DESIGN AND VALIDATION OF A DIGITAL CORE FOR WIRELESS COMMUNICATION WITH RFID-ENABLED DEVICES

A THESIS IN
Electrical Engineering

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment of
the requirments for the degree

MASTER OF SCIENCE

by
KUMAR SWAMY HOSUR SATYAMURTHY

B.E., Visvesvaraya Technological University, 2009

Kansas City, Missouri
2011

DESIGN AND VALIDATION OF A DIGITAL CORE FOR WIRELESS COMMUNICATION WITH RFID-ENABLED DEVICES

Kumar Swamy Hosur Satyamurthy, Candidate for the Master of Science Degree

University of Missouri-Kansas City, 2011

ABSTRACT

A digital core was designed and successfully tested to communicate with RFID-enabled devices. This digital core comprises of a communication block and a control block. The communication block is made up of two blocks, namely, the receiver block and the transmitter block whereas the control block controls the operation of the digital core. VHDL (Very High Speed IC Hardware Description Language) was used to design the digital block according to the ISO15693 standard specified for RFID-enabled devices. The developed VHDL code was successfully simulated and the design was transferred onto a CPLD. A test circuit was developed on a bread board with the CPLD and a commercially available RFID reader was used to successfully communicate with the designed digital core. Then a printed circuit board (PCB) was designed to validate the above design with all components including the CPLD for communication. This PCB meets all the specifications provided for communication by the ISO15693 standard. The developed digital core can be translated into a layout which can be part of an implantable or embedded RFID passive sensors chip.

The faculty listed below, appointed by the Dean of the School of Computing and Engineering have examined a thesis titled "Design and Validation of a Digital Core for Wireless Communication with RFID-enabled Devices, " presented by Kumar Swamy H.S., candidate for the Master of Science in Electrical Engineering degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Walter D. Leon-Salas, Ph. D., Committee Chair
Department of Computer Science and Electrical Engineering

Ghulam Chaudhry, PhD.,
Department of Computer Science and Electrical Engineering

Deb Chatterjee, Ph. D.
Department of Computer Science and Electrical Engineering

CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

Figure Page

LIST OF TABLES

ACKNOWLEDGEMENTS

First, I would like to thank my parents, aunt and my late uncle for their support and encouragement which has inspired me to come all the way to graduate school. I always admire you for your will to achive excellence in life. The values taught by you are taking me a long way and I am sure one day it will help me to reach my goal. Thank you dear sisters for your love and affection which brings a smile on my face even during the toughest of time. I would like to thank my dad Professor H N Satyamurthy for supporting me financially without which none of these would be possible. I could not have continued taking classes or have completed my thesis without their support.

My thesis advisor, Prof. Walter D. Leon-Salas, has been a great help in directing my research work, and suggested a thesis topic that I continue to enjoy. He has worked on a couple of research projects that he has allowed me to be a part of, and I have learned a lot that is sure to be helpful to me in whatever career path I end up taking. I also appreciate the graduate research positions that I have had with my advisor, as well as other various graduate teaching positions in the CSEE department, that have allowed me to stay focused on my studies.

I would also like to thank Prof. Ghulam Chaudhry and Prof. Deb Chatterjee who are on my thesis committee for their support and for everything that they have taught me in the multiple courses that I have had with them. I would like to acknowledge helpful conversations that I have had with my fellow colleagues who have done excellent job in enlightening me about VHDL, microcontrollers, circuit design and many other topics which helped me to get a good start of my thesis.

CHAPTER 1

INTRODUCTION

1.1 RFID Enabled Sensors

In recent years Radio Frequency Identification (RFID) technology have become very popular in many service industries, purchasing and distribution logistics, manufacturing companies and many more. RFID systems provide information about products, goods, people and animals, to name a few. The use of barcode labels which revolutionized identification systems a few years ago are now being found to be inadequate in increasing numbers of cases due to their line of sight requirement, their low storage capacity and inability to be reprogrammed. To overcome this problem RFID-based solutions have been proposed which use electronic data storage devices to store large amounts of data. They can be reprogrammed and do not require line of sight. Moreover, RFID systems provide contactless transfer of power and data from the reader to the data storage device [1]. Because of this unique property, namely battery is not a requirement, RFID technology has gained popularity. A RFID system is made up of basically two components a RFID reader and an RFID tag.

Electronic sensing is another technology which is seeing exponential growth across various industries. Electronic sensors measure physical quantities and convert them into a electric signal which can be used to understand the behavior of a physical quantity or the environment in which it is placed [2]. The ability of the sensor to detect the smallest change is often very critical to the application in which it is used, called as resolution of the sensor. Sensors are classified based on their applications such as pressure sensors, temperature sensors, biosensors, etc.; some of these sensors are passive sensors which do not require a battery for operation. The industry is moving towards implementation of networks of wireless sensors that can operate in demanding environment and provide clear advantages in cost, size, power, flexibility and ability to re-program.

From the above discussion we can see that RFID technology along with sensors can be used to address many challenges in the field of engineering and medicine. In the past two decades many industries have embraced RFID enabled sensors to provide quality service to their customers. RFID-enabled sensors can be classified based on power consumption into two type's active and passive sensors. An active RFID sensor requires a battery to operate. While this increases its communication range it limits their application only where battery replacements are possible and affordable [3]. In applications where battery replacement is not possible such as implantable sensors, sensors embedded in concrete and construction material it is desirable to use passive RFID sensors. Passive RFID sensors are less complex and hence very economical, they use passive circuits which have no internal power source and use backscattering techniques to transmit data back to the reader. Passive RFID sensors have many advantages such as low power consumption, low cost, small size, flexibility, durability, long shelf life and are easy to control. With these many advantages they have unlimited applications in consumer goods, healthcare system, engineering and others industries.

This thesis is part of a research effort that sims at developing a passive RFID-enabled sensor which can be used in implantable applications. Such a sensor will have two parts the analog core and the digital core. The analog core will be used to generate the required voltage levels to power circuit from the incoming RF input and also to convert the analog signals from the sensor to digital signals which can be easily sent back to the reader. The function of the digital core is to decode the incoming signal, execute commands and send the data back to the reader. The control logic required to communicate with the reader will be performed by this digital core. In this thesis work we have developed a digital core that complies with the ISO15693 standard and can be used in RFID sensors for different applications. A testbed to test the digital core has also been developed.

1.2 Prior Work

In the past decade, a plethora of RFID sensors related research have been carried out especially after the influential and most widely-cited paper by Udo Karthaus [4], in which a fully integrated passive RFID transponder IC with very low RF input power was designed. Research in this field has been galvanized in recent years due to advances in technology coupled with demand in numerous applications and exponential adoption for commercial uses [5].

One of the earliest works in the field of RFID tag was done by Ulrich Kasier et al [6] in 1995 who have presented an integrated circuit for a battery-less transponder system. The system works in half-duplex mode. First, the energy is transmitted to and stored in the transponder. A data telegram is sent back to the reader unit. This technique allows for greater reader distance over simple receivers that use full-duplex systems. The transponder consists of a LC tank, a supply capacitor and a transponder circuit. The LC tank serves as the RF interface to receive energy from the reader unit and to send back the data. The supply capacitor stores the charge during supply first phase and supplies current in second phase. The antenna was able to get an operating range of up to 2 m, depending on the size of the antenna and allowable field strength. The main drawback of this design is the use of LC tank. Tuning an LC circuit for the desired frequency is complex and also damping is a major factor to worry. Aging will have the most effect on damping of the LC tank, as damping increases

LC tank will discharge very quickly and the circuit might not be able to complete the transmission of data. Because of these factors it is not reliable to use this design in implantable applications.

After the work by Ulrich Kaiser another work by Qiuting Huang et al [7] is worth mentioning. In 1998 they came up with a low-power, single-chip, one-channel, fully implantable microtransponder system for low-frequency biomedical sensors application. This circuit is powered externally by an RF source of 27/40 MHz and the chip is battery less. The RF power received by the circuit via a small antenna is in millivolts range so the data acquisition and transmission systems are optimized for low power consumption. The on-board sensors communicate wirelessly with the external monitoring units through backscatter modulation. The sensor and the transponder are implanted. Power and communication is provided via coupled loop antennas between the transponder and a mobile interrogation unit (MIU). The microtransponder consists of two main modules: the data acquisition unit (DAU) and the RF/DC converter (RFC). The data acquisition unit (DAU) is an interface to the sensor and it does all the pre-processing of sensor signals. The RF/DC converter provides wireless powering and communication. The signals from the sensors are converted to low-duty-cycle pulse-position-modulation (PPM) signals and amplitude modulated for transmission. The main disadvantage of this design is the use of amplitude modulated (AM) signals which are highly sensitive to noise. Once the transponder is implanted the AM signals will have to travel through the skin, blood and other muscles inside the body which will greatly attenuate the AM signal decreasing the signal to noise ratio.

In the past 10 years technological advancement has been faster than ever and thus the quality and quantity of research work done on RFID sensors has increased. Because of this now we have numerous applications using RFID sensors.

Udo Karthaus and Martin Fischer have developed a fully integrated passive UHF RFID transponder IC [4]. The reading range is around 4.5 m at 500 mW base-station transmit power operating in the 868/915 MHz band. The IC includes DC power supply generation, phase shift keying backscatter modulator, pulse width modulation demodulator, EEPROM and logic circuitry to handle protocol for wireless write and read access to the IC's EEPROM and for anti-collision. The DC power supply generation unit is a Schottky diode-based voltage multiplier circuit which allows for a high efficiency conversion of received RF input signal energy in to DC supply voltage. Use of UHF band frequencies is the main cause of concern of this design, line-of-sight is required for communication, atmospheric moisture, physical obstructions and time of the day affect the signal transmission and degradation of signal reception. UHF signals are greatly attenuated by moisture than lower frequencies. Therefore the sensor circuit should consume little power and have considerable reading distance for implantable applications.

Another interesting work has been done by Vijay Pillai et al [8] who designed a fully-integrated dual-battery and passive RFID tags which works in both UHF and microwave bands and consumes low power of 700 nA at 1.5 V. This design uses a small 100 mA.hr capacitive battery for lifetimes exceeding ten years. But the use of battery and UHF band prohibits this design from use in implantable applications.

Most recent work in the field of passive RFID sensors has been done by Daniel Yeager et al [5]. They have developed an addressable sensor tag for bio-signal acquisition. This sensor tag is a fully passive 900 MHz RFID tag IC with addressability. It contains a 1.25 µV rms integrated noise chopper-stabilized micro-power sensor interface amplifier and an 8-bit ADC. The communication range with the off-the-shelf reader is around 3m, enabling previously impossible recording scenarios like in-flight recording from small insects [5]. For accurate signal amplification and digitization precise supply and reference voltages are provided by the ultra-low-power linear regulators, bandgap reference and bias current generator. The sensor signals are first amplified using a low-noise chopper-stabilized amplifier then digitized using an 8-bit successive approximation ADC (SAR). Then random numbers are used for anti-collision and encryption of data. Finally the controller logic encodes all the data in to a Gen 2-compatible packet in response to reader commands. This design cannot be used for implantable applications because of the use of 900 MHz RF signals which greatly attenuates due to moisture.

Melexis, one of the leading manufacturers of smart mixed-signal ICs, have come up with a wireless sensor tag, the MLX90129, which works in the 13.56 MHz band. This IC provides a precise acquisition chain for external resistive sensors with a wide range of interface possibilities. It can be accessed via a front-end RFID interface or through a SPI port. It is designed to work as a temperature sensor with no other components connected to it. It has an option to add a battery. With SPI port connectivity there are numerous application possibilities. This IC is designed for low-power, low-voltage battery or battery-less applications and it is mounted on a printed cirucit board (PCB). This PCB having so much capability cannot be used for implantable applications because the size of the PCB is around

1 cm^2, which is large for implantation, also the materials from which the PCB is manufactured might prove to be hazardous for implantable applications.

1.3 Plan of Development

The development of the digital core is suitable for any implantable application and can be easily modified to be used with most of the readers available in the market. To achieve this we had to follow a communication standard, so we decided to follow the ISO15693 standard for contactless integrated circuits/ vicinity cards. The main reason for choosing this standard is because it works at high frequency band (13.56 MHz carrier frequency), this has better read range and data rate compared to low frequency signals and its ability to read objects with water and metal content is better than ultra high frequency signals. The main working principle of the RFID-enabled sensors and the standard followed to develop the digital core are described in chapter 2. Chapter 3 will focus on the detail description of the digital core. Working principle and design techniques of every module in the digital core is described in this chapter. Chapter 4 will discuss about testing the designed digital core using two methods and the obtained results. Summary and conclusions drawn from the results, future work are discussed in chapter 5.

CHAPTER 2

BACKGROUND

2.1 Introduction

The main idea behind the passive RFID-enabled sensors is the use of radio frequency (RF) for contactless power supply and communication with an integrated circuit. These RFID-enables sensors mainly consist of two basic components: a RFID reader and an RFID tag. The block diagram of a generalized RFID communication system is shown in Fig 2.1.

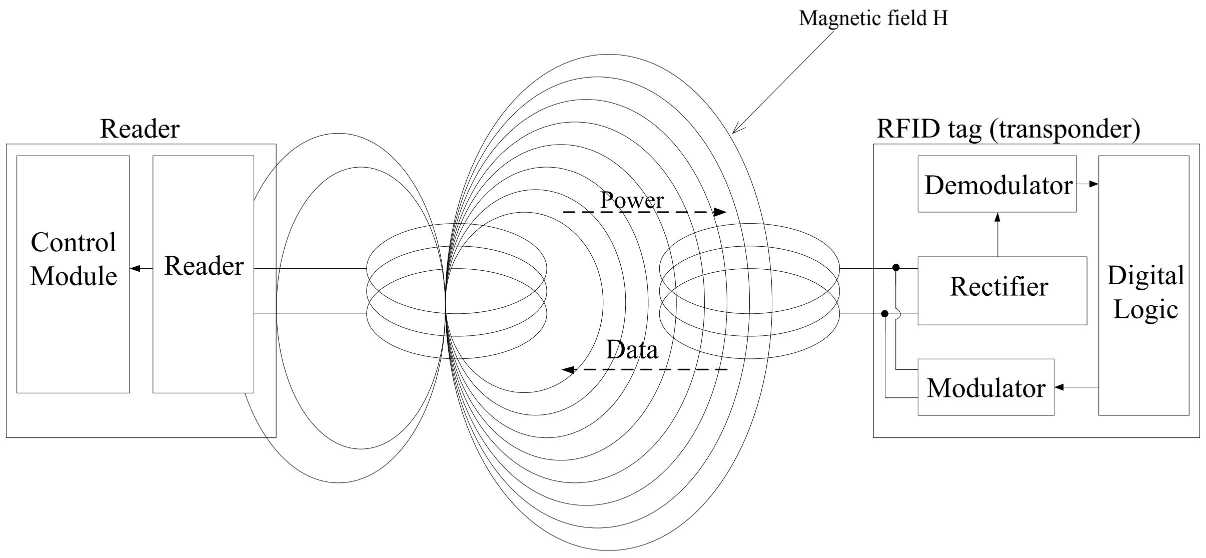


Fig 2.1: RFID Contactless Communication System [5].

This RFID reader can read and/or write data in the RFID tag depending on the application in which it is used. The RFID reader generally consists of a transmitter, receiver, coupling element and a control unit whereas the RFID tag consists of a coupling element and an electronic microchip. This electronic microchip is a mixed signal IC where both analog core and the digital core are used in conjunction to communicate with the reader. The RFID tag is activated only when it comes in the interrogation range of the reader during which the power required to activate the tag is supplied through the coupling element of the tag.

2.2 RFID Tag with Sensor Interface

The RFID tag described in the previous section has two parts to it the analog core and the digital core. The analog core is responsible for stable power supply to the RFID tag whereas, the digital core process the incoming digital signal, performs required function and sends back a response signal to the reader.

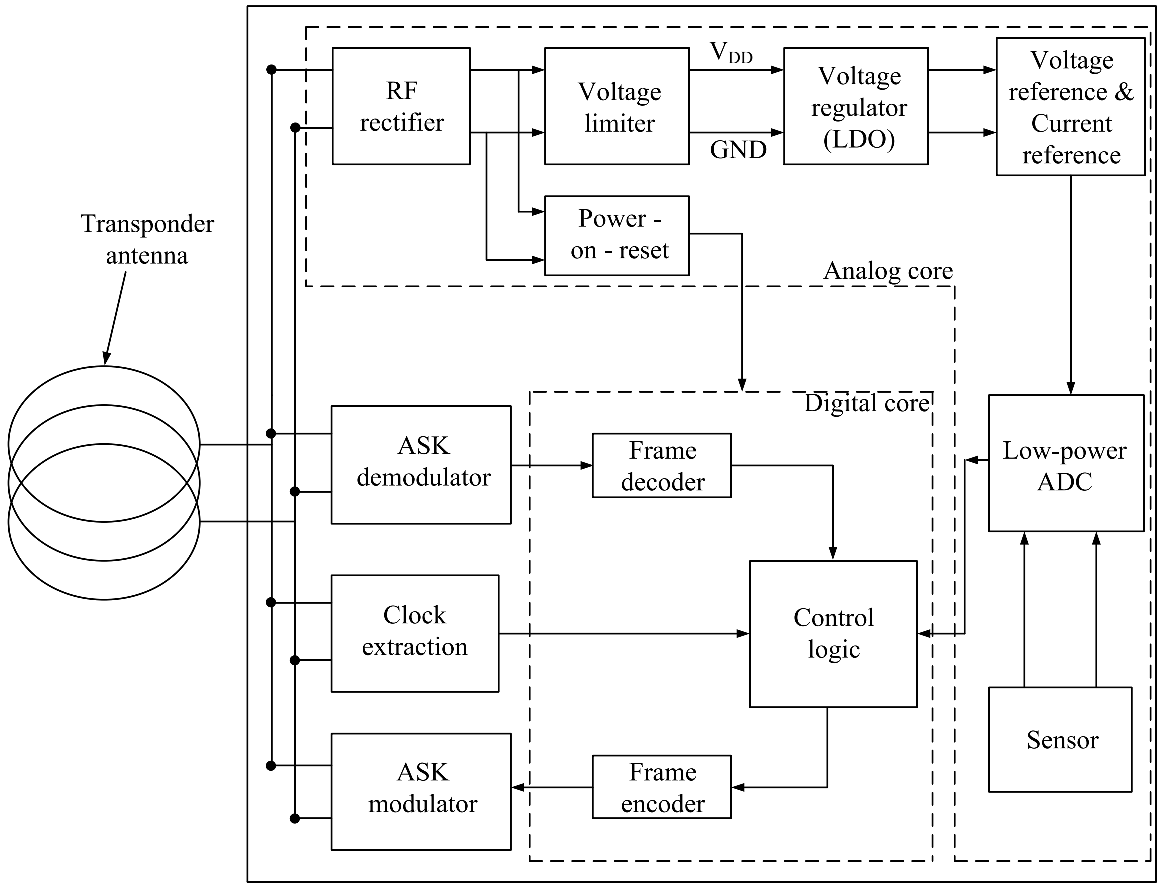


Fig 2.2: Block diagram of the RFID Tag

From the above block diagram we can see that the power rectifier, voltage limiter, voltage regulator, voltage rectifier, ASK modulator & demodulator, clock extraction unit, low-power ADC and sensor forms the analog core of the RFID tag and the frame decoder, frame encoder & control logic forms the digital logic. The digital logic is the crucial component of the tag which communicates with the tag. In the analog core the power rectifier is a full wave rectifier which rectifies the incoming RF signal and sends it to the limiter. The voltage limiter is designed to protect the tag from incoming signal which might spike up suddenly to high voltage (as much as 30v) which the tag cannot withstand. Voltage regulator is used to provide stable voltage supply to the entire tag as the incoming signal varies in strength. The voltage reference provides required references to different analog blocks of the tag. The long term goal of this thesis is to use this digital core for implantable application with sensors, so we have given provision to use a low-power ADC to convert the incoming analog signal from a sensor into a digital signal. Then this digital signal can be encoded in a packet and sent to the reader. The digital core is made up of a frame decoder, control logic and a frame encoder. The frame decoder extracts information from all the fields of the incoming data packet, control logic then takes action based on the information in the incoming packet and the frame encoder will generate a data packet to respond to the reader which is sent out using load modulation. Detailed explanation of the digital core and the prototype designed to use with this has been explain in the next chapter.

For communicating with the RFID tag, the reader uses many different protocols defined by the international organization for standardization (ISO), in this thesis we chose to use the ISO15693 standard. To understand the design of the above shown block diagram and its components we have to get to know the protocols used in the ISO15693.

2.3 ISO15693

The international organization for standardization widely known as ISO is an international standard setting body composed of representatives from various national standards organizations. This organization promulgates worldwide proprietary industrial and commercial standards. Many different standards have been defined for RFID applications. We chose to use the ISO 15693 standard for vicinity cards, i.e. cards which can be read from a greater distance as compared to proximity cards. The reasons for choosing this ISO standard is because they operate at 13.56 MHz which corresponds to high frequency range. Low frequency signals which operate at 125 KHz have better ability to read objects with water or metal content but they suffer from low read range and low data rate, on the flip side the Ultra high frequency signals which operate at 860 – 930 MHz band have very high read range and high data rate but the signal is attenuated by atmospheric moisture. Physical obstructions and time of the day affect the signal transmission and degrades signal reception. Considering all these factors we choose to use ISO15693 which uses 13.56 MHz carrier frequency, this has better read range and data rate compared to low frequency signals and its ability to read objects with water and metal content is better than ultra high frequency signals. Also the effect of environmental conditions on this signal is less than ultra high frequency signals.

The communication between the reader and the tag occurs through the following sequence as defined by the above ISO15693 standard:

1) Activation of the RFID tag by the RF operating field of the reader.
2) Transmission of the command by the reader.
3) Transmission of the response by the tag.

This technique is called Reader talk first. In this reader talk first technique, initially the tag is activated by the RF field of the reader. Then the tag waits for a communication signal from the reader called the request data packet, once this packet arrives depending on the request from the reader the tag will send a response data packet. The format of the request and response data packets which includes the data rate, encoding schemes, packet delimiters are all defined by the ISO15693 standard. In the following sections we will discuss the specified formats for both request and response data packets.

2.4 Communication from Reader to Tag

The communication signal used for reader to tag uses different modulation, data coding techniques and different packet format. The communication signal from the reader to tag uses modulation principle of ASK (amplitude modulation) with 10% or 100% modulation indices [9] [10]. The request data packet format used for the reader to tag consists of seven fields they are start of frame (SOF), request flags (Flags), command code, parameters, data, CRC and end of frame (EOF) as shown in Fig 2.3

SOF	Flags	Command code	Parameters	Data	CRC	EOF

Fig 2.3: General Request Format.

The SOF and EOF are packet delimiters which specify the start and end of a packet, the flags used in the second field indicates the use of subcarrier, data rate and other variables to the tag. The command code specifies the action to be performed by the tag for example to indicate if it's a read operation or a write operation. Parameters and data fields are optional and it depends on the command code. The CRC used is calculated as per the definition in

ISO13239. It's a two byte CRC and it is appended to each request and response packet, within each frame before the end of frame. The CRC is calculated on all the bytes after the SOF up to the CRC field. To understand the read data command which our digital tag uses initially we have to see the description of each field of the general request format. Following sections will discuss every field in-detail.

2.4.1 Start of Frame (SOF) and End of Frame (EOF)

The SOF defines the data coding mode the reader is to use for the following command frame. The SOF sequence described in Fig 2.4 selects 1 out of 4 data coding mode. The EOF for either of the coding modes is described in Fig 2.5 [9] [10].

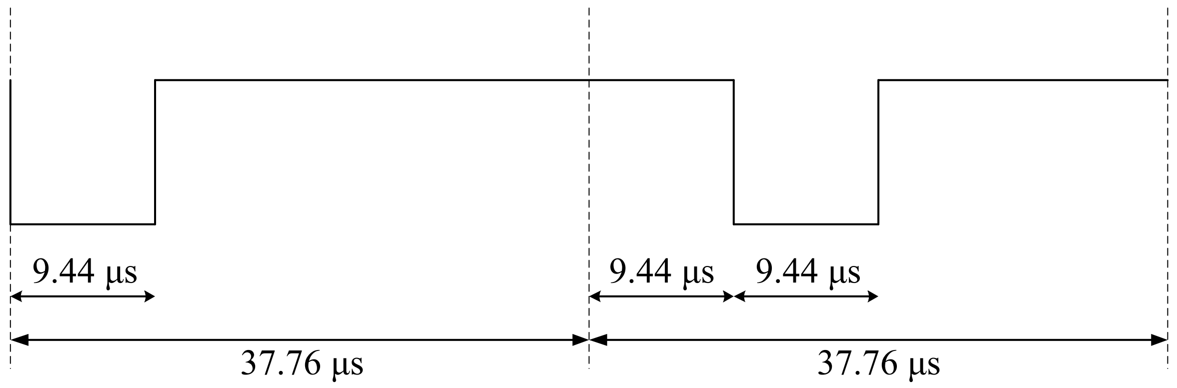


Fig 2.4: SOF to select 1 out of 4 data coding mode.

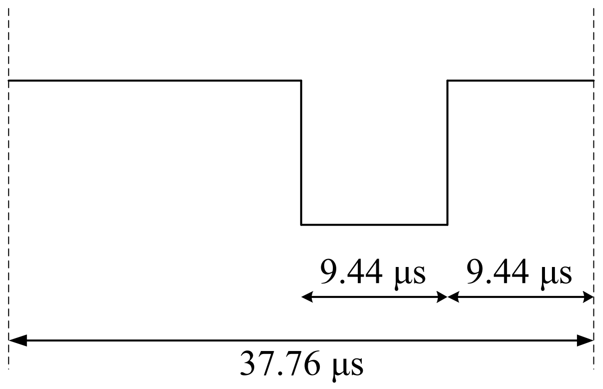


Fig 2.5: EOF for either data coding mode.

2.4.2 Data Coding and Data Rate

The ISO15693 specifies two data coding rate for the signals coming from the reader to the tag they are 1 out of 256 and 1 out of 4. The selection of data coding rate and modulation index is done by the reader and indicated to the tag within the Start of Frame (SOF). In this thesis we use 1 out of 4 data coding method where the value of 2 bits is represented by the position of one pause. The position of pause on 1 out of 4 successive time periods of 18.88 μs ($256/f_{c)}$, determines the value of 2 bits. Four successive pairs of bits form a byte, where the least significant pair of bits is transmitted first. So, transmission of one byte takes 302.08 μs and the resulting data rate is 26.48 Kbits/s ($f_c/512$) [10]. Fig 2.6 illustrates the 1 out of 4 coding example for transmission of E1h (1110 0001b) by the reader and Fig 2.7 illustrates the technique of pulse position coding. The flags, command, data and CRC fields are sent using this data coding method.

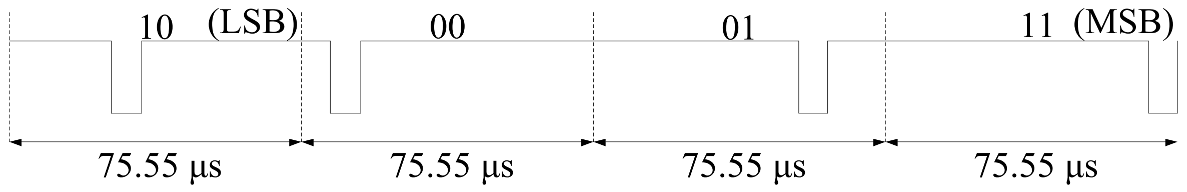


Fig 2.6: One out of Four coding example.

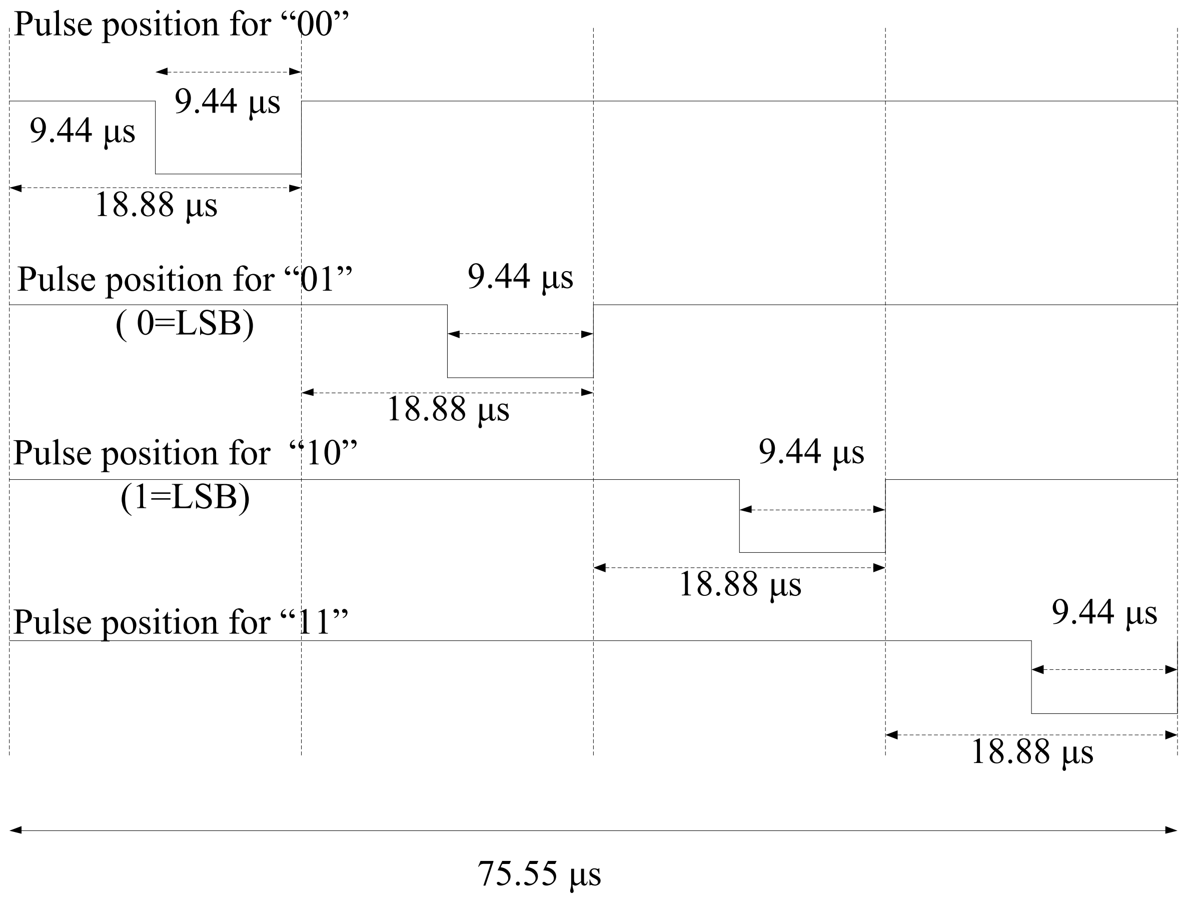


Fig 2.7: One out of Four Data coding mode.

