</td>
<td></td>
</tr>
<tr>
<td>F[7]</td>
<td>OUT PIN_187</td>
<td></td>
</tr>
<tr>
<td>F[8]</td>
<td>OUT PIN_185</td>
<td></td>
</tr>
<tr>
<td>F[9]</td>
<td>OUT PIN_183</td>
<td></td>
</tr>
<tr>
<td>F[10]</td>
<td>OUT PIN_182</td>
<td></td>
</tr>
<tr>
<td>F[12]</td>
<td>OUT PIN_193</td>
<td></td>
</tr>
<tr>
<td>F[13]</td>
<td>OUT PIN_213</td>
<td></td>
</tr>
<tr>
<td>F[14]</td>
<td>OUT PIN_207</td>
<td></td>
</tr>
<tr>
<td>F[15]</td>
<td>OUT PIN_203</td>
<td></td>
</tr>
<tr>
<td>F[16]</td>
<td>OUT PIN_201</td>
<td></td>
</tr>
<tr>
<td>F[17]</td>
<td>OUT PIN_197</td>
<td></td>
</tr>
<tr>
<td>F[18]</td>
<td>OUT PIN_195</td>
<td></td>
</tr>
<tr>
<td>F[19]</td>
<td>OUT PIN_188</td>
<td></td>
</tr>
<tr>
<td>F[20]</td>
<td>OUT PIN_186</td>
<td></td>
</tr>
<tr>
<td>LE</td>
<td>OUT PIN_184</td>
<td></td>
</tr>
</tbody>
</table>

### 4.2 System Blocks
This section describes briefly the functionality and specifications of each block in the overall system architecture.

4.2.1 Por_gen

The functionality of por_gen.v block is to generate a power-on reset (PoR), which detects the power applied to chip and generates a reset impulse that goes to the entire circuit placing it into a known state. The duration of the power-on reset signal is approximately 3.142s.

![Figure 22 por_gen interface block.](image)

4.2.2 pll

This block has been designed with a Phase-Locked Loop (ALTPLL) Megafunction.

The Phase-Locked Loop (PLL) is a closed-loop frequency-control system that compares the phase difference between the input signal and the output signal of a voltage-controlled oscillator (VCO). The negative feedback loop of the system forces the PLL to be phase-locked.
The PLL consists of a pre-divider counter (N counter), a phase-frequency detector (PFD) circuit, a charge pump, loop filter, a VCO, a feedback multiplier counter (M counter), and post-divider counters (K and V counters).

The PFD detects the differences in phase and frequency between its reference signal ($f_{\text{REF}}$) and feedback signal (Feedback), controls the charge pump, and controls a loop filter that converts the phase difference to a control voltage. This voltage controls the VCO.

Based on the control voltage, the VCO oscillates at a higher or lower frequency, which affects the phase and frequency of the Feedback signal. After the $f_{\text{REF}}$ signal and the Feedback signal have the same phase and frequency, the PLL is said to be phase-locked.

Inserting the M counter in the feedback path causes the VCO to oscillate at a frequency that is M times the frequency of the $f_{\text{REF}}$ signal. The $f_{\text{REF}}$ signal is equal to the input clock ($f_{\text{IN}}$) divided by the pre-scale counter (N).
The reference frequency is described by the equation \( f_{REF} = f_{IN}/N \). The VCO output frequency is \( f_{VCO} = f_{IN} \times M/N \), and the output frequency of the PLL is described by the equation \( f_{OUT} = (f_{IN} \times M)/(N \times K) \) for the signals.

### 4.2.3 RF_gen

This module instantiate frequency_sweep.v and trigger_gen.v. The main function of this module is to create a framework between the top.v module and frequency_sweep.v and trigger_gen.

![RF_gen RTL diagram.](image)

**Figure 24 RF_gen RTL diagram.**

#### 4.2.3.1 Trigger_gen

This module generates the trigger signal used in the frequency sweep module.

This signal is generated from the user defined parameter period of trigger. The trigger pulse generated is of 1us. See the code or the simulation results for more information.
4.2.3.2 Frequency_sweep

The function of this module is to generate the proper signal to RF generator from the parameters defined by the user. The start frequency, end frequency and delta parameters are used to generate the proper RF out signal. The first version of this code was simulated and was working correctly. But when wants to synthetize the code, Quartus was not able to synthetize the code. So, it has been to modify the code in order to make it synthesizable.
4.2.4 **Mcu_param_reg**

The main objective in this block is to create a bank of registers width 64 address and 8 bit width per address. This block uses a bidirectional signal to specify the address and the data.

The different parameters used in this project are described following.

- **Period of Trigger**

  This parameter is allocated in the `param[23:16]`, the width of this parameter is 8 bit. Thus, the address for Trigger period parameter is `AD=6'b000010`. The minimum value corresponds to `8'd100`, equivalent to `8'h64`, this value generate a trigger with a period of 100us. Furthermore, the max value is restricted by the max value of the register, which is `8'd255` or `8'hFF`, corresponds to a period of 255us.

- **Final and initial frequencies**
For these parameters is necessary 21 bits. Since the initial frequency is 1.8GHz, is represented with all zero’s, and the max frequency is 19GHz. Consider the minimum value of delta in order to obtain the max number of steps.

The value_max is represented as:

\[
value_{\text{max}} = \frac{freq_{\text{max}} - freq_{\text{min}}}{\Delta_{\text{min}}} = \frac{18\text{GHz} - 1.9\text{GHz}}{10\text{kHz}} = 1610000
\]

The number of bits necessary to represent this value is:

\[
n = \left\lceil \frac{\log 1610000}{\log 2} \right\rceil = \left\lceil 20.618 \right\rceil = 21 \text{ bits}
\]

So is needed 3 registers to represent the initial and end frequency parameters.

