HARDWARE EMULATION OF A SECURE PASSIVE RFID SENSOR SYSTEM

A Thesis Presented

by

MICHAEL TODD

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial
fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2010

Department of Electrical and Computer Engineering

HARDWARE EMULATION OF A SECURE PASSIVE RFID SENSOR SYSTEM

A Thesis Presented

by

MICHAEL TODD

Approved as to style and content by:

DEDICATION

To my family & fiancé.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Wayne Burleson for providing me with the opportunity to expand my knowledge of electrical and computer engineering, and for his guidance and support throughout my research. I would also like to thank Professor Christof Paar for guidance in the state-of-the-art of security engineering. Thanks should also be given to Professor Kevin Fu and his team in the Computer Science department here at UMass for their pioneering work in advanced RFID related topics. I would like to thank Professor Wayne Burleson, Sandip Kundu, and Russell Tessier for their service on my thesis committee. I would also like to thank the entire VLSI Circuits and Systems group for both their friendship and academic support, specifically Serge Zhilyaev, Basab Datta, and Ibis Benito.

A great many thanks should also be given to my family Mark, Lori, Nathan, Jason, and Benjamin for their encouragement and support throughout my life and academic career and without whom none of this would be possible. Finally, I would like to thank my beautiful bride-to-be Amanda Burk. She has been there through every step of my research, through good times and struggles, with truly unwavering devotion and love.

ABSTRACT

HARDWARE EMULATION OF A SECURE PASSIVE RFID SENSOR SYSTEM

SEPTEMBER 2010

MICHAEL TODD, B.S., UNIVERSITY OF MASSACHUSETTS AMHERST

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Wayne Burleson

Passively powered radio frequency (RFID) tags are a class of devices powered via harvested ultra high frequency (UHF) radiation emitted by a reader device. Currently, these devices are relegated to little more than a form of wireless barcode, but could be used in a myriad of applications from simple product identification to more complex applications such as environmental sensing. Because these devices are intended for large scale deployment and due to the limited power that can be harvested from RF energy, hardware and cost constraints are extremely tight.

The Electronic Product Code (EPC) Global Class 1 Generation 2 (Gen2) specification [EPC08] is currently the de facto communication standard for passively powered RFID. One issue restricting deployment and a cause for some privacy concerns is a lack of security in the Gen2 protocol. We will demonstrate a potential solution to this problem by using a novel block cipher designed for low power and area constrained devices to encrypt and transmit sensor data. This will be done while maintaining backward compatibility with the original standard and will require no substantial changes to the reader. Our solution will also provide one way authentication, data integrity checking and will provide security against replay attacks.

In this thesis we will demonstrate an FPGA emulation of a Gen2 compatible RFID tag which will serve as a test bed for several novel features. We will leverage prior work involving several aspects of a tag [QL09] [PP07] as well as incorporate a novel low power encryption cipher [AB07] and external temperature sensor. Demonstrated in [CT08], FPGA emulation will allow for the independent verification of several components. This thesis will provide insight into the future of RFID and will provide insight into tag design as well as possible future updates to the Gen2 standard.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

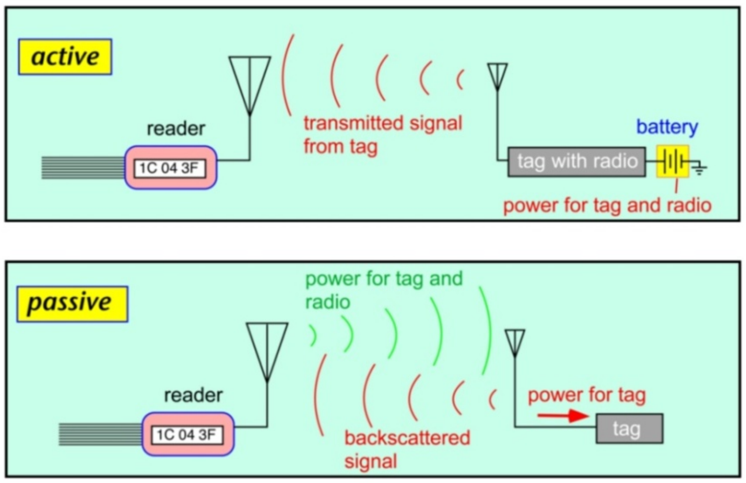
LIST OF ALGORITHMS

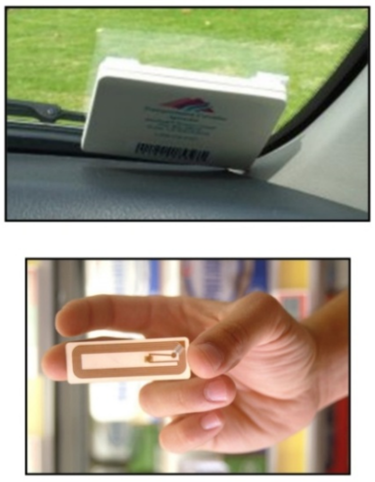
CHAPTER 1

INTRODUCTION TO RFID

1.1 Motivations

Passive RFID tags are essentially a low cost integrated circuit (IC) and antenna that harvest RF energy and use it to power a small amount of analog circuitry, digital logic, and non-volatile memory. Passive tags do not have a battery and thus lack a way to actively transmit data. Instead, a process known as *backscatter* is used to reflect RF energy transmitted to the tag back to the reader. Fig. 1a from [DD08] illustrates the difference between an active and a passive RFID tag. An example of each is given in Fig. 1b, showing that active tags are much larger and are used in applications such as automatic toll collection. Passive tags are significantly smaller and are typically used in inventory tracking applications [TF06], [WI10].





a) b)

Figure 1: From [TF06], [WI10], [DD08] Passive vs. Active RFID. a) Passively powered RFID tags lack an independent power supply and must derive power and data from RF energy emitted by a reader, and communicate by reflecting RF energy back to the reader. b) Passive tags are significantly smaller and lower cost than active tags and could possibly be used in a greater number of applications.

Passive tags are primarily intended to function as a form of "wireless barcode" used for tracking, inventorying, and even preventing theft of merchandise. Arguably, the most high profile deployment of RFID technology was a mandate in 2004 when Wal-Mart told its 100 top suppliers to provide RFID tags on all cases and pallets delivered to Wal-Mart by January 2005, with the next 100 to follow a year later [DD08]. Such high volume deployment inherently forces tight area constraints on these devices to reduce cost. Most claim these devices are generally in the range of 5k gates with a target price of 1-5¢ [PP09].

1.2 EPC Class-1 Generation-2 RFID Standard

The EPCglobal Class-1 Generation-2 Radio-Frequency Identity Protocol for Communications at 860 MHz – 960MHz, which we will refer to as Gen2 throughout the remainder of this work, "defines the physical and logical requirements for a passive-backscatter, Interrogator-talk-first (ITF), radio-frequency identification (RFID) system" [EPC08]. This standard has been widely accepted as the primary standard for passively powered RFID tags since its inception by EPCglobal in 2004 and the International Organization for Standardization (ISO) 18000-6C in 2006.

There have been several published works involving designs which incorporate Gen2 directly [AM07], [CT08], [AM07] or seek to improve upon specific portions of the protocol [DB06], [QL09], [PP07], [AN07]. It is our goal to incorporate and build upon many of these works and create a type of guide to implementing Gen2 compliant tags. We *do not* however make the claim that our implementation is either 100% Gen2 compliant or fully optimized for area and power, but we will discuss in detail how we were able to design, validate, and emulate a design capable of communicating with a Gen2 compliant RFID reader.

1.3 Prior Work

1.3.1 WISP

The Wireless Sensing Platform (WISP) is a prototype RFID tag shown in Fig. 2. The WISP was designed by Intel and is a partially Gen2 compliant RFID tag constructed from a low power MSP430 microcontroller, analog circuitry, temperature sensor, and accelerometer [SA08]. It is in essence a software programmable RFID tag and has been utilized in several publications [HC07], [DH08], [SC09].

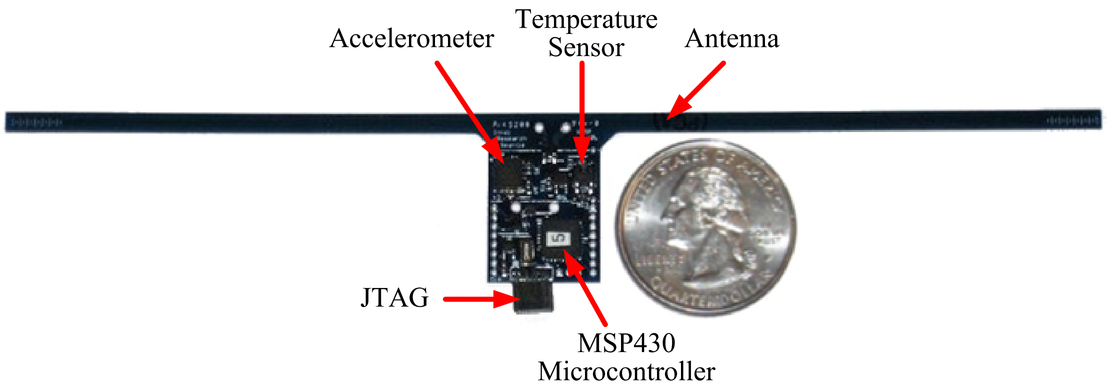


Figure 2: Wireless Sensing Platform. WISP is designed from a low power MSP430 microcontroller, and includes several onboard sensors.