2.4.3 CRC

The CRC used in the communication between the reader and the tag is defined as per the definition in ISO13239 as shown in Fig 2.8 [9]. The cyclic redundancy check (CRC) is calculated on all the data contained in a message from the SOF till the CRC field.

CRC type	Length	Polynomial	Direction	Preset	Residue
ISO13239	16 bits	$x^{16} + x^{12} + x^{5} + 1$	Backward	'FFFF'	'F0B8'

Fig 2.8: CRC specifications.

Upon reception of the request packet the tag verifies if the CRC value is valid, if it is invalid then it discards the frame and does not answer the reader. Similarly upon reception of the response packet CRC can be verified. As we can see from Fig 2.9 that CRC is 2 bytes long and it is transmitter least significant bit first and each byte is transmitter least significant byte first.

LSBit LSByte MSBit	LSBit MSByte MSBit
CRC 16 (8 BITS)	CRC 16 (8 BITS)

Fig 2.9: CRC Transmission rules [9].

2.4.4 Read/Write data Request Packet

As we have explained earlier we have designed our tag for read and write data operations. In the previous section we have seen the format for general request format now we shall see the request format for reading and writing data in the tag. To read/write data from the tag we have to specify the command code in the request packet. According to the ISO15693 standard 0x20/0x21 is the command code used to read/write data from the tag respectively. When the tag receives a read/write command the request is processed and in case of read command the tag sends back its 32 bits value or in case of write command it will report the success of the operation in response.

Request SOF	Request Flags	Read Single Block	Block number	CRC 16	Request EOF
	8 bits	0x20	8 bits	16 bits	

Fig 2.10: Read Single Block request format.

Request SOF	Request Flags	Write Single Block	Block number	Data	CRC 16	Request EOF
	8 bits	0x21	8 bits	32 bits	16 bits	

Fig 2.11: Write Single Block request format.

The request SOF and EOF are defined in section 2.4.1, the CRC 16 is discussed in section 2.43 and data in other fields are sent as shown in section 2.4.2.

The ISO command to request a read/write from the tag is Iso15 022001/ Iso15 4221data crc respectively where Iso15 specifies the ISO standard used, 02 (0000 0010)/ 42 (0100 0010) refers to the 8 bit data in the request flags field, 20 (0010 0000)/ 21 (0010 0001) refers to the 8 bit data in the read/write single block (command code) field and 01 (0000 0001) refers to the 8 bit data in the block number field. The request flag field is as shown in Table 2.1 it consists of eight bits. The bit 3 (Inventory flag) of the request field defines the content of the 4 MSB's (bits 5 to 8). When bit 3 is reset (0), bits 5 to 8 define the tag selection criteria and when bit 3 is set (1), bits 5 to 8 define inventory parameters [9] [10].

Table 2.1: Request Flags 1 to 4 definitions [9].

Bit	Request flag	Level	Definition
Bit 1	Subcarrier flag	0	A single subcarrier frequency shall be used by the tag
		1	Two subcarriers shall be used by the tag
Bit 2	Data_rate _flag	0	Low data rate is used
		1	High data rate is used
Bit 3	Inventory flag	0	Flags 5 to 8 meaning according to Table 2.1.1
		1	Flags 5 to 8 meaning according to Table 2.1.2
Bit 4	Protocol extension flag	0	No protocol format extension

Table 2.2: Request Fields 5 to 8 when bit 3 is reset (0).

Bit	Requested Flag	Level	Definition
Bit 5	Select Flag	0	Request shall be executed by the tag
		1	Request shall be executed by the tag in selected state
Bit 6	Address Flag	0	Request shall be executed by all tags, UID field absent
		1	Request is executed by matched tag, UID field present
Bit 7	Option Flag	0	
Bit 8	RFU	0	

Table 2.3: Request Fields 5 to 8 when bit 3 is set (1).

Bit	Requested Flag	Level	Definition
Bit 5	AFI Flag	0	AFI field is not present
		1	AFI field is present
Bit 6	Nb_slots Flag	0	16 slots
		1	1 slot
Bit 7	Option Flag	0	
Bit 8	RFU	0	

The flag field in the read/write command can be understood from the above tables 2.1, 2.2 and 2.3. As defined earlier the ISO command for read/write specifies 02 (0000 0010) for the flag field which means the reader uses and expects a data packet in single subcarrier frequency, high data rate and the request shall be executed by all the tags.

2.5 Communication from Tag to Reader

The RFID tag communicates with the reader via an inductive coupling area in which the carrier is loaded to generate a subcarrier with frequency f_s. The subcarrier is generated by switching in a load in the tag. The tag designed in this thesis supports one subcarrier response format. This is indicated by the reader using the first bit of the flag field in request packet. In this format the frequency f_s of the subcarrier load modulation is 423.75 kHz (f_c/32). The tag responds using high frequency data rate (26.48 Kbits/s i.e. f_c/512) format and this selection is again indicated by the reader in the second bit of the flag field in request packet. The response data packet consists of the following field's namely start of frame (SOF), response flags (Flags), parameters, data, CRC and end of frame (EOF) as shown in Fig 2.11.

SOF	Flags	Parameters	Data	CRC	EOF

Fig 2.12: General Response Format [9].

All the fields in the response format are defined as per the ISO15693 standard, we will discuss about their definition in the following sections. The data bits in the response format are encoded using Manchester coding and depend on the data rate indicated by the reader.

2.5.1 Bit Representation and Coding

In this thesis we have designed the digital tag to respond with one subcarrier using high data rate but, with small modification it can respond with low data rate. For high data rate, logic 0 and logic 1 bit representations are as shown below.

Logic 0 starts with 8 pulses of 423.75 kHz ($f_c/32$) followed by an unmodulated time of 18.88 μs as shown in Fig 2.13.

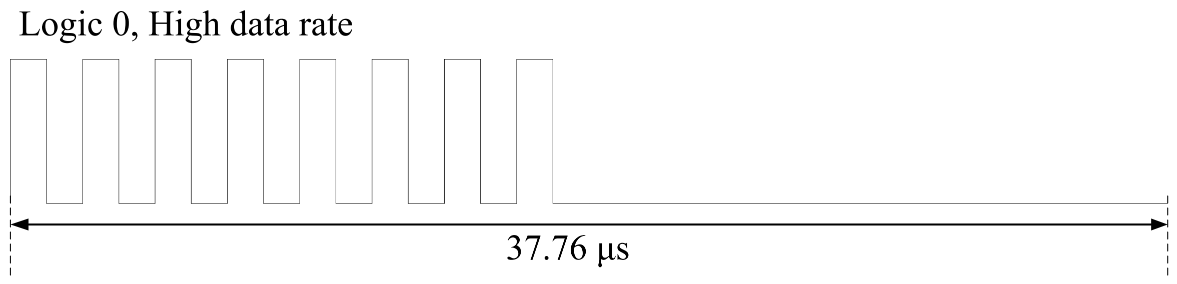


Fig 2.13: Logic 0 representation.

Logic 1 starts with an unmodulated time of 18.88 μs followed by 8 pulses of 423.75 kHz ($f_c/32$) as shown in Fig 2.14 .

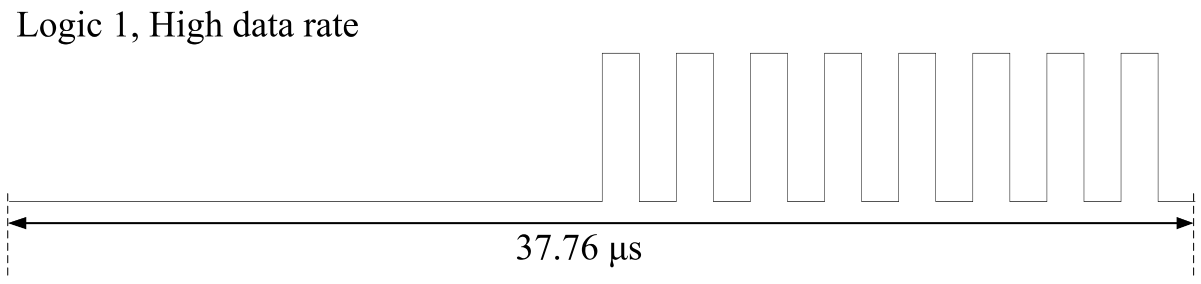


Fig 2.14: Logic 1 representation.

2.5.2 Start of Frame (SOF) and End of Frame (EOF)

The SOF for high data rate is made up of 3 parts namely, an unmodulated time of 56.64 μs followed by 24 pulses of 423.75 kHz and a logic 1 which also corresponds to the high data rate as shown below in Fig 2.15.

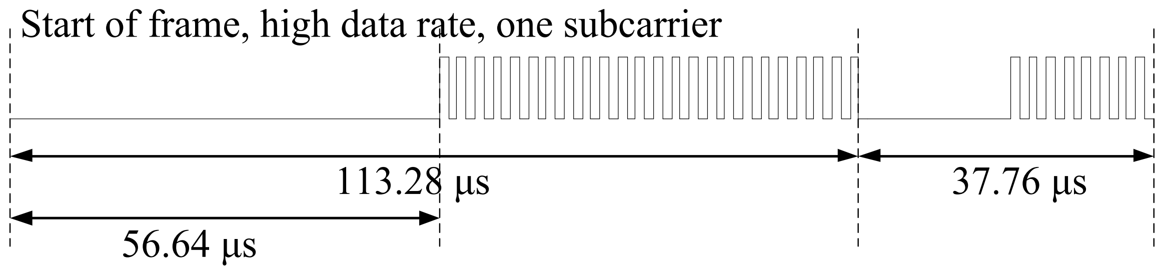


Fig 2.15: SOF Representation for response packet.

The EOF for high data rate is made up of 3 parts namely, a logic 1 which also corresponds to the high data rate followed by 24 pulses of 423.75 kHz and an unmodulated time of 56.64 μs as shown below in Fig 2.16.

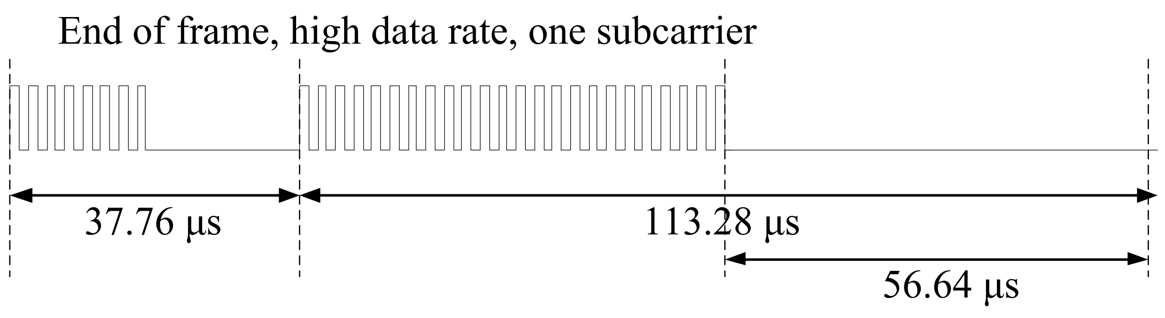


Fig 2.16: EOF Representation for response packet.

Flags field in the response packet is an 8 bit flag and it indicates to the reader the actions it has performed and whether other fields are present or not. The parameter and data fields are optional. The CRC is calculated as discussed in section 2.4.3.

2.5.3 Read/Write data Response Packet

The request format for read/write operations were explained in section 2.4.4 and the general response format were explained in section 2.5.1 and 2.5.2, now we shall see the response format for reading and writing data in the tag. To read/write data from the tag we have to specify the command code in the response packet. According to the ISO15693 standard 0x20/0x21 is the command code used to read/write data from the tag respectively. When the tag receives a read/write command the request is processed and in case of read command the tag sends back its 32 bits value or in case of write command it will report the success of the operation in response. The response format for a read and write operation is as shown in Fig 2.17 and Fig 2.18.

Response SOF	Response Flags	Data	CRC 16	Response EOF
	8 bits	32 bits	16 bits	

Fig 2.17: Read Single Block response format.

Response SOF	Response Flags	CRC 16	Response EOF
	8 bits	16 bits	

Fig 2.18: Write Single Block response format.

The response SOF and EOF are defined in section 2.5.2, the CRC 16 is discussed in section 2.43 and data in other fields are sent as shown in section 2.5.1.

The response format a read/write from the tag for Iso15 022001/ Iso15 4221 data crc respectively uses 00 (0000 0000) 8 bit data in the response flags field. The response flag field is as shown in Table 2.2 and it consists of eight bits [9] [10].

Table 2.4: Response flags 1 to 8 definitions [9].

Bit	Request flag	Level	Definition
Bit 1	Error flag	0	No error
		1	Error detected
Bit 2	RFU	0	
Bit 3	RFU	0	
Bit 4	Extension flag	0	No extension
Bit 5	RFU	0	
Bit 6	RFU	0	
Bit 7	RFU	0	
Bit 8	RFU	0	

The flag field in the read/write command can be understood from the above tables 2.4, as defined earlier the ISO command for read/write specifies 00 (0000 0010) for the flag field which means no error was detected by the tag and the operation was successful. If error flag is set by the tag in the response then error code field is present but in our design we do not respond to errors, if error is detected then the packet is dropped and the tag waits for another request from the reader.

2.6 RFID Tag Transmission Protocol

The ISO15693 standard defines the transmission protocol for exchange of instructions and data between the reader and tag. As explained in section 2.3 it is based on the concept of reader talk first, this means the tag does not start transmission unless it has received and properly decoded an instruction sent by a reader. The protocol is based on an exchange of request packet from the reader to tag and a response packet from the tag to the reader. Each request and response is contained in a frame. The frame delimiters SOF and EOF are explained in-detail in the previous sections. The protocol is bit-oriented, the number of bits transmitter in a frame is a multiple of eight. A single–byte field is transmitted Least significant bit first. A multiple-byte field is transmitted Least significant byte first and each byte is transmitter Least significant bit first. The settings of the flags fields in the packets indicate the presence of optional fields. When the flag is set, the field is present else if flag is reset then the field is absent [9]. The protocol timing defined by the ISO15693 states that the response frame for any request frame should be sent to the reader after a certain time t1 and consecutive request frames are sent in intervals of certain time. The protocol timing diagram is shown below in Fig 2.19 and the timing specifications are summarized in Table 2.5.

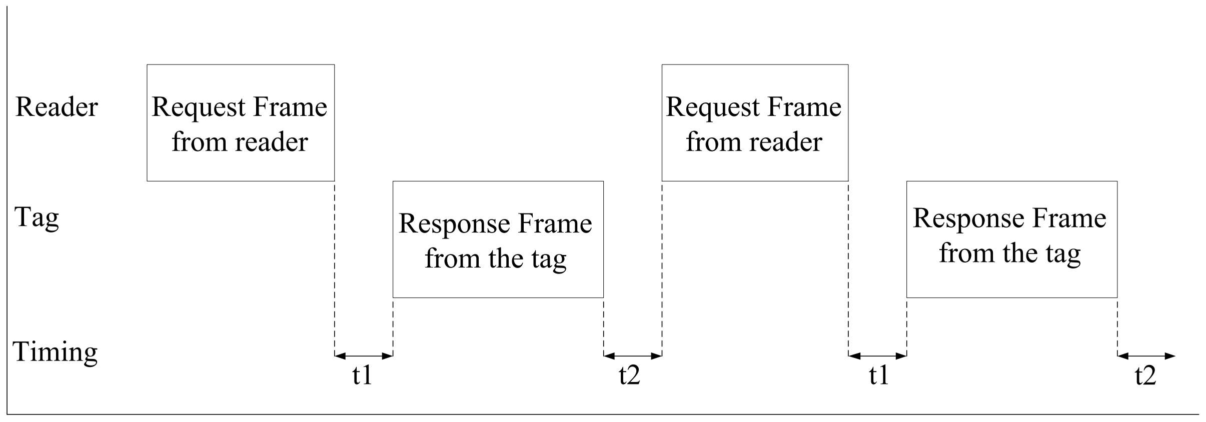

Reader
Tag
Timing
Request Frame from reader
Response Frame from the tag
Request Frame from reader
Response Frame from the tag
t1
t2
t1
t2

Fig 2.19: Transmission Protocol Timing.

Time	Minimum	Nominal	Maximum

t1	$\frac{4192}{\text{fc}} = 309.02\ \mu s$	$\frac{4224}{\text{fc}} = 311.5\ \mu s$	$\frac{4256}{\text{fc}} = 313.9\ \mu s$
t2	$\frac{4192}{\text{fc}} = 309.02\ \mu s$	-	-

Table 2.5: Time values for packets transmission (f_c = 13.56 MHz).

Summarizing this chapter, we have seen the basic block diagram of the digital core used in the development of the RFID-enabled sensor and the ISO standard on which it has been designed. The ISO standard ISO15693 which defines communication protocols, formats used for data packets and the timing constraints used to design the digital core has been explained in this chapter. Next chapter will discuss about the design of the digital core which works according to the standard defined in this chapter.

CHAPTER 3

THE DIGITAL CORE

3.1 Introduction

In this chapter we will see detailed description of the digital core used in the development of the RFID tags. The digital core is designed to conform to the ISO 15693 standard. To understand the design of the digital core we have to visualize it as a big state machine which steps through different states based on the input conditions and present state of the machine. This state machine is a Mealy state machine whose output values are determined by its current state and by the values of its input.

3.2 Design Constraints

The combination of extreme low-power and high reliability requirements make implantable electronic systems differ from other electronic-systems implementations. Most of the implantable medical devices available today depend on a non-rechargeable battery for operation. Use of a non-rechargeable battery reduces the shelf-life of the implantable device, thus making it less reliable. To increase the operating lifetime of an implantable device we are going to use passive RFID technology. RFID technology for implantable applications place a number of constraints on power management, design environment, targets used to implement those designs. As system complexity and activity increases, without a proportional increase in available energy, design challenges become more persistent [11].

Most of the implantable systems utilize fair amount of analog signal processing due to the fact that most of the systems on the chip are analog devices which are used for power harvesting, regulation, signal acquisition from sensors when compared to digital signal processing. Most of this design is done as subthreshold analog design to take advantage of low bias currents and high gain. The subthreshold region offers low current, high gain at given current and maximum voltage swing due to low V_{gs} and V_{dsat}. Another reason that aides us in the use of subthreshold analog design is these systems can be very slow as most body responses are measured in milli-seconds rather than in nano-seconds. This is an important distinction, as the need to do things slowly translates to lower power consumption. Design of digital blocks for implantable applications can take variety of approaches. At the system level and IC level, this often includes turning off blocks and/or clocks when they are not required to perform a task, lowering power supply voltage down to the sum of p- and n-channel thresholds, using libraries that offer gates with multiple thresholds, selectively duplicating functions, circuits or tasks at the expense of area to save power, and balancing the decision to implement functions in hardware versus firmware. At the IC level, minimizing power dissipated in the clock network is important [11].

3.3 Digital Core Design

The digital core handles the communications and control operations of the RFID tag. It decodes the incoming data packet from the reader, executes commands and sends data back to the reader. The control logic performing reception, acquisition and transmission of data is also implemented by the digital core. The generalized block diagram of the RFID tag is as shown in Fig 3.1, it consists of both analog core and digital core. The block diagram of the digital core is as shown in Fig 3.2. From the block diagram we can see that there are three main modules which make up the digital core namely Frame decoder, controller and the encoder.

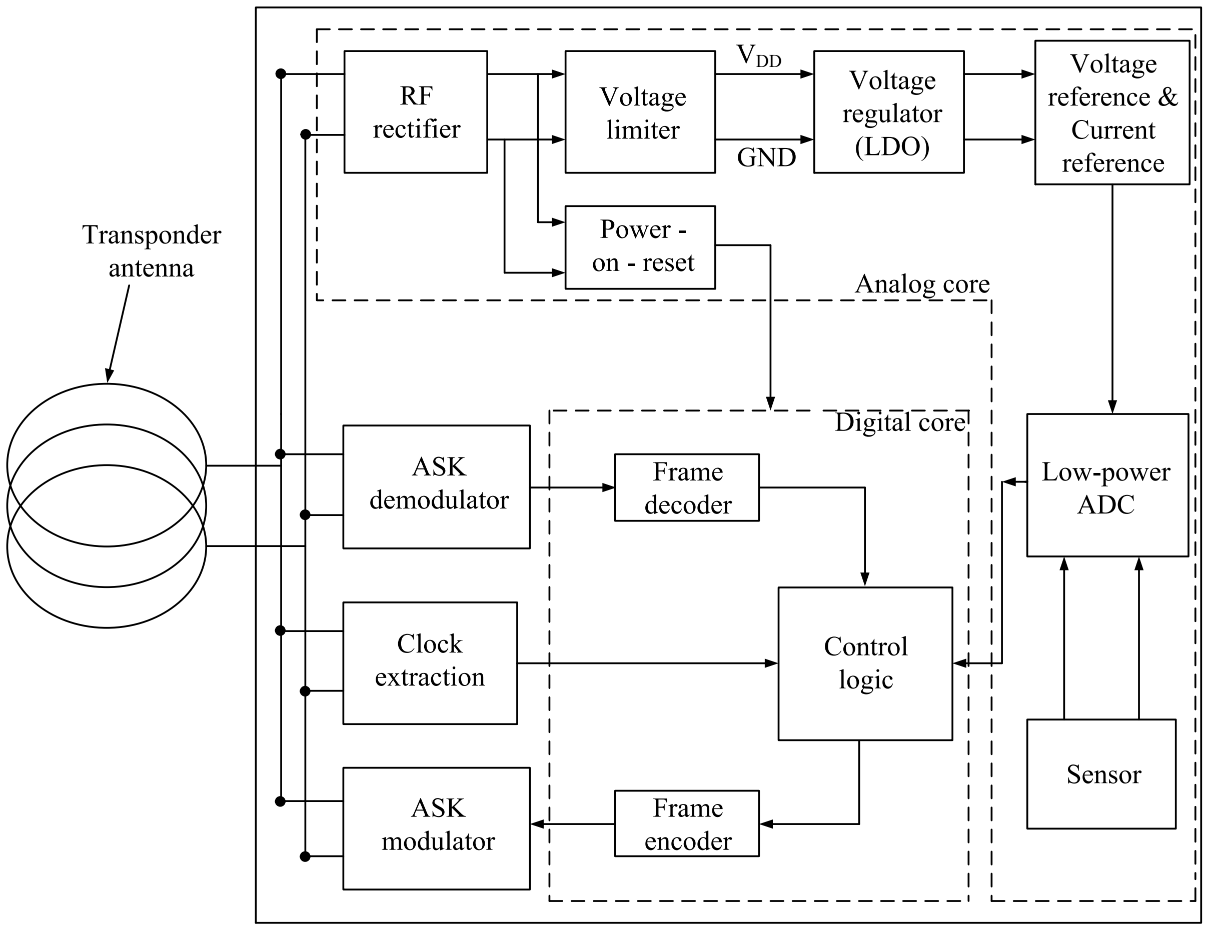


Fig 3.1: Block diagram of the RFID Tag.

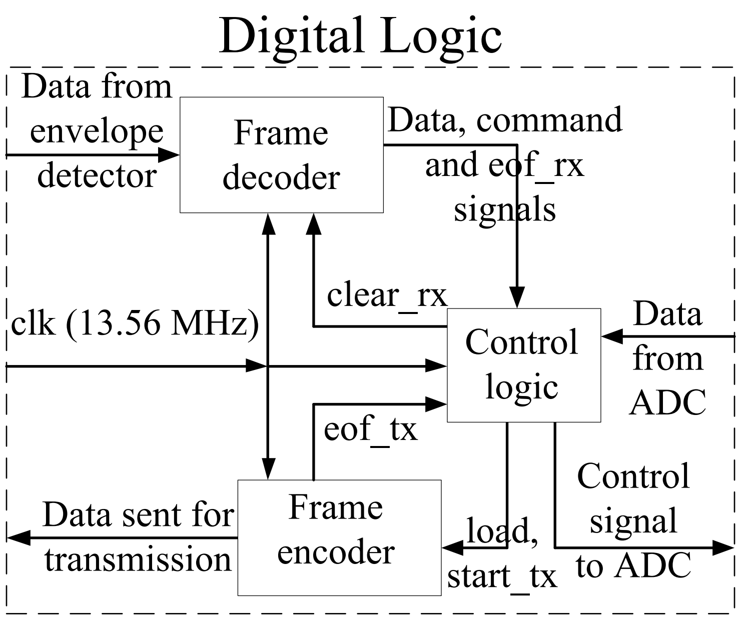


Fig 3.2: Digital logic of the RFID Tag.

As we said earlier the digital core works on a state machine which is operated and controlled by the control module in the tag. In the next subsequent sections we will discuss about the architecture of every module and state machines on which they work.

3.4 Control Module

The control module is the main module of the RFID tag as it controls the operation of both frame decoder and frame encoder in the tag. This module has been designed to operate as a state machine. This state machine works in four, states namely: IDLE, COMPARE, TRANSMIT and WAIT_STATE. To help visualize the operation of the state machine a state diagram is shown below in Fig 3.3. The VHDL code for the above control module is shown in appendix and the VHDL entity for this control module is given below

```
entity controller is
Port (data_in : in  STD_LOGIC_VECTOR (15 downto 0);
Clk           : in  STD_LOGIC;// 13.56 MHz clock signal
Rst           : in  STD_LOGIC;
eof_rx        : in STD_LOGIC; // End of frame signal from the decoder
eof_tx        : in STD_LOGIC; // End of frame signal from the transmitter
frame_error   : in STD_LOGIC; // Frame error signal from the decoder
adc_data      : in STD_LOGIC_VECTOR (7 downto 0);
data_to_tx    : out STD_LOGIC_VECTOR (39 downto 0);
load          : out STD_LOGIC;
start_tx      : out STD_LOGIC;
clear_rx      : out STD_LOGIC);
                        end controller;
```

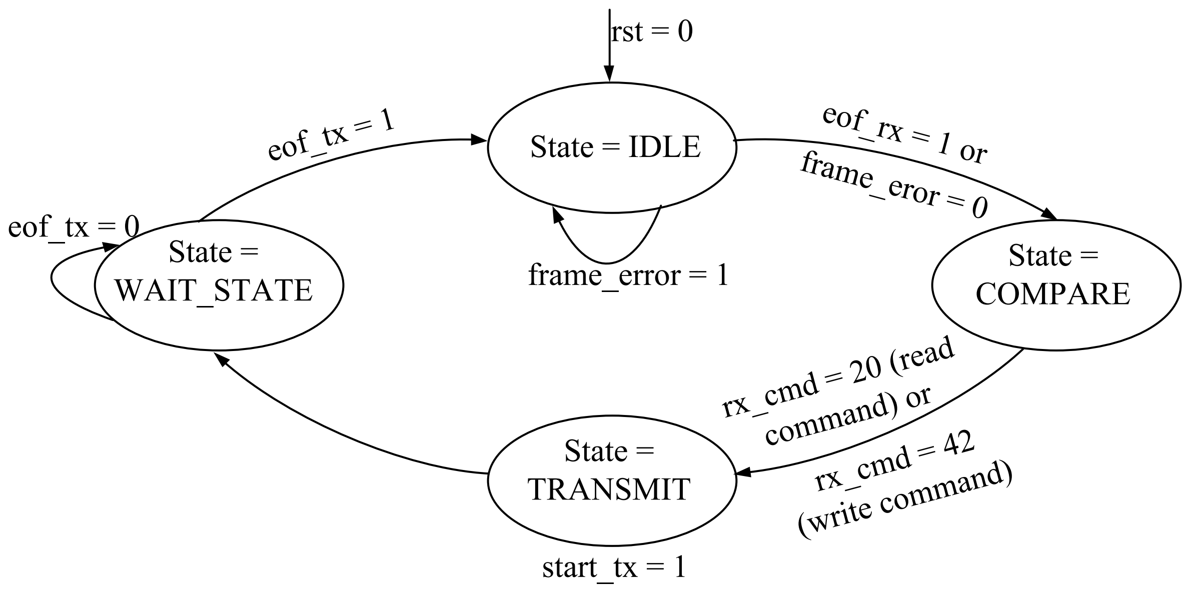


Fig 3.3: State diagram for the Control Module state machine.

During power-on the state machine is always reset to the IDLE state. When the state machine is in IDLE state most of the control signals and both frame decoder and frame encoder modules are reset and it is ready to receive a data packet from the reader. When the reader sends a data packet, the frame decoder decodes the incoming packet and sends a **end of frame** signal (eof_rx) to the controller. Based on this signal the controller will step into the COMPARE state. If frame error occurs then the controller will drop the packet and remains in the IDLE state where it waits for another packet but, if the packet is received without error then the controller will go to COMPARE state. In COMPARE state the received command is used to determine if the requested operation is read or write, once this is determined then the requested operation is preformed and the state machine goes into the TRANSMIT state. In this state a response data packet is generated using frame encoder module and sent to the reader using load modulation, during this time a WAIT_STATE is initiated to wait for a signal from the frame encoder which indicates completion of packet transmission. When the frame encoder indicates the **end of transmission** (eof_tx) of a response packet the control module goes back to IDLE state resetting both the frame decoder and frame encoder.