- **Initial_freq_parameter**

This parameter is allocated in param[52:32], the width of this parameter is 21 bits. Thus, the addresses to configure the parameter are:

<table>
<thead>
<tr>
<th>AD</th>
<th>param</th>
</tr>
</thead>
<tbody>
<tr>
<td>6'b000100 = 6'd4</td>
<td>param[39:32]</td>
</tr>
<tr>
<td>6'b000101 = 6'd5</td>
<td>Param[47:40]</td>
</tr>
<tr>
<td>6'b000110 = 6'd6</td>
<td>Param[55:48]</td>
</tr>
</tbody>
</table>
- **Final_freq_parameter**

This parameter is allocated in param[84:64], the width of this parameter is 21 bits. Thus, the addresses to configure the parameter are:

<table>
<thead>
<tr>
<th>AD</th>
<th>param Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6'b001000</td>
<td>6'd8</td>
</tr>
<tr>
<td>6'b001001</td>
<td>6'd9</td>
</tr>
<tr>
<td>6'b001010</td>
<td>6'd10</td>
</tr>
</tbody>
</table>

- **Delta**

For this parameter is necessary 10 bits, since the minimum delta is 10KHz and the maximum delta is 10MHz.

The max value represented is:

\[
value_{max} = \frac{\Delta_{max} - \Delta_{min}}{\Delta_{min}} = \frac{10MHz - 10KHz}{10kHz} = 999
\]

The number of bits necessaries to represent this value is:

\[
n = \left\lceil \frac{\log 999}{\log 2} \right\rceil = [9.964] = 10\ bits
\]

So is needed 2 registers to represent delta parameter.
This parameter is allocated in param[137:128], the width of this parameter is 10 bits. Thus, the addresses to configure the parameter are:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6'b010000</td>
<td>6'd16</td>
<td>6'b010001</td>
<td>6'd17</td>
</tr>
</tbody>
</table>

The value 0 is not allowed in delta parameter. The min value of delta must be 10’d1 which represent a delta of 10KHz. Moreover, the maximum value of delta is 10MHz represented as 10’d10000.

4.2.5 Decoder_16

The four MSB of the address signal are used to select the proper chip. In this case are used to select the FPGA. The correct address to select the FPGA is 4’hA.

![Decoder_16 RTL design.](image)

4.3 Simulation Results
To verify the system architecture, simulations were performed using ModelSim-Altera 6.5b (Quartus II 9.1) Starter Edition. Each block was simulated for proper functionality before being tested in-circuit.

In figure 28 checks that the pulse period of trigger signals is 1us. Furthermore, we can see how the system works when a reset signal is applied. In this case all the counters put to 0 and the default period is 100us. The trigger signal is represented in blue. Figure 29 and 30 checks that the trigger period generated match with the trigger period parameter defined by the user. In the first case the period is 100us and in the second one the period is 127us.

![Figure 28 Simulation of trigger_gen, check pulse period.](image1)

![Figure 29 Simulation of trigger_gen, check trigger period (100us).](image2)
Figure 30 Simulation of trigger_gen, check trigger period (127us).

Figure 31 corresponds to the simulation of sweep_frequency block. Notice that when a rising edge of start signal is detected then the sweep frequency process is enabled. Until the sweep frequency process has finished the start external signal is ignored. In this example the start frequency and end frequency are set to 35 and 40 respectively. Delta parameter is set to 2. That means that the process starts in 35 and has a step of 2. So, it will never reach 40 since is not multiple. As we can see in the simulation when the 39 frequency is reached the process stop and wait for another start signal. This process has been checked for all max and min parameters and the functionality is the expected and desired.
The framework for the trigger_gen and sweep_frequency is shown in figure 32. In this figure the trigger generate is directly connected to sweep_frequency module. Notice that the periods and the sweep_frequency are correct.

A simulation for the whole system is represented in figure 32. In this figure is represented the process to write and read values in the register bank. To select the FPGA chip the ADDR_H signal has to be 4’hA. Moreover, the ALE signal is used to set the address for the register bank.
4.4 System Architecture Synthesis

After VHDL modeling, Altera’s Quartus II Web Edition, version 9.1, was used to synthesize the different blocks in the system architecture. Table 4 shows the number of HDL lines of code written to specify the functionality of each block, the number of Les needed to implement each block in the CYCLONE chip, and the ratio of the implementation Les to the available Les in the actual device. The device utilization is also given as a reference. The utilization figures are based on the Altera Cyclone EP1C12Q240I7 FPGA chip, which contains 12060 Les.

<table>
<thead>
<tr>
<th>Module name</th>
<th>Lines of HDL</th>
<th>Logic Elements</th>
<th>Device Utilization (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoder32</td>
<td>38</td>
<td>30</td>
<td>0.24</td>
</tr>
<tr>
<td>Trigger_gen</td>
<td>59</td>
<td>141</td>
<td>1.17</td>
</tr>
<tr>
<td>Por_gen</td>
<td>39</td>
<td>40</td>
<td>0.33</td>
</tr>
<tr>
<td>PLL1</td>
<td>148</td>
<td>113</td>
<td>0.94</td>
</tr>
<tr>
<td>Mcu_param_reg</td>
<td>120</td>
<td>323</td>
<td>2.67</td>
</tr>
<tr>
<td>Frequency_sweep</td>
<td>115</td>
<td>422</td>
<td>3.5</td>
</tr>
<tr>
<td>TOTAL</td>
<td>519</td>
<td>1069</td>
<td>9</td>
</tr>
</tbody>
</table>

Tabla 4: VHDL specification and synthesis of each block in the system architecture.
As shown in this table, only nine percent of the Cyclone EP1C12Q240I7 is used. Although this device is a low capacity chip, it is largely sufficient to implement an entire RF controller. This result makes the case for using high capacity chips for implementing complex controllers such as the ones used for multi-channel based on intelligent computational approaches such as neural networks.

Table 5 shows the maximum delay and the resulting clock frequency of the synthesized implementation of each block in the system architecture.