The WISP has been proven successful in many applications including medical devices [DH08], recognizing human activities [MB09], and light sensing [RS06], though the relatively large size and high cost would make such a device impractical for high volume deployment. Also, modularizing the software required to both perform the Gen2 protocol, and any additional computation in the face of intermittent power from harvested RF energy provides a significant challenge. Our work will build upon the work pioneered by the WISP in two ways. First, we will leverage the idea of directly incorporating additional functionality into a Gen2 RFID tag with our block cipher and external sensor. This will provide a low cost solution to a Gen2 compliant, environmental sensing, and secure RFID tag. Second we will prototype our device on an FPGA. This will create a sister device to the WISP, a hardware programmable RFID tag.

1.3.2 FPGA RFID Sensing Platform

The idea of a hardware programmable, Gen2 compliant RFID sensing platform was first demonstrated in [CT08] and is shown in Fig. 3. FPGA emulation provides two major advantages over simulation. The first is proof of compliance with Gen2, as we will be using an RFID reader known to be compliant, discussed further in section 5.2. The second is a reduction in simulation time. Communication in Gen2 occurs on the order of milliseconds, which can lead to long simulation time as we integrate a greater number of modules in our design. We will replicate and build upon the work presented in [CT08] by introducing a way to secure the sensor data.

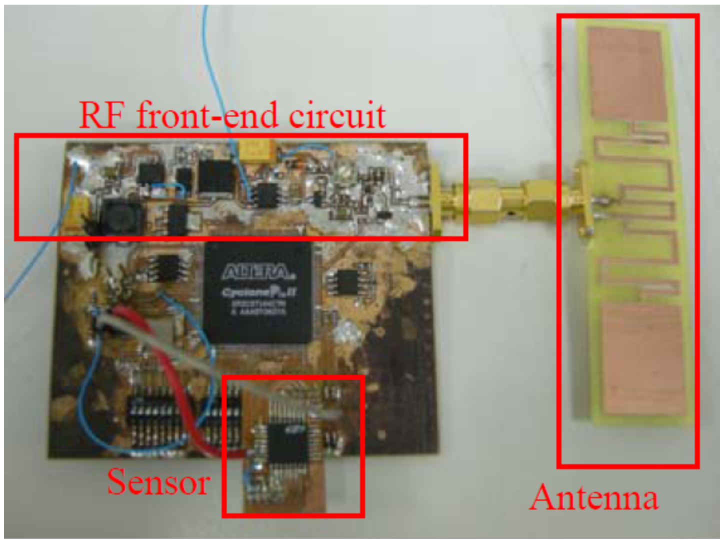


Figure 3: From [CT08], FPGA emulation of Gen2 compliant RFID sensing platform.

1.4 Thesis Outline

In this thesis we will begin with an overview of the Gen2 protocol followed by a block by block breakdown of each portion of the design. Chapter 3, justifies using a Gen2 RFID tag for environmental sensing and the security implications. Chapter 4 discusses the enhancements required to implement our approach to security. Finally, we will explore the validation and emulation aspects of our work including how we were able create a proof of concept for our design.

CHAPTER 2

DESIGNING A GEN2 TAG

2.1 Design Methodology

Both the design and validation portions of our project were done in the Verilog Hardware Definition Language (HDL) with compilation and simulation being done in the Xilinx ISE 9.2.04i environment. As previously mentioned, in a commercially viable RFID tag gate count and power consumption are of primary concern. The focus of our project is ease of reuse in the HDL code. This will allow our work to serve as a testbench for changes and optimizations to Gen2 and security protocols, as well as reducing both ramp-up and debug time. To accomplish this goal, implementation was guided by several important design rules:

1. Make the design highly modularized, rather than integrated.
2. Thoroughly commented code with external references where appropriate.
3. The manner by which each module is enabled, reset, and completion is indicated should remain standard.

It is our desire that by utilizing said guidelines we will be able to greatly reduce the complexity of the design at the cost of size and allow components of the design to be replaced with ease. With that being said we have made a reasonable effort to reduce the size of our design. Discussed further in section 5.5, we have synthesized our design and will report the NAND2 Gate Equivalent (GE) for each portion of the design.

2.2 Gen2 Basics

In this protocol the interrogator (reader) modulates a signal in the UHF frequency range (860-960MHz) in order to communicate with a receiver (tag). The tag uses this signal not only to receive data, but to power the device via RF energy harvesting. The reader receives data from a tag by transmitting a continuous wave while the tag modulates the reflection coefficient of its antenna in a process known as backscatter. This can be thought of as basically creating an open circuit or short circuit across the antenna to generate a 0 or 1. Fig. 4 is an illustration of a basic RFID system.

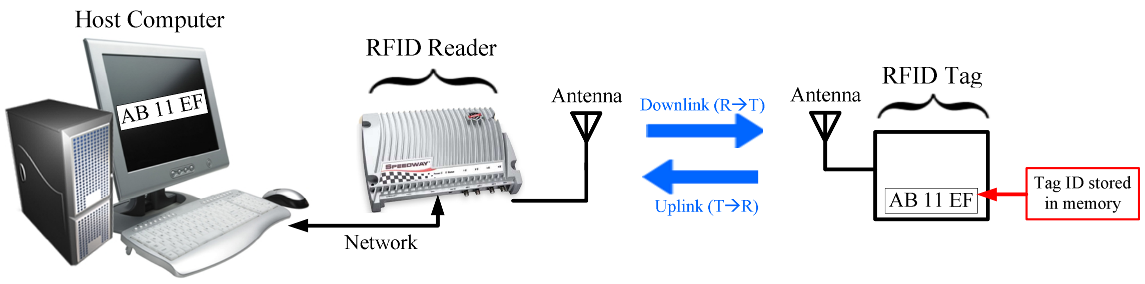


Figure 4: Basic RFID system diagram adapted from [DD08]. A computer sends commands through the reader device to the tag. The tag then responds with stored information accordingly.

The reader controls several aspects of each communication session such as both downlink (R→T) data rate between 50-215 kbps, and uplink (T→R) data rate between 5-640 kbps. The reader also controls what information the tag backscatters. Tags must support a basic set of commands given in Table 1. Tags are also assumed to have some form of non-volatile memory such as flash, where they store the Electronic Product Code, denoted as EPCID throughout this work, which can be thought of as the ID of the tag, as well as other information.

Table 1: From [GEN2] set of commands required by a Gen2 compliant RFID tag. There are also several non-mandatory commands, as well as opcodes reserved for custom commands to allow for flexibility.

Command	Opcode
QueryRep	00
ACK	01
Query	1000
QueryAdjust	1001
Select	1010
NAK	11000000
Req_RN	11000001
Read	11000010
Write	11000011
Kill	11000100
Lock	11000101

2.3 Typical Tag Inventory Session

Fig. 5 depicts a typical reader-tag inventory session, both data flow and in time. RN16 is a 16 bit pseudo random number generated by the tag used in access control and to confirm that the reader intended to speak to this tag when the reader transmits RN16 back to the tag. EPCID is the ID of the individual tag. This sort of inventorying session is an example of a passive tag used as little more than a wireless barcode. It should also be noted that CW stands for continuous wave, which is the unmodulated 900MHz RF energy emanated by the reader to power the tag during computation and is reflected during backscatter.

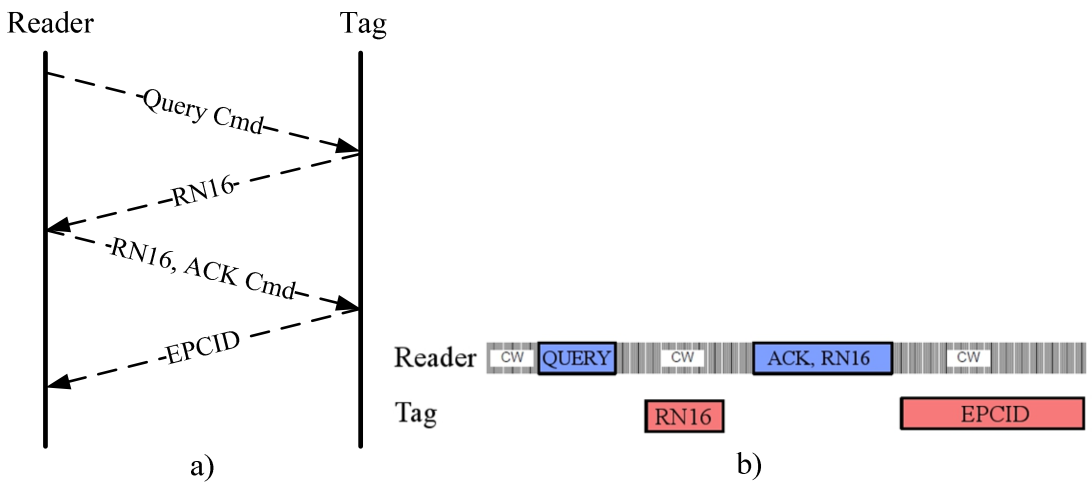


Figure 5: Typical tag inventory session. a) Data flow representation. b) In time representation.