3.5 Frame Decoder

This module decodes the incoming data packet and extracts information sent by the reader. The incoming data packet format is as shown in Fig 3.4 and Fig 3.5. If it's a read request data packet then it contains 6 fields which include packet delimiters SOF and EOF along with 40 bits of data which represent flag bits, block number and the calculated CRC for error detection. If the incoming data packet is a write request data packet then it contains 7 fields with packet delimiters SOF and EOF along with 72 bits of data which represents flag bits, block number, data to be written in the block and the calculated CRC for error detection as shown in Fig 3.5.

Request SOF	Request Flags	Read Single Block	Block number	CRC 16	Request EOF
	8 bits	0x20	8 bits	16 bits	

Fig 3.4: Read Single Block request format.

Request SOF	Request Flags	Write Single Block	Block number	Data	CRC 16	Request EOF
	8 bits	0x21	8 bits	32 bits	16 bits	

Fig 3.5: Write Single Block request format.

To do the job of decoding these packets we designed a state machine which was implemented using VHDL and verified on a CPLD. VHDL (very high speed IC hardware description language) is a high level hardware description language used in electronic design automation to describe digital and mixed signal systems such as FPGA's and integrated circuits. VHDL was used because it offers many benefits such as the management of complex designs, explore alternative design techniques, provides feedback to produce better results, increase productivity and most importantly portable design data.

3.5.1 State Machine Design for Frame Decoder

The process of decoding the incoming data packet in the frame decoder can be understood by first describing the state machine on which the frame decoder works. The state machine has two processes in it: **ASK process** and **CLK process** which depend on each other to step through different states as shown in Fig 3.6 and 3.7. The **ASK process** is used to step through different states depending on the condition of the **CLK process.** If the **CLK process** goes to **ERROR** state during any point in the decoding process then **ASK process** will go into **IDLE** state, else it normally steps through every state. The **CLK process** is the main process where the incoming data is decoded and stored in a register to be transferred to control module for analysis. The reason for using two process can be understood by referring to chapter 2, section 2.4.2 where I have shown that incoming data from the reader uses 1 out of 4 data coding technique. In this technique 2 data bits are represented by a pause in any one of four consecutive 18.88 μs time periods, so by using two processes we can count between two pauses and depending on the value of the counter we can determine the value of the incoming data bit. The **ASK process** process always works on the falling edge of the incoming signal (ask_in) and **CLK process** works on the rising edge of the 13.56 MHz clock. The VHDL code for the frame decoder is given in appendix; the VHDL entity for this module is given below:

```
entity frame_decoder is
port ( ask_in  : in  STD_LOGIC; // Incoming signal from the data slicer
rst            : in  std_logic;           // Reset signal
clk            : in  STD_LOGIC;    // 13.56 MHz clock signal
SOF            : out std_logic;      // Start of frame signal
EOF            : out std_logic;      // End of frame signal
ask_out        : out std_logic;
clk_out        : out std_logic;
frame_error    : out std_logic;  // Frame error signal
data_out       : out std_logic_vector(15 downto 0)); // Data extracted from the incoming
packet
end frame_decoder;
```

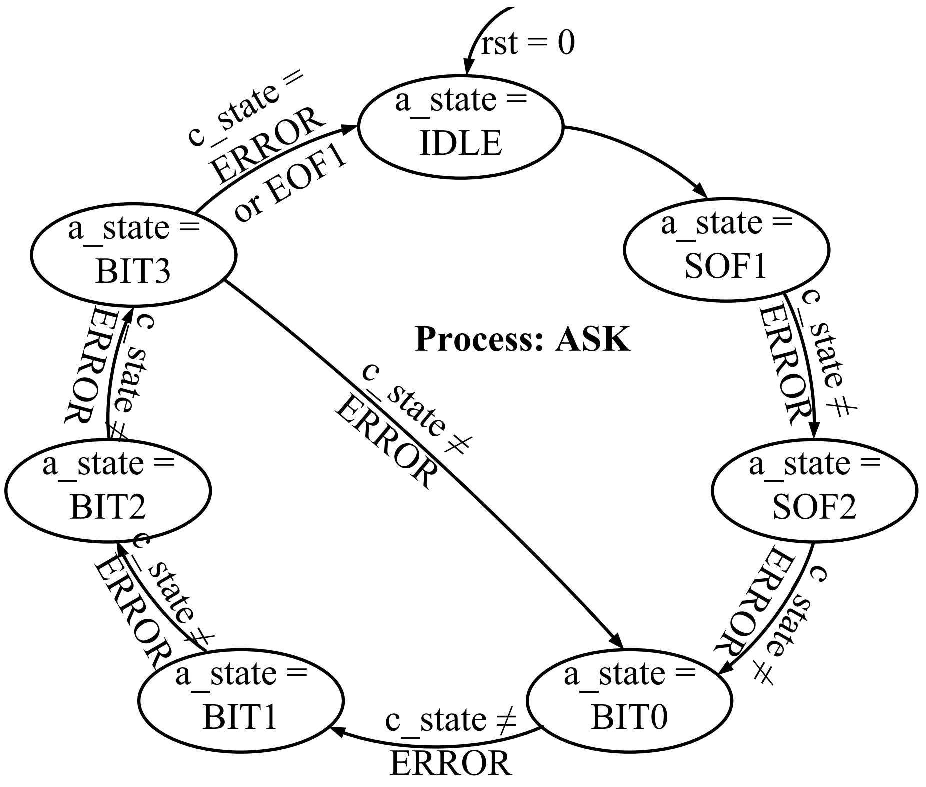


Fig 3.6: State diagram for the **ASK** process.

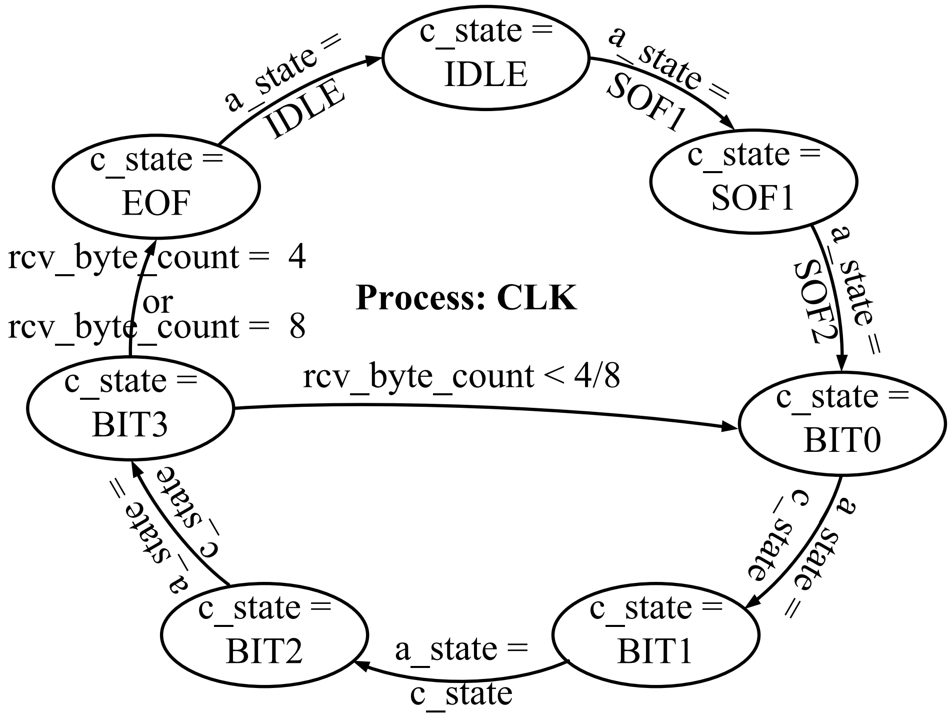


Fig 3.7: State diagram for the **CLK** process.

The states used in the two processes are IDLE, SOF1, SOF2, SOF3, BIT0, BIT1, BIT2, BIT3, INTER, BYTE_READY, HALT, EOF1, ERROR and a_state, c_state are the two signals used to handles the states in ASK process and CLK process respectively. To see and understand how the count value between two pauses determines the value of data bits let us see the following steps:

1) Fig 3.8 shows the **start of frame** format of the incoming data packet. Initially the **ASK process** steps into SOF1 state due to which **c_state** also steps into SOF1 state and the 11 bit counter in **CLK process** starts counting. Next, **a_state** will step into SOF2 state, when this happens the counter in the **c_state** stops counting. If the counter value is 512 (±32, due to effect of noise) then **c_state** decodes this signal as **start of frame** and steps into **BIT0 state.** When determining the counter value we use the first 6 bits of the counter ie., from counter(10) to counter(5). The least significant bits are neglected to include the effect of noise.

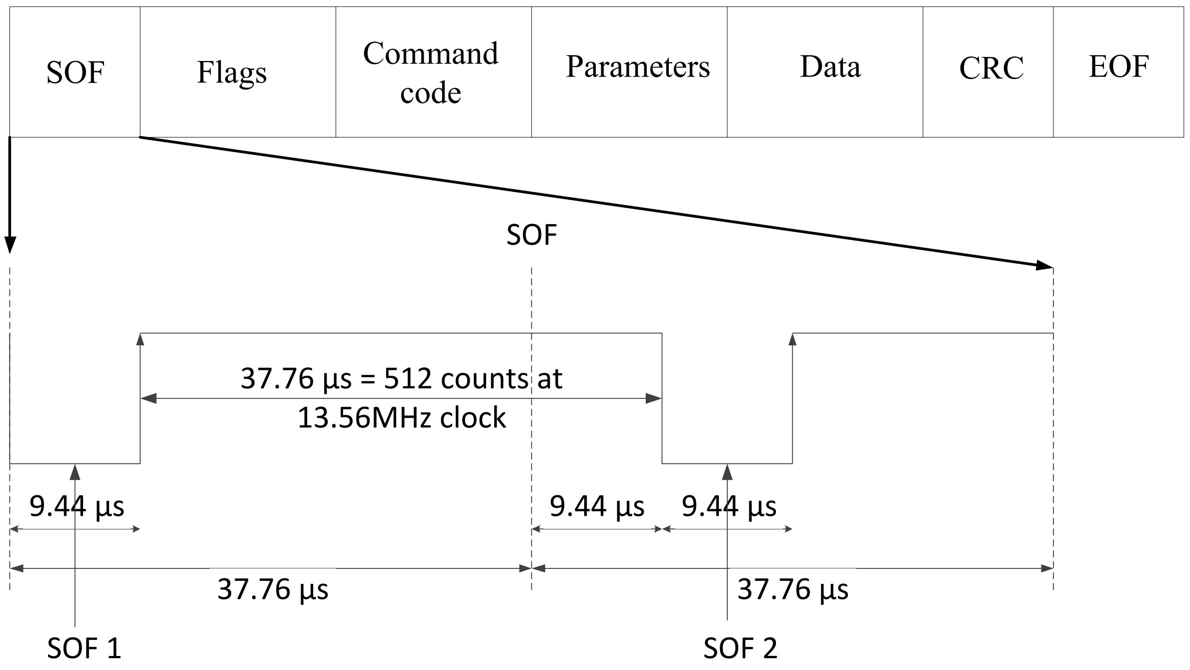


Fig 3.8: SOF with two pauses.

2) Fig 3.9 shows how dibits are decoded in BIT0 state of the **CLK process** after S0F detection. The BIT0 state determines the value of the first dibit of a byte. According to the 1 out of 4 data coding technique, a byte of data can be divided into four dibits. Detection of the first dibit of a byte is refered as BIT0 state. The **CLK proces** may go to BIT0 state any number of times depending on the number of bytes in the incoming data packet. Fig 3.9 shows the BIT0 state after detection of **start of frame** in the incoming data packet. The method of determing the value of the first bit after **start of frame** is similar to the method of determing the **start of frame** as shown earlier in step 1. When the **c_state** steps into BIT0 state the counter starts counting until **a_state** steps into BIT0 state. On the falling edge of incoming signal **a_state** steps into BIT0 state. When this happens the counter stops and depending on the value of the counter the value of the incoming bit is determined. Suppose after **start of frame** signal, a dibit 00 occurs, then the counter value should be 384 (±32, due to effect of noise). Similarly if other dibits such as 01, 10 or 11 occur after the start of frame signal then depending on the counter value the dibits are decoded.

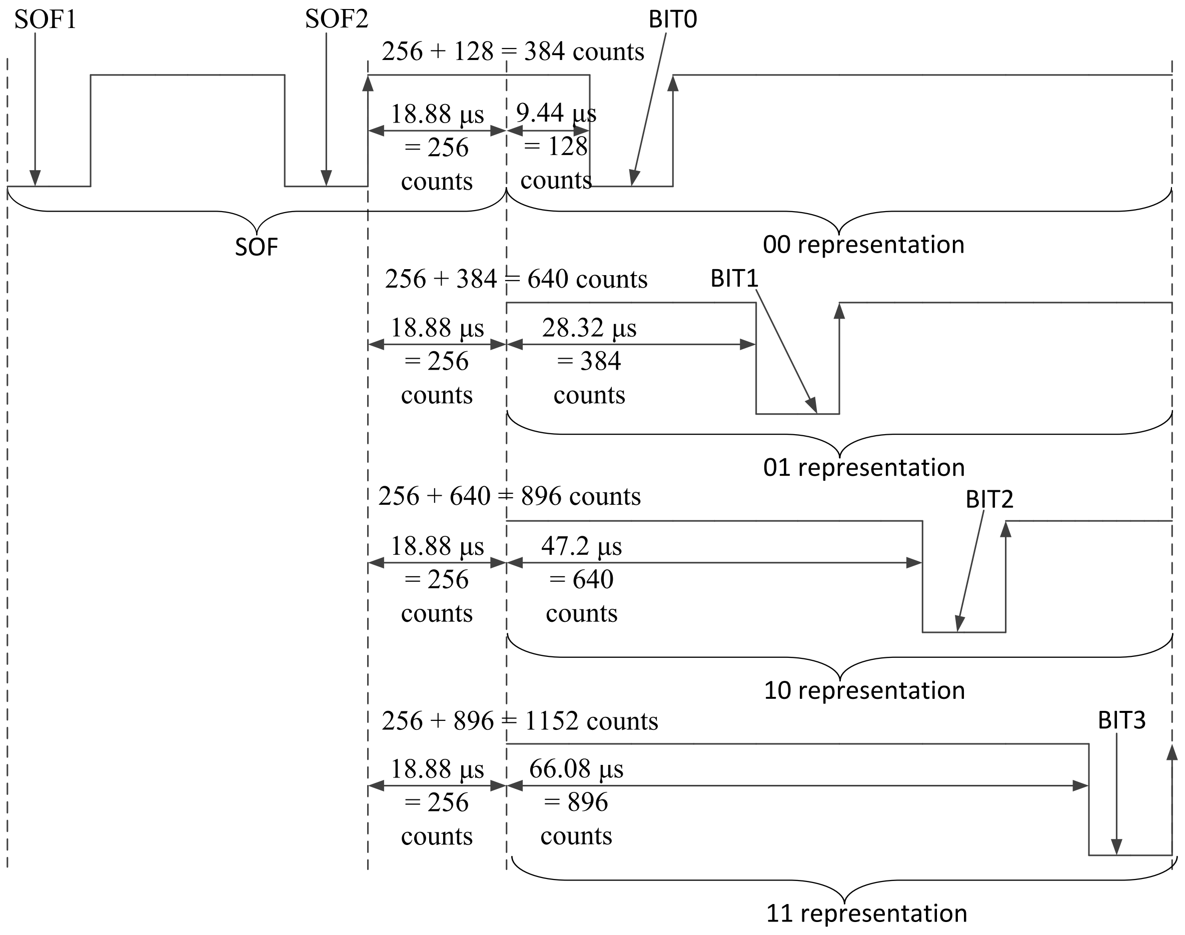


Fig 3.9: BIT0 state of the **c_state** process after **start of frame** detection.

3) Fig 3.10 shows how dibits are decoded in BIT1 state of the **CLK process** after dibit 00 detection in BIT0 state. Once the first bit of the incoming data byte is detected then **c_state** process steps into BIT1 state. When the **c_state** steps into BIT1 state the counter starts counting until **a_state** steps into BIT1 state. On the falling edge of incoming signal (ask_in) **a_state** steps into BIT1 state. When this happens the counter stops and depending on the value of the counter the value of the incoming bit is determined. Suppose dibit 01 occurs in BIT1 state after dibit 00 in BIT0 state then the counter value should be 1152 (±32, due to effect of noise). Similarly if other

dibits such as 00, 10 or 11 occurs after BIT0 state then depending on the counter value the dibits are decoded.

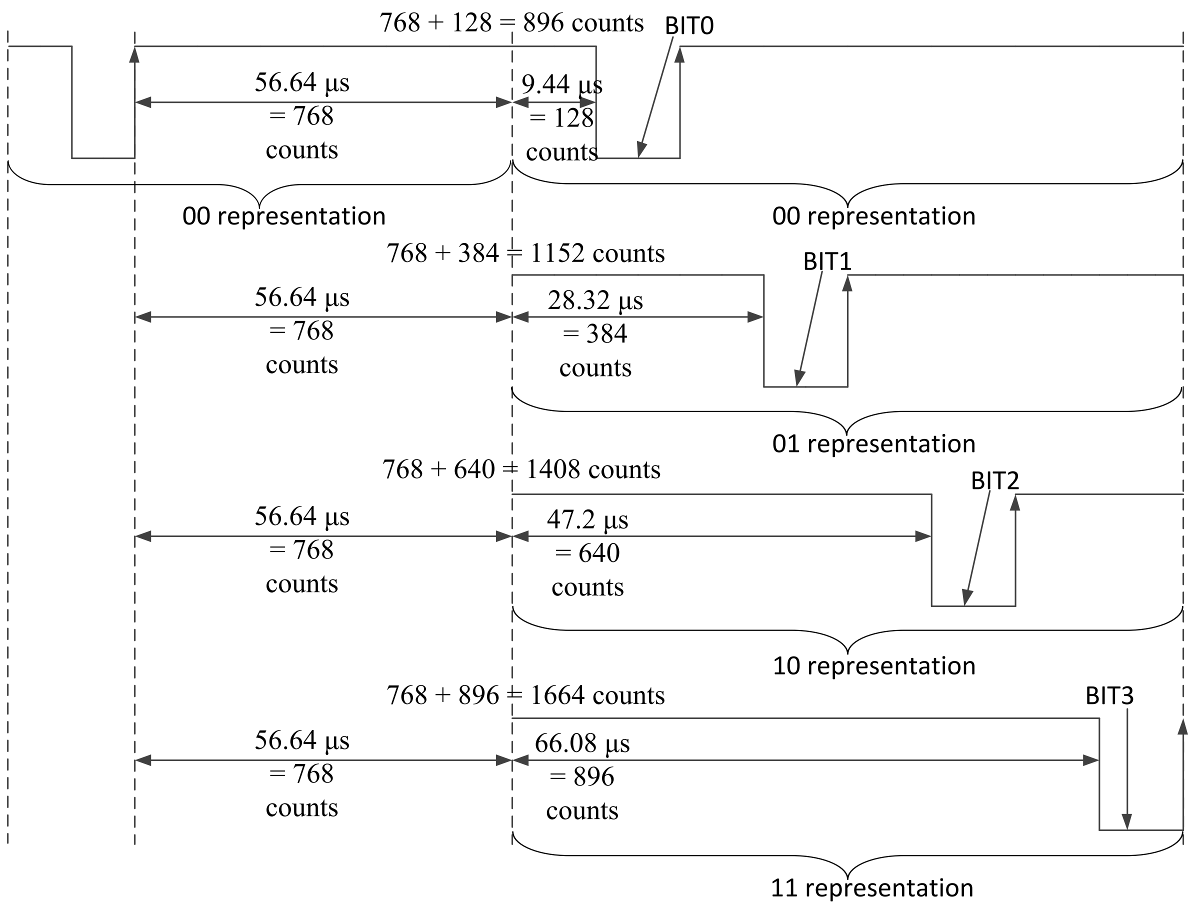


Fig 3.10: BIT1 state of the **c_state** process after dibit 00 detection.

Similarly when **CLK process** decodes dibits in BIT1 state it steps into BIT2 state followed by BIT3 state to decode the incoming data bits. Again the counter value between the two pauses determines the value of the dibits. When all the data bytes are decoded the value is stored in **data_out** register (as seen in the VHDL entity), which is sent out to the control module for analysis. When the **CLK process** steps into **end of frame (EOF)** state the **ASK process** steps into **IDLE** state and on the next rising edge **c_state** also steps into **IDLE** state.

So at the end of decoding process both the processes go back into **IDLE** state and wait for another incoming data packet.

3.6 Frame Encoder

The frame encoder module is responsible for sending a response back to the reader. We know that the RFID tag works on the principle of reader talk first, when the RFID tag receives a command from the reader the control module in the RFID tag processes the incoming packet and determines the type of operation to be performed. The RFID tag responds after a time interval **t1 (311.5 µs)** specified in Table 2.5 of section 2.6, which describes the RFID tag transmission protocol. After time **t1 (311.5 µs)** the frame encoder starts generating a response packet and simultaneous transmission of response packet takes place. The read and write response packet format is as shown below in Fig 3.11 and 3.12 respectively.

Response SOF	Response Flags	Data	CRC 16	Response EOF
	8 bits	32 bits	16 bits	

Fig 3.11: Read Single Block response format.

Response SOF	Response Flags	CRC 16	Response EOF
	8 bits	16 bits	

Fig 3.12: Write Single Block response format.

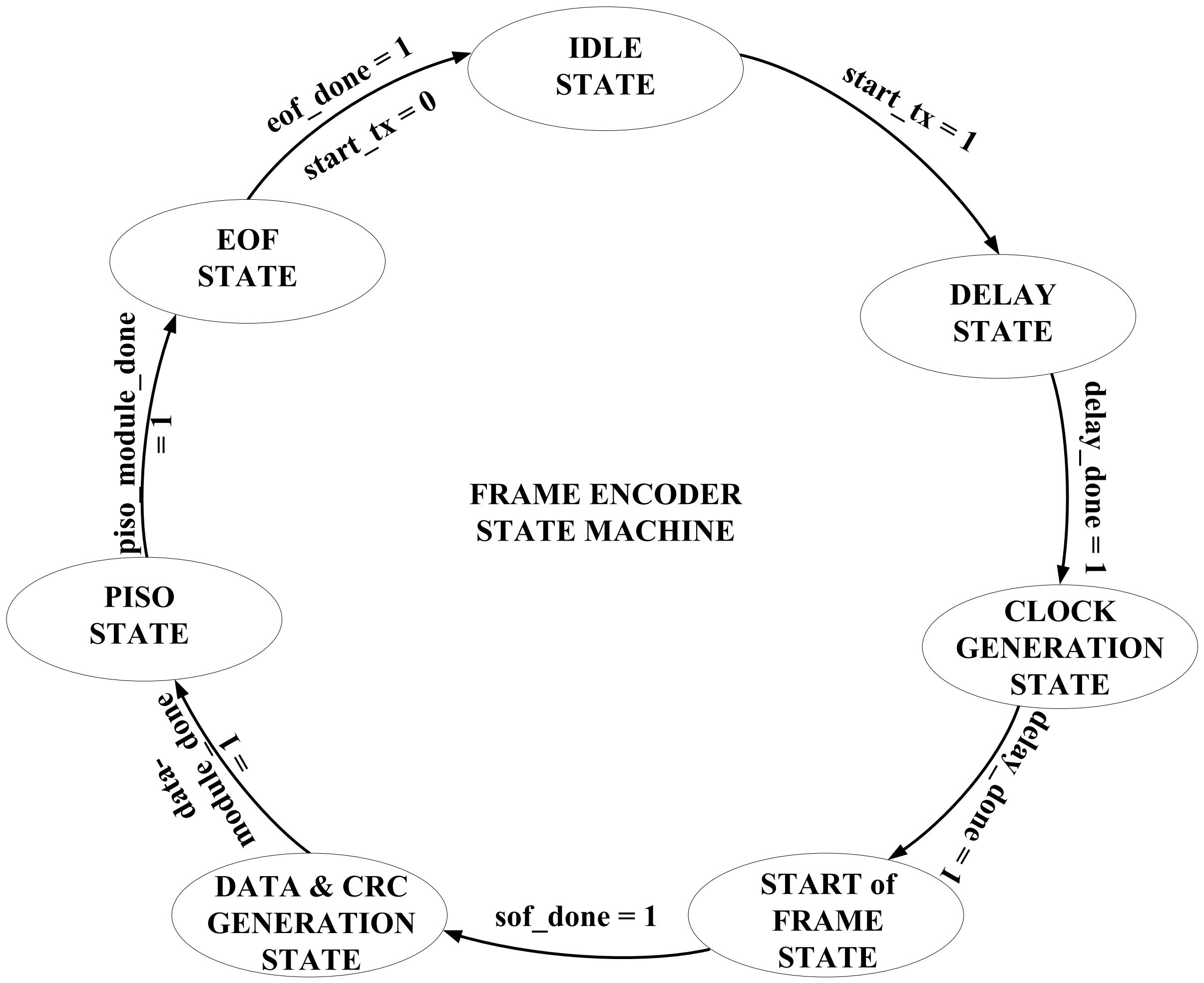


Fig 3.13: State diagram for the Frame Encoder module.

The frame encoder uses a state machine to generate the response packets; the designed state machine follows the state diagram shown in Fig 3.13. As we know from the response format shown in Fig 3.11 and Fig 3.12 we need to generate SOF, FLAGS and/or DATA, CRC and EOF fields before we send the response packet. To generate all these fields we have designed different modules which work in a sequence, each module depends on the input signal coming from the previous module. The architecture of the frame encoder is shown below in Fig 3.14. A response frame starts by generating a delay, once a delay of 311.5 μs is generated then it raises the signal **delay_done** which starts the clock module. The clock module generates 3 sub-clocks from the incoming master clock of 13.56 MHz frequency. The three different clocks used inside the frame decoder are clk_512 (26.484 KHz), clk_256 (52.968 KHz) and clk_32 (423.75 KHz). The clk_512 is used for CRC, FLAGS and/or DATA modules, clk_256 is used for SOF and EOF modules and clk_32 is used for modulation during data transmission. The clk_512 corresponds to the high frequency data rate (26.484 Kbits/s i.e. $f_c/512$) chosen for data transmission, the use of clk_256 for SOF and EOF and clk_32 for data transmission will be explained later. The start of clock module initiates the process of generating a response frame. As explained earlier, to generate different fields of the response packet we have designed different modules, all these modules are connected to a 4:1 mux which switches its output depending on the completion of every module. The completion of a module is specified by the **done** signal of that module, this signal is used to start the succeeding module, switch the 4:1 mux and in some cases to disable it to save power.

The ISO 15693 standard uses Manchester codes (see section 2.4) for encoding data and a clock frequency of $f_c/32$ for sub-carrier modulation before data transmission. So, to follow this rule we have designed a Manchester module which will convert the incoming data into Manchester codes before sending it to the 4:1 mux for transmission. The Manchester module has to convert both data and the CRC checksum of the data, to facilitate this 2:1 mux is used. The 2:1 mux has inputs from both data module and CRC module. The **data_module_done** signal from the data module is used to switch the output of 2:1 mux. The designed CRC module generates CRC value in parallel but the Manchester module requires data to come-in serially, so we designed another module called PISO (parallel in serial out) module. The PISO module takes 16 bit CRC value in parallel from the CRC module and sends them out serially to the 2:1 mux. The 2:1 mux will switch its output from data module output to PISO module output when data module sends the **data_module_done** signal. Once all the data (40 bits from data module and 16 bits from CRC module) is sent out through Manchester module the frame encoder switches the 4:1 mux to the end of frame output. The **piso_module_done** signal will switch the 4:1 mux and also starts the end of frame module. The end of frame module generates the **end of frame** signal and then it raises eof done signal. This signal disables **EOF** module to save power and the control modules on receiving this signal disables the entire frame encoder module by lowering the **start_tx** signal.