<table>
<thead>
<tr>
<th>Type</th>
<th>Slack</th>
<th>Required Time</th>
<th>Actual Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst-case tsu</td>
<td>N/A</td>
<td>None</td>
<td>11.991 ns</td>
</tr>
<tr>
<td>Worst-case tco</td>
<td>N/A</td>
<td>None</td>
<td>26.694 ns</td>
</tr>
<tr>
<td>Worst-case tpd</td>
<td>N/A</td>
<td>None</td>
<td>25.068 ns</td>
</tr>
<tr>
<td>Worst-case th</td>
<td>N/A</td>
<td>None</td>
<td>0.530 ns</td>
</tr>
<tr>
<td>Clock Setup: 'pll1:pll1</td>
<td>altpll:altpll_component</td>
<td>_clk1'</td>
<td>7.972 ns</td>
</tr>
<tr>
<td>Clock Setup: 'INT_CLK'</td>
<td>16.895 ns</td>
<td>50.00 MHz ( period = 20.000 ns )</td>
<td>Restricted to 320.10 MHz ( period = 3.124 ns )</td>
</tr>
<tr>
<td>Clock Hold: 'pll1:pll1</td>
<td>altpll:altpll_component</td>
<td>_clk1'</td>
<td>0.918 ns</td>
</tr>
<tr>
<td>Clock Hold: 'INT_CLK'</td>
<td>0.934 ns</td>
<td>50.00 MHz ( period = 20.000 ns )</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Total number of failed paths</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Tabla 5: Block Synthesized and frequencies.

Default parameters of the synthesis tool were used to synthesize the blocks of the system architecture. No optimization has been performed while synthesizing these blocks. As the table shows, the slowest block is the PLL1
with can run at a frequency of 83.14 MHz. This means that the remaining blocks in the system architecture can slow down in order to run at the same speed as the RF controller. Ignoring latency considerations outside the FPGA chip, the RF controller can produce a throughput of 21.321.961 power tracking points per second.

4.5 Summary

This chapter describes the architectural components of the system architecture of the proposed RF controller. Next, it presents the results obtained from the synthesis of the controller into an FPGA implementation. Cursory examination of synthesis results shows that this implementation can be readily expanded to handle RF control with more functionality.
5 Prototype Testing

This chapter presents additional findings regarding the testing of the physical prototype used to implement the RF generator.

Is used a spectrum analyzer in order to analize the functionality of the RF generator. The equipment used to check the sytem is shown in figure 34. The powers supply, the FPGA, the LCD and the the RF generator are all together in one device. In the picture we see the deviced named and the spectrum analyzer.

Figure 34 System to test the RF generator.
In order to check the functionality the user define the parameters, as initial frequency, end frequency, period of trigger and delta. For this purpose there is a GUI programmed in Visual Basic. As shown in figure 36.

In this example the period of trigger is set to 200 us, initial frequency is 4GHz, end frequency is 10 GHz and the delta is 10 times 10 KHz. Setting the
holding mode in the spectrum analyzer it can be appreciated the frequency sweep generated by the RF generator, which is shown in figure 37.

Still working in another way to introduce the user defined parameters. This way consist in use a LCD touchscreen to set the parameters. The user interface is pretty much the same but the communication protocol is different. Currently there are some problems but the LCD is programmed correctly as shown in figure 38.
Figure 38 RF generator equipment with LCD.

Figure 39 Interface of LCD screen.
6 Conclusions

FPGAs have been widely used in control systems to leverage their many benefits including higher data throughput, reliability, concurrency, and expected performance. FPGA-based control systems can alleviate traditional hardware design constraints such as system control complexity, degree of modifiability, and universal communication abstraction. Software engineering concepts may also be instilled in a FPGA-based control system with DPR. Despite the many benefits, the accessibility of hardware designs remains a critical issue for developers. The complexity and the prolonged design time often impede their use in developing embedded applications, including control.

A good technique to use is to synthesize the code before being simulated with another external program. During the realization of this thesis a first version of the sweep_frequency.v code was created and simulated with Modelsim. However, when the first version was introduced in Quartus, it was unable to synthesize the code. So a second version has to be created with the corresponding wasting of time. Since, another focus to solved the problem had to be applied.

A future work is finishing the programing of the communication between the LCD and the MCU. Furthermore, more functionality and more devices can be controlled by the same MCU and FPGA.
7 Appendix: source code listing
7.1 Top.v

module top//Module name

//Parameter declaration-------------------------
#(parameter ADDR_PAR_REG=10 //Parameter register high-order address

) //Parameter Definition

//------------------------------------------
//Port Declaration---------------------

(input INT_CLK, //The default on the PCB, 50MHz, generating CLK_50

input P34, //PC reset control line, active low reset FPGA

input [15:12] ADDR_H, //High-order address input

input [7:0] AD, //Low multiplexed address data bus

input nRD, //Read control line, active low

input nWR, //Write control line, active low

input ALE, //Address Latch Enable

//Low address latch signal, falling latch

//Stanby line of control

//Temporary debugging line

//Output port---------------------

(output MCU_RST, //Chip power-on reset signal

// Allow

output reg [7:0] LAN_A, //Latch the address for the LAN low address

//Other control lines

output LE,

//Output control signal

output [20:0] F //Pulses generated

//Alternate signal life

);

//Signal Definitions

//Power-on Reset

wire nPOR;

por_gen por_gen(.CLK(INT_CLK),.nRST(nPOR));

//Global Reset control signal

wire nGCLR;
assign nGCLR=nPOR & P34;

//PLL1 system clock generation
wire CLK_50;
pl1 pll1 (.inclk0 ( INT_CLK ),.cl ( CLK_50 ));

//PLL1 system clock generation

//SCI Sample low address latch, or only at power microcontroller reset
wire [7:0] ADDR_L; //Low address

always @(negedge ALE or negedge nGCLR)
begin
  if(!nGCLR)
    LAN_A<=0;
  else
    LAN_A<=AD;
end
assign ADDR_L=LAN_A;

//Address decoding chip high output parameter register select
signal do
  //Because LAN uses 15 address, FPGA's address high four available addresses is 8-F
  wire [15:0] nCS; //Address decoding output high, as chip select
  decoder16 decoder16(.ADDR(ADDR_H),.nCS(1'B0),.nRW_EN(nCS));