2.4 Gen2 Tag Hardware

Fig. 6 illustrates the basic building blocks of a Gen2 RFID tag. The two primary modules of the tag are the analog frontend and digital backend. The frontend is responsible for regulating the incoming RF signal to generate V_{DD} for the digital logic, generating a clock signal, and demodulating the incoming data. The frontend also contains the backscatter transistor used to alter the reflection coefficient of the antenna during T→R communication. Though this work will fully concentrate on the portion labeled Digital Components & Control Logic, a cursory understanding of the remainder of the design is still required for prototyping.

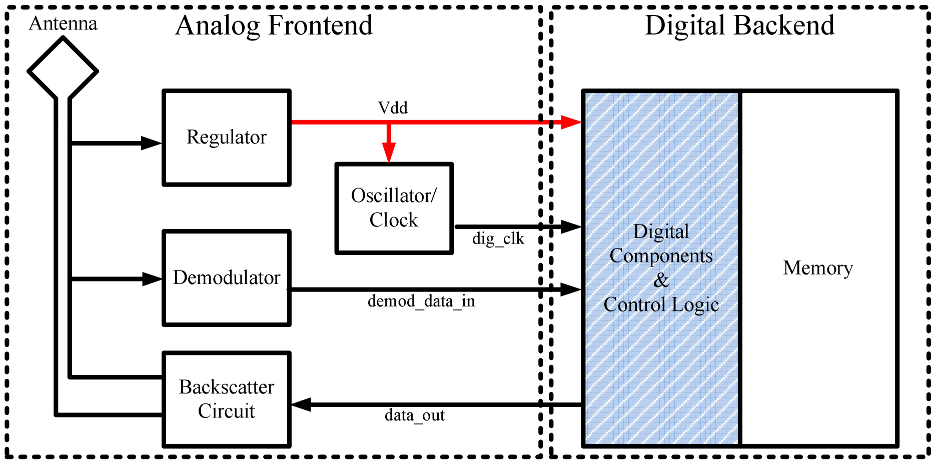


Figure 6: The two primary components of a traditional passive RFID tag. Our work will concentrate on the design and modification of the digital components and control logic.

2.5 Basic Digital Backend

Fig 7. is an expanded view of the components that make up the basic digital backend of a Gen2 RFID tag. The basic blocks consist of the incoming data decoder, the backscatter clock generator, output data encoder, cyclic redundancy checkers (CRC5/16), pseudo random number generator (PRNG), memory controller, and command handler finite state machine. Next we will delve into the implementation details for several of these components.

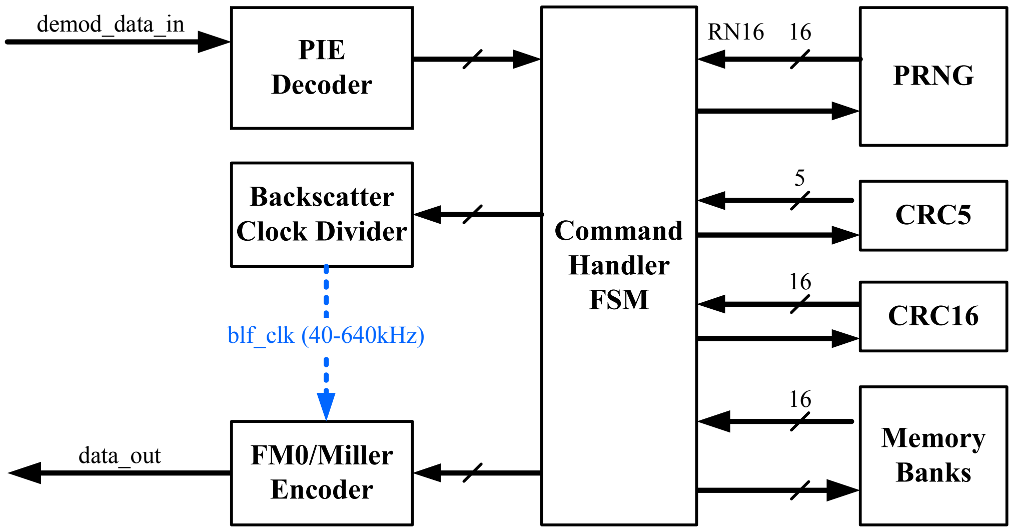


Figure 7: Block diagram depiction of the digital backend of Gen2 RFID tag.

2.6 PIE Decoder

Pulse interval encoding (PIE) is the encoding scheme used for R→T transmissions. Fig. 8 demonstrates how binary symbols are represented in this encoding scheme.

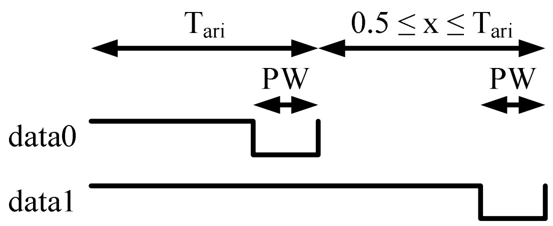


Figure 8: From [EPC08] PIE encoding. Encoding is achieved by varying the amount of time each symbol is high.

The reason for using this encoding scheme can be explained through understanding Fig. 9 from [DD08]. Here we can see that to simplify the decoder, amplitude modulation (AM) is used. Recalling that the tag also harvests power from this signal, whenever the amplitude of the incoming RF signal is low, the tag is not receiving power. Therefore, we must use an encoding scheme in which the incoming RF signal is high for the majority of the time. Though in Fig. 9 the low in the 0 and 1 seems to be a significant portion of the symbol, in reality this will be a much smaller fraction.

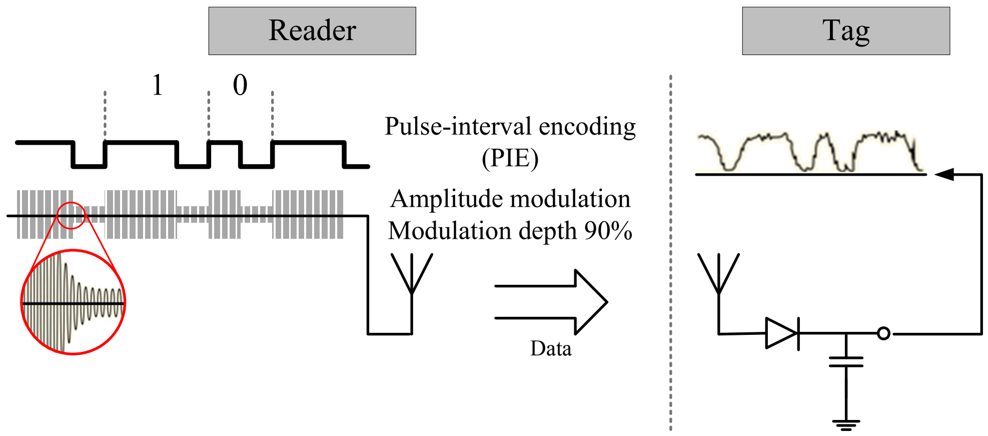


Figure 9: Schematic depiction of reader-to-tag data link adapted from [DD08].

In order to decode the data, we sample the incoming demodulated data when the data is high. Fig. 10 illustrates the R→T calibration symbol (RTcal) which is transmitted at the beginning of each reader command. We sample RTcal and divide this number by two resulting in the pivot. For the duration of the current command we interpret any symbol shorter (less samples) than the pivot to be a 0 and any symbol larger to be a 1. A detailed description of the PIE decoder state machine is given in Fig 11.

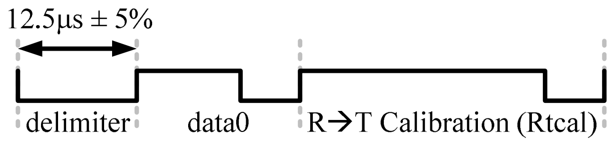


Figure 10: From [EPC08] the RTcal symbol. Symbol is transmitted at the beginning of each reader command and is used to calibrate the R-->T data link.