The VHDL code for this module is given in appendix, but the VHDL entity for this module is as given below:

```
entity transmitter is
generic ( n : integer := 39);
Port (start     : in STD_LOGIC; // This will start the delay module
Data            : in std_logic_vector(n downto 0); // Data to be transmitter
Clk             : in std_logic; // 13.56Mhz clock
Load            : in std_logic; // This will load the data onto Data and CRC module
eof_done        : inout std_logic;
tx_out          : out std_logic);
end transmitter;
```

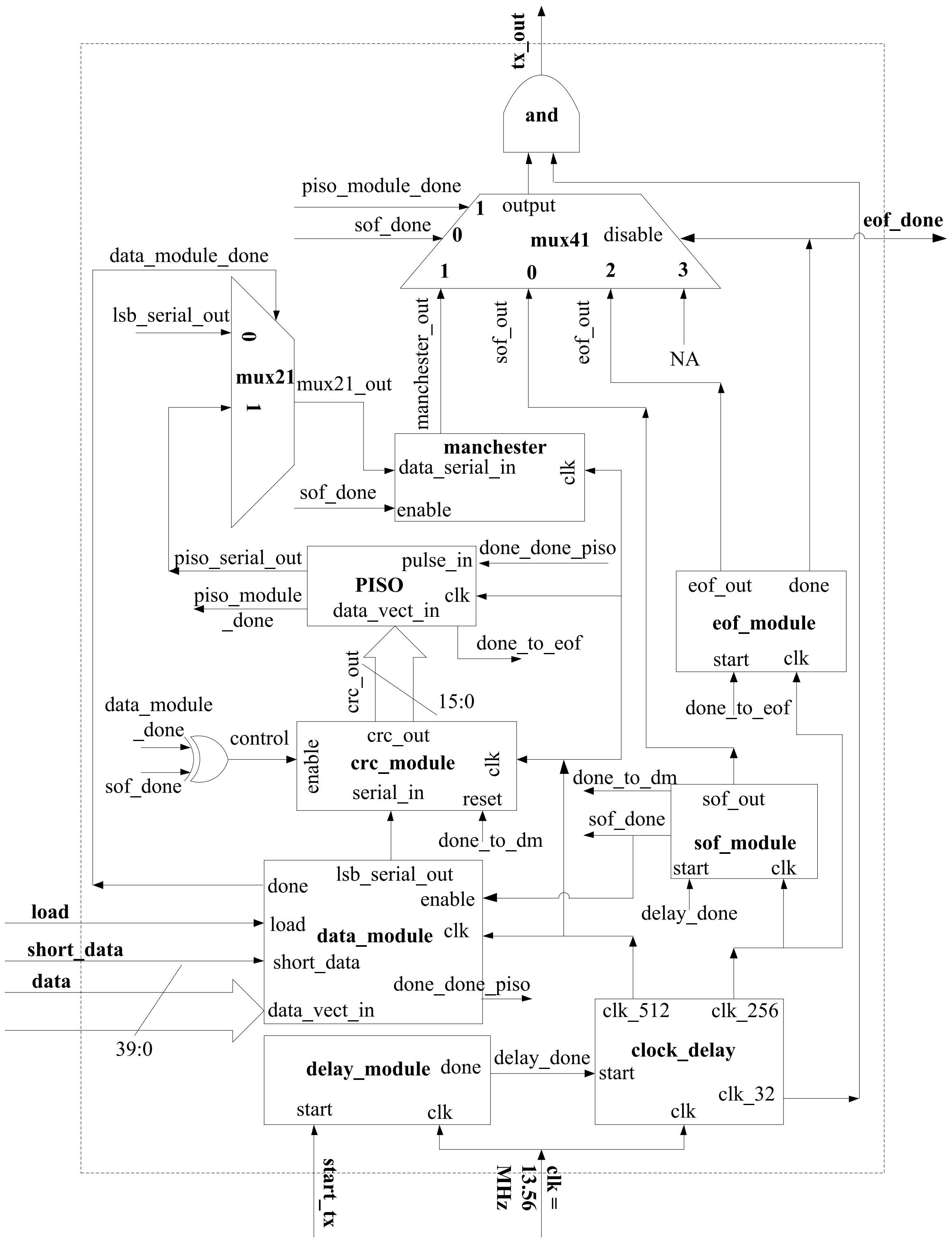


Fig 3.14: Architecture of the Frame Encoder.

The following sections will discuss all the modules used to design the frame encoder in detail. We start with start of frame module and end of frame module as their design technique is similar, later we will discuss about data module, crc generation, PISO module and other supporting modules.

3.6.1 SOF Module

The SOF for high data rate is made up of 3 parts namely, an unmodulated time of 56.64 μs followed by 24 pulses of 423.75 kHz and a logic 1 which also corresponds to the high data rate as shown below in Fig 3.15.

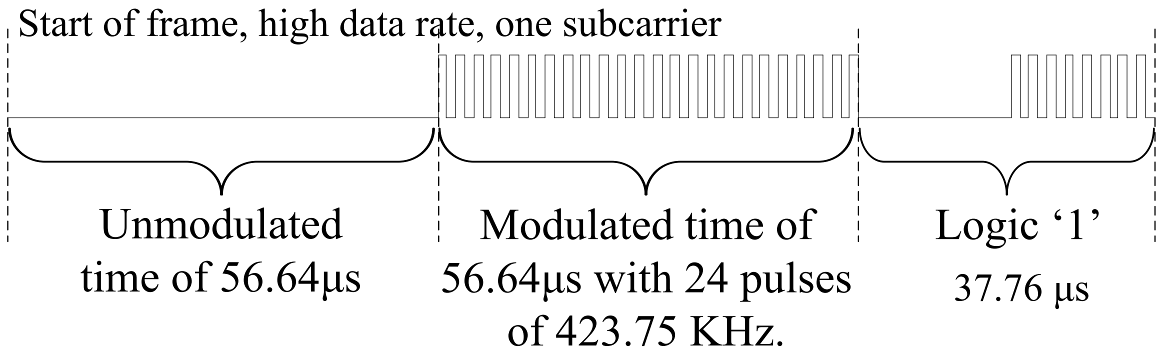

Start of frame, high data rate, one subcarrier
Unmodulated time of 56.64μs
Modulated time of 56.64μs with 24 pulses of 423.75 KHz.
Logic '1' 37.76 μs

Fig 3.15: SOF representation for response packet.

The VHDL code for this module is given in appendix, the entity for this module is given below.

```
entity sof_module is
port(start      : in std_logic; // delay_done signal from the delay_module
clk             : in std_logic; // clk_256 (52.968 KHz)
enable          : in std_logic;
sof_out         : out std_logic;
done            : inout std_logic; // sof_done signal to switch MUX41
done_early      : out std_logic);
end sof_module;
```

To generate the **start of frame** (SOF) signal as shown above we approached many different types of designs, of which one design suited for the state machine described above. In this design we make use of a shift register to shift out desired data on a particular clock. To generate SOF we made use of clk_256 which has a time period of 18.88 μs to shift out data '00011101' which corresponds to SOF signal.

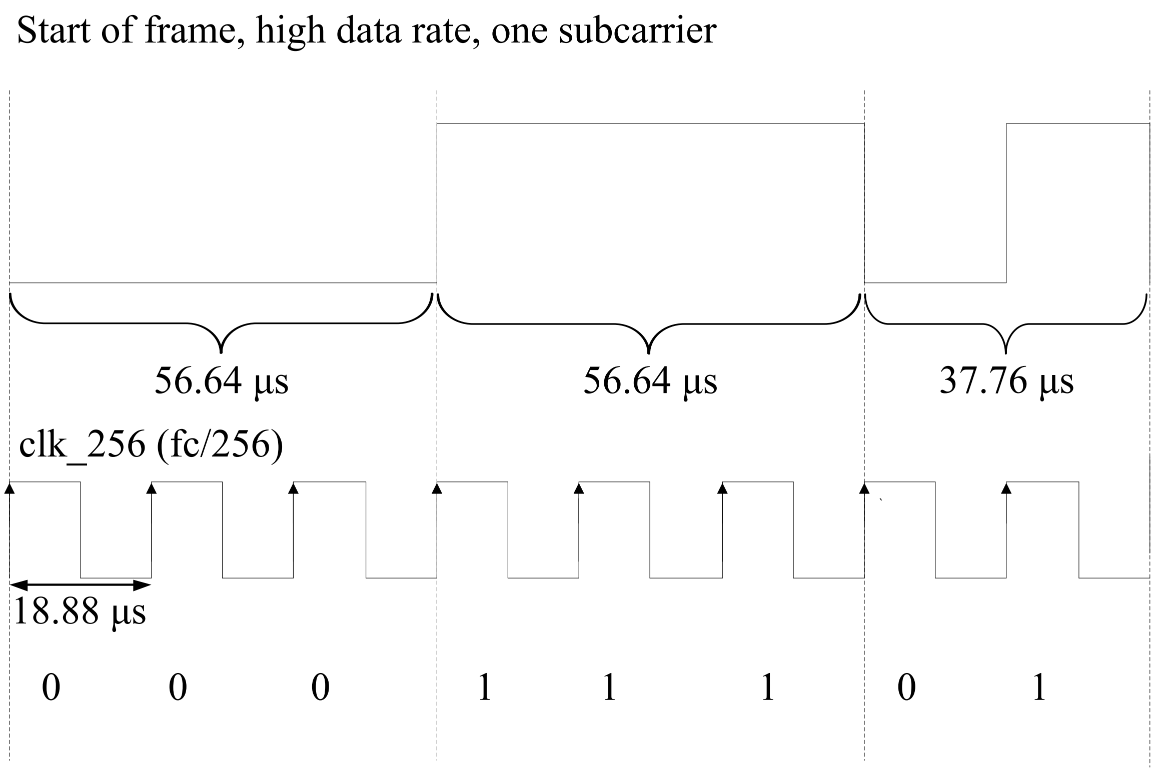


Fig 3.16: Use of clk_256 to generate SOF.

The reason for using clk_256 is that its time period 18.88 μs is a multiple of 56.64 μs, so on the rising edge of every clock cycle of clk_256 we can shift out data which will represent the SOF for the response packet as shown in Fig 3.16. After generating the SOF signal we modulate the outgoing data using clk_32 as shown below in Fig 3.17.

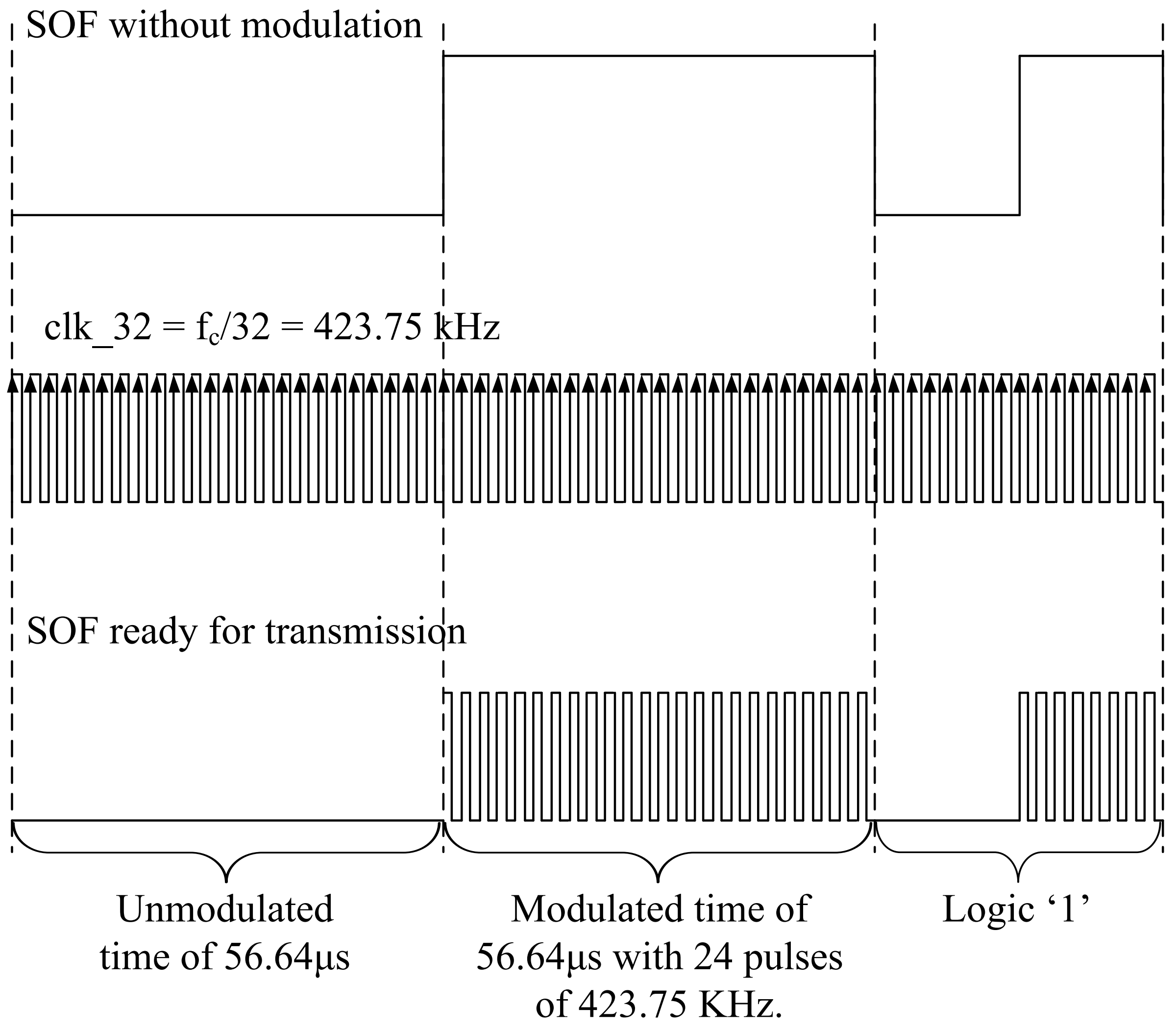

SOF without modulation
clk_32 = f_c/32 = 423.75 kHz
SOF ready for transmission
Unmodulated time of 56.64µs
Modulated time of 56.64µs with 24 pulses of 423.75 KHz.
Logic '1'

Fig 3.17: SOF after modulating with clk_32 ($f_c/32 = 423.75$ kHz).

This method is the simplest way of generating an SOF frame. The VHDL code written for this module will perform the function as described by this method. Once the frame is sent this module will send a signal sof_done to initiate the next module and to stop itself to save power.

3.6.2 EOF Module

The EOF for high data rate is made up of 3 parts namely, a logic 1 which also corresponds to the high data rate followed by 24 pulses of 423.75 kHz and an unmodulated time of 56.64 μs as shown below in Fig 3.18.

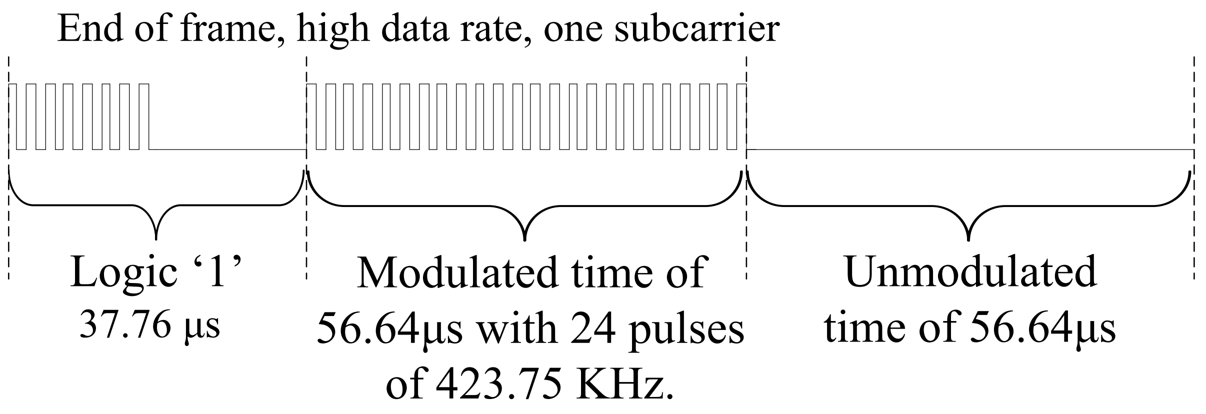

End of frame, high data rate, one subcarrier
Logic '1' 37.76 μs
Modulated time of 56.64μs with 24 pulses of 423.75 KHz.
Unmodulated time of 56.64μs

Fig 3.18: EOF representation for response packet.

The VHDL code for this module is given in appendix, the entity for this module is given below.

```
entity eof_module is
port( start     : in    std_logic; // Done signal from the PISO module
clk             : in    std_logic; // clk_256 (52.968 KHz)
enable          : in    std_logic
eof_out         : out   std_logic; // End of frame signal
done            : inout std);
end eof_module;
```

Similar to start of frame (SOF) to generate the end of frame (SOF) signal as shown above we approached many different types of designs, of which one design suited for the state machine described above. In this design again we make use of a shift register to shift out desired data on a particular clock. To generate EOF we made use of clk_256 which has a time period of 18.88 μs to shift out data '10111000' which corresponds to EOF signal.

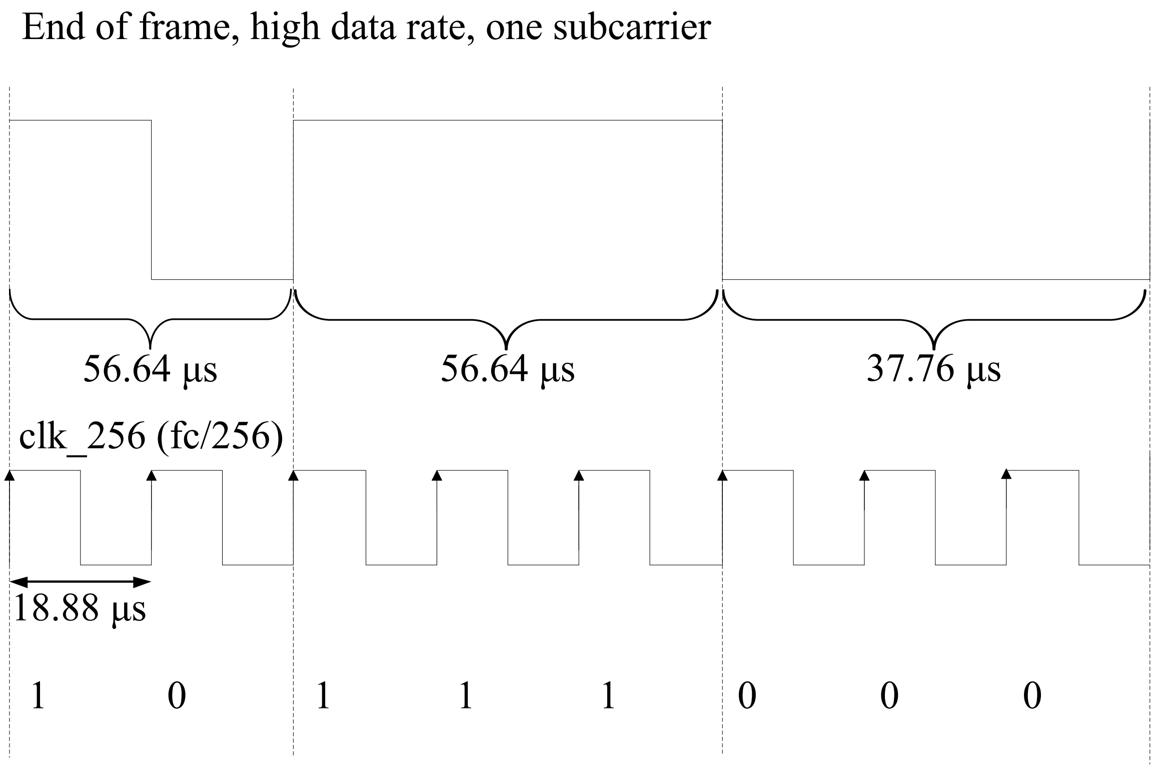


Fig 3.19: Use of clk_256 to generate EOF.

The reason for using clk_256 is similar to that of SOF, that its time period 18.88 µs is a multiple of 56.64 µs, so on rising edge of every clock cycle of clk_256 we can shift out data which will represent the EOF for the response packet as shown in Fig 3.19. After generating the EOF signal we modulate the outgoing data using clk_32 as shown below in Fig 3.20.

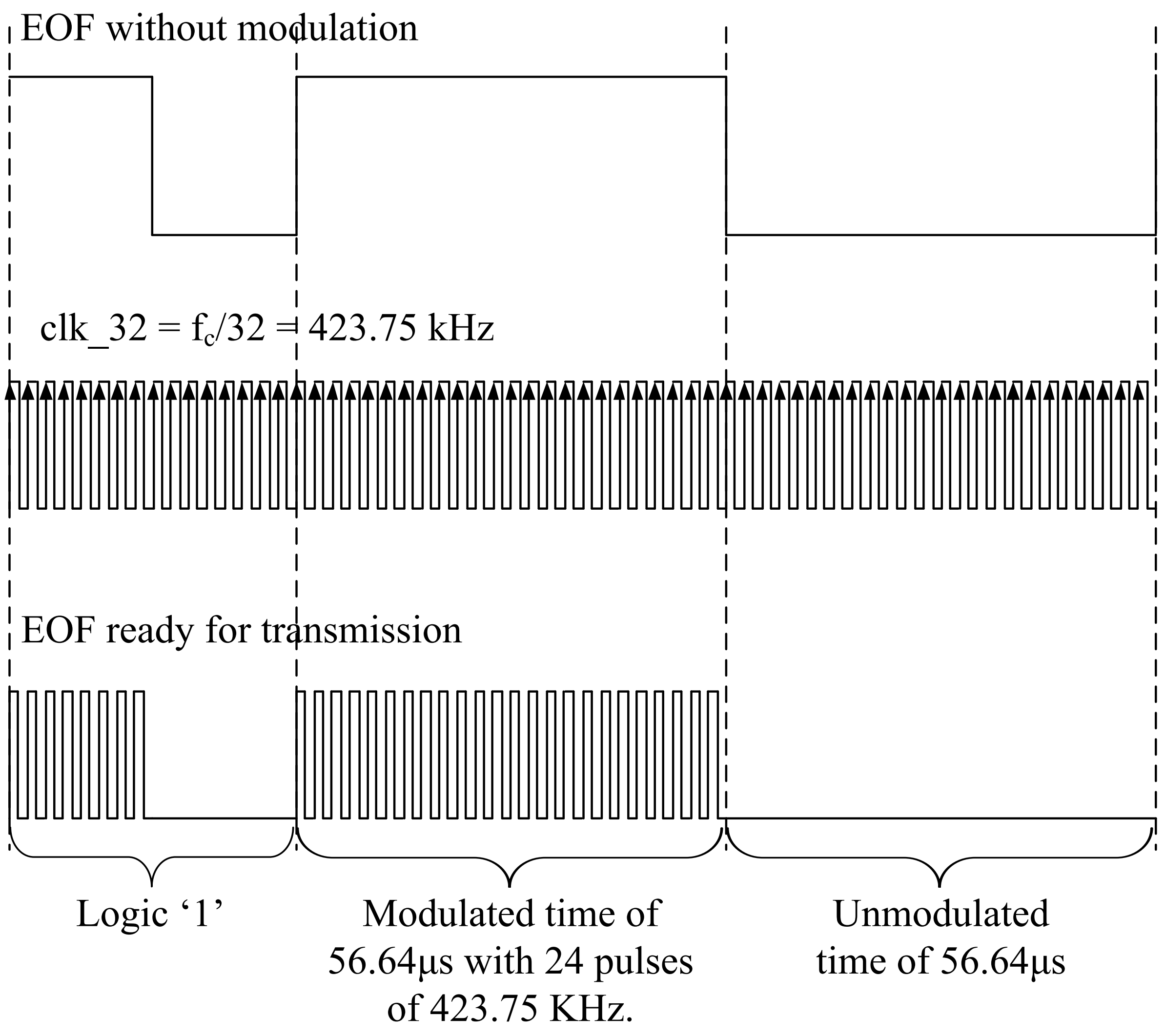

EOF without modulation
clk_32 = f_c/32 = 423.75 kHz
EOF ready for transmission
Logic '1'
Modulated time of 56.64μs with 24 pulses of 423.75 KHz.
Unmodulated time of 56.64μs

Fig 3.20: EOF after modulating with clk_32 ($f_c/32 = 423.75$ kHz).

This method is the simplest way of generating an EOF frame. The VHDL code written for this module will perform its function based on the method described above. Once the frame is done this module will send a signal **sof_done** to initiate the next module and to stop itself to save power.

3.6.3 Data Module

The VHDL entity for the data module is given below; complete VHDL code for this module is given in the appendix.

```
entity data_module is
port( data_vect_in    : in std_logic_vector(39 downto 0);
clk                   : in std_logic;
load                  : in std_logic; // Signal on which data is stored in local register
enable : in std_logic : // Signal on which transmission starts
lsb_serial_out        : out std_logic; // Data sent to Manchester and CRC module
done                  : out std_logic; // Pulse sent to mux21 and XOR gate
done_to_piso          : out std_logic);
end data_module;
```

The main idea behind the data module is to load data (8 bits flag and/ or 32 bits data) from the control module before transmission into a shift register and then to shift **them** out during transmission. The control module loads data after the comparison state using load signal and the data module transmits data during transmission state using **enable** signal. When the **load** signal goes high the incoming data is stored in a local register. During transmission of this data the **load** signal goes low and **enable** signal goes high. When the **enable** signal goes high, data in the local register is shifted out on the rising edge of the clock (clk_512). The data is sent out least significant bit first to the mux 2:1 and also to CRC module as shown in Fig 3.14. The mux 2:1 will send this incoming data out for transmission, whereas the CRC module will use the data to calculate the CRC checksum. The state diagram given below in Fig 3.21 describes the design of the data module. Once shifting of 40 bits are completed the data module generates two control signals namely **done** and **done_to_piso**, **done** signal is generated to inform the crc module and mux 2:1 module whereas the **done_to_piso** is generated to start the PISO module. Once these signals are received, the CRC module is disabled, the mux 2:1 switches its output from **data module** to **PISO**

module and the PISO module will output the CRC serially starting from least significant bit of the least significant byte. Seen from the code in appendix, these two signals are generated on the transmission of 39th bit this is because the state machine will detect this signal on the rising edge of the 40th bit which helps to synchronize the PISO module with data module. If we raise these two signals on the 40th bit instead of 39th bit then the state machine will detect them on the rising edge of the 41st clock cycle which will create a gap in transmission.

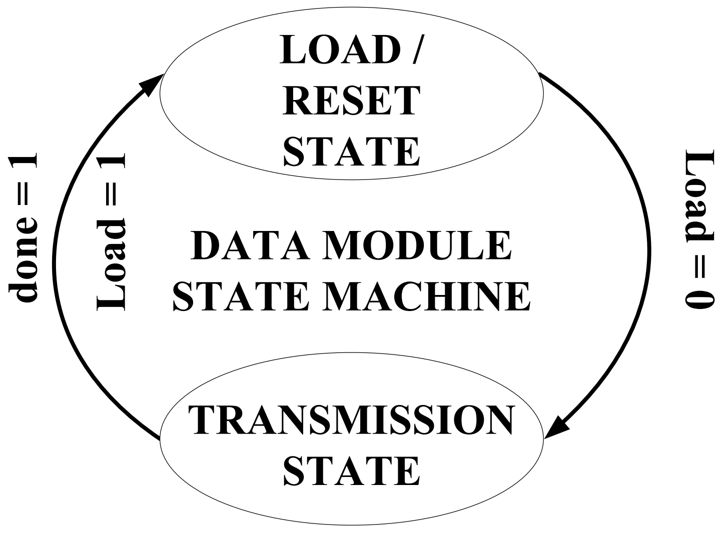


Fig 3.21: Flowchart for Data Module design.

3.6.4 CRC and PISO Module

The CRC (Cyclic Redundancy Check) method is a widely used and highly reliable way of detecting transmission errors. It can only detect errors but it cannot correct them. The calculation of CRC is a cyclic procedure which incorporates the current CRC value of the data and the CRC value of all previous data bytes. The CRC value of the entire data block is obtained by calculating CRC value of every individual byte in a data block. Mathematically we calculate the CRC by division of a polynomial using a generator polynomial. The remainder obtained from this division is the CRC value of the data used for division. The calculation of CRC always starts with an initial value, for large data blocks the CRC value from the preceding data byte is used as starting value for the subsequent data byte. To evaluate the received data for errors we perform CRC calculation on the received data along with the CRC value appended to it, the result should be same as a pre-determined residue (0xF0B8) [1].

When a data block is transmitted, the CRC value of the data is calculated within the transmitter and this value is appended to the end of data block and transmitted with it. The CRC value of the received data, including the appended CRC byte, is calculated in the receiver. The result is always a pre-determined residue (0xF0B8), unless there are transmission errors in the received block. Checking for this residue is a very easy method of analyzing the CRC checksum and avoids use of complex methods of comparing checksums. Care should be taken to start both the CRC calculation from the same initial value (0xFFFF) [1]. The great advantage of using CRC is the reliability of error recognition that is achieved in a small number of steps even where multiple errors are present [12]. A 16-bit CRC is suitable for checking the data integrity of data blocks up to 4 Kbytes in length, above which the performance degrades. The data blocks transmitted in RFID systems are much less than 4 Kbytes, which means we can use 16-bit CRC [1]. The specifications of the 16-bit CRC used in RFID systems are given in the ISO15693 standard shown in Fig 3.22.

CRC type	Length	Polynomial	Direction	Preset	Residue
ISO13239	16 bits	$x^{16} + x^{12} + x^{5} + 1$	Backward	‘FFFF’	‘F0B8’

Fig 3.22: CRC specifications.

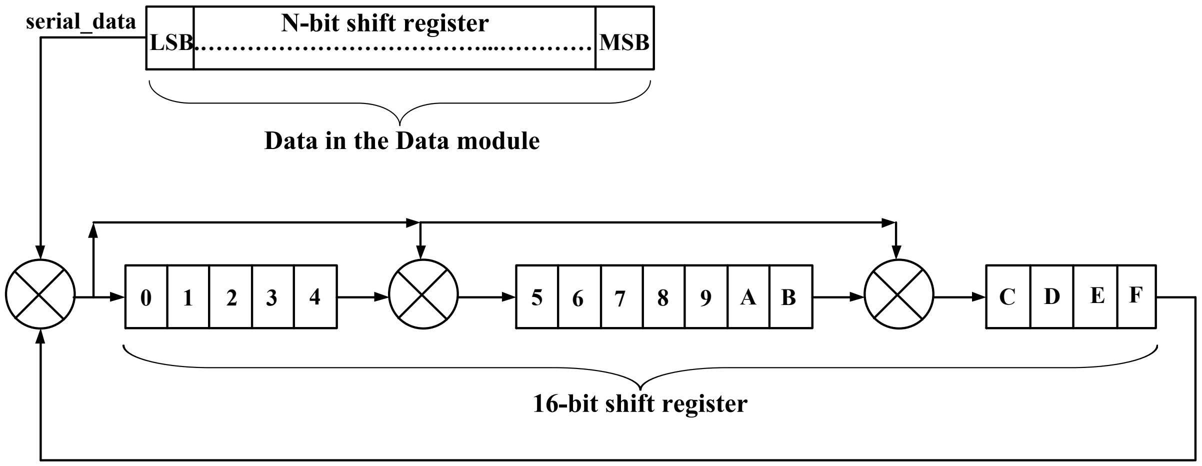


Fig 3.23: CRC operating principle.

When CRC algorithms were first developed, priority was given to realize a simple CRC processor made up of shift registers and XOR gates as shown in Fig 3.23 [1]. The CRC-16 can be calculated using shift registers as shown in Fig 3.21. To this end, a 16-bit shift registers is first set to its starting value (FFFFh in this case). The calculation is then initiated by shifting in the data bits obtained by the **data module** starting with the least significant bit of the least significant data byte into the 16-bit shift register. The polynomial division is based upon the XOR logic gating of the CRC bits shown in Fig 3.24. When all the bits have been shifted out of the register, the calculation is complete and the content of the 16-bit shift register represents the desired CRC [12]. The polynomial used in CRC-16 calculation called the generator polynomial is given by, $X^{16} + X^{12} + X^5 + 1$. So the XOR logic gates X1, X2 and X3 represent the polynomial X^{16}, X^{12}, X^5 respectively.