  //SCI (Software Configuration Management) parameter register,
  //can be read back only at power MCU reset or reset 64*8bit
  wire [511:0] PARAM; //Parameter Register Group
  mcu_param_reg mcu_param_reg(.CLK(nWR),.nRST(nGCLR),.nWR(1'B0),.nRD(nRD),.nCS(nCS[ADD_PAR_REG]),.ADDR(ADDR_L[5:0]),.DATA(AD),.PARAM(PARAM));

//0[0]write parameters indicating signal indicates a parameter update,
//then reset the unit of work, 0 indicates that the update is completed,
//one of the reset control signal
wire FLAG_PRM_UPDT; //Write parameters indicating signal
assign FLAG_PRM_UPDT=PARAM[0]; //0[0] Write parameters indicating signal

wire nRST; //Timing unit reset control signal
// assign nRST=nGCLR&&!(!FLAG_PRM_UPDT);

wire [7:0] trigg_T;
wire [20:0] ini_F;
wire [20:0] end_F;
wire [9:0] delta;

assign trigg_T = PARAM[23:16];
assign ini_F = PARAM[52:32];
assign end_F = PARAM[84:64];
assign delta = PARAM[137:128];

RF_gen RF_gen
.clk(CLK_50),
.rst(nGCLR),
//.start(start),
.start(FLAG_PRM_UPDT),
.param_trig_T(trigg_T),
.param_ini_F (ini_F),
.param_end_F(end_F),
.param_delta(delta),
.RF_freq(F),
.LE (LE)
);

//MCU reset control, nPOR is the power-on reset, about 20ms delay releasing the reset signal,
//the final output is open-drain output that allows an external manual reset
wire ENA1M;
div div50(.RST(nGCLR),.CLK(CLK_50),.COUT(ENA1M));
defparam div50.MOD=50,
div50.WID=6,
div50.ALoad=0;

reg [19:0] MCU_COUNT; //MCU reset control the internal count register
reg MCU_RST1; //MCU reset output signal to control the internal

always @(posedge CLK_50 or negedge nPOR)
begin
  if (!nPOR)
    begin
      MCU_RST1<=1;
      MCU_COUNT<=0;
    end
  else if (ENA1M)
    begin
      if(MCU_COUNT==20'HFFFFF)
        MCU_RST1<=0;
      else
        MCU_COUNT<=MCU_COUNT + 20'd1;
    end
end

assign MCU_RST=MCU_RST1?1'B1:1'BZ;
endmodule
7.2 top_tb.v

`timescale 1 ns / 1 ns
`define CLK_HALF_PERIOD 10
`define Address_T_TRIGG 8'd2
`define Address_INI_F 8'd4
`define Address_END_F 8'd8
`define Address_DELTA 8'd16

`define Value_T_TRIGG 8'd100 // Set to 100 us
`define Value_INI_F_7_0 8'h00
`define Value_INI_F_15_8 8'h00
`define Value_INI_F_20_16 8'h00
`define Value_END_F_7_0 8'h10
`define Value_END_F_15_8 8'h00
`define Value_END_F_20_16 8'h00 // Set to 1.8GHz
`define Value_DELTA_7_0 8'hE7
`define Value_DELTA_9_8 8'h03 // Set to 10MHz

module top_tb;

// ------------------Input Ports-----------------------
reg INT_CLK;
reg P34;
reg[15:12] ADDR_H;
reg nRD;
reg nWR;
reg ALE;

// ------------------Inout Ports-----------------------
reg [7:0] AD_in;
wire [7:0] AD_tb, AD_out;

// ------------------Output Ports-----------------------
wire MCU_RST;
wire [7:0] LAN_A;
wire LE;
wire [20:0] F;

top DUT ( INT_CLK (INT_CLK),
         P34 (P34),
         ADDR_H (ADDR_H),
         AD (AD_tb),
         nRD (nRD),
         nWR (nWR),
         ALE (ALE),
         MCU_RST (MCU_RST),
         LAN_A (LAN_A),
         LE (LE),
         F (F) );

assign AD_tb = ((ADDR_H==4'ha) && nRD) ? AD_in : 8'bZ;
assign AD_out = ((ADDR_H==4'ha) && !nRD) ? AD_tb : 8'bZ;

initial begin
  INT_CLK = 1'b0;
P34 = 1'b1;
ADDR_H = 4'hA;
nRD = 1'b1;
end
nWR = 1'b1;
ALE = 1'b1;
//------------------ WRITE PROCESS ------------------
// Select address from AD_in, the address is set producing a nedege of ALE
AD_in = `Address_T_TRIGG;
repeat (20) @ (posedge INT_CLK);
ALE = 1'b0;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b1;
repeat (4) @ (posedge INT_CLK);
AD_in = `Value_T_TRIGG;
repeat (3) @ (posedge INT_CLK);
NWR = 1'b0;
repeat (2) @ (posedge INT_CLK);
NWR = 1'b1;
//******CONFIGURE INI_F_PARAM***************
// Configure [7:0] INI_F_PARAM
repeat (2) @ (posedge INT_CLK);
AD_in = `Address_INI_F;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b0;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b1;
repeat (4) @ (posedge INT_CLK);
AD_in = `Value_INI_F_7_0;
repeat (3) @ (posedge INT_CLK);
NWR = 1'b0;
repeat (3) @ (posedge INT_CLK);
NWR = 1'b1;
// Configure [15:8] INI_F_PARAM
repeat (2) @ (posedge INT_CLK);
AD_in = `Address_INI_F + 8'd1;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b0;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b1;
repeat (4) @ (posedge INT_CLK);
AD_in = `Value_INI_F_15_8;
repeat (3) @ (posedge INT_CLK);
NWR = 1'b0;
repeat (3) @ (posedge INT_CLK);
NWR = 1'b1;
// Configure [20:16] INI_F_PARAM
repeat (2) @ (posedge INT_CLK);
AD_in = `Address_INI_F + 8'd2;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b0;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b1;
repeat (4) @ (posedge INT_CLK);
AD_in = `Value_INI_F_20_16;
repeat (3) @ (posedge INT_CLK);
NWR = 1'b0;
repeat (3) @ (posedge INT_CLK);
NWR = 1'b1;
//******************************
//******CONFIGURE END_F_PARAM***************
// Configure [7:0] END_F_PARAM
repeat (2) @ (posedge INT_CLK);
AD_in = `Address_END_F;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b0;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b1;
repeat (4) @ (posedge INT_CLK);
AD_in = `Value_END_F_7_0;
repeat (3) @ (posedge INT_CLK);
nWR = 1'b0;
repeat (3) @ (posedge INT_CLK);
nWR = 1'b1;
// Configure [15:8] END_F_PARAM
repeat (2) @ (posedge INT_CLK);
AD_in = `Address_END_F + 8'd1;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b0;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b1;
repeat (4) @ (posedge INT_CLK);
AD_in = `Value_END_F_15_8;
repeat (3) @ (posedge INT_CLK);
nWR = 1'b0;
repeat (3) @ (posedge INT_CLK);
nWR = 1'b1;
// Configure [20:16] END_F_PARAM
repeat (2) @ (posedge INT_CLK);
AD_in = `Address_END_F + 8'd2;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b0;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b1;
repeat (4) @ (posedge INT_CLK);
AD_in = `Value_END_F_20_16;
repeat (3) @ (posedge INT_CLK);
nWR = 1'b0;
repeat (3) @ (posedge INT_CLK);
nWR = 1'b1;
//******************************************************************************