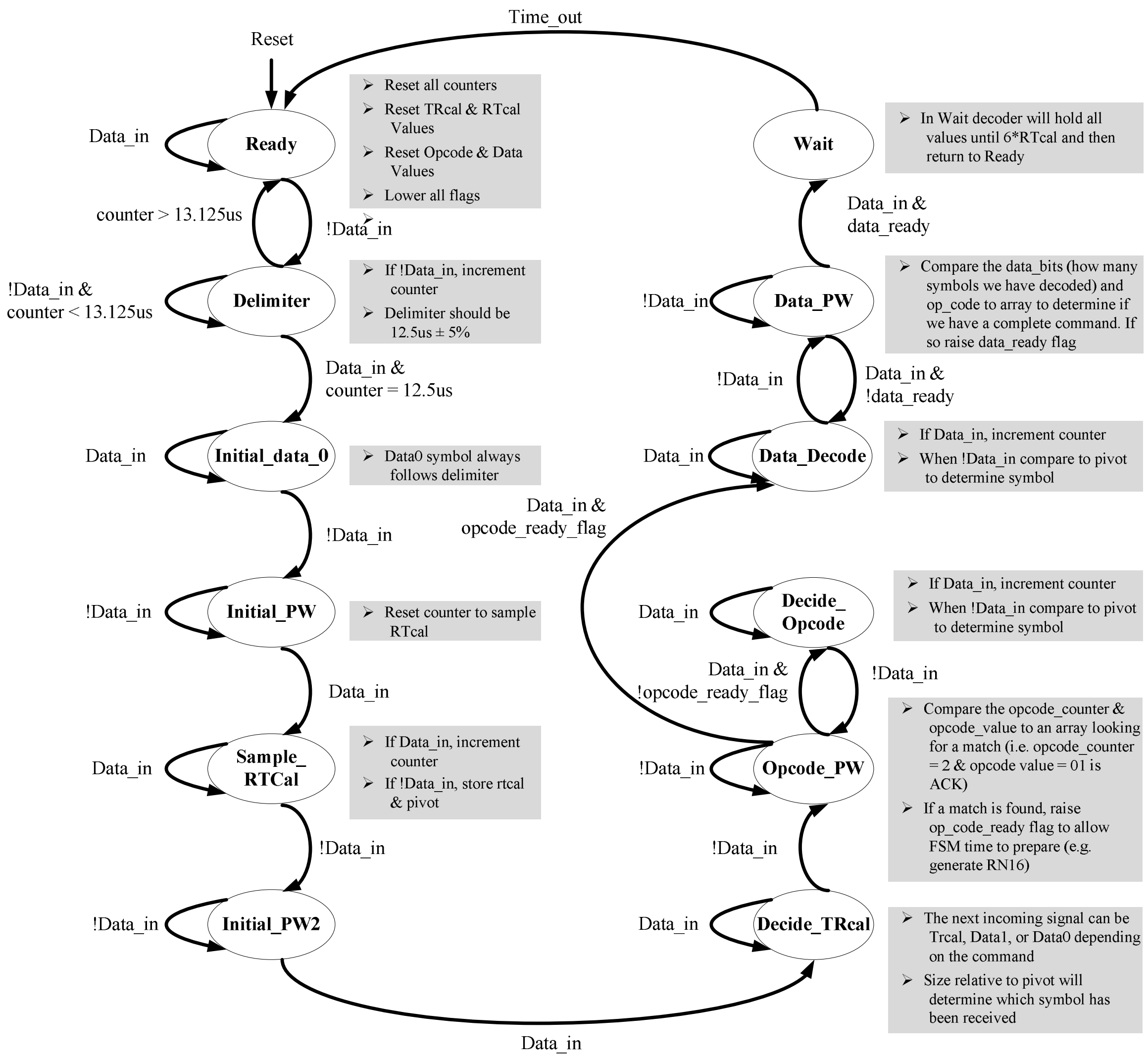

Time_out
Reset
Data_in
Ready
Reset all counters
Reset TRcal & RTcal Values
Reset Opcode & Data Values
Lower all flags
counter > 13.125us
!Data_in
!Data_in & counter < 13.125us
Delimiter
If !Data_in, increment counter
Delimiter should be 12.5us ± 5%
Data_in & counter = 12.5us
Data_in
Initial_data_0
Data0 symbol always follows delimiter
!Data_in
!Data_in
Initial_PW
Reset counter to sample RTcal
Data_in
Data_in
Sample_ RTCal
If Data_in, increment counter
If !Data_in, store rtcal & pivot
!Data_in
!Data_in
Initial_PW2
Data_in
Wait
In Wait decoder will hold all values until 6*RTcal and then return to Ready
Data_in & data_ready
!Data_in
Data_PW
Compare the data_bits (how many symbols we have decoded) and op_code to array to determine if we have a complete command. If so raise data_ready flag
!Data_in
Data_in & !data_ready
Data_in
Data_Decode
If Data_in, increment counter
When !Data_in compare to pivot to determine symbol
Data_in & opcode_ready_flag
Data_in
Decide_ Opcode
If Data_in, increment counter
When !Data_in compare to pivot to determine symbol
Data_in & !opcode_ready_flag
!Data_in
!Data_in
Opcode_PW
Compare the opcode_counter & opcode_value to an array looking for a match (i.e. opcode_counter = 2 & opcode value = 01 is ACK)
If a match is found, raise op_code_ready flag to allow FSM time to prepare (e.g. generate RN16)
!Data_in
Data_in
Decide_TRcal
The next incoming signal can be Trcal, Data1, or Data0 depending on the command
Size relative to pivot will determine which symbol has been received

Figure 11: PIE Decoder state diagram. The concept is to sample RTcal and divide by 2 to calculate the pivot. Each subsequent symbol shorter than the pivot is interpreted as a 0 any longer are interpreted as a 1.

Our decoder utilizes an array containing all supported opcodes, and the bit length of each command. After receiving a symbol, we compare with the array, and upon finding a match, raise the opcode_ready flag. This allows the command handler FSM time to perform certain tasks in parallel with the command decoding such as pseudo random number generation or retrieving values from memory. We then use the same array to indicate we have received all of the data associated with the command and raise the data_ready flag.

2.7 CRC5 & CRC16

Included in the Gen2 are 5-bit and 16-bit cyclic redundancy checks, called CRC5 and CRC16 respectively. Both of these circuits are constructed from a linear feedback shift register (LFSR) and are used to safe guard information against bit errors during transmission. Schematics for both circuits are given in Fig. 12 from [EPC08].

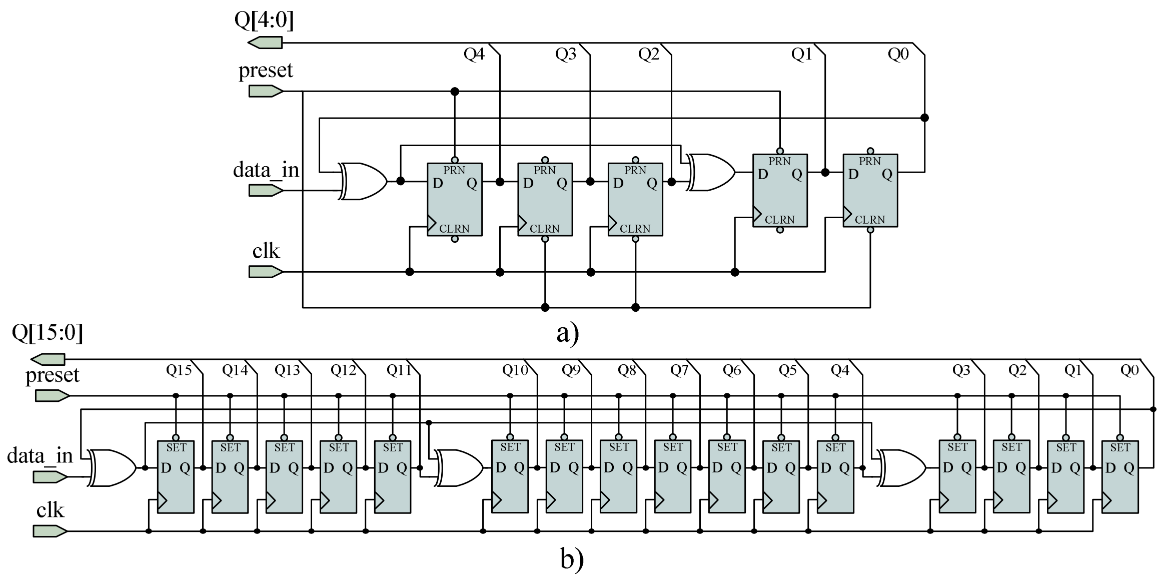


Figure 12: From [EPC08] Schematic view of the a) 5 bit CRC and b) 16 bit CRC.

CRC5 is only utilized when confirming a Query command was properly received. For this reason we were able to modularize the control logic. We used a 22 bit shift register which resets the CRC5 if a Query opcode is received and loads the contents of the Query command when the PIE decoder data_ready flag is raised and begins to shift in the data. If the Query command is properly received based on Appendix F of [EPC08], the crc5_valid flag is raised.