The VHDL code written for this module does a structural description of the circuit given in Fig 3.24. A separate behavioral description of D flip-flop is provided. Appendix gives the code used for this module.

The VHDL entity is given below,

```
entity crc_module is
port ( serial_in: in  STD_LOGIC; //Serial data coming from data module
clk            : in  STD_LOGIC; // clk_512 = 26.484 KHz clock
enable         : in  std_logic;
reset          : in  STD_LOGIC;
crc_out        : out std_logic_vector(15 downto 0)); // 16-bit CRC available in parallel
end crc_module;

entity d_ff is
port ( data : in  STD_LOGIC; // D1 input
clk         : in  STD_LOGIC; // clk_512 = 26.484 KHz clock
set         : in STD_LOGIC;
Q           : out  STD_LOGIC); // D0 output
end d_ff;
```

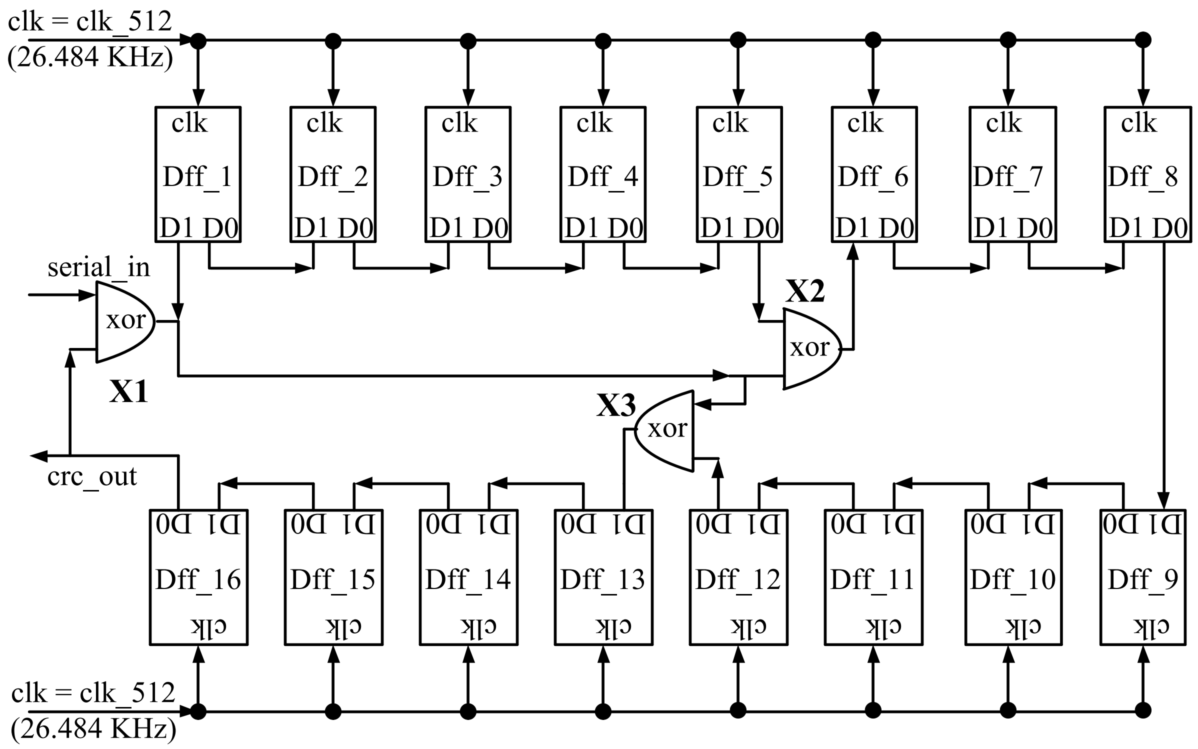


Fig 3.24: The circuit to realize CRC-16 calculation.

The signal **serial_in** is the serial data coming in from the **data module**. **Enable** signal starts the CRC module. Sixteen D flip-flops are used as shift registers to shift out data serially into the register array, at the end of calculation all the N-bits in the **data module** would have been shifted out and the content of the 16-bit shift register is the CRC checksum of the N-bit data.

The **PISO module** is basically a 16-bit parallel in and serial out module which receives CRC data parallelly from the **CRC module** and shifts out data serially for transmission via mux 4:1. This module was designed because as we know the CRC module computes using shift registers. At the end of computation the data in the shift register is the computed CRC value, but this value is available parallelly. To transmit data serially we came up with a small PISO module. The flow diagram for the operation of **PISO module** is shown in Fig 3.25.

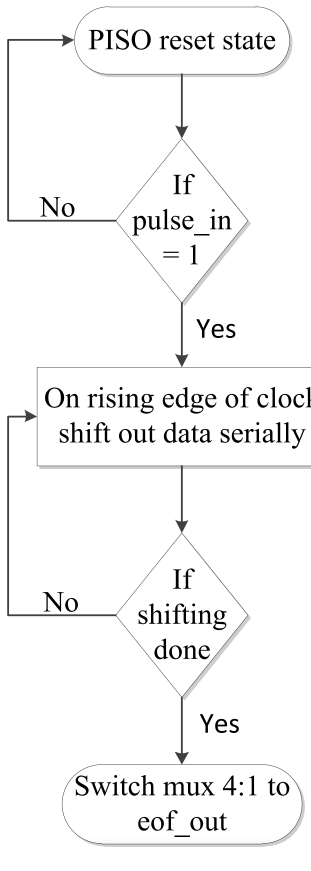

PISO reset state
If pulse_in = 1
No
Yes
On rising edge of clock shift out data serially
If shifting done
No
Yes
Switch mux 4:1 to eof_out

Fig 3.25: Flowchart representing operation of the PISO module.

The VHDL code for this module is a simple behavioral description of the above flowchart, which can be seen in appendix. The VHDL entity for this model is given below,

```
entity PISO is
port( pulse_in          : in std_logic; // Data module sends this pulse to start PISO
clk               : in std_logic;
data_vect_in      : in std_logic_vector(15 downto 0); // 16 bit parallel data from CRC
serial_out        : out std_logic; // CRC data sent out serially
done_to_eof       : out std_logic; // module done pulse sent to eof module
done              : out std_logic); // done pulse sent to switch the mux41
end PISO;
```

3.6.5 Manchester Module

The data bits in the transmitter must be Manchester encoded before transmission according to the ISO15693 standard. In Manchester code the value of a bit is defined by the change in level (negative or positive transition) within a bit window (Tbit). A logic 0 in this module is coded by a negative transition and a logic 1 is coded by a positive transition as shown in Fig 3.26. The no transition state is not permissible during data transmission and is recognized as an error. The Manchester code is often used for data transmission from transponder to the reader based upon load modulation using a subcarrier.

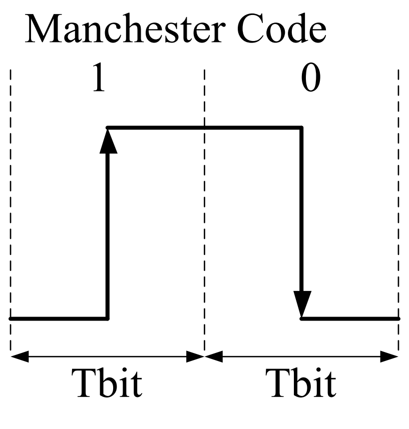


Fig 3.26: Manchester code.

To convert the data into Manchester coder before transmission we thought of many different approaches and the simplest way of doing it was to just XOR the data with the clock. The reason why it would work is that the data from the data module is being sent at a rate of $f_c/512$ = 26.48 Khz, so if we use the same clock then XOR of these two will give us the Manchester encoded data as shown in Fig 3.27.

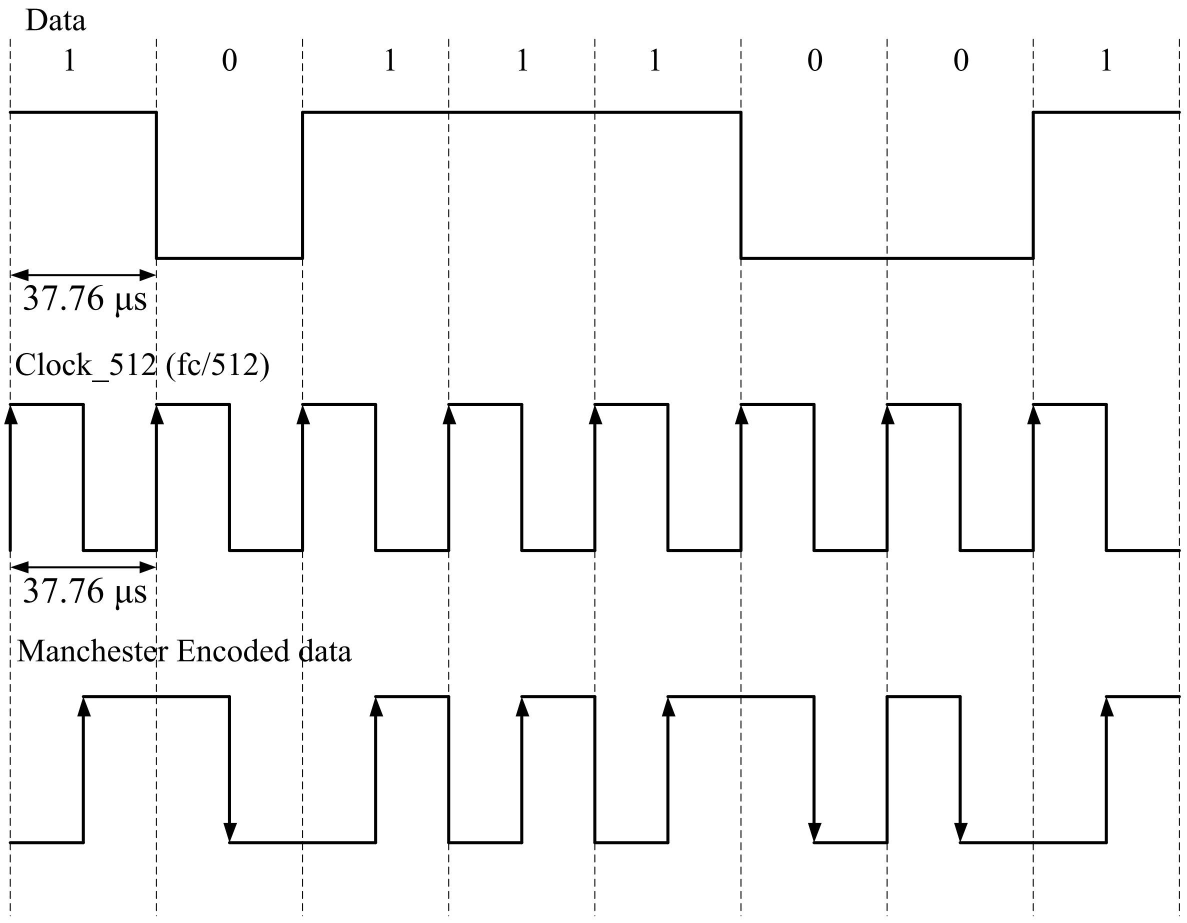


Fig 3.27: Example for Manchester Encoding.

The VHDL code written for this module is a simple behavioral model which takes in the incoming data and XOR gates it with clock (clk_512). The VHDL entity for this module is given below,

```
entity manchester is
port( data_serial_in  : in std_logic; // Serial data coming in from 2:1 mux
clock                 : in std_logic; // clk_512 = 26.484 KHz clock
enable                : in std_logic;
manchester_out        : out std_logic); // Manchester encoded data going out
end manchester;
```

3.6.6 Delay Module

The flowchart describing operation of the delay module is shown below in Fig 3.28. From

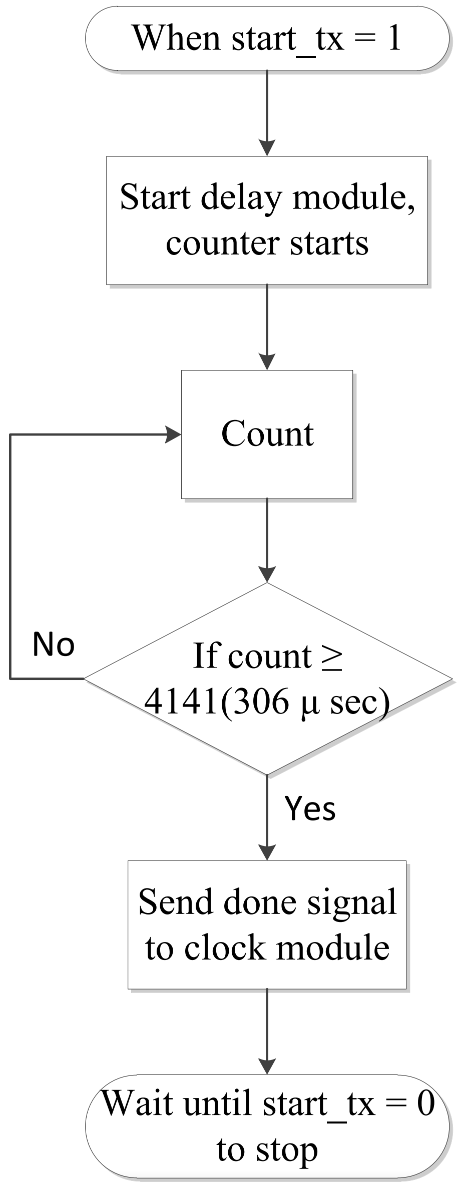

When start_tx = 1
Start delay module, counter starts
Count
No
If count ≥ 4141(306 μ sec)
Yes
Send done signal to clock module
Wait until start_tx = 0 to stop

Fig 3.28: Flowchart describing the operation of delay module.

The RFID command transmission follows certain timing protocol described in Table 2.5. So to provide a nominal delay of **t1** which is equal to 311.5 μs (4224/ f_c) we have designed this delay module. This delay of 311.5 μs is calculated from the time end of frame of received data to the start of frame of transmission data. In our design we have to account for the time the control module takes to change state and send control signals to start the transmitter. So inside the delay module we wait for 306 μs before we send **done** signal to start the clock module. The VHDL code for this module is again a behavioral model which resets all signals when **start_tx** signal is 0 and starts counting when it becomes 1. The VHDL entity for this module is given below,

```
entity delay_module is
port ( start    : in std_logic; // Start signal from the control module
clk             : in std_logic;  // 13.56 MHz clock extracted from the incoming RF signal
enable          : in std_logic;
done            : inout std_logic); // Done signal to SOF module to start Transmission of
response
end delay_module;
```

3.6.7 Clock Module

The clock module is a simple clock divider which divides the incoming 13.56 MHz clock into three smaller clocks namely **clk_32** (fc/32 = 423.75 KHz), **clk_256** (fc/256 = 52.97 KHz) and **clk_512** (fc/512 = 26.484 KHz). We make use of a ten bit counter which starts counting once it receives start signal. The 5th, 7th and 8th bits of the counter are used to derive clk_32, clk_256 and clk_512 respectively. The **start** signal to the clock module is given by the **done** signal from the delay module. The VHDL code for this module is written in behavioral model and the code is presented in appendix, but the VHDL entity is given below. The state diagram describing the operation of this module is given below in Fig 3.29.

```
entity clock_delay is
   Port ( clk    : in  STD_LOGIC;
          start     : in  STD_LOGIC;
          clk_32  : out  STD_LOGIC;
          clk_256 : out  STD_LOGIC;
          clk_512 : out  STD_LOGIC);
end clock_delay;
```

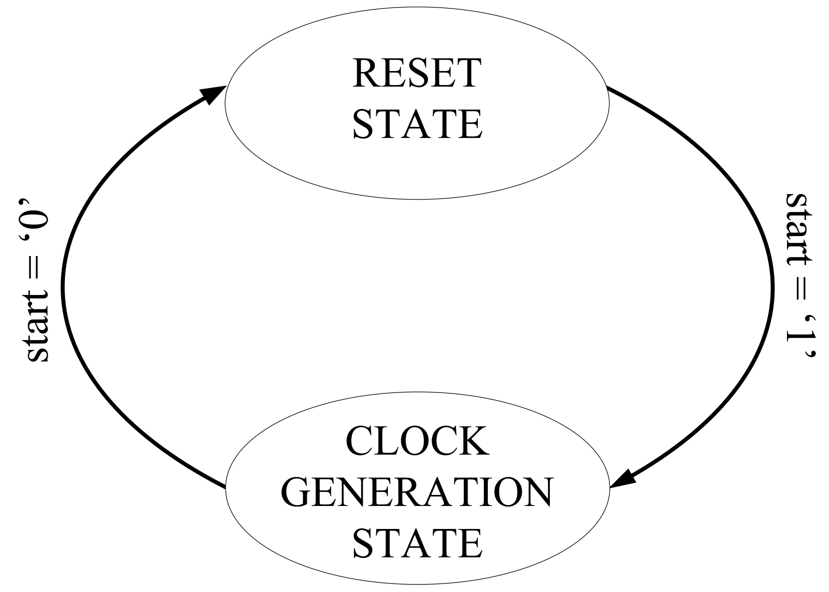


Fig 3.29: State diagram describing the operation of clock module.

3.6.8 Data Routing and Carrier Multiplication

From the architecture of the frame encoder shown in Fig 3.14 we can see that two mux's have been used for routing data. The 2:1 mux shown in Fig 3.30 is used to route data from data module and PISO module in a sequence. Initially the data module sends out 40 bits of data which is routed into the Manchester module followed by 16 bit CRC by PISO module. The equation for this 2:1 mux derived from the truth table shown in Fig 3.31 is given by

output = (Input(0) and (not sel)) or (Input(1) and (sel)).

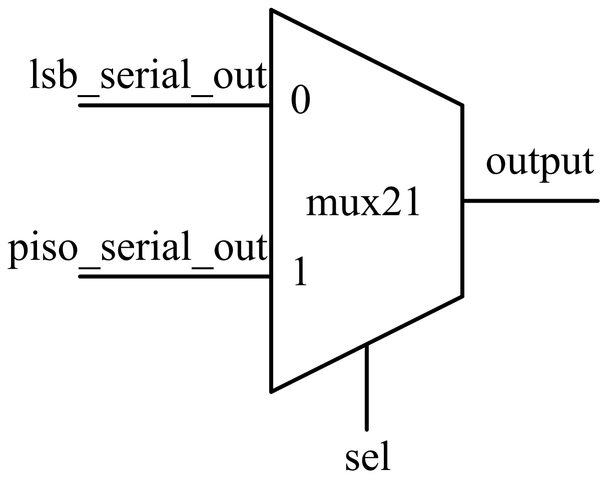


Fig 3.30: Mux 2:1.

sel data_module_done	output
0	Input 0 (lsb_serial_out)
1	Input 1 (piso_serial_out)

Fig 3.31: Truth table for Mux 2:1.

The 4:1 mux shown in Fig 3.32 is used to route data from start of frame module, Manchester module and end of frame module in a sequence. Initially the start of frame sends out its data bits 00011101 shown in Fig 3.14 at 52.97 KHz (fc/256) rate, once it is done the **sof_done** signal is raised which switches the mux output from Input 0 to Input 1. Similarly once Manchester module is done sending all 56 bits (40 bits data and 16 bits CRC) it will raise **piso_module_done** which switches the 4:1 mux output from Input 1 to Input 2. Once end of frame is done sending out all its data bits it raises **eof_done** signal which will disable the output of 4:1 mux. From the equation given below for the designed 4:1 mux we can see how the output of mux never changes to the fourth input **NA** as this condition never occurs**.**

output = ((Input(0) and (not sel(0)) and (not sel(1))) or (Input(1) and sel(0) and (not sel(1))) or (Input(2) and sel(0) and sel(1))) and disable.

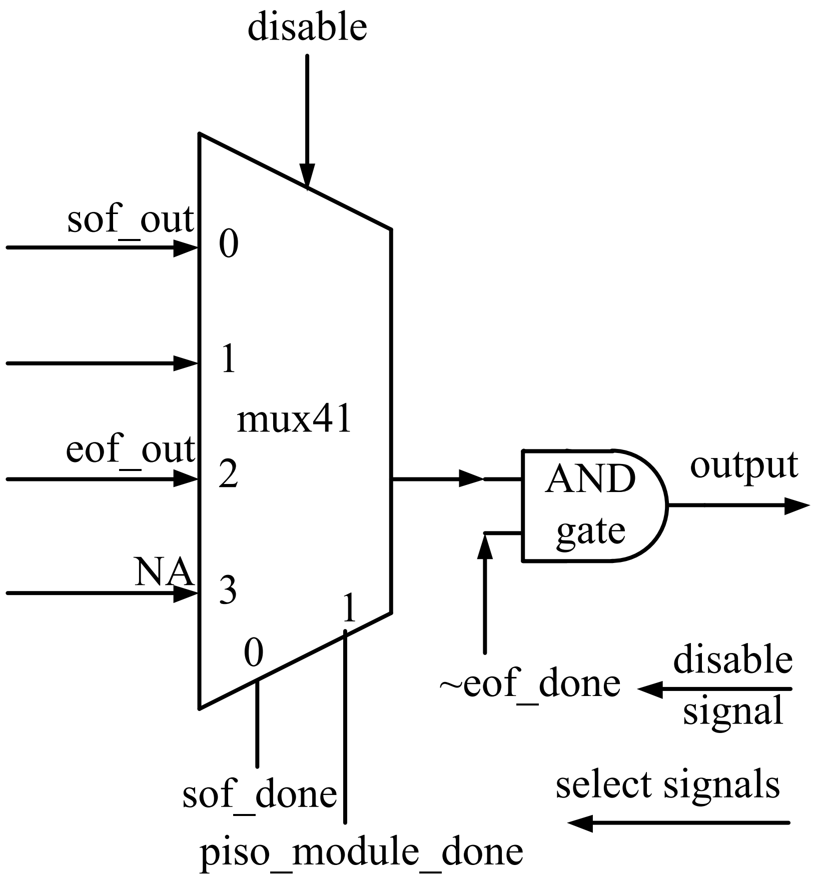


Fig 3.32: Mux 4:1.

The VHDL code is the behavioral description of the above shown equation and it is given in appendix, but the VHDL entity for this module is given below,

```
entity mux41 is
port(   input   : in std_logic_vector(2 downto 0); // 3 input signals
sel             : in std_logic_vector(1 downto 0); // 2 select signals
disable         : in std_logic;
output          : out std_logic);
end mux41;
```

Carrier multiplication is required to be performed on the data before transmission. The data bits coming out of 4:1 mux are sent out for transmission through load modulation. But before load modulation the data has to be multiplied with the subcarrier with frequency f_s = 423.75 KHz (f_c/32) to achieve this multiplication we use AND gate as shown in Fig 3.14. We logically AND data coming from 4:1 mux with a clk_32 and then send that signal for load modulation.

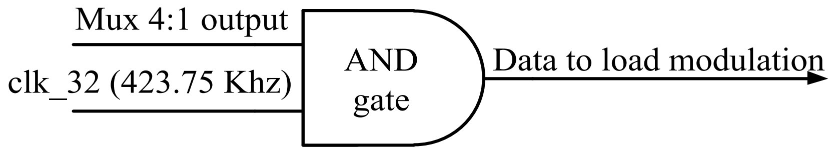
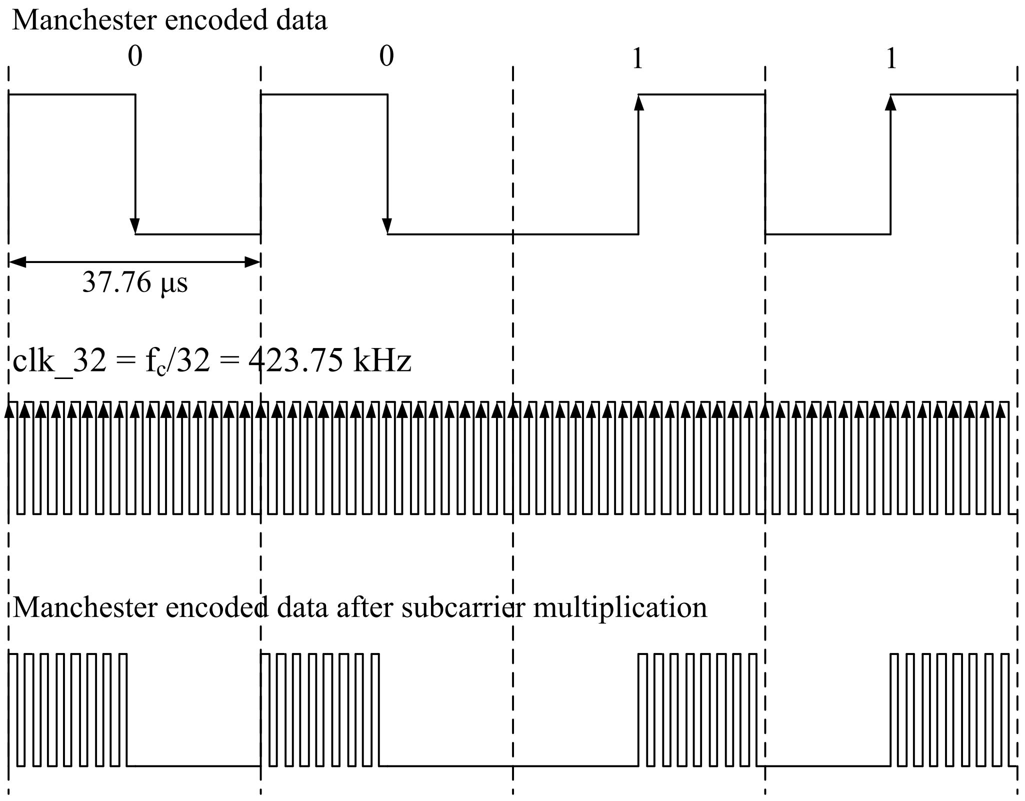


Fig 3.33: Subcarrier multiplication using AND gate.



Manchester encoded data
0
0
1
1
37.76 μs
clk_32 = f_c/32 = 423.75 kHz
Manchester encoded data after subcarrier multiplication

Fig 3.34: Subcarrier multiplication.

From the above Fig 3.34 we can see how the data with subcarrier looks before it is sent out through load modulation. The VHDL code for this module is a simple behavioral description of the AND gate and it is given in appendix.

Summarizing this chapter, we have seen the architecture of the digital core in section 3.3. From the architecture we have seen the part that digital core plays for successful operation of the digital core. The digital core has three modules: 1) frame decoder, 2) control module and 3) frame encoder. The Frame decoder decodes the incoming data packet from the reader, the control module executes a command according to the data it receives and the frame encoder responds to the reader. The operating principle of all three modules was discussed in-detail, especially the complex process of frame encoder. The frame encoder has eight sub modules which were individually discussed with state diagrams, flowcharts, block diagrams and waveforms. The VHDL code written for all the modules and sub-modules are given in appendix, the flowcharts, waveforms and in some cases block diagrams help us to understand the written code.

In the next chapter we will see results for the above design. Simulation results for all sub modules of the frame encoder, design of the test bed for evaluating the VHDL code and results obtained through the test bed will be discussed. Another test bed using a PCB was designed. Design and results from that PCB will be seen in the next chapter.

CHAPTER 4

RESULTS

4.1 Introduction

This chapter will focus on the results obtained through simulation using Xilinx ISE and from the developed test-bed. During the design phase each module was separately designed and tested. The simulated outputs of every module are shown below and they are explained in brief.

4.2 Validation of the Design

The developed code was simulated successfully to see the response given by the transmitter for a read command sent by the reader, now to test it practically a test-bed has been developed.

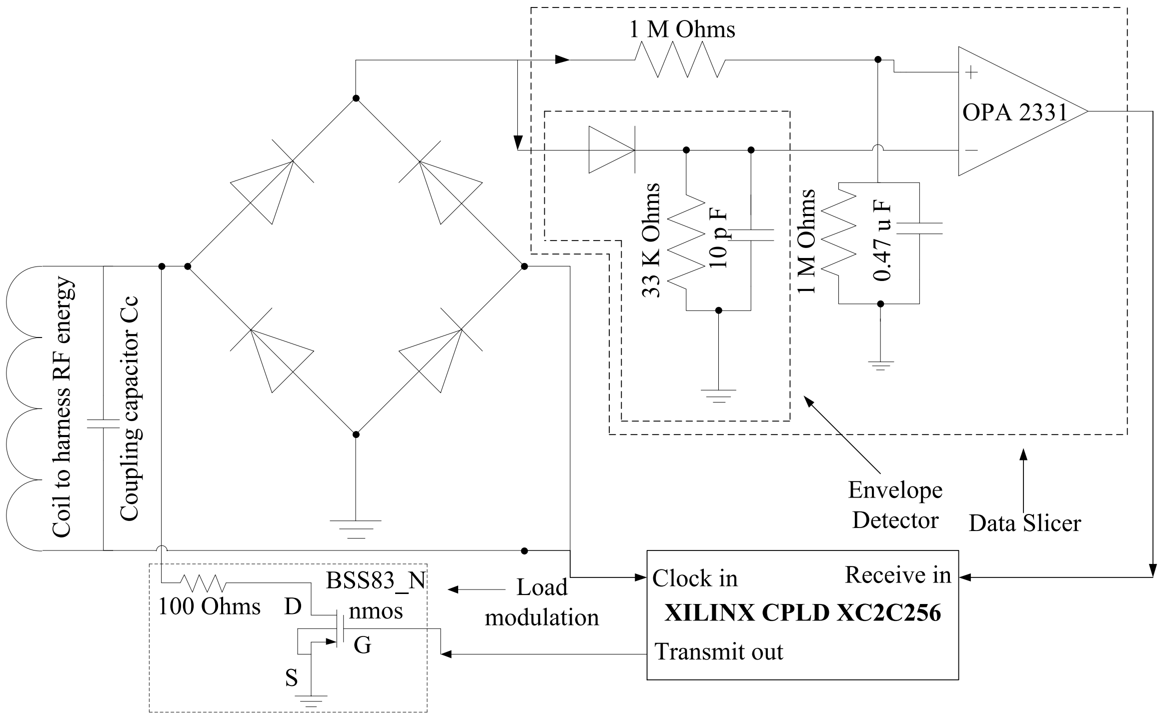


Fig 4.1: Test-bed to test the developed digital core.