//*****CONFIGURE DELTA_PARAM************
// Configure [7:0] DELTA_PARAM
repeat (2) @ (posedge INT_CLK);
AD_in = `Address_DELTA;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b0;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b1;
repeat (4) @ (posedge INT_CLK);
AD_in = `Value_DELTA_7_0;
repeat (3) @ (posedge INT_CLK);
nWR = 1'b0;
repeat (3) @ (posedge INT_CLK);
nWR = 1'b1;
// Configure [15:8] DELTA_PARAM
repeat (2) @ (posedge INT_CLK);
AD_in = `Address_DELTA + 8'd1;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b0;
repeat (2) @ (posedge INT_CLK);
ALE = 1'b1;
repeat (4) @ (posedge INT_CLK);
AD_in = `Value_DELTA_9_8;
repeat (3) @(posedge INT_CLK);
nWR = 1'b0;
repeat (3) @(posedge INT_CLK);
nWR = 1'b1;
.dataGridViewTextBoxColumn
 repeat (3) @(posedge INT_CLK);
 nWR = 1'b0;
 repeat (3) @(posedge INT_CLK);
 nWR = 1'b1;
 nWR = 1'b0;
 repeat (3) @(posedge INT_CLK);
 nWR = 1'b1;
 nWR = 1'b0;
 repeat (3) @(posedge INT_CLK);
 nWR = 1'b1;
 nWR = 1'b0;
repeat (3) @(posedge INT_CLK);
AD_in = 8'hFF;
repeat (2) @(posedge INT_CLK);
ALE = 1'b0;
repeat (2) @(posedge INT_CLK);
ALE = 1'b1;
repeat (4) @(posedge INT_CLK);
AD_in = 8'd1;
repeat (3) @(posedge INT_CLK);
nWR = 1'b0;
repeat (3) @(posedge INT_CLK);
nWR = 1'b1;
repeat (4) @(posedge INT_CLK);
AD_in = 8'd0;
repeat (3) @(posedge INT_CLK);
nWR = 1'b0;
repeat (3) @(posedge INT_CLK);
nWR = 1'b1;

end

initial begin

// Set pulse of start signal
repeat (80000) @(posedge INT_CLK);
AD_in = 8'd0;
repeat (2) @(posedge INT_CLK);
ALE = 1'b0;
repeat (2) @(posedge INT_CLK);
ALE = 1'b1;
repeat (4) @(posedge INT_CLK);
AD_in = 8'd1;
repeat (3) @(posedge INT_CLK);
nWR = 1'b0;
repeat (3) @(posedge INT_CLK);
nWR = 1'b1;
repeat (4) @(posedge INT_CLK);
AD_in = 8'd0;
repeat (3) @(posedge INT_CLK);
nWR = 1'b0;
repeat (3) @(posedge INT_CLK);
nWR = 1'b1;

end

always #`CLK_HALF_PERIOD INT_CLK =! INT_CLK;
endmodule
module mcu_param_reg //mcu_param_reg(.CLK(),.nRST(),.nWR(),.nRD(),.nCS(),.ADR(),.DATA(),.PARAM());

//--------------Parameter declaration------------------------
#(parameter WID_D=8, //Data Width
    DEEP=64  //Number of registers
)
//Parameter definition

//--------------Port declaration---------------------
//Input port
input CLK,
input nRST,
input nCS,nWR,nRD,
//Data and address
input [5:0] ADDR,
inout [WID_D-1:0] DATA,
//Other control lines

//Data and address

//--------------Output port---------------------
//Clock
//Reset
//Allow
//Data and address
output [WID_D*DEEP-1:0] PARAM
//Other control lines

);
//Port Definition

//--------------Internal signal declaration--------------------------
wire [DEEP-1:0] nRWEN;
//--------------------------
decoder64 decoder64 (.ADDR(ADDR),.nCS(nCS),.nRWEN(nRWEN));

genvar i;
generate for (i = 0; i < DEEP; i = i + 1)
begin :
    wr_rd
    wr (.CLK(CLK),.nRST(nRST),.nWR(nWR),.nRD(nRD),.nCS(nCS),.nRWEN(i),.DATA(DATA),.PARAM((PARAM[(WID_D*(i+1)-1):WID_D*i])));
end
generate

defparam wr[0].rd.DEF_D=8'h00,   // 0 Write parameter indicates the signal
                             // 1 Parameter
update
update is completed, the default updates is completed