The CRC16 is used both to confirm certain commands were properly received, and to protect transmitted data, making implementation slightly more challenging, but the basic operating principles remain the same as the CRC5. We use a larger shift register and each time the CRC16 is used, data is loaded into crc16_data_in, crc16_reset is raised, and the number of bits to be clocked into the CRC16 is loaded into crc16_count. We then shift crc16_data_in by one position and decrement crc16_count until we reach 0. Depending on whether we are computing or verifying a CRC16 value we will either raise the crc16_valid flag or compare to a fixed value according to Appendix F of [EPC08] and raise the crc16_valid flag.

2.8 Backscatter Encoding

Gen2 requires that the tag support two data encoding schemes for T→R communication, FM0 and Miller. In keeping with design modularity, we designed a transmission circuit (tx_fsm) shown in Fig. 13, charged with selecting the correct encoding scheme and transmitting the appropriate preamble and data. Inputs to tx_fsm are the number of bits to be transmitted, a 2 bit m value for selecting the encoder, and trext for selecting the preamble length. A 129-1 multiplexer selects the appropriate bit to transmit based on a counter value. The counter is incremented on the negative edge of the blf_clk and data is clocked into the encoders on the following positive blf_clk edge. When all bits have been transmitted the tx_complete flag is raised.

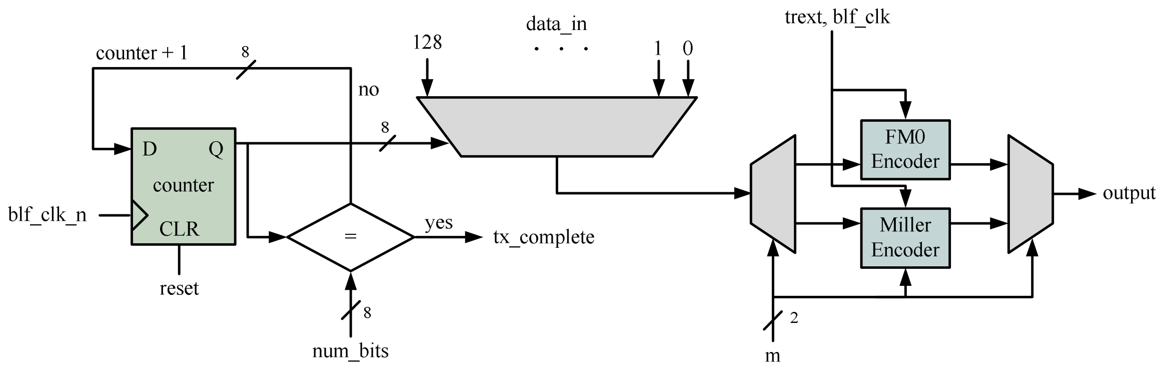


Figure 13: Transmission encoder circuit. The input register is preloaded with the data being transmitted to the reader. A counter is used to select the appropriate bits. The m value selects the appropriate encoding scheme. When the counter reaches the number of bits the tx_complete flag is raised. Trext is used to indicate the length of the preamble.

2.8.1 FM0 Encoding

Fig. 14 from [EPC08] is a waveform of all possible FM0 sequences. The output is toggled at the start of each symbol. FM0 encoding requires a phase change on the negative edge of the blf_clock when transmitting a 1 and no phase change when transmitting a 0. Therefore, the data rate in FM0 encoding is equal to the backscatter link frequency.

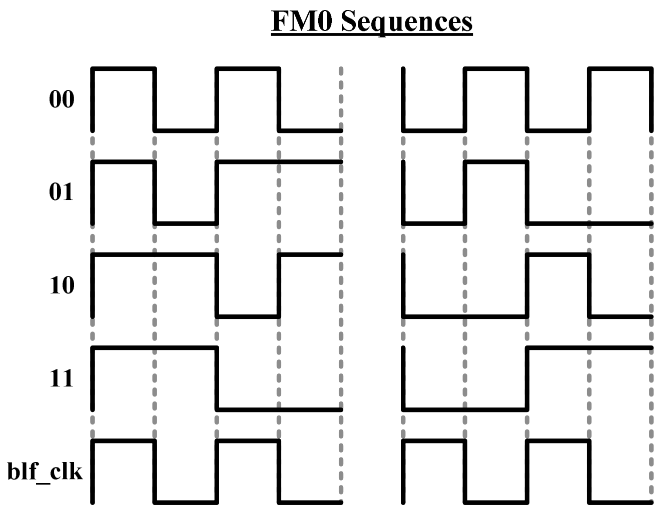


Figure 14: From [EPC08] FM0 encoding sequences.

2.8.2 Miller Encoding

Miller encoding works in a manner similar to FM0. The difference between a 0 and 1 is based on 180º phase shift, but with a more complex set of guidelines. To begin with, there is what is known as a subcarrier value (m) that determines the number of clock cycles required to transmit a single bit of data. Therefore, the data rate is the backscatter link frequency/m. A tag is required to support m values of 2, 4, & 8. Fig. 15 is a representative set of Miller encoded sequences. Also shown in Fig. 15 are the rules for when it is appropriate to change phase. This occurs on the boundary between consecutive 0's and on the m^th clock edge of every data 1 symbol.

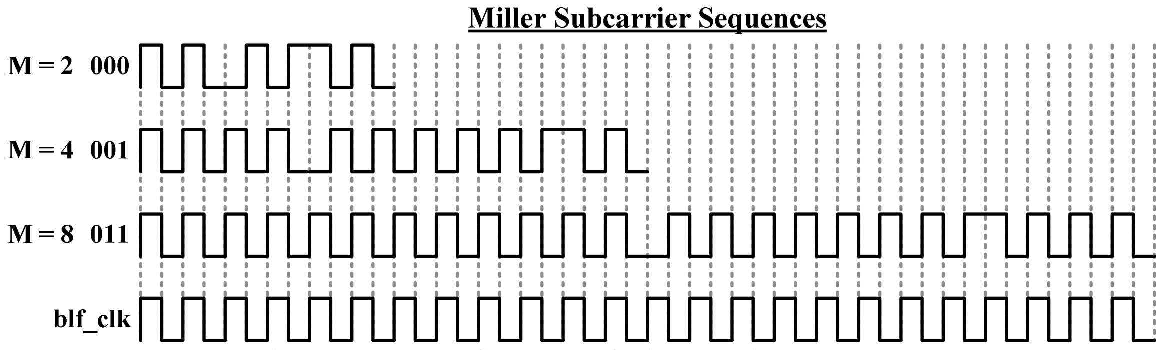


Figure 15: From [EPC08] subset of Miller encoding sequences.

2.9 Memory

As mentioned in section 1.1, Gen2 tags must contain some form of non-volatile memory used to store information such as the EPCID and access passwords. Section 6.3.2 of [EPC08] states this memory must be partitioned into 4 banks User, TID, EPC, and Reserved. Data is accessed in 16 bit words. For simplicity we chose to simulate memory with a set of 2-dimentional register arrays, addressed by the 2 bit mem_bank value, and the mem_word byte. We use a bit called write_enable to distinguish between memory reads and writes.

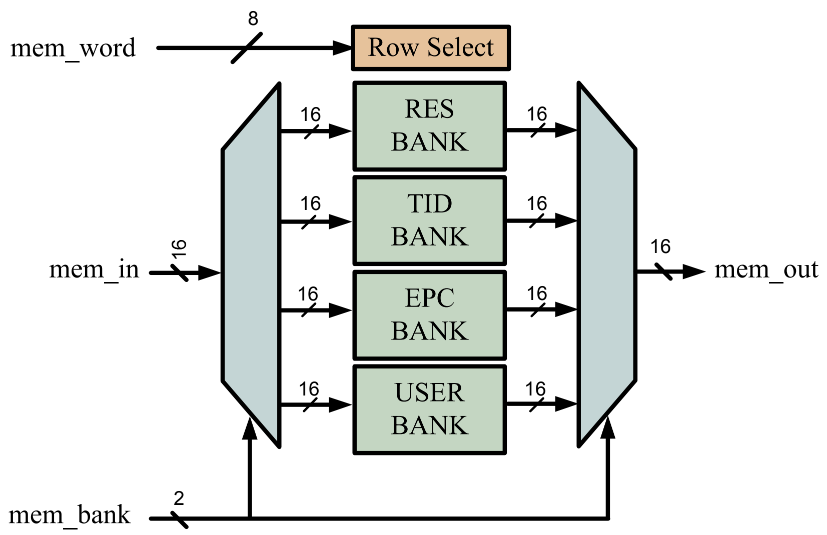


Figure 16: Memory configuration. For our prototype we chose to emulate non-volatile memory with a set of four 2-dementional arrays.

Because the register array is not actually non-volatile we also included a mem_reset input which resets the register arrays to a specific state. The act of resetting the memory also serves to hard code information such as the EPCID. It should be noted that we did not include any form of access control to prevent reads or writes to and from specific memory locations, though such a scheme would be a straightforward extension to the current architecture.