The test-bed consists of a full wave rectifier, a data slicer, CPLD and a load modulation unit. The coil used to harness the RF energy and data is connected to the full wave rectifier; the rectified output is then fed to a data slicer. The data slicer unit consists of a low-pass filter and a comparator. The data slicer is used to demodulate the incoming signal by removing all the high frequency components of the signal, the capacitor and resistor form a low-pass filter to filter out the carrier frequency. The filtered signal is then compared with a reference signal to get a clean full rail DC signal which is then sent to the CPLD for analysis. The developed code was mapped into the CPLD, which decodes the incoming signal and performs required operation. The response packet to the reader is generated by the CPLD and sent out through load modulation unit. The load modulation unit consists of an NMOS (BSS23_N by NXP) transistor for switching and a 100Ω resistor. The resistor is used to load the coil so that the reader can see the incoming signal. The test-bed was successful and the results are shown in the next section.

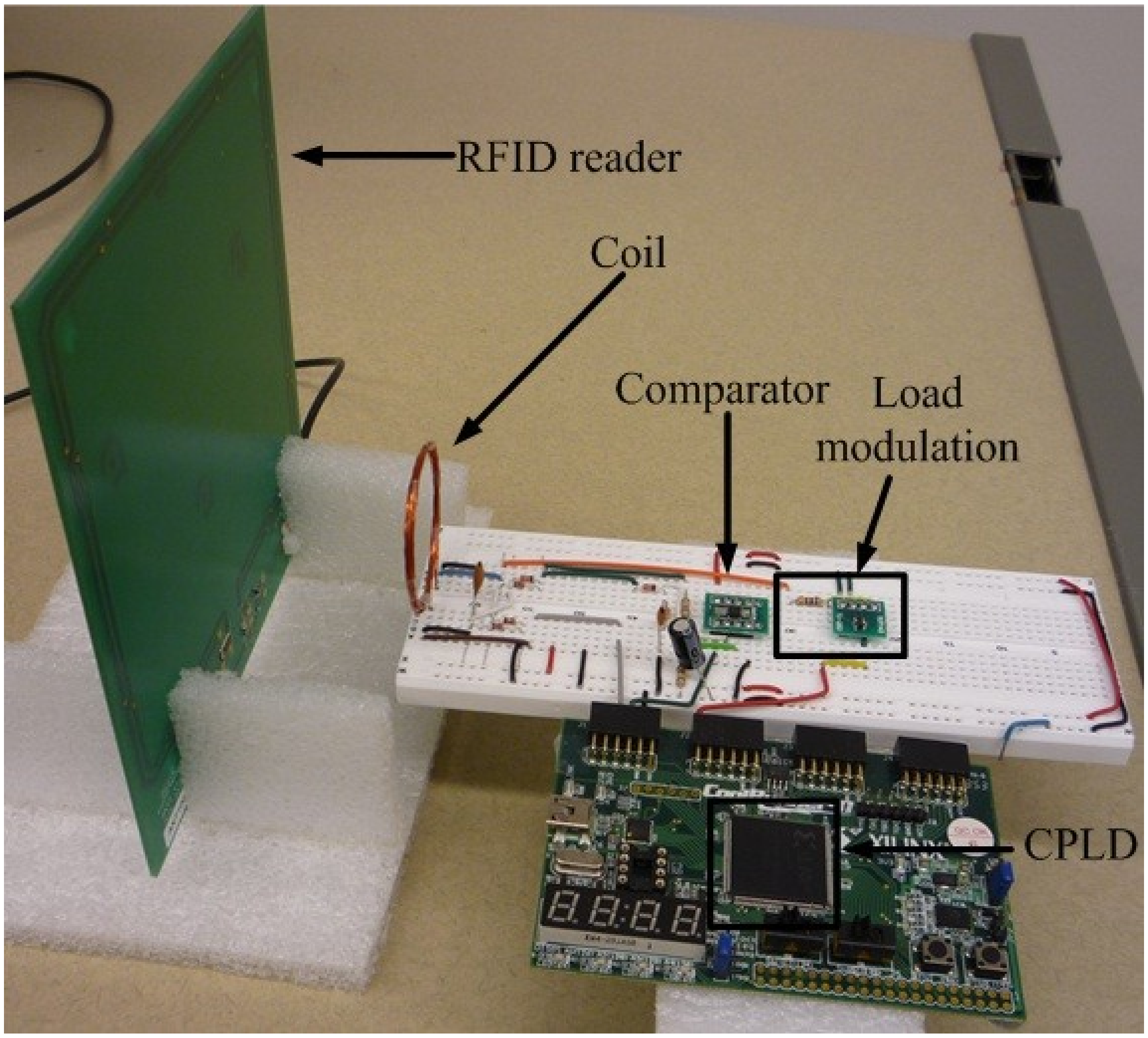


Fig 4.2: RFID reader and test-bed setup.

Finally a printed circuit board was designed and fabricated to incorporate the designed test-bed as shown below in Fig 4.3.

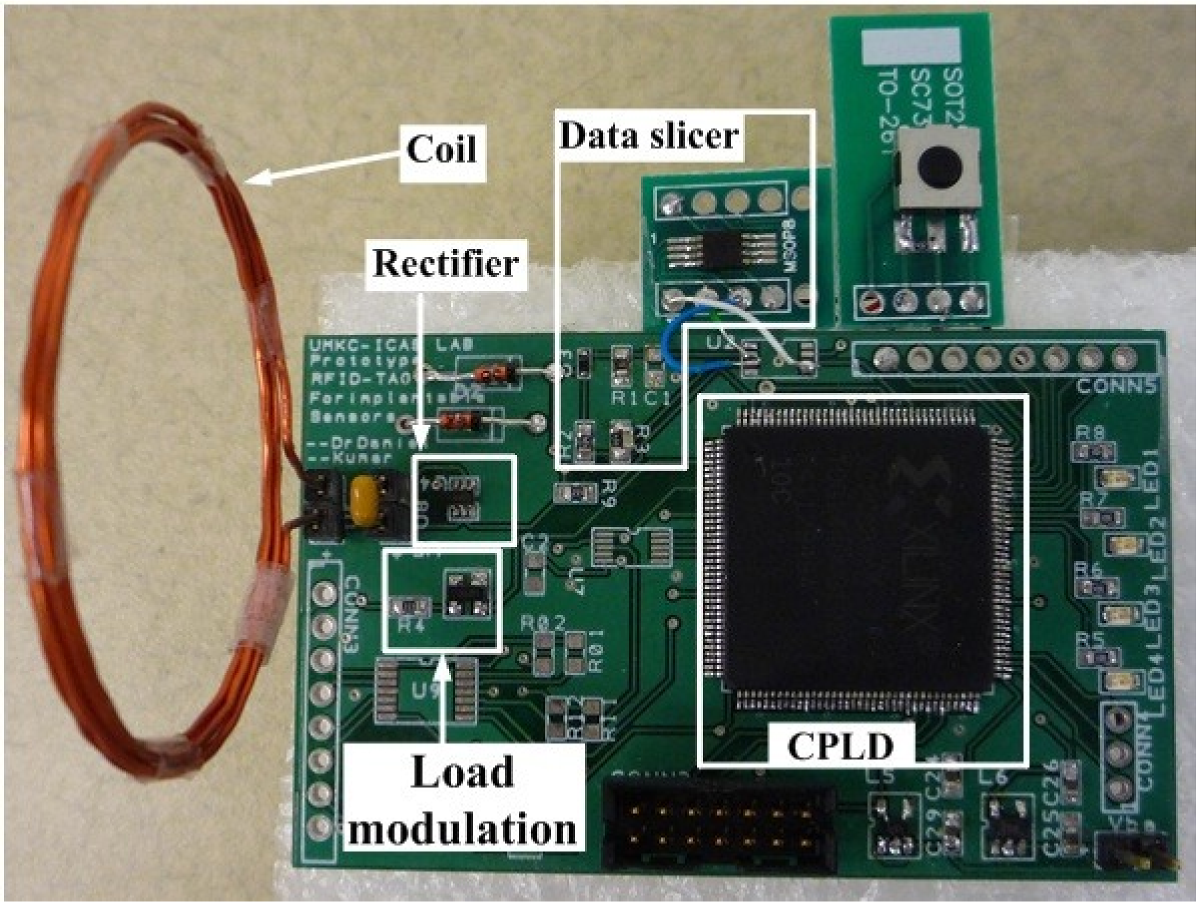


Fig 4.3: Test-bed on a printed circuit board.

4.3 Simulated Results

The simulated results are presented in two sections: transmitter results and control module results. The transmitter results have simulation results of all the sub blocks and the final transmitter output, this is followed by the simulated results of the control block which shows transition from one state to another in a sequential order.

4.3.1 Transmitter Results

The transmitter code consists of twelve sub-modules the simulation results are shown below with brief explanation.

Start of Frame Module:

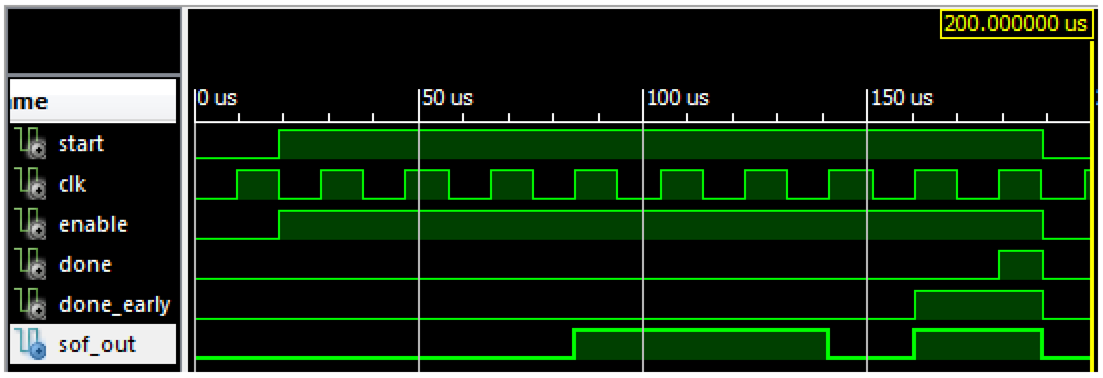


Fig 4.4: Simulated output for Start of Frame Module.

The simulation result of the start of frame module is seen above in Fig 4.4, where we can see the **sof_out** signal which is the start of frame signal generated by the module. The Start of Frame module is controlled by the enable signal which starts the module, generates the start of frame signal and stops the module when it is done. We can also see two control signals **done** and **done_early** which are enable signals for succeeding modules, the **done** signal switches the 4:1 mux whereas the **done_early** signal starts the data module. The done_early signal is raised earlier to allow data module to start at the precise moment when the start of frame module ends.

End of Frame Module:

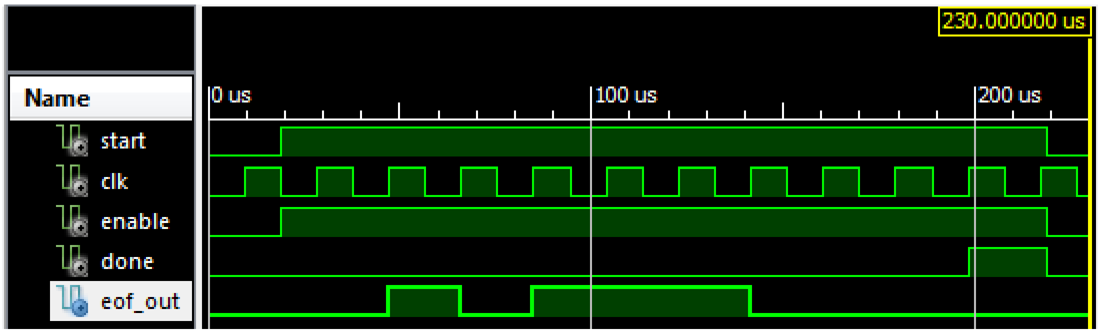


Fig 4.5: Simulated output for End of Frame Module.

The simulation result of the end of frame module is seen above in Fig 4.5, where we can see the **eof_out** signal which is the end of frame signal generated by the module. The end of frame module is controlled by the **enable** signal which starts the module, generates the start of frame signal and stops the module when it is done. The control signal **done** is generated when the end of frame module has generated the required signal. This control signal will disable the mux 4:1 which in-effect switches off the transmitter also it indicates the control module to change its state.

Clock Module:

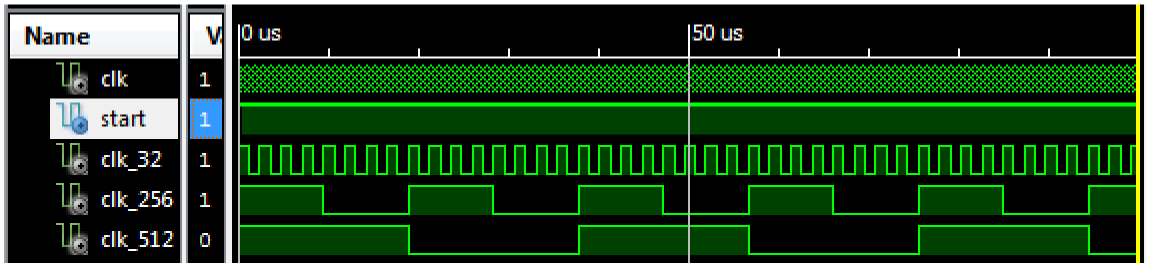


Fig 4.6: Simulated output for Clock Module.

The clock module generates all the clocks required for the operation of the transmitter. The master clock of 13.56 MHz extracted from the incoming RF field is divided into three clocks **clk_32** (13.56 MHz/32), **clk_256** (13.56 MHz/256) and **clk_512** (13.56 MHz/512) as shown in Fig 4.6 which drive all the sub systems in the transmitter module.

Delay Module:

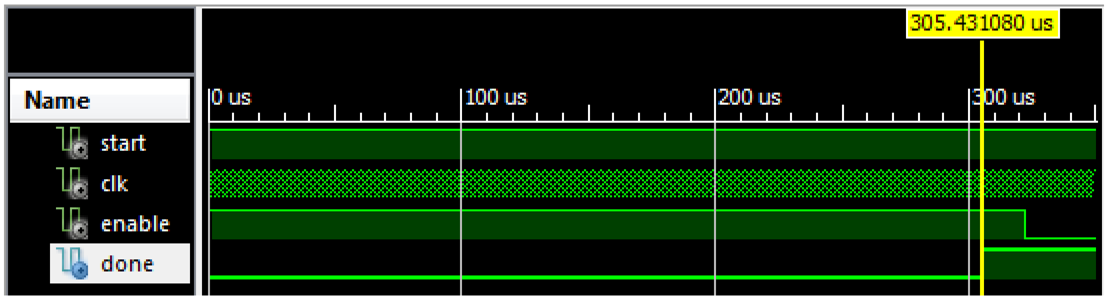


Fig 4.7: Simulated output for Delay Module.

From the section 2.6 we know that RFID command transmission follows certain timing protocol described in Table 2.5. So to provide a nominal delay of t1 which is equal to 311.5 µs (4224/ f_c) we have designed this delay module. From Fig 4.7 shown above we can see that the delay module generates a delay of 305.43 µs and then raises the done signal but the delay is supposed to be 311.5 µs. The reason is, it takes 6 µs for the control module to detect end of frame signal from the received data, change its state and send start signal to the delay module inside the transmitter.

CRC Module:

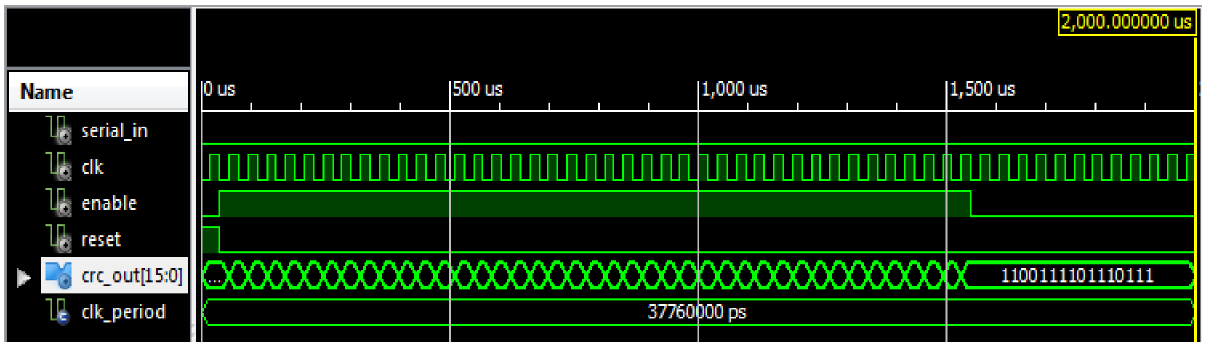


Fig 4.8: Simulated output for CRC Module.

The simulated result of the CRC module is shown above in Fig 4.8. We know from chapter 3 the operation of the CRC module. Once the data module starts sending out data, the

CRC module receives data serially and calculates the CRC value. So for 40 bits data it takes 40 clock cycles to calculate the CRC. On the 41st clock the CRC value is obtained, from the above simulated result we can see the CRC value for forty ‘0’ bits is CF77 h. This value is sent out through Manchester and 4:1 mux modules.

Data Module:

The data module sends out data in least significant bit first, the data sent out is routed into two modules namely, CRC module and Manchester module for data transmission. The data required to be sent out is initially loaded into the data module by the control module after the COMPARE state as explained in section 3.4. When the **load** signal goes low and **enable** signal goes high the data is sent out as **lsb_serial_out**. The simulated result for a test data is shown below in Fig 4.9.

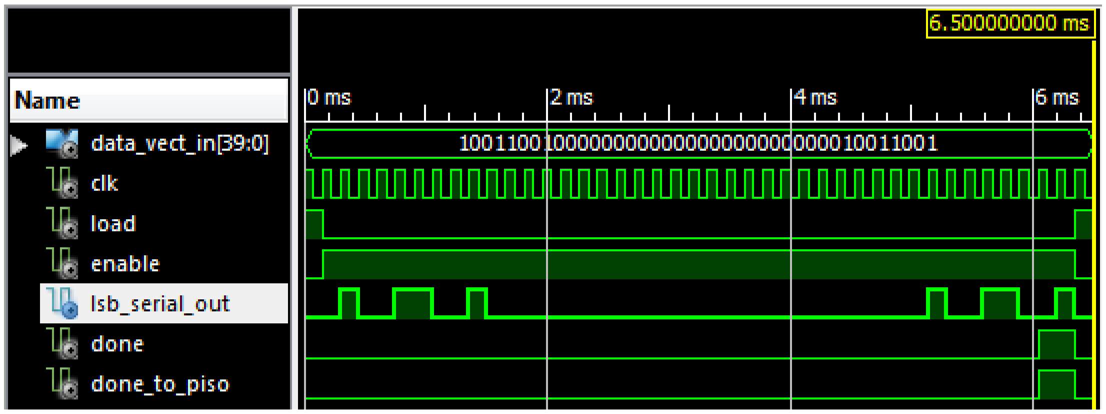


Fig 4.9: Simulated output for Data Module.

Manchester Module:

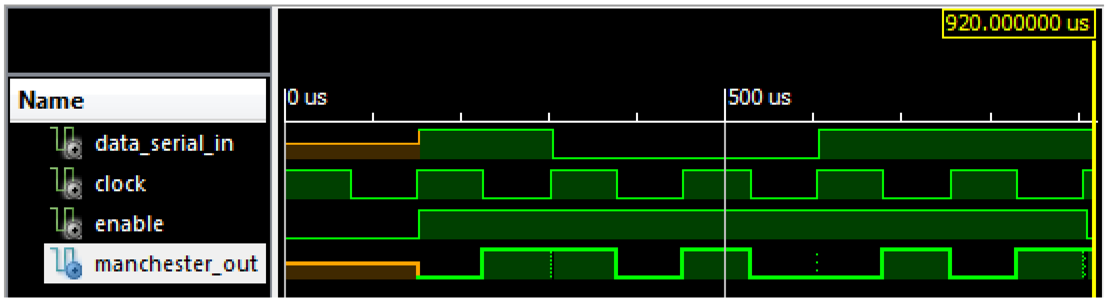


Fig 4.10: Simulated output for Manchester Module.

We know from chapter 3 that the data sent for transmission should be Manchester modulated, which is performed by this module. From the simulated result shown above in Fig 4.10 we can see that test data 10011 is Manchester modulated. ‘1’ is represented by a low to high going signal and ‘0’ is represented by a high to low signal. We can see small spikes between two signals this is because of the testing conditions that were given in the test bench, but the final result has no such spikes.

PISO Module:

The PISO module is used to send CRC data serially into the Manchester module and then out for transmission. The CRC module generates the CRC value in parallel, but the Manchester module requires data to be serial. To solve this problem we came up with this module. The 16 bit CRC value is initially sent saved in a temporary register and then sent out least significant bit first to the Manchester. The simulated result is shown below in Fig 4.11.

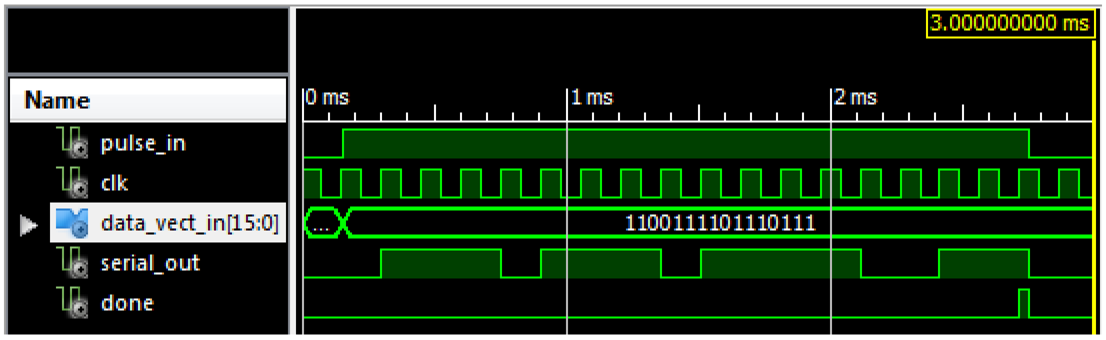


Fig 4.11: Simulated output for PISO Module.

Digital Core Transmitter Simulated Output:

The transmitter module simulated output is seen below in Fig 4.12 and Fig 4.13. The Fig 4.12 shows all the control signals, output from every module inside the transmitter module and also the final transmitted signal **tx_out**. The white arrow in Fig 4.12 points to two signals: **tx** and **tx_out** which are final transmitted signals before and after modulation. The Fig 4.13 is a magnified sample of the transmitted signal, which clearly shows clock multiplication before transmission.

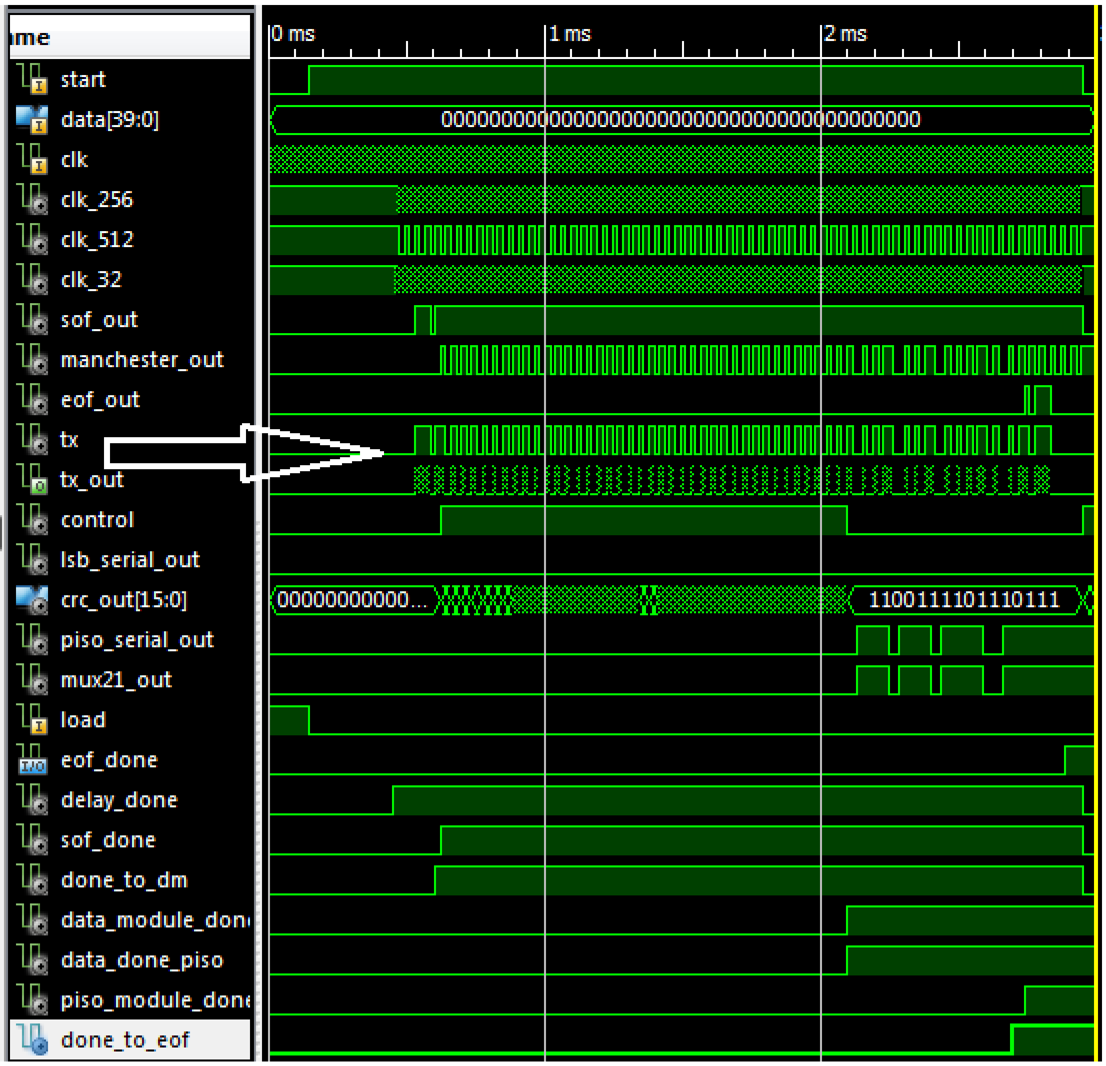


Fig 4.12: Simulated output for Transmitter Module.

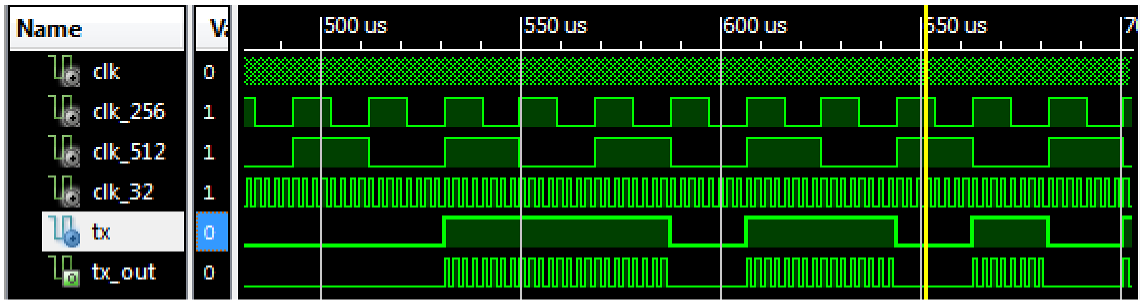


Fig 4.13: Magnified Sample of the transmitted output **'tx_out'**.

The transmitted response at the output of the CPLD was captured using high speed osciloscope as it is shown below in Fig 4.41:

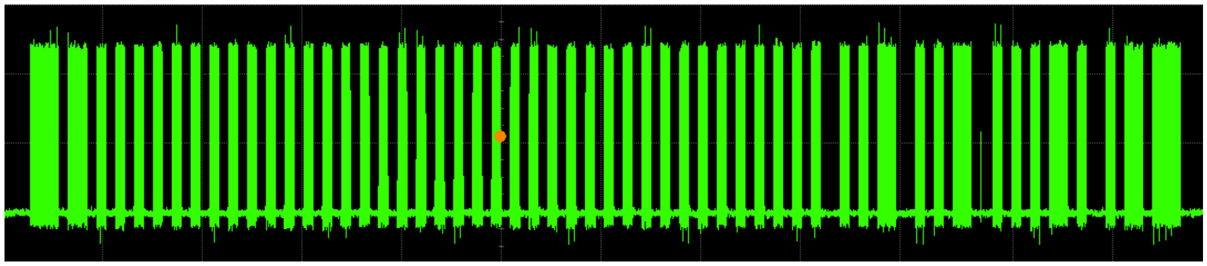


Fig 4.14: Transmitter response at the output of the CPLD.

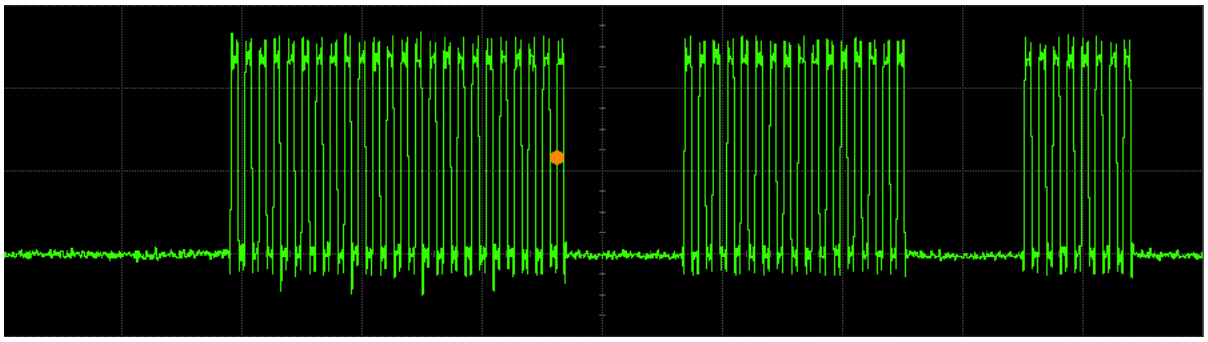


Fig 4.15: Start of frame signal generated by the CPLD.