// 0 Parameter

wr[1].rd.DEF_D=8'h00, //1
wr[2].rd.DEF_D=8'd100, //2 [7:0]trigg_T -> Trigger
period, 1us unit

wr[3].rd.DEF_D=8'h00, //3
wr[4].rd.DEF_D=8'h00, //4 [7:0]ini_F -> Inicial
frequency, 10 KHz unit

wr[5].rd.DEF_D=8'h00, //5 [15:8]ini_F
wr[6].rd.DEF_D=8'h00, //6 [20:16]ini_F
wr[7].rd.DEF_D=8'h00, //7
wr[8].rd.DEF_D=8'h00, //8 [7:0]end_F -> Final
frequency, 10 KHz unit

wr[9].rd.DEF_D=8'had, //9 [15:8]end_F
wr[10].rd.DEF_D=8'h01, //a [20:16]end_F (value =
1AD0 -> set to 3GHz)

wr[11].rd.DEF_D=8'h00, //b
wr[12].rd.DEF_D=8'h00, //c
wr[13].rd.DEF_D=8'h00, //d
wr[14].rd.DEF_D=8'h00, //e
wr[15].rd.DEF_D=8'h00, //f
wr[16].rd.DEF_D=8'h01, //10 [7:0]delta -> user defined
frequency step

wr[17].rd.DEF_D=8'h00, //11 [9:8]delta
wr[18].rd.DEF_D=8'h00, //12
wr[19].rd.DEF_D=8'h00, //13
wr[20].rd.DEF_D=8'h00, //14
wr[21].rd.DEF_D=8'h00, //15
wr[22].rd.DEF_D=8'h00, //16
wr[23].rd.DEF_D=8'h00, //17
wr[24].rd.DEF_D=8'h00, //18
wr[25].rd.DEF_D=8'h00, //19
wr[26].rd.DEF_D=8'h00, //20
wr[27].rd.DEF_D=8'h00, //21
wr[28].rd.DEF_D=8'h00, //22
wr[29].rd.DEF_D=8'h00, //23
wr[30].rd.DEF_D=8'h00, //24
wr[31].rd.DEF_D=8'h00, //25
wr[32].rd.DEF_D=8'h00, //26
wr[33].rd.DEF_D=8'h00, //27
wr[34].rd.DEF_D=8'h00, //28
wr[35].rd.DEF_D=8'h00, //29
wr[36].rd.DEF_D=8'h00, //30
wr[37].rd.DEF_D=8'h00, //31
wr[38].rd.DEF_D=8'h00, //32
wr[39].rd.DEF_D=8'h00, //33
wr[52].rd.DEF_D=8'h00, //34
wr[53].rd.DEF_D=8'h00, //35
wr[54].rd.DEF_D=8'h00, //36
wr[55].rd.DEF_D=8'h00, //37
wr[56].rd.DEF_D=8'h00, //38
wr[57].rd.DEF_D=8'h00, //39
wr[58].rd.DEF_D=8'h00, //3a
wr[59].rd.DEF_D=8'h00, //3b
wr[60].rd.DEF_D=8'h00, //3c
wr[61].rd.DEF_D=8'h00, //3d
wr[62].rd.DEF_D=8'h00, //3e
wr[63].rd.DEF_D=8'h00; //3f

endmodule
module por_gen
#
  parameter RST_CNT_WID=27
  //parameter RST_CNT_WID=4 //SIMULATION
)

    // input Ports
    input CLK, //Clock
    //input LOCK, //PLL lock indicator

    // output Ports
    //output reg nRST2,
    output reg nRST
    );

    //Power-on reset delay
    reg [RST_CNT_WID-1:0] RST_CNT = 0;
    always @(posedge CLK )
    begin
      if(RST_CNT[RST_CNT_WID-1:RST_CNT_WID-4]==4'HF)
        begin
          nRST<=1;
        end
      else
        begin
          nRST<=0;
          RST_CNT<=RST_CNT+1;
        end
    end

    /*always @(posedge CLK )
    begin
      if(RST_CNT[RST_CNT_WID-1:RST_CNT_WID-2]==2'H3)
        begin
          nRST2<=1;
        end
    end
    */
endmodule
7.5  read.v

module read
    //read(.nCS(),.nRD(),.PARAM(),.DATA());
    //Parameter declaration-----------------------------
    #(
        parameter DAT_WID=8
    ) //Parameter definition
    //----------Port declaration-----------------------
    ( //----------Input port------------------------
        //Clock
        //Reset
        //Allow
        input  nCS,
        input  nRD,
        //Data and address
        input  [DAT_WID-1:0] PARAM,
        //Other control lines
    ) //----------Output port------------------------
    //Clock
    //Reset
    //Allow
    //Data and address
    output  [DAT_WID-1:0] DATA
    //Other control lines
); //Port Definition

//----------Internal signal declaration--------------

//--------------------------------------------------
assign  DATA=(nCS||nRD)?{DAT_WID{1'b0}}:PARAM;
endmodule
7.6  RF_gen.v

module RF_gen(clk,rst,start,param_trig_T,param_ini_F,param_end_F,param_delta,RF_freq,LE);

    //Parameter declarations
    // These parameters define the size of the registers,
    // those registers that comes from the register bank
    parameter L_trig_T = 8;
    parameter L_ini_F = 21;
    parameter L_end_F = 21;
    parameter L_delta = 10;
    parameter L_RF = 21;

    //Port declarations
    input clk,rst,start;
    input [L_trig_T-1:0] param_trig_T;
    input [L_ini_F-1:0] param_ini_F;
    input [L_end_F-1:0] param_end_F;
    input [L_delta-1:0] param_delta;
    output [L_RF-1:0] RF_freq;
    output LE;

    wire int_trigger;

    //Module instantiation
    defparam trigger_gen.size_cnt_pulse = 6;
    defparam trigger_gen.size_cnt_period = 8;
    trigger_gen trigger_gen
    (    .clk (clk),
    .nRST (rst),
    .T_trigger (param_trig_T),
    .trigger (int_trigger)
    );

    defparam frequency_sweep.length_reg_f = L_ini_F;
    defparam frequency_sweep.length_reg_delta = L_delta;
    defparam frequency_sweep.length_RF = L_RF;

    frequency_sweep frequency_sweep
    (    .clk (clk),
    .nRST (rst),
    .start (start),
    .trigger (int_trigger),
    .inicial_f (param_ini_F),
    .final_f (param_end_F),
    .delta (param_delta),
    .RF_freq (RF_freq),
    .LE (LE)
    );
endmodule
7.7 trigger_gen.v

module trigger_gen (clk,nRST,T_trigger,trigger);
// Parameter declarations
parameter size_cnt_pulse = 6;
parameter size_cnt_period = 8;
//Ports declarations
input clk,nRST;
input [7:0] T_trigger; // parameter from mcu_param_reg that sets the period of trigger
output trigger;