CHAPTER 3

SECURE SENSING WITH A GEN2 TAG

3.1 Environmental Sensing with a Gen2 Tag

The combination of a moderate read range of 10-30 ft [DD08], non-volatile memory, low cost reader hardware, and power harvesting make Gen2 tags well suited to function as the communication frontend for a sensor. This idea is evident in the design of the WISP which includes a temperature sensor and accelerometer and could be used for a variety of applications [HC07], [DH08], [SC09]. The basic principle being that in the best case, the power harvested by the tag could be used to power the sensor. While at the very least, the tag could serve as the communication frontend for the sensor, eliminating the power required to transmit data and allowing the sensor to last longer if it has a battery or harvest less energy from another source such as solar.

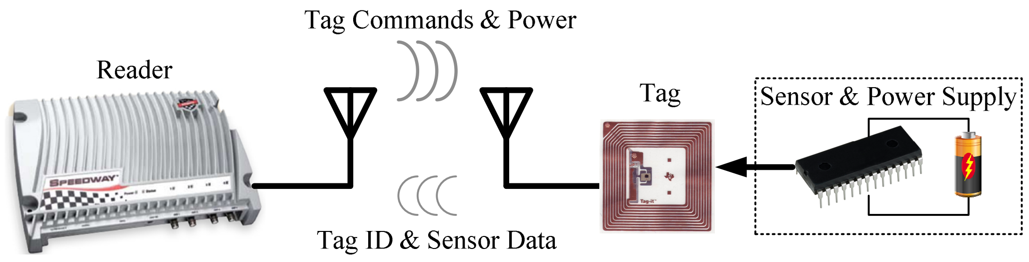


Figure 17: RFID tag with sensor block diagram adapted from [FT09], [IPJ08]. Power for the sensor could ideally come from the tag's harvesting circuitry, but could also be from a separate source such as a battery or solar. With no power drawn from the sensor to transmit the data battery life of the sensor would be extended or the sensor would function in less ideal conditions.

In [CT08] temperature sensor data was incorporated into the EPCID value and transmitted over an unsecured channel. We would like to take this a step further and provide a level of security by incorporating a novel low power block cipher. In this chapter we will justify the need for security, and explain how this was accomplished.

3.2 Security Motivations

Fig. 18 illustrates an example application in which we would want to protect sensor data. We will reference this application throughout this chapter. In this example, the bridge has been embedded with several sensors with RFID frontends. If we assume that our bridge is located in a remote location, constant monitoring of the tag could be considered challenging and expensive. Instead, periodically a vehicle equipped with an RFID reader passes over the bridge and collects the sensor data. The vehicle then returns to a secure location and downloads the information for later processing.

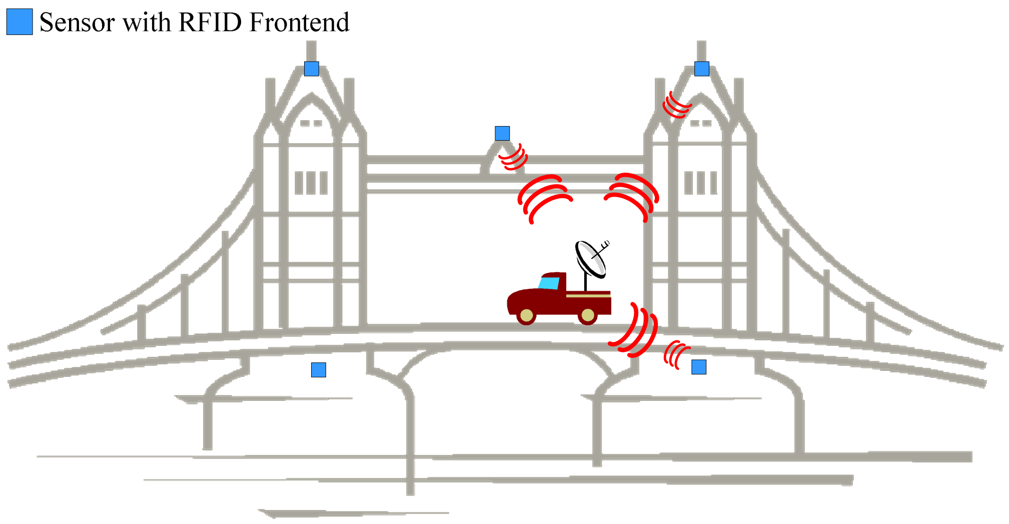


Figure 18: RFID sensor application illustration. Here we depict a vehicle periodically driving over the bridge to retrieve sensor data. This could simplify the infrastructure required to monitor the bridge, though several security implications must be considered.

This scenario could justify several layers of security. We would not want to reveal the sensor data to an untrusted reader because this might allow a malicious party to discover and target a weak point in the bridge if it were a mechanical stress sensor. We would also like to obscure the data in such a way that even if the sensor data remains the same, it will appear to change each time to an untrusted party; thereby stopping an adversary from inferring information about the sensor without knowing the actual value.

Finally, we would like to provide one way authentication in the sense that the trusted reader has a way to verify the origin of the data.

3.3 Security in Gen2

With the primary intended function of Gen2 tags being low cost and high volume inventory tracking, security is virtually nonexistent. While the protocol does contain a 16 bit PRNG, it mainly serves in an access control role, allowing tags to verify the intended recipient of reader commands, as well as slotting tags in a response queue. Providing security has been the focus of several research papers. In [AM07] an Advanced Encryption Standard (AES) block cipher was added with the protocol modified in the manner depicted in Fig 19.

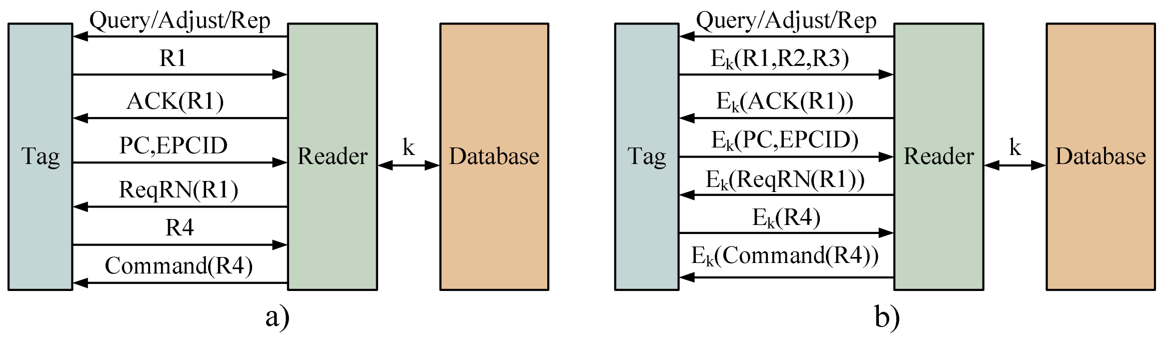


Figure 19: From [AM07] standard vs. secure command flow. a) Standard Gen2 command flow. b) Security enhanced command flow. K is the secret 128 bit AES key, and Ek() indicates the data is encrypted with AES using key k.

The security scheme in Fig. 19 suffers from several major drawbacks. The first being key distribution. According to [AM07], “We assume that the secret key is securely delivered to the reader from the database before the communication starts between the reader and the tag.” This immediately raises the question: How does the database know which tag it is communicating with before receiving the EPCID of the tag? The second drawback to this scheme is the reader must have a constant uplink to the backend database, increasing the difficulty in implementing this scheme. The final drawback is the vulnerability to replay attacks. An untrusted reader could read information from the tag and replay the data without knowing the secret key. The reader does not have a way to validate the origin of the data.

[DB06] proposes a security scheme which will serve as the basis of our protocol. A tag computes $R_T = H(K_{TS}, C_R)$ where H() is a cryptographic functions such as a block cipher, K_{TS} is a shared secret key stored on the tag and reader (or backend server), and C_R is a unique challenge sent from the reader to the tag. The tag then replies with its EPCID and R_T. This process is described in Fig. 20.

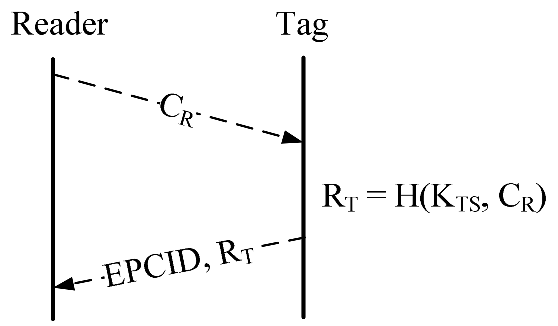


Figure 20: Security scheme proposed in [DB06]. C_R represents a unique challenge from the reader to the tag. K_{TS} is a unique key shared by the reader and tag. H() is a cryptographic function such as a block cipher.

This scheme provides several advantages over the previous scheme. Protection against replay attacks is provided by the length of C_R. Because an adversary does not know C_R ahead of time, they must store all possible C_R and corresponding R_T values to clone the tag. Authentication is provided in the strength of the function H. With EPCID, the reader or backend server can select the appropriate K_{TS} and decrypt R_T. If C_R is recovered, than the origin of the information can be assumed to be the correct tag. The only drawback to this scheme is that if C_R is static, R_T is also static. This could allow an untrusted reader to imply information about the contents of the tag by querying the tag with a constant C_R. We will build upon this scheme in deriving our protocol to protect sensor data.