From the above figures 4.14 and 4.15 we can see the response of the CPLD for a read request from the RFID reader. In Fig 4.15 we can see manchester encoded and subcarrier modulated **start of frame** signal generated by the CPLD. The request packet by the reader and response packet generated by the CPLD is shown below in Fig 4.16.

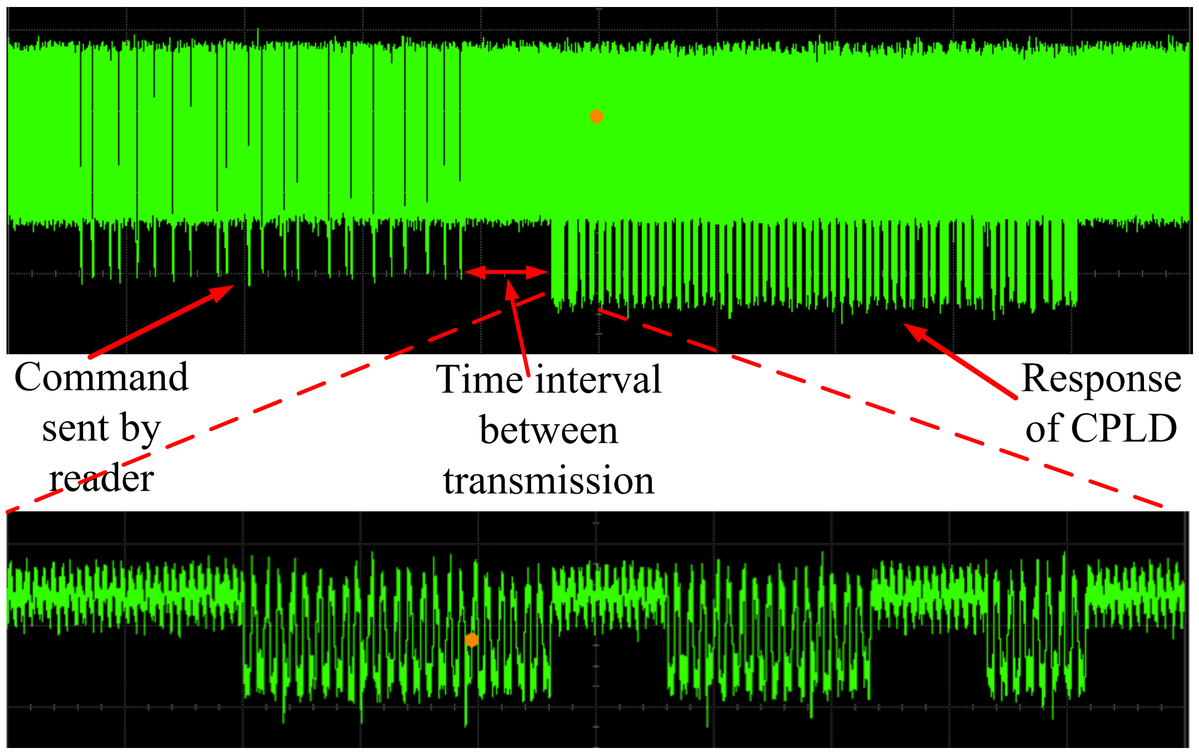


Fig 4.16: The request and response packets seen at the coil.

4.3.2 Control Module Results

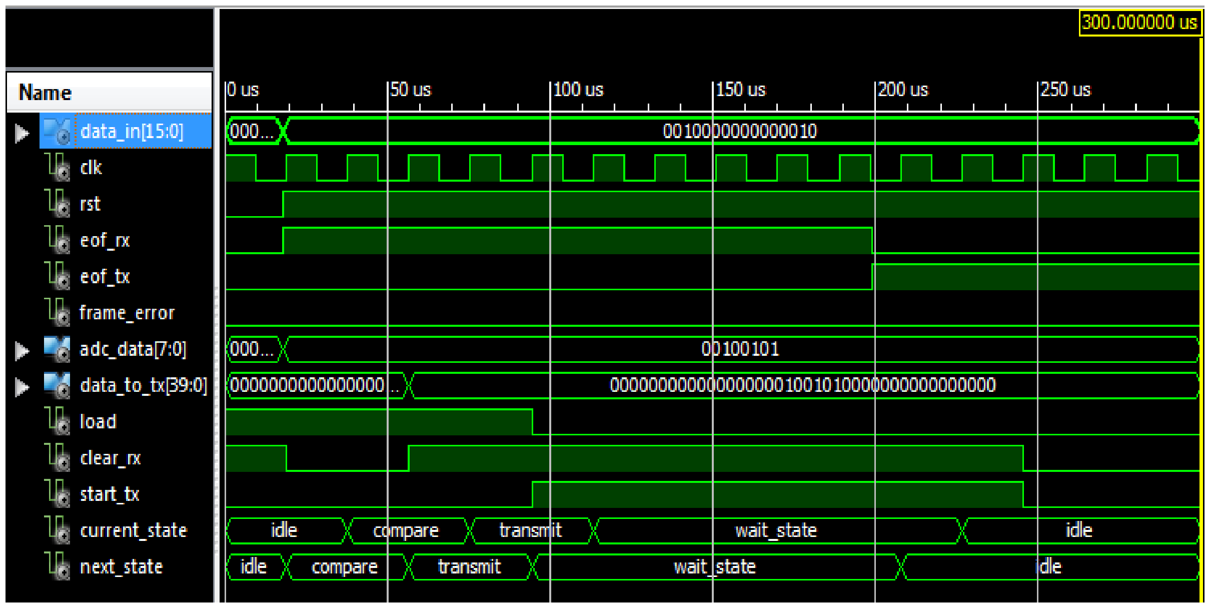
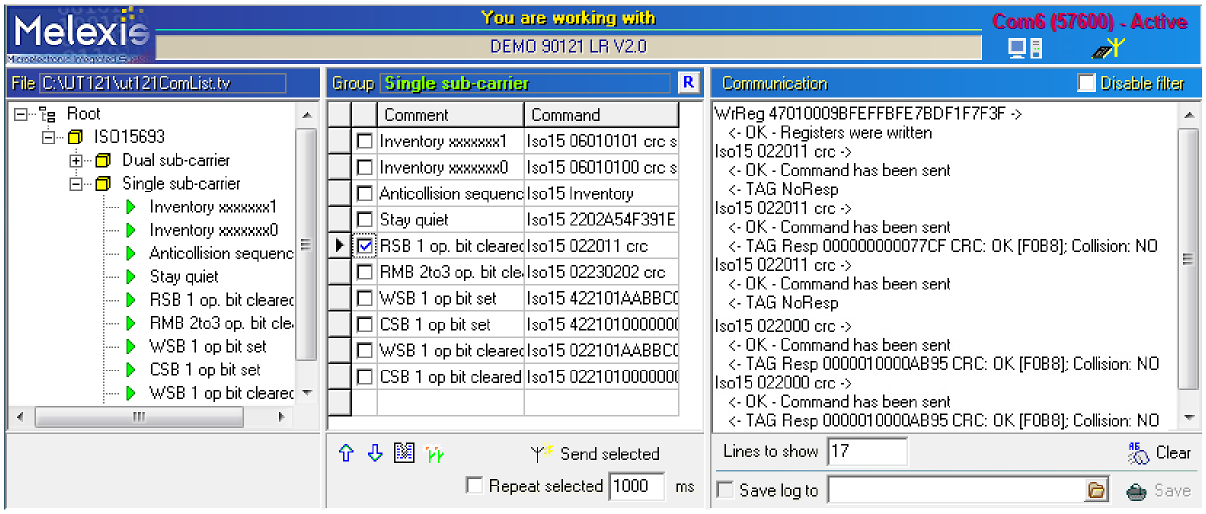


Fig 4.17: Simulated output for Control Module.

The control module is the main state-machine which controls and synchronizes the operation of the designed digital core in the RFID tag. As we have seen in section 3.4 the control module steps through many states. From the simulation result shown above in Fig 4.17 we can see that the control module steps through **idle-state, compare-state, transmit-state and wait_state**. During the idle-state most of the signals are reset, the frame decoder module is reset by the **clear_rx** signal and the transmitter module is reset by the **start_tx** signal. During the compare state the received signal is analyzed and data to be sent out is initialized into the data module. During the transmit module the **clear_rx** resets the receiver and the **start_tx** signal initiates the transmitter module. During the wait_state it waits for end of frame signal from the transmitter on receiving that signal it switches to idle state.

4.3.3 RFID Reader Output



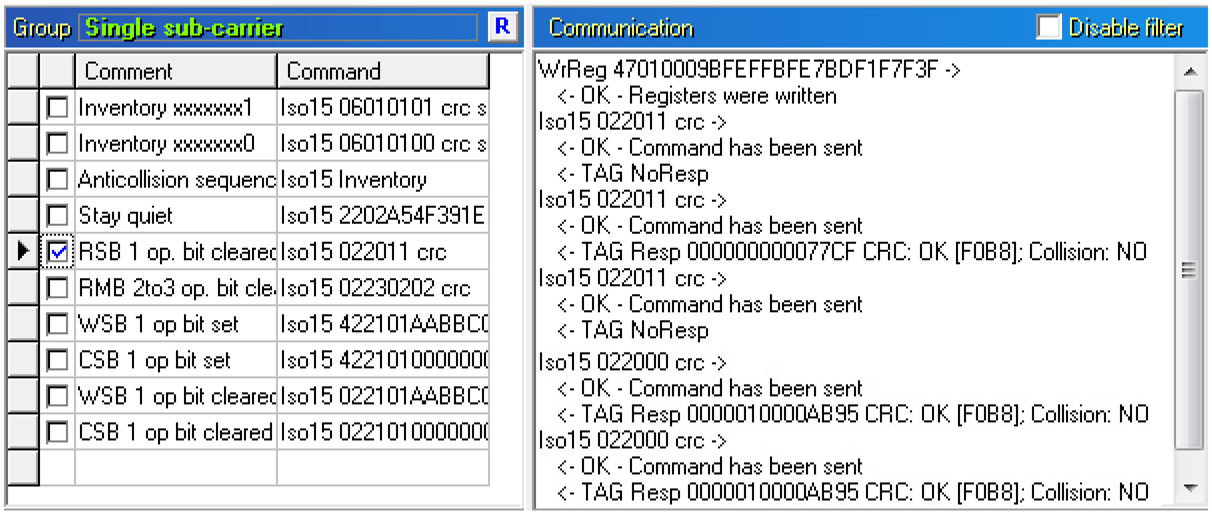
	Comment	Command
☐	Inventory xxxxxxx1	Iso15 06010101 crc s
☐	Inventory xxxxxxx0	Iso15 06010100 crc s
☐	Anticollision sequenc	Iso15 Inventory
☐	Stay quiet	Iso15 2202A54F391E
☑	RSB 1 op. bit clearec	Iso15 022011 crc
☐	RMB 2to3 op. bit cle	Iso15 02230202 crc
☐	WSB 1 op bit set	Iso15 422101AABBC(
☐	CSB 1 op bit set	Iso15 4221010000000
☐	WSB 1 op bit clearec	Iso15 022101AABBC(
☐	CSB 1 op bit cleared	Iso15 0221010000000

```
WrReg 47010009BFEFFBFE7BDF1F7F3F ->
  <- OK - Registers were written
Iso15 022011 crc ->
  <- OK - Command has been sent
  <- TAG NoResp
Iso15 022011 crc ->
  <- OK - Command has been sent
  <- TAG Resp 000000000077CF CRC: OK [F0B8]; Collision: NO
Iso15 022011 crc ->
  <- OK - Command has been sent
  <- TAG NoResp
Iso15 022000 crc ->
  <- OK - Command has been sent
  <- TAG Resp 0000010000AB95 CRC: OK [F0B8]; Collision: NO
Iso15 022000 crc ->
  <- OK - Command has been sent
  <- TAG Resp 0000010000AB95 CRC: OK [F0B8]; Collision: NO
```

Fig 4.18: RFID communication through Melexis user interface.

The Digital core was tested using the user interface provided by Melexis corp. The test-bed developed on the bread board receives commands sent by this user interface through the

RFID reader shown in Fig 3.36. The reply sent by the digital core can be seen in the communication window as shown in Fig 4.16.



		Comment	Command
	☐	Inventory xxxxxxx1	Iso15 06010101 crc s
	☐	Inventory xxxxxxx0	Iso15 06010100 crc s
	☐	Anticollision sequenc	Iso15 Inventory
	☐	Stay quiet	Iso15 2202A54F391E
▶	☑	RSB 1 op. bit clearec	Iso15 022011 crc
	☐	RMB 2to3 op. bit cle	Iso15 02230202 crc
	☐	WSB 1 op bit set	Iso15 422101AABBC0
	☐	CSB 1 op bit set	Iso15 422101000000
	☐	WSB 1 op bit clearec	Iso15 022101AABBC0
	☐	CSB 1 op bit cleared	Iso15 022101000000

```
WrReg 47010009BFEFFBFE7BDF1F7F3F ->
   <- OK - Registers were written
Iso15 022011 crc ->
   <- OK - Command has been sent
   <- TAG NoResp
Iso15 022011 crc ->
   <- OK - Command has been sent
   <- TAG Resp 000000000077CF CRC: OK [F0B8]; Collision: NO
Iso15 022011 crc ->
   <- OK - Command has been sent
   <- TAG NoResp
Iso15 022000 crc ->
   <- OK - Command has been sent
   <- TAG Resp 0000010000AB95 CRC: OK [F0B8]; Collision: NO
Iso15 022000 crc ->
   <- OK - Command has been sent
   <- TAG Resp 0000010000AB95 CRC: OK [F0B8]; Collision: NO
```

Fig 4.19: Command and Communication Window as seen in the Melexis User Interface.

CRC value for the reply sent by the digital core is calculated by the Melexis software and if it matches with the CRC value in the received signal then it accepts the packet and says there was no collision else it displays YES for collision.

4.4 RFID Communication using HyperTerminal

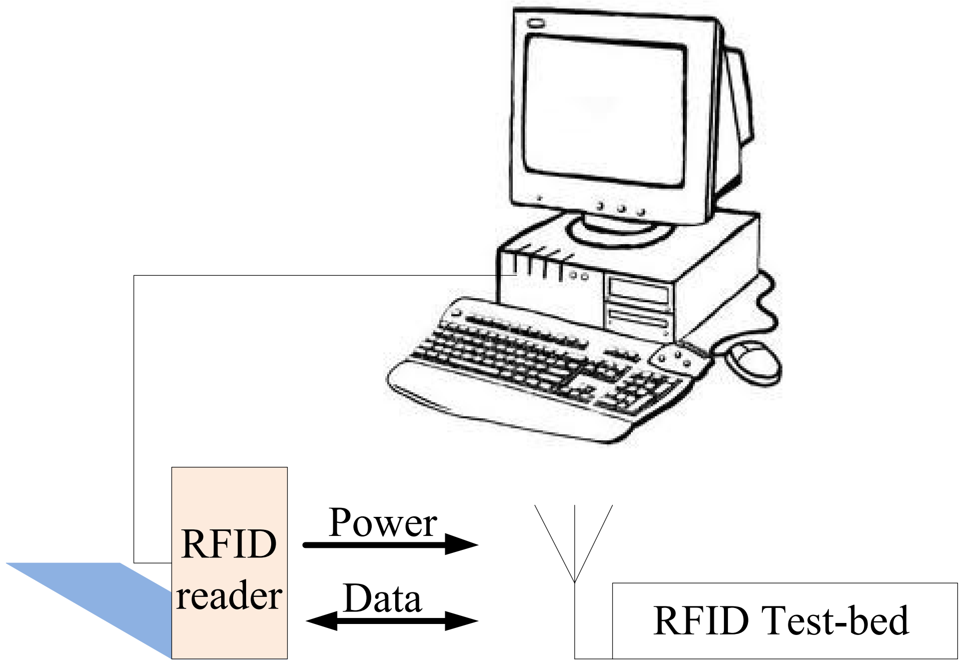


Fig 4.20: RFID communication using HyperTerminal.

We can also use HyperTerminal to communicate with the RFID tag. HyperTerminal is a terminal emulation program capable of communicating to systems through COM ports, TCP/IP Networks and Dial-Up Modems. When the HyperTerminal program runs we have to setup the COM port properties such as bit rate, data bits, flow control, parity and stop bits. Once initialized the COM port is open and we can start to type in commands and receive response from the tag.

The commands that we type into the hyper terminal are :

1) For read single block, the commnd is **Iso15 022011 crc.**
2) For write single block, the command is **Iso15 422101<u>32-bit data</u> crc (ex: Iso15 422101AABBCCDD crc).** Write single block allows user to send 32-bit data in one packet, as an example I have shown AABBCCDD as 32-bit data.

4.5 Analog Component design:

4.5.1 Clock Extractor

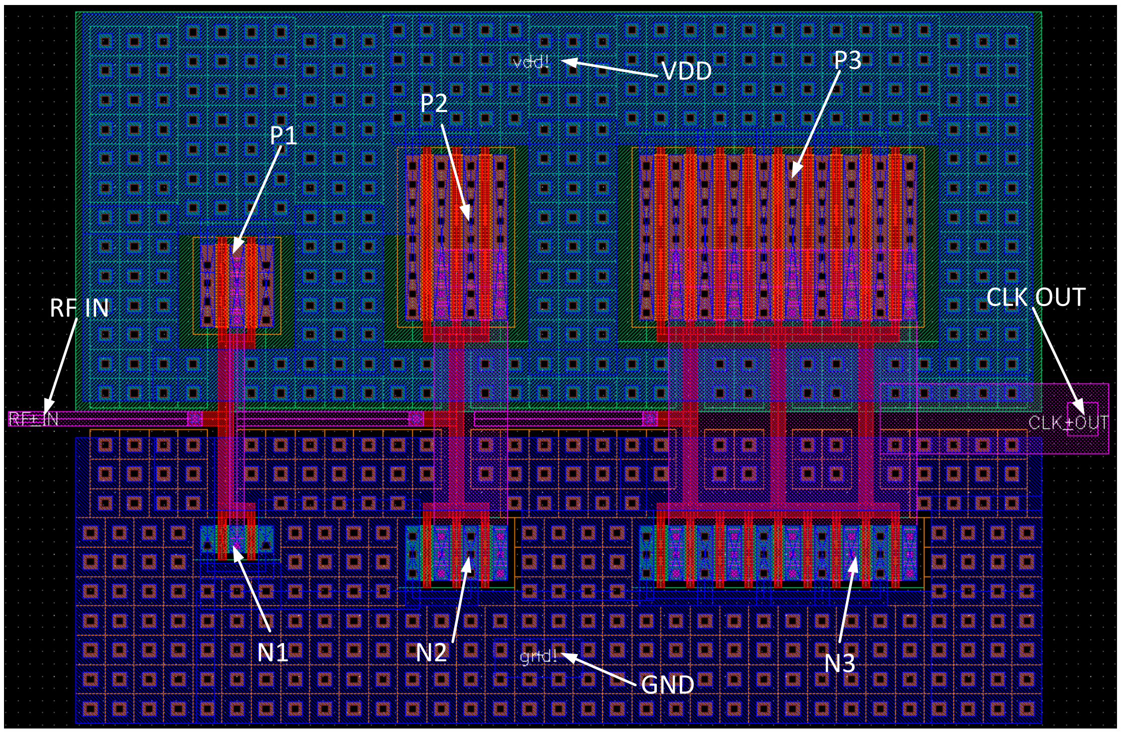


Fig 4.31 Clock extractor layout design

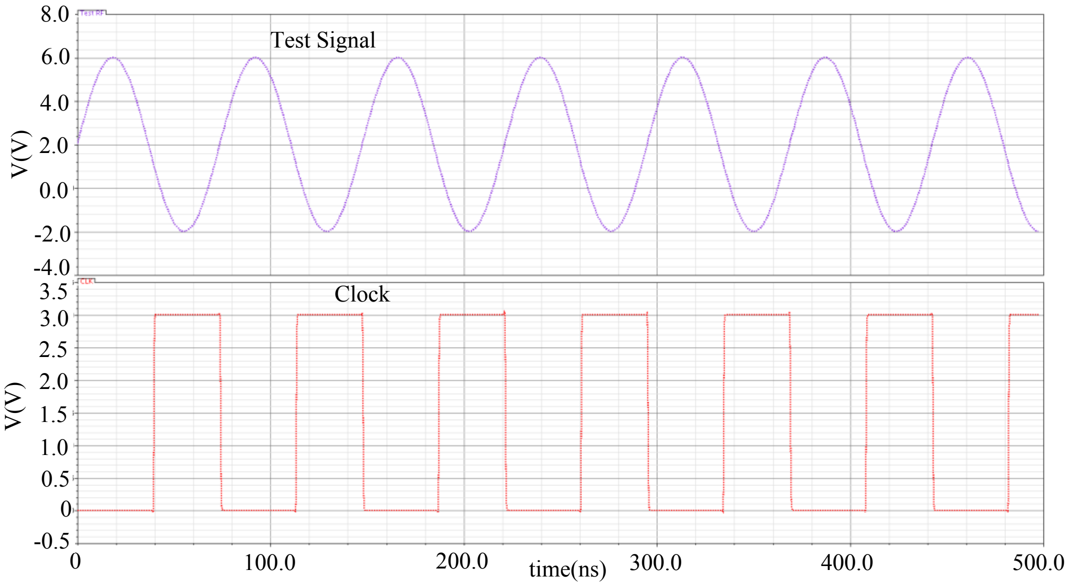


Fig 4.32 Clock extractor post-layout simulation

The layout of the clock extractor is shown in Fig. 4.31. The layout measures 86.25 µm x 65.75 µm. The post layout simulation of the clock extractor is shown in Fig 4.32. The post layout plots correspond to the schematic waveforms confirming a functional layout design.

4.5.2 Modulator and Demodulator

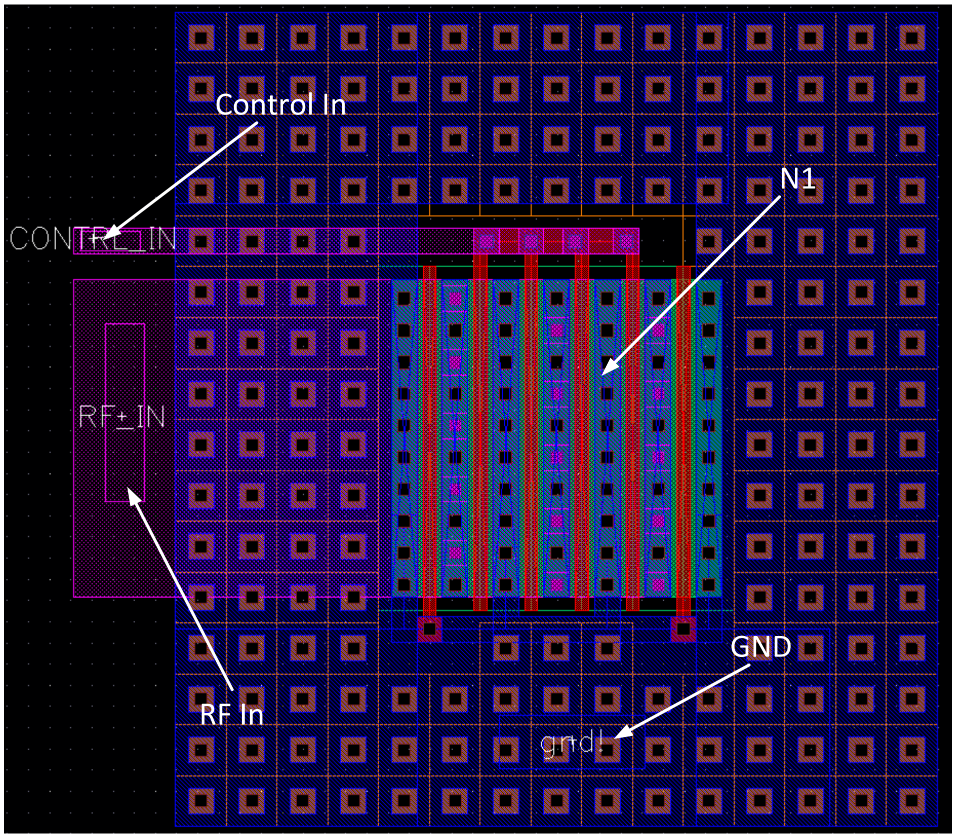


Fig 4.33 Modulator layout design

The modulator layout is shown in Fig 4.33. The layout measures 30.25 µm x 28.5 µm. Post-layout simulation is shown in Fig 4.34 which corresponds to the pre-layout simulation confirming a functional layout design. The demodulator layout is shown in Fig 4.35. The layout measures 355.260 µm x 441.400 µm.

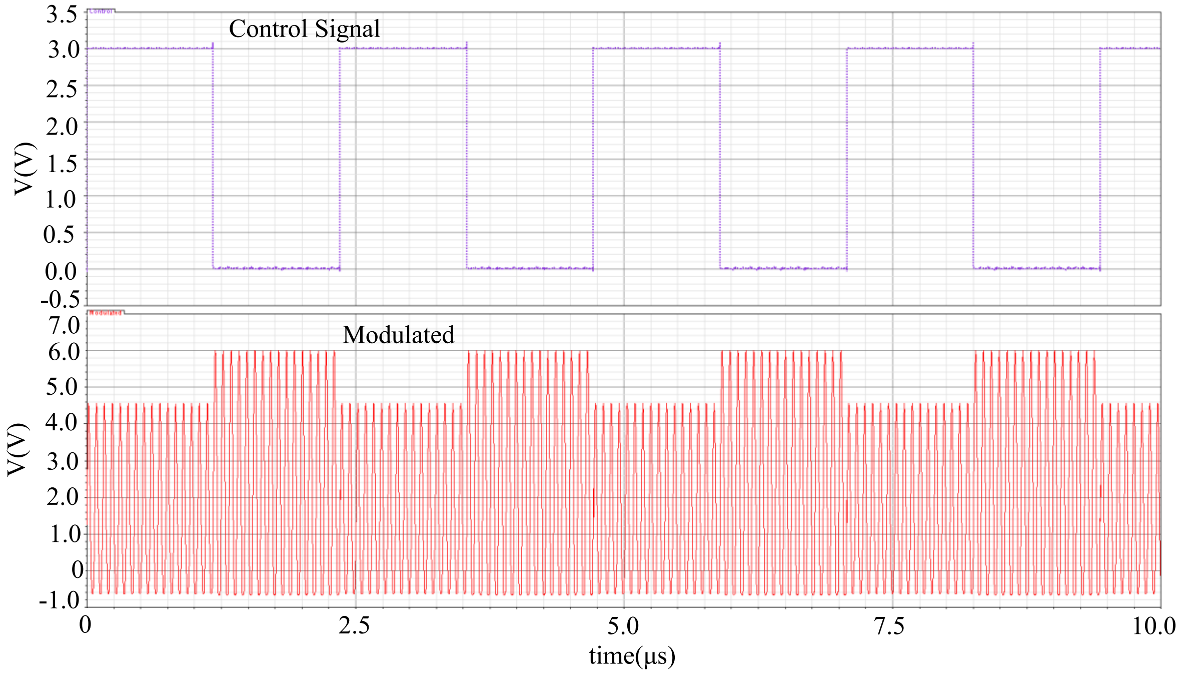


Fig 4.34 Modulator post-layout simulated waveform

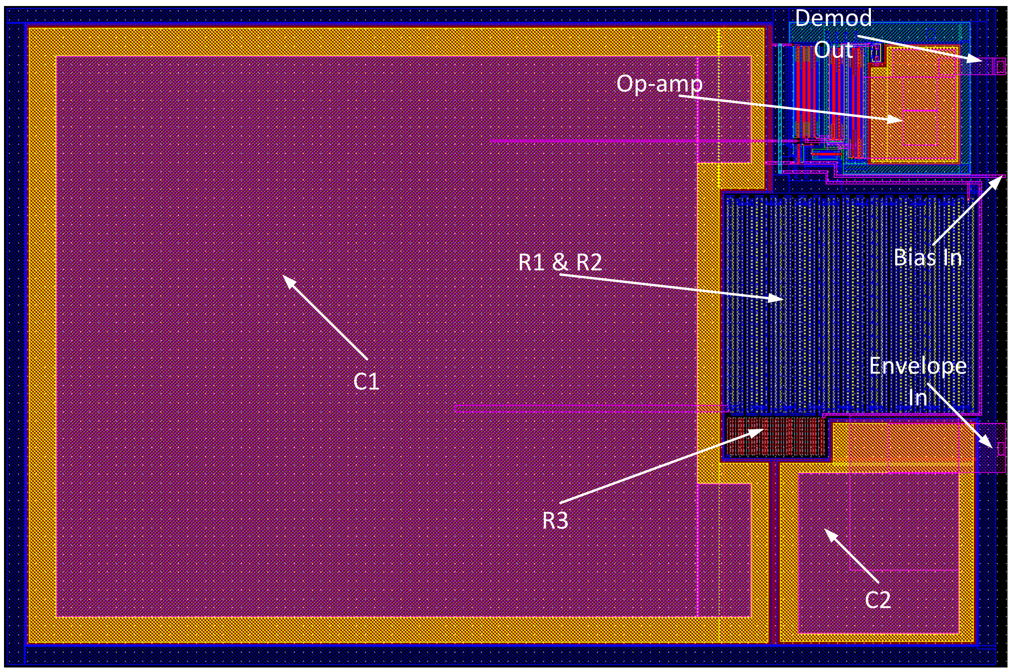


Fig 4.35 De-Modulator layout design

4.5.3 Final Chip layout

The layout of the entire analog core is shown in Fig 4.36. This shows the final chip submission layout. The final layout measures at 3.0 mm x 2.4 mm and was designed on 0.5 µm CMOS technology. An in-detail pin configuration and description is provided in Table 4.1.