//Internals signals
// SBC used to generate a signal of 1us
// 50cycles, T=20ns @50MHz => 1 us
reg [size_cnt_pulse-1:0] step_1u;
// SBC used to generate the trigger signal with the user defined period
reg [size_cnt_period-1:0] cnt_period;
// User defined period of trigger signal
wire [7:0] period;
// // Signals to sincronize rst, since rst is a external signal an is not sincronized with clock
// reg rst_sync1 = 1'b0 ,rst_sync2 = 1'b0 ;

//Minimum user defined period must be 100us, maximum 2^8 - 1 = 255us
// Notice that period is always the user defined value in the parameter
// ¡OJO!reset signal doesn't have any influence in period net
assign period = (T_trigger > 8’d100) ? (T_trigger):(8’d100);

// // Process to synchronize reset signal, necesary to obtain a clear
// // MAKE SYNC_RESET GLOBAL! for the whole system
// always @(posedge clk)
// begin
//   // rst_sync1 <= rst;
//   // rst_sync2 <= rst_sync1;
// end
always @(posedge clk)
begin
  if(!nRST)// Synchronous reset
  begin
    step_1u <= 'b0;
    cnt_period <= 'b0;
  end
  else
  begin
    step_1u <= step_1u + 1'b1;
    if(step_1u == 'd49) // 50cycles, T=20ns @50MHz => 1 us
    begin
      cnt_period <= cnt_period + 1'b1;
      step_1u <= 'b0;
      if (cnt_period == period - 1) //When cnt_period is equal to period, initialize to 0.
        cnt_period <= 'b0;
    end
  end
end
end

// // Generate the output signal trigger, with a user defined period
// and a width pulse of 1us.
// assign trigger = (cnt_period== 1'b0 && !rst_sync2)? 1'b1 : 1'b0;
// Generate the output signal trigger, with a user defined period
// and a width pulse of 1us.
assign trigger = (cnt_period== 1'b0 && nRST)? 1'b1 : 1'b0;

endmodule
module wr_rd//Module
name(.CLK(),.nRST(),.nWR(),.nRD(),.nCS(),.DATA(),.PARAM());
//Parameter declaration-------------------
#(
  parameter WID_D=8, //Data Width
      DEF_D=0  //Register Reset Value
);
//Parameter definition
//Port declaration---------------------
//Input port----------------------------
//Clock
//Reset
//Allow
//Data and address
//Other control lines
//Output port---------------------
//Clock
//Reset
//Allow
//Data and address
//Other control lines
);
//Port Definition

//Internal signal declaration--------------
always @(posedge CLK or negedge nRST)
begin
  if (!nRST)
  begin
    PARAM<=DEF_D;
    end
  else if (!(!nCS) && (!nWR))
  begin
    PARAM<=DATA;
    end
end

assign DATA=((!nCS) && (!nRD))?PARAM:{WID_D{1'bZ}};
endmodule
7.9 decoder_16.v
//-----------------------------------------
//  20050322
//Function Description:
//  4-32Decoder
//Signal description:
//  ADDR:Enter the address line
//  nRW_EN:Output write enable, active low
//  nCS:Strobe signal, active low
//  
//-----------------------------------------
module decoder16(ADDR,nCS,nRW_EN);
  input [3:0] ADDR;
  input nCS;
  output [15:0] nRW_EN;
  reg [15:0] nRW_EN;
  always @(ADDR or nCS)
    begin
      if(nCS)
        nRW_EN=16'hffff;
      else
        case(ADDR)
          4'h0: nRW_EN=16'hfffe; // nCS[ADDR_PAR_REG] bit
          4'h1: nRW_EN=16'hfffd; // nCS[ADDR_PAR_REG] bit
          4'h2: nRW_EN=16'hfffb; // nCS[ADDR_PAR_REG] bit
          4'h3: nRW_EN=16'hfff7; // nCS[ADDR_PAR_REG] bit
          4'h4: nRW_EN=16'hffef; // nCS[ADDR_PAR_REG] bit
          4'h5: nRW_EN=16'hffdf; // nCS[ADDR_PAR_REG] bit
          4'h6: nRW_EN=16'hffbf; // nCS[ADDR_PAR_REG] bit
          4'h7: nRW_EN=16'hff7f; // nCS[ADDR_PAR_REG] bit
          4'h8: nRW_EN=16'hfeff; // nCS[ADDR_PAR_REG] bit
          4'h9: nRW_EN=16'hffb; // nCS[ADDR_PAR_REG] bit
          4'ha: nRW_EN=16'hffbf; // nCS[ADDR_PAR_REG] bit
          4'h0: nRW_EN=16'h7fff; // nCS[ADDR_PAR_REG] bit
          4'h1: nRW_EN=16'h0fff; // nCS[ADDR_PAR_REG] bit
          4'h2: nRW_EN=16'hfff; // nCS[ADDR_PAR_REG] bit
          4'h3: nRW_EN=16'hff7f; // nCS[ADDR_PAR_REG] bit
          4'h4: nRW_EN=16'h7ff; // nCS[ADDR_PAR_REG] bit
          4'h5: nRW_EN=16'h0ff; // nCS[ADDR_PAR_REG] bit
          4'h6: nRW_EN=16'h0f7; // nCS[ADDR_PAR_REG] bit
          4'h7: nRW_EN=16'h0f; // nCS[ADDR_PAR_REG] bit
          4'h8: nRW_EN=16'h0; // nCS[ADDR_PAR_REG] bit
          4'h9: nRW_EN=16'hf0; // nCS[ADDR_PAR_REG] bit
          4'ha: nRW_EN=16'hf0f; // nCS[ADDR_PAR_REG] bit
        endcase
      end
endmodule
7.10 decoder_32.v