3.4 Securing Sensor Data

Chapter 4 describes in detail all of the enhancements to the basic digital backend we have incorporated into our platform, but in order to properly describe our sensor data protection protocol we must assume that our tag contains a sensor and a block cipher. For the reasons described in section 4.2, we will be using PRESENT, a novel block cipher with an 80 bit key and 64 bit data size.

3.4.1 Guidelines

Focusing on our example from Fig. 18, we have come up with a specific set of guidelines for our security scheme:

1. Full backwards compatibility with the Gen2 standard.
2. Reader does not need an uplink to a backend secret key database or decryption mechanism to gather data from the tag.
3. Defense against replay attacks.
4. Tag data integrity checking.
5. Authentication of the origin of the data.

Expanding on our example we can justify these requirements. First, if we say that the bridge is in a remote location, we could assume it is difficult and expensive to maintain an uplink to a backend server. To keep the system costs down, we would like to use an off-the-shelf reader, meaning it does not support encryption or decryption. This is also why we must be fully backwards compatible with the Gen2 protocol.

Next, we would like to ensure that the data is secure, verifiable, and that an adversary cannot infer the status of the sensor from the ciphertext. In the bridge analogy we would not like to tell an adversary what the status of any one sensor is as it may divulge weaknesses. We would like to confirm what point on the bridge the data came from to properly direct maintenance. Lastly, if a sensor normally emits a static value until something is wrong, we would not like an adversary to infer what the status of the sensor is just by seeing that the ciphertext has changed.

3.4.2 Adversary

As with any security protocol, a detailed description of our threat model and adversary are required. In our design we assumed the following:

1. The adversary is fully aware of all aspects of our protocol.
2. The adversary can request the data from the tag an unlimited number of times.
3. The adversary can change any data being transmitted between the reader and tag.
4. Each tag contains an 80 bit stored secret symmetric key (k_i) that our adversary does not know.

3.4.3 Security Protocol

Fig. 21 & 22 depict the manner by which we propose to transfer encrypted data from the tag to the reader. Data collection is separated into two distinct sections. In the first dubbed *online*, the reader collects encrypted data and ID pairs from the tags. In the second *offline* portion, the reader returns to a secure location to download the data to a backend server and perform the security checks.

Fig. 21 is a depiction of the Reader-Tag online portion. The start of this protocol exactly matches the original Gen2 protocol. In our extension, once the EPCID has been received by the reader, the reader emits a "Data Request" command and an 80 bit true random number (TRN_{80}). This number must be 80 bits to match the key size of the block cipher we are using. At this point the tag will XOR TRN_{80} with the stored symmetric key (k_i), concatenate a 40 bit binary counter value (CV_{40}) with the 24 bits of sensor data (D_{24}), and encrypt it using the result of the 80 bit XOR operation.

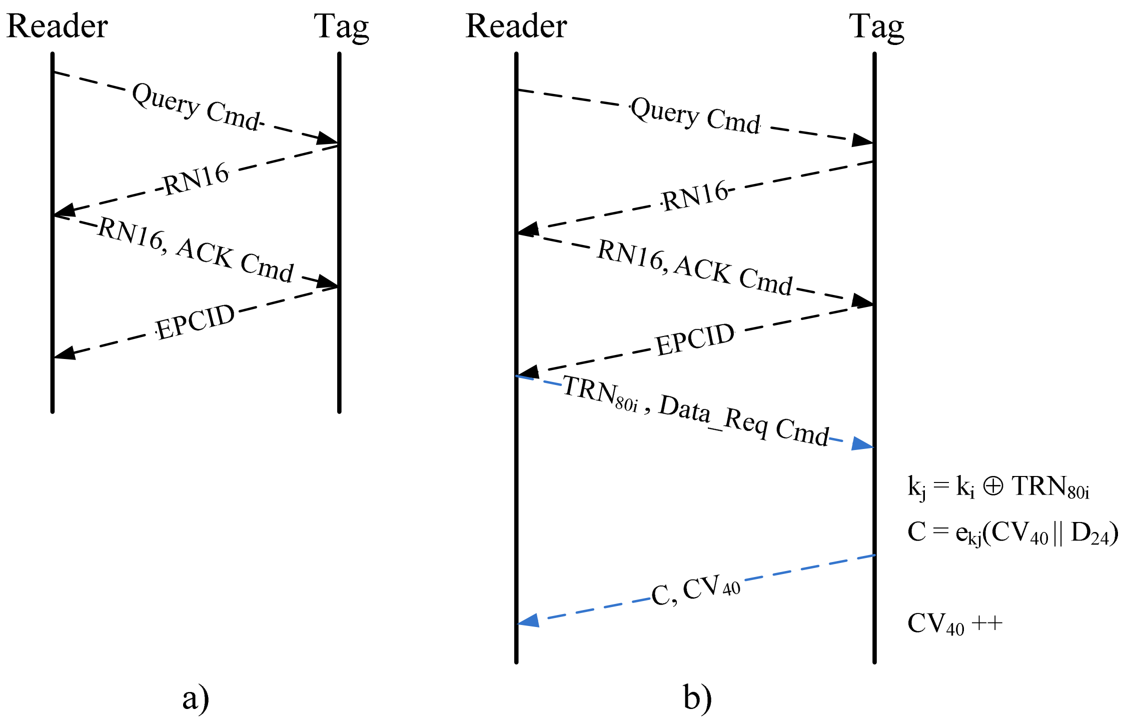


Figure 21: Online portion of secure data transfer. a) Standard Gen2 protocol for transmitting the EPCID of the tag b) Our security extension which begins with the same set of commands. After EPCID has been transferred, an 80 bit true random number (TRN80) is sent to the tag. The tag then computes an XOR of TRN_{80} and the stored secret key k. The result of the XOR is used as the key to encrypt 24 bits of sensor data (D_{24}) and a 40 bit counter value (CV_{40}).

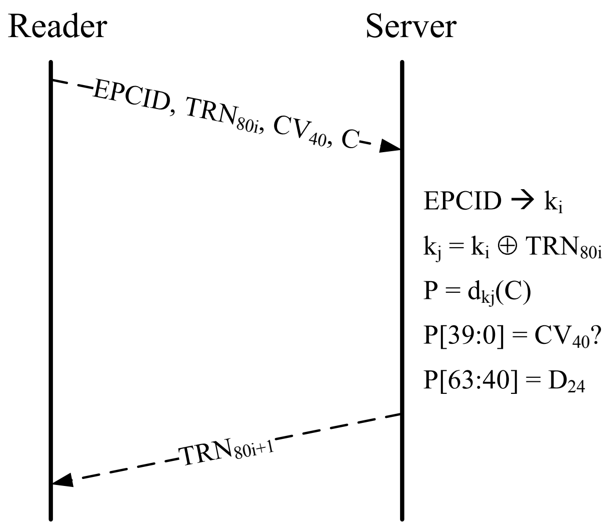


Figure 22: Offline tag data upload and integrity check. The communication between the reader and server is assumed to be over a secure channel. Using the previous TRN, the tag ID, and counter value, the server is able to decrypt the message and ensure that there were no transmission errors. This is the 2nd step in our two part secure data transfer scheme.

Fig. 22 depicts the second portion of our scheme, the offline data download. After all tags have been queried, the reader returns back to a secure location and uploads the tag information to the symmetric key database. Using the EPCID the server is able to retrieve the appropriate shared symmetric key. Then the server can perform an XOR between the key and TRN_{80} and decrypt the ciphertext C. The server can than determine the validity of C by checking if the decrypted version of CV_{40} matches the received version. The reader then receives the next TRN_{80} from the server. This removes the need for the reader to carry a TRNG. The reader will need a separate, unique TRN_{80} for each tag it plans to query.

3.5 Security Analysis

3.5.1 Security Basis

The security of the protocol described above is derived from what is known as unilateral authentication, using random numbers from [AM01]. This scheme is shown by Eq. 1, 2:

$$A \leftarrow B : r_B \tag{1}$$
$$A \rightarrow B : E_K\left(r_B, B^*\right) \tag{2}$$

Here A is the tag, B is the reader, r_B is a true random number, B* is the data requested by the reader, and $E_K()$ represents a cryptographic function such as a block cipher encrypted with key K. B decrypts the message sent by A and checks that the random number is correct. Part of the novelty of our scheme is that the block cipher is in a modified version of counter mode. Fig. 23 depicts the standard version of block cipher counter mode versus the modified version.