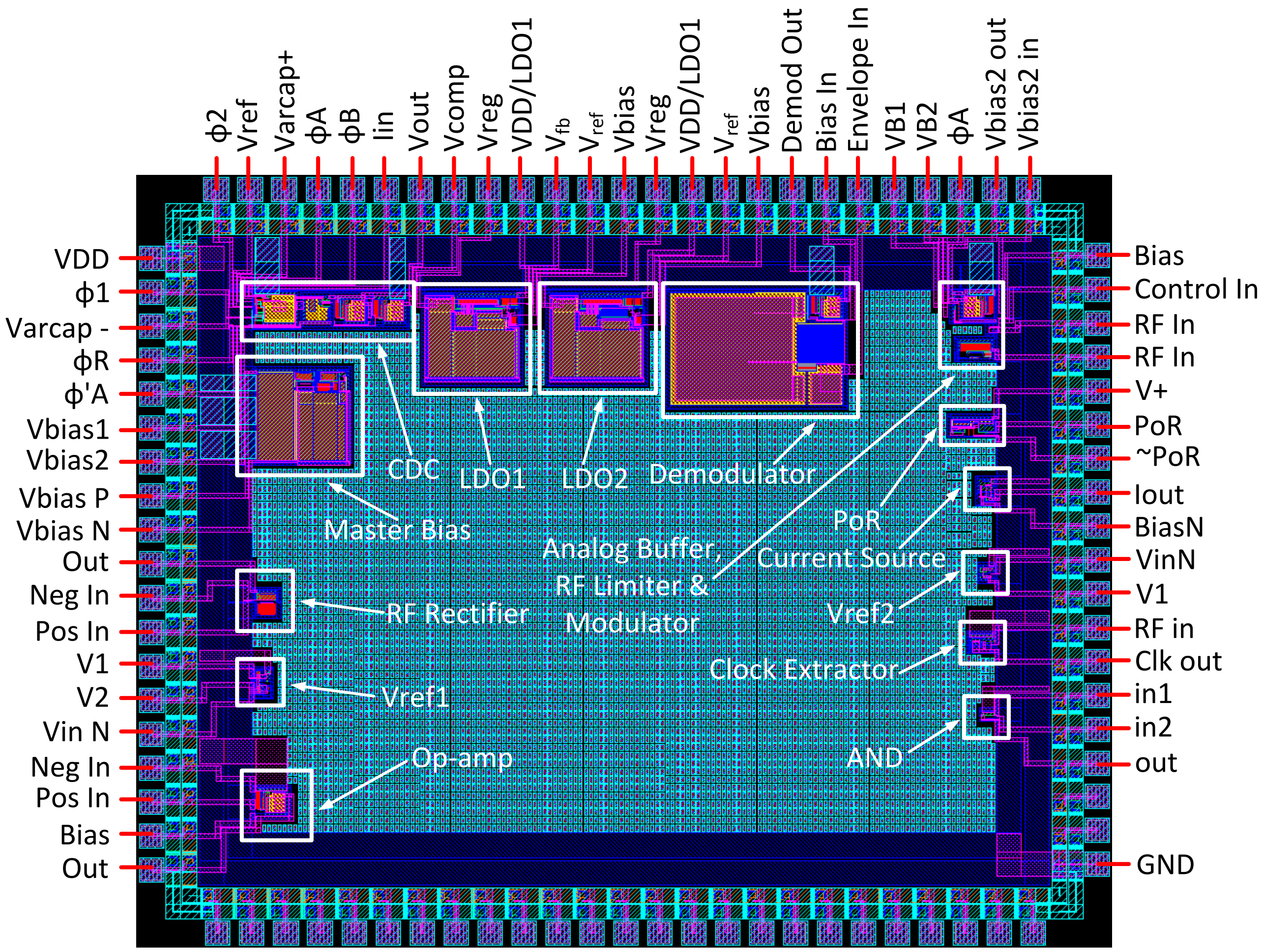


Fig 4.36 Final chip layout of the analog core

Table 4.1 In-detail pin configuration for final chip layout

	Pins	Analog/Digital	Input Range	Connectivity
	VDD	Analog	3 V	in/out
	GND	Analog	0 V	in/out
CDC	ϕA	Digital	0 V-3 V	input
	ϕB	Digital	0 V-3 V	input
	ϕ1	Digital	0 V-3 V	input
	ϕ2	Digital	0 V-3 V	input
	ϕR	Digital	0 V-3 V	input
	ϕ'A	Digital	0 V-3 V	input
	Varcap -	Analog	5.71 pF - 6.2 pF	in/out
	Varcap +	Analog	5.71 pF - 6.2 pF	in/out
	Iin	Analog	10 nA	input
	Vref	Analog	3 V	input
	Vbias1	Analog	750 mV	input
	Vbias2	Analog	1.5 V	input
	Vout	Analog	1.5V - 2 V	output
	Vcomp	Analog	0V - 3V	output
LDO w/o resistor	Vreg	Analog	3V	output
	VDD	Analog	0V - 10V	input
	Vfb	Analog	From resistors	input
	Vref	Analog	2 V	input
	Vbias	Analog	750 mV	input
Fixed LDO	Vreg	Analog	3V	output
	VDD	Analog	0V - 10V	input
	Vref	Analog	2V	input
	Vbias	Analog	750 mv	input
Demod	Demod Out	Analog	0 V- 3 V	output
	Bias In	Analog	1 V	input
	Envelope In	Analog	From Diode	input
Analog Buffer	VB1	Analog	750 mV	input
	VB2	Analog	1 V	input
	ϕA	Digital	0 V - 3 V	input
	Vbias2 Out	Analog	1.5 V	output
	Vbias2 In	Analog	1.5 V	input
	Bias	Analog	750 mV - 1 V	input
Mod	Control In	Analog	0 V- 3 V	input
	RF In	Analog	4 VP-P, 13.56 MHz	in/out
RF Limiter	RF In	Analog	0 V- 30 V, 13.56 MHz	in/out
POR	V+	Analog	0 V- 10 V, 13.56 MHz	input
	PoR	Analog	0V - 3 V	output

	~PoR	Analog	gnd	output
Curren t Source	Iout	Analog	10 nA	output
	BiasN	Analog	750 mV	input
Clk Extr	RF In	Analog	0 V - 10 V, 13.56 MHz	input
	Clk Out	Analog	0 V - 3 V, 13.56 MHz	output
AND	in 1	Analog	0 V - 3 V	input
	in 2	Analog	0 V - 3 V	input
	out	Analog	0 V - 3 V	output
Maste r Bias	VbiasP	Analog	2.212 V	output
	VbiasN	Analog	550 mV	output
Vref1	V1	Analog	1.5 V	output
	V2	Analog	750 mV	output
	Vin N	Analog	550 mV	input
Vref2	V1	Analog	1 V	output
	Vin N	Analog	550 mV	input
Rectifier	Neg In	Analog	10 VP-P, 13.56 MHz	input
	Pos In	Analog	10 VP-P, 13.56 MHz	input
	Out	Analog	8 V	output
Op-amp	Neg In	Analog	2 V	input
	Pos In	Analog	2 V, 10 VP-P, 1 KHz	input
	Bias	Analog	750 mV	input
	Out	Analog	0 V - 3 V	output

4.6 Robustness of the RFID Tag

The designed RFID Tag is robust with an error rate of < 7% when the matching network of the coil is good. But the only disadvantage of this matching network is that the DC level of the clock signal shifts when the distance between the RFID tag and RFID reader is varied. Due to the change in the clock level the CPLD cannot recognize it and thus response will not be given for a request packet. To overcome this problem we have designed a clock extraction circuit which is part of a chip sent for fabrication. Once the chip is fabricated we can test the performance of the clock extraction circuit and recalculate the error rate.

Summarizing this chapter we can say that the digital core has been successfully designed and tested to communicate with the commercially available RFID readers. The simulated results clearly show the response of the developed digital core to the read command sent by the reader.

CHAPTER 5

CONCLUSION AND FUTURE WORK

1) A digital core was designed and successfully tested to work with commercially available RFID reader. The design is compatible with the ISO standard and is robust with an error rate of less than 7%.
2) The design needs to be tested with the analog core. Depending on the test results the digital core might have to be customized, so the design is still evolving.
3) The analog components required to work with this digital core have been designed by and sent for fabrication.
4) If the analog core satisfies the power requirement of the digital core then we can add more functionality to the designed RFID Tag.
5) Once the design is completed and validated then the VHDL code can be translated in to a layout which can be part of an RFID sensor chip.
6) This synthesis report generated by Xilinx ISE shows this design constitutes 2589 basic gates, 294 sequential circuits and 10 combinational circuits.

APPENDIX

VHDL CODE

The VHDL code used to design the digital core is presented in the next pages. Each code has a heading which describes the module name for which the code was designed.

Frame Decoder:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

use ieee.numeric_std.all;

entity frame_decoder is

   port ( ask_in: in  STD_LOGIC; rst: in  std_logic;

           clk: in  STD_LOGIC; SOF: out std_logic;

           EOF: out std_logic; ask_out: out std_logic;

           clk_out: out std_logic; frame_error: out std_logic;

           data_out: out std_logic_vector(15 downto 0));

end frame_decoder;

architecture behavioral of frame_decoder is

 type TState is (IDLE, SOF1, SOF2, SOF3, BIT0, BIT1, BIT2, BIT3, INTER,
BYTE_READY, HALT, EOF1, ERROR);

 signal a_state : TState;

 signal c_state, next_state : TState;

 signal counter: std_logic_vector(10 downto 0);

 signal dibit: std_logic_vector(1 downto 0);

 signal rcv_byte: std_logic_vector(7 downto 0);

 signal bit_decoding: std_logic;

 signal rcv_byte_count : integer range 0 to 7;

 signal error_flag: std_logic;


begin

 ask_out     <= ask_in; clk_out     <= clk; frame_error <= not error_flag;
```

```
---------------------- ASK process -----------------------------
process(ask_in, rst)
begin
       if rst = '0' then
         a_state <= IDLE;
       elsif falling_edge(ask_in) then
               if a_state = IDLE then
                       a_state <= SOF1;
               elsif a_state = SOF1 then
                 if c_state = ERROR then
       a_state <= IDLE;
     else
                         a_state <= SOF2;
                       end if;
               elsif a_state = SOF2 then
                 if c_state = ERROR then
       a_state <= IDLE;
     else
           a_state <= BIT0;
                       end if;
               elsif a_state = BIT0 then
                       if (c_state = EOF1) or (c_state = ERROR) then
                         a_state <= IDLE;
                 else
                   a_state <= BIT1;
                       end if;
```

```
                    elsif a_state = BIT1 then
                      if c_state = ERROR then
            a_state <= IDLE;
          else
                      a_state <= BIT2;
                            end if;
                    elsif a_state = BIT2 then
                      if c_state = ERROR then
            a_state <= IDLE;
          else
            a_state <= BIT3;
                            end if;
        elsif a_state = BIT3 then
          if (c_state = EOF1) or (c_state = ERROR) then
                  a_state <= IDLE;
                      else
            a_state <= BIT0;
                            end if;
       end if;
     end if;
end process;
   --------------------- CLK process ----------------------------
  process(clk, rst)
  begin
    if rst = '0' then
            counter <= (others => '0'); SOF <= '1'; EOF <= '1';
```

```
                    c_state <= IDLE; bit_decoding   <= '0';
                    rcv_byte <= (others => '0'); data_out <= (others => '0');
                    dibit <= "10"; rcv_byte_count <= 0; error_flag     <= '0';
        elsif rising_edge(clk) then
                    counter <= counter + 1;
if (a_state = SOF1 and c_state = IDLE) then
                        counter     <= (others => '0');
                              error_flag  <= '0';
                              c_state     <= SOF1;
                elsif (a_state = SOF2 and c_state = SOF1) then
if (counter(10 downto 5) = "010000") or (counter(10 downto 5) = "001111") then  -- from
543 - 480 or 0x21F - 0x1E0
                                     c_state      <= BIT0;
                                     bit_decoding <= '1';
                                     SOF          <= '0';
                                     counter      <= (others => '0');
                              else
                                error_flag  <= '1';
                                c_state     <= ERROR;
                              end    if;
                    elsif (c_state = a_state) and bit_decoding='1' then
          ------------------------ dibit decoding -------------------------
                              if (counter(10 downto 5)="000100") or (counter(10 downto
5)="000011") then      --128  --> -3 (159 - 96)
                    dibit <= dibit + 1;
                              elsif (counter(10 downto 5)="001100") or (counter(10 downto
5)="001011") then   --384  --> -2 (415 - 352)
                    dibit <= dibit + 2;
```

```
                              elsif    (counter(10   downto   5)="010100")   or   (counter(10   downto
5)="010011") then  --640  --> -1 (671 - 608)

                    dibit <= dibit + 3;

                              elsif    (counter(10   downto   5)="011100")   or   (counter(10   downto
5)="011011") then  --896  --> +0

                    dibit <= dibit;

                              elsif    (counter(10   downto   5)="100100")   or   (counter(10   downto
5)="100011") then  --1152 --> +1 (1183 - 1120)

                    dibit <= dibit + 1;

                              elsif    (counter(10   downto   5)="101100")   or   (counter(10   downto
5)="101011") then  --1408 --> +2

                    dibit <= dibit + 2;

                              elsif    (counter(10   downto   5)="110100")   or   (counter(10   downto
5)="110011") then  --1664 --> +3

                    dibit <= dibit + 3;

                              else

          error_flag <= '1';

                              end if;

        if c_state = BIT0 then

                                next_state <= BIT1;

                              elsif c_state = BIT1 then

                                next_state <= BIT2;

                              elsif c_state = BIT2 then

                                next_state <= BIT3;

                              elsif c_state = BIT3 then

                                next_state <= BYTE_READY;

                              end if;

                              c_state <= INTER;
                      elsif c_state = INTER then
    if error_flag = '1' then
                                c_state <= ERROR;
                              else
            rcv_byte(7 downto 0)  <= dibit & rcv_byte(7 downto 2);
            counter <= (others => '0');
            c_state <= next_state;
                              end if;
        elsif c_state = BYTE_READY then
if rcv_byte_count = 0 then
                        data_out(7 downto 0) <= rcv_byte;
                              elsif rcv_byte_count = 1 then
                        data_out(15 downto 8) <= rcv_byte;
                              end if;
                              if rcv_byte_count = 4 then
                                c_state       <= EOF1;
                                    bit_decoding   <= '0';
                                rcv_byte_count <= 0;
                              else
                              rcv_byte_count <= rcv_byte_count + 1;
            c_state <= BIT0;
                              end if;
             elsif c_state = EOF1 then


              if a_state = IDLE then    --wait for EOF pulse
                        c_state <= IDLE;
```

```
                                        dibit   <= "10";
                                        EOF     <= '0';
                        else
                          c_state <= EOF1;
                              end if;
        elsif c_state = ERROR then
                                bit_decoding   <= '0';
                                      dibit         <= "10";
                                      SOF           <= '1';
                                      EOF           <= '1';
                                      rcv_byte_count <= 0;
                                      data_out       <= "1111" & '0' & counter;
                                      error_flag     <= '1';
    if a_state = IDLE then   --wait for other state machine to recognize error state
                                        c_state <= IDLE;
                                      else
                                        c_state <= ERROR;
                                      end if;
        end if;
      end if;
    end process;
    end behavioral;
```

Start of Frame Module:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity sof_module is
port( start  : in    std_logic; clk    : in    std_logic;       enable : in    std_logic;
sof_out: out   std_logic;          done   : inout std_logic;          done_early  : out   std_logic );
end sof_module;
architecture Behavioral of sof_module is
signal data : std_logic_vector(7 downto 0) := "00011101";
signal i    : integer; signal clock: std_logic;
begin
clock <= clk and enable;
 process(clock,start)
begin
        if(start = '0') then
                sof_out <= '0'; done <= '0'; done_early <= '0';           i <= 0;
        elsif (clock='1' and clock'event) then
                if i < 8 then
                        sof_out <= data(7-i);
                end if;
                i <= i + 1;
                if i = 7 then
                        done_early <= '1';
                end if;
                if i = 8 then
                        done <= '1';
                end if;
        end if;
 end process;
end Behavioral;
```

End of Frame Module:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity eof_module is
port( start : in std_logic; clk : in std_logic; enable : in std_logic;
        eof_out : out   std_logic; done : inout std_logic        );
end eof_module;
architecture Behavioral of eof_module is
signal data : std_logic_vector(7 downto 0) := "10111000";
signal i   : integer range 0 to 8; signal clock: std_logic; signal count: integer range 0 to 100;
begin
clock <= clk and enable;
 process(clock,start)
begin
        if(start = '0') then
                eof_out <= '0'; done <= '0'; i <= 0; count <= 0;
        elsif (clock='1' and clock'event) then
                 if count > 0 then
                        if i < 8 then
                                eof_out <= data(7-i);
                        end if;
                        if i > 7 then
```

```
                                done <= '1';
                         end if;
                          i <= i + 1;
                  end if;
                  count <= count + 1;
         end if;
 end process;
end Behavioral;
```

Data Module:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity data_module is
port( data_vect_in      : in std_logic_vector(39 downto 0);
clk : in std_logic; load : in std_logic; enable : in std_logic;
lsb_serial_out : out std_logic; -- data sent to Manchester and CRC module
done : out std_logic;  -- pulse sent to eof_encoder
done_to_piso   : out std_logic -- pulse sent to PISO module  );
end data_module;
architecture Behavioral of data_module is
signal clock: std_logic;
signal ireg       : std_logic_vector(39 downto 0);
signal lsb        : std_logic :='0';
signal i          : integer range 0 to 41;
```

```
begin
clock <= clk AND (enable);
process(clock,load)
begin
if load ='1' then
        lsb_serial_out  <= '0'; done <= '0'; done_to_piso <= '0'; i <= 0; ireg <= data_vect_in;
elsif(clock='1' and clock'event) then
        if (i < 40) then
                lsb <= ireg(i);       -- sending lsb data for manchester
        end if;
        lsb_serial_out <= lsb;
        if i > 38 then
        done <= '1'; -- this will send a pulse to crc indicating completed sending data to
        manchester
        done_to_piso <= '1'; -- this will send a pulse to crc indicating completed sending data
to PISO
        end if;
        i <= i + 1;
end if;
end process;
end Behavioral;
```

CRC Module:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity crc_module is

   port ( serial_in    : in  STD_LOGIC; clk : in  STD_LOGIC; enable : in  std_logic;

            reset  :  in    STD_LOGIC;    crc_out  :  out  std_logic_vector(15  downto  0));  end
crc_module;

architecture structural of crc_module is

 component d_ff is

   port ( data : in  STD_LOGIC;

          clk  : in  STD_LOGIC;

                                set  : in  STD_LOGIC;

          Q    : out  STD_LOGIC);

 end component;

 signal q     : std_logic_vector(15 downto 0);  signal x     : std_logic_vector(2 downto 0);

 signal clock : std_logic;

begin

  clock <= clk AND (enable);

 x(0)  <= serial_in xor q(15);

 x(1)  <= q(4) xor x(0);

 x(2)  <= q(11) xor x(0);


 U1: d_ff port map (data => x(0), clk => clock, set => reset, Q => q(0));

 U2: d_ff port map (data => q(0), clk => clock, set => reset, Q => q(1));

 U3: d_ff port map (data => q(1), clk => clock, set => reset, Q => q(2));

 U4: d_ff port map (data => q(2), clk => clock, set => reset, Q => q(3));
```

```
U5: d_ff port map (data => q(3), clk => clock, set => reset, Q => q(4));

U6: d_ff port map (data => x(1), clk => clock, set => reset, Q => q(5));

U7: d_ff port map (data => q(5), clk => clock, set => reset, Q => q(6));

U8: d_ff port map (data => q(6), clk => clock, set => reset, Q => q(7));

U9: d_ff port map (data => q(7), clk => clock, set => reset, Q => q(8));

U10: d_ff port map (data => q(8), clk => clock, set => reset, Q => q(9));

U11: d_ff port map (data => q(9), clk => clock, set => reset, Q => q(10));

U12: d_ff port map (data => q(10), clk => clock, set => reset, Q => q(11));

U13: d_ff port map (data => x(2), clk => clock, set => reset, Q => q(12));

U14: d_ff port map (data => q(12), clk => clock, set => reset, Q => q(13));

U15: d_ff port map (data => q(13), clk => clock, set => reset, Q => q(14));

U16: d_ff port map (data => q(14), clk => clock, set => reset, Q => q(15));


crc_out(0) <= not q(15);

crc_out(1) <= not q(14);

crc_out(2) <= not q(13);

crc_out(3) <= not q(12);

crc_out(4) <= not q(11);

crc_out(5) <= not q(10);

crc_out(6) <= not q(9);

crc_out(7) <= not q(8);

crc_out(8) <= not q(7);

crc_out(9) <= not q(6);

crc_out(10) <= not q(5);

crc_out(11) <= not q(4);

crc_out(12) <= not q(3);
```

```
 crc_out(13) <= not q(2);
 crc_out(14) <= not q(1);
 crc_out(15) <= not q(0);

end structural;
 -------------------------------------------- D-ff code -----------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity d_ff is
    port ( data : in  STD_LOGIC; clk  : in  STD_LOGIC; set  : in STD_LOGIC;
             Q    : out  STD_LOGIC); end d_ff;
architecture behavioral of d_ff is
begin
process (data,clk,set)
begin
  if set = '1' then
    Q <= '1';
  elsif (clk'event and clk = '1') then
    Q <= data;
  end if;
end process;
end behavioral;
```

PISO Module:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity PISO is
port( pulse_in   : in std_logic;
                    clk : in std_logic; data_vect_in : in std_logic_vector(15 downto 0);
                    serial_out   : out std_logic; done_to_eof  : out std_logic;      done:               out
std_logic);
end PISO;
architecture Behavioral of PISO is
          signal i            : integer :=0;
          signal temp : std_logic_vector(15 downto 0):= (others =>'0');
begin
          temp <= data_vect_in;
process(clk,pulse_in)
          begin
                    if pulse_in = '0' then --or rst ='1' then
                              serial_out  <= '0';
                              done        <='0';
                              done_to_eof <='0';
                              i <= 0;
                    elsif(clk='1' and clk'event) then
                              if(i < 16) then
                                        serial_out <= temp(i); -- CRC trasmitted as Least Significant
                              byte first & Least significant bit first.
                                        i <= i + 1;
                              end if;
```

```
                if i = 15 then
                        done_to_eof <= '1';
                end if;
                if i = 16 then
                        done <= '1';
                end if;
        end if;
end process;
end Behavioral;
```

Manchester Module:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity manchester is
port( data_serial_in : in std_logic;
        clock : in std_logic; enable : in std_logic; manchester_out : out std_logic );
end manchester;
architecture Behavioral of manchester is
signal clk: std_logic;
begin
clk <= clock AND (enable);
process(clk,data_serial_in)
    begin
        manchester_out <= clk xor data_serial_in;
        end process;
end Behavioral;
```

Delay Module:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity delay_module is
port ( start  : in std_logic; clk    : in std_logic; enable : in std_logic; done      :  inout  std_logic
);
end delay_module;
architecture Behavioral of delay_module is
signal count    : integer range 0 to 5000;
signal clk_13MHz: std_logic;
begin
clk_13MHz <= clk and enable;
process(clk_13MHz, start)
 begin
        if start = '0' then
                done <= '0';
                count <= 0;
        elsif (clk_13MHz'event and clk_13MHz ='1') then
                count <= count + 1;
                if (count = 4141) then
                        done <= '1';
                        count <= 0;
                end if;
```

```
            end if;
end process;
end Behavioral;
```

Clock Module:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity clock_delay is
   Port ( clk : in  STD_LOGIC; start : in  STD_LOGIC; clk_32  : out  STD_LOGIC;
          clk_256 : out  STD_LOGIC; clk_512 : out  STD_LOGIC  );
end clock_delay;
architecture Behavioral of clock_delay is
signal  count  : std_logic_vector (9 downto 0);
signal  clk_divide_32  : std_logic;
signal  clk_divide_256: std_logic;
signal  clk_divide_512: std_logic;
begin
 process(clk,start)
  begin
   if(start='0') then -- The Push Button is Acitve low. When we push the button rst ='1' else rst
='0'
                    clk_divide_32   <= '0';
                    clk_divide_256 <= '0';
                    clk_divide_512 <= '0';
                    count <= (others => '0');
```

```
            elsif(clk'event and clk='1') then
                        clk_divide_32    <= count(5);
                        clk_divide_256  <= count(7);
                        clk_divide_512  <= count(8);
   end if;
 end process;
clk_32   <= not clk_divide_32;
clk_256 <= not clk_divide_256;
clk_512 <= not clk_divide_512;
end Behavioral;
```

Mux 4:1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity mux41 is
port(    input : in std_logic_vector(2 downto 0); sel : in std_logic_vector(1 downto 0);
            disable : in std_logic; output  : out std_logic:='0'  );
end mux41;
architecture Behavioral of mux41 is
begin
            output <= ((Input(0) and (not sel(0)) and (not sel(1))) or (Input(1) and sel(0) and (not
sel(1)))
                                    or (Input(2) and sel(0) and sel(1))) and disable;


end Behavioral;
```

Mux 2:1

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;use IEEE.NUMERIC_STD.ALL;
entity mux21 is
port(    input   : in std_logic_vector(1 downto 0); sel     : in std_logic; output  : out std_logic );
end mux21;
architecture Behavioral of mux21 is
begin
        output <= (Input(0) and (not sel)) or (Input(1) and (sel));
end Behavioral;
```

AND operator:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity andoperator is
   Port ( clk_in : in  STD_LOGIC; data_in : in  STD_LOGIC; data_out : out  STD_LOGIC);
end andoperator;
architecture Behavioral of andoperator is
signal data: std_logic;
begin
process(clk_in, data_in)
        begin
                data_out <= clk_in and data_in;
        end process;
end Behavioral;
```

REFERENCES

[1] Klaus F, *RFID Handbook, Radio-Frequency Identification Fundamentals and Applications*, 2nd ed. New York: Wiley, 2003.

[2] Wikipedia.org.

[3] Li Y; et al, "Battery-free RFID-enabled wireless sensors," *Microwave Symposium Digest (MTT), 2010 IEEE MTT-S International* , vol., no., pp.1528-1531, 23-28 May 2010.

[4] Karthaus, U.; Fischer, M.; , "Fully integrated passive UHF RFID transponder IC with 16.7-µW minimum RF input power," *Solid-State Circuits, IEEE Journal of* , vol.38, no.10, pp. 1602- 1608, Oct. 2003.

[5] Yeager, D.; Fan Z; Zarrasvand, A.; George, N.T.; Daniel, T.; Otis, B.P.; , "A 9 mu A, Addressable Gen2 Sensor Tag for Biosignal Acquisition," *Solid-State Circuits, IEEE Journal of* , vol.45, no.10, pp.2198-2209, Oct. 2010.

[6] Kaiser, U.; Steinhagen, W.; , "A low-power transponder IC for high-performance identification systems," *Solid-State Circuits, IEEE Journal of* , vol.30, no.3, pp.306-310, Mar 1995.

[7] Qiuting H; Oberle, M.; , "A 0.5-mW passive telemetry IC for biomedical applications," *Solid-State Circuits, IEEE Journal of* , vol.33, no.7, pp.937-946, Jul 1998.

[8] Pillai, V.; Heinrich, H.; Dieska, D.; Nikitin, P.V.; Martinez, R.; Rao, K.V.S.; , "An Ultra-Low-Power Long Range Battery/Passive RFID Tag for UHF and Microwave Bands With a Current Consumption of 700 nA at 1.5 V," *Circuits and systems I: Regular Papers, IEEE Transactions on* , vol.54, no.7, pp.1500-1512, July 2007.

[9] STMicroelectronics, LRI512 data sheet, http://pdf1.alldatasheet.com/datasheet-pdf/view/22777/STMICROELECTRONICS/LRI512.html.

[10] ISO/1EC 15693:2000: Identification cards - contactless integrated circuit(s) cards - vicinity cards. http://www.iso.org/iso/catalogue_detail.htm?csnumber=30995.

[11] Gerrish, P.; Herrmann, E.; Tyler, L.; Walsh, K.; , "Challenges and constraints in designing implantable medical ICs," *Device and Materials Reliability, IEEE Transactions on*, vol.5, no.3, pp.435-444, Sept. 2005 doi: 10.1109/TDMR.2005.858914.

[12] Rankl, W.; Effing, W. (1996) *Handbuch der Chipkarten*, 2nd edn, Carl Hanser Verlag, Munich.

VITA

Kumar Swamy H.S. was born on June 22, 1988, in Bangalore, India. He was educated in local private schools and graduated from Seshadripuram Pre-University in 2005. Then he went on to receive his Bachelor of Engineering degree in Electronics and Communication from Visvevsaraya Technological University, Bangalore, India with magna cum laude, in 2009.

After receiving his Bachelor's degree Mr Kumar Swamy came to the United States to pursue his graduate studies. He began his master's program in Electrical Engineering at the University of Missouri – Kansas City. Due to his academic excellence he was awarded "Dean's International Computing & Engineering Award" Scholarship for both the graduate school years. Upon completion of the degree requirements, Mr Kumar Swamy received job offers from many companies including Qualcomm Inc, ALTERA corporation. Currently he is pursuing his career as an Electrical Engineering specializing in Hardware design at Qualcomm Inc.

Mr Kumar Swamy recently published a paper given below,

Hosur Satyamurthy, K.; Leon-Salas, Walter D.; Timpson, Erik;, "Experimental Evaluation of an Intravascular Differential Pressure Flow Meter Using MEMS Pressure Sensors," *, IEEE SENSORS Conference: Regular Papers* , Oct 2011.