///////////////Signal description////////////////////
// ADDR: Enter the address line
// WR_EN: Output write enable, active low
// nCS: Strobe, active low
///////////////////////////////////////////
module decoder32(ADDR,nCS,nRWEN);

input [4:0] ADDR;
input nCS;
output [31:0] nRWEN;
reg [31:0] nRWEN;

always @(ADDR or nCS)
begi
  if(nCS)
    nRWEN=32'hffffffff;
  else
case(ADDR)
  5'h00:
    nRWEN=32'hffffffff; 5'h10:
  5'h01:
    nRWEN=32'hfffeffff; 5'h11:
  5'h02:
    nRWEN=32'hffffdfff; 5'h12:
  5'h03:
    nRWEN=32'hffffbfdf; 5'h13:
  5'h04:
    nRWEN=32'hffff7fff; 5'h14:
  5'h05:
    nRWEN=32'hffeeffff; 5'h15:
  5'h06:
    nRWEN=32'hffdfdfdf; 5'h16:
  5'h07:
    nRWEN=32'hffbbffff; 5'h17:
  5'h08:
    nRWEN=32'hff7f7ffe; 5'h18:
  5'h09:
    nRWEN=32'hfeefefef; 5'h19:
  5'h0a:
    nRWEN=32'hfd07dfe; 5'h1a:
  5'h0b:
    nRWEN=32'hf7ff7ffe; 5'h1b:
  5'h0c:
    nRWEN=32'hefefeef; 5'h1c:
  5'h0d:
    nRWEN=32'hdf07dfe; 5'h1d:
  5'h0e:
    nRWEN=32'hbf7ffeef; 5'h1e:
  5'h0f:
    nRWEN=32'hf7fffff; 5'h1f:
endcase
end
endmodule
module decoder64 // Module name decoder64(.ADDR(), .nCS(), .nRWEN());
{
    // Allow
    input  nCS,
    // Data and address
    input [5:0] ADDR,
    // Output permit
    output [63:0] nRWEN
    );

    // ----------------- Internal signal declaration -------------------
    wire nCS0, nCS1;

    assign nCS0 = nCS | ADDR[5];
    assign nCS1 = nCS | (!ADDR[5]);

    decoder32 dec0 (.ADDR(ADDR[4:0]), .nCS(nCS0), .nRWEN(nRWEN[31:0]));
    decoder32 dec1 (.ADDR(ADDR[4:0]), .nCS(nCS1), .nRWEN(nRWEN[63:32]));

    endmodule
7.12 frequency_sweep.v

`define min_FREQ 1.9e9
`define max_FREQ 18e9
`define min_DELTA 10e3
`define max_DELTA 10e6

module frequency_sweep
(clk,nRST,start,trigger,inicial_f,final_f,delta,RF_freq,LE);
//Parameter declarations
parameter length_reg_f = 21;
parameter length_reg_delta = 10;
// Ports declarations
input clk,nRST,start,trigger;
// Input signal inicial_f has to have a length of 21 bits,
// since the min frequency is 1.9GHz and the max frequency is 18GHz,
// with a delta of 10kHz, for the max freq supose a value of 1.610.000
// that is represented with 21 bits.
input [length_reg_f - 1 : 0] inicial_f,final_f;
// Input signal delta has to have a length of 10 bits,
// since the min delta is 10KHz and the max delta is 10MHz,
// with a step of 10kHz, for the max freq supose a value of 999
// that is represented with 10 bits (2^10 -1 = 1023).
input [length_reg_delta - 1 : 0] delta;
output reg[ length_RF-1 : 0] RF_freq;
output reg LE;

reg start_le;
reg [length_RF-1:0] RF_freq_aux;
wire trig_pos_edge,e_cnt;
reg trigger_aux;
integer i;

// start_le signal indicate when the start signal is detected
// Once the start signal is detected, start signal is ignored until
// sweep frequency process is finished
always @ (posedge clk)
begin
  if (!nRST)
    begin
      start_le <= 1'b0;
    end
  else
    begin
      // start_le condition is used to ignore the start signal
      //if it has already been detected
      if(start == 1'b1 && start_le == 1'b0)
        start_le <= 1'b1;
      // If a trigger arrived an the next frequency step is
      // bigger the final_f
      // it's means that the sweep frequency process has
      finished.
      else if( e_cnt && (RF_freq_aux + delta > final_f))
        start_le <= 1'b0;
    end
end

// Process responsible to register trigger in order to detect the
// rising edge
always @(posedge clk)
begin
  if (!nRST)
    trigger_aux <= 1'b0;
  else
    trigger_aux <= trigger;
end

// trig_pos_edge indicates a raising edge of trigger signal
// It's necessary in order to create a pulse of a clock duration
and a1(trig_pos_edge,trigger,!trigger_aux);

// e_cnt signal enables sweep frequency, if there is a trigger and
// start signal is detected
// then sweep frequency process could work.
and a2(e_cnt,trig_pos_edge,start_le);

// Sweep frequency process
always @(posedge clk)
begin
  if (!nRST)
    begin
      i <= 1'b0;
      RF_freq_aux <= inicial_f;
      LE <= 1'b0;
    end
  else
    begin
      // If trigger and start signals are set properly, sweep
      frequency has to start
      if(e_cnt)
        begin
          // If the next is bigger than final_f, sweep
          // frequency process has to stop
          // Furthermore, is used to avoid undesired situation:
          // final_f = ini_f -> Process stays quiet.
          // the steps between ini_f and final_f are not
          // multiple of delta, then sweep frequency
          // process is the lowest immediate multiple
          if(RF_freq_aux + delta > final_f)
            begin
              LE = 1'b0;
              i = 1'b0;
              RF_freq_aux = inicial_f;
            end
          else
            begin
              // Otherwise increase in user defined delta
              RF_freq_aux = inicial_f + i * delta;
              i = i + 1;
              LE = 1'b1;
            end
        end
    end
end

// Register the output signal
// LE signal is set from the start process to the end
always @(posedge clk)
begin
  if (LE)
    RF_freq <= RF_freq_aux;
  else
// Otherwise the RF_freq is the initial frequency
    RF_freq <= inicial_f;
end
endmodule
8 References