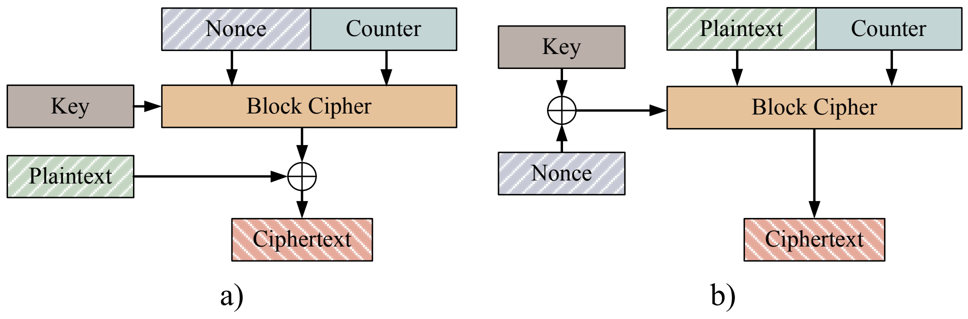


Figure 23: a) Standard block cipher counter mode. The number used only once (nonce) represents a true random number value. b) Modified version of counter mode in which the counter and plaintext are concatenated and encrypted with a key composed of an XOR of the secret key and the nonce.

The reason for this modification is the key space of PRESENT (80 bits) is greater than the ciphertext space (64 bits) and provides greater protection against replay attacks. This version further justified given the size of the plaintext we are protecting. Sensor data is on the order of only several bits and here we can use the counter value as padding for the plaintext. We will delve further into the implications of our design below.

3.5.2 CV_{40}

Table 2: Bit length for the subset of reader commands required in our security protocol.

Reader Commands	Data Bits
Query	22
ACK	18
Data_Req	112

Table 3: Bit length for data transmitted by the tag in our security protocol. We are assuming FM0 encoding and a short preamble.

Tag Replies	Data Bits	End of Signaling Bits	Preamble Bits
RN16	16	1	6
EPCID	128	1	6
Sensor Data	120	1	6

The impact of the 40 bit counter value is twofold. First it is used to help ensure the adversary cannot infer information about our data by passively observing the ciphertext. Because C is comprised of $CV_{40} \parallel D_{24}$, should D_{24} and TRN_{80} remain the same, the 40 bits of CV_{40} will ensure that our data cycles at least 2^{40} times before repeating. To establish how long this would take we must first calculate the time required for each round of data transfer. Tables 2 and 3 contain the number of bits required for each reader command and tag reply respectively assuming that a CRC16 value is transmitted with C and CV_{40} to protect against transmission errors.

Section 6.3.1.2.4 of Gen2 states 6.25μs and 9.375μs are the shortest possible 0 and 1 PIE values supported respectively. If we assume an equal distribution of 0s and 1s, FM0 encoding, and 640kHz T→R we can calculate the minimum time required for each round. Finally, we also assume that the Data_Req command and sensor data transfer are repeated $2^{40} - 1$ times because once EPCID has been transferred data can be requested an infinite number of times. Table 4 summarizes these calculations.

Table 4: Data transfer timing calculation summary.

Parameter	Value	Unit
Data1	9.375	μs
Data0	6.25	μs
RTcal	15.625	μs
TRcal	33.3	μs
BLF	640	kHz

Reader Subtotal	1286.425	μs
Tag Subtotal	445.3125	μs
Total (Single Exchange)	1.7317375	ms

Total (Repeated Exchanges)	1.18026E+15	μs
	37.42570437	years

Table 4 shows that assuming both the fastest transmit and receive values allowed by the Gen2 protocol our adversary would need approximately 37.5 years to transmit the data required to view one data cycle, a sufficiently long time for our application. This number could increase further if the bit length of the counter value is increased.

The second quality of CV_{40} has to do with verifying the integrity of the data received by the reader. When C is decrypted offline, if CV_{40} does not match the expected value then it is clear that the data portion is not valid. As we stated previously, CRC16 is used to protect against transmission errors which could cause the same result.

3.5.3 TRN_{80}

The motivation behind the 80 bit random number is protection against replay attacks. Without this, an adversary could collect data from the tag, and while they may not be able to decrypt the data, an adversary could replay this data to the reader. The reader could still properly decrypt the information, but since a Gen2 tag has no real time clock or way to ensure data freshness, the data would seem valid. By including this number, the adversary would need to store 2^{80} different TRN_{80} values as he would not know what number will serve as the challenge from the trusted reader. He would also need 2^{64} different combinations of C, as our block cipher has an output of 64 bits and 2^{40} different CV_{40} values so that he could properly reply with old data. It is critical TRN_{80} be a true random number as opposed to a pseudo random number to avoid the possibility of the adversary guessing what number will be used next, forcing him to store all possible combinations. This would result in a total data storage of $80*2^{80} + 2^{64}*64 + 2^{40}*40$ bits = $1.2*10^{13}$ terabytes of information per tag, an unfeasibly large amount for our application.

3.6 Security Protocol Proof of Concept

The sensor data security scheme outlined above fits within the framework Gen2 by making use of the opcode space reserved for custom commands to transmit our Data_Req command and random number. However, with the limited RFID reader software available we needed a way to practically emulate our protocol using only basic Gen2 commands. Fig. 24 illustrates the difference between our actual protocol and the proof of concept. In the latter, we use the 16 bit password parameter of the access command to construct the TRN_{80} value. The 32 bit access password is delivered in two 16 bit packets by consecutive access commands. In our proof of concept we string together these packets to form the 80 bit value required. Though it is clear that this would severely reduce the security of our scheme if deployed in this manner, it serves as a sufficient proof of concept to demonstrate our protocol.

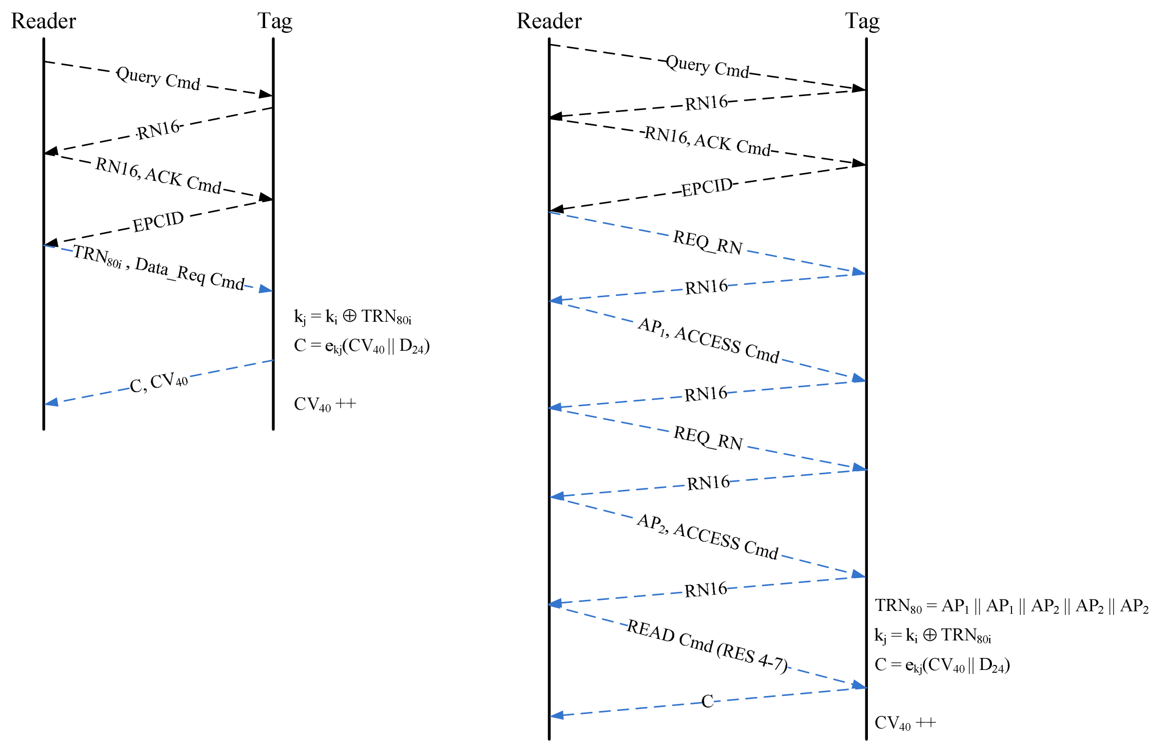


Figure 24: Sensor data security scheme POC command flow. a) Actual sensor data security scheme. b) Proof of concept of sensor data security scheme. Here we have utilized only basic Gen2 commands. By concatenating and reusing the 16 bit access passwords (AP_1 & AP_2) we can mimic our 80 bit true random number.

CHAPTER 4

ENHANCING THE DIGITAL BACKEND

4.1 Enhancement Overview

Fig. 25 is a block diagram depiction of our enhanced Gen2 digital backend. Our design leverages work on clocking and pseudo random number generation from [QL09] and [PP07] respectively. We will also introduce a novel block cipher known as PRESENT [AB07], and the circuitry required to communicate with the temperature sensor. Justification and implementation details for each new block can be found throughout this chapter.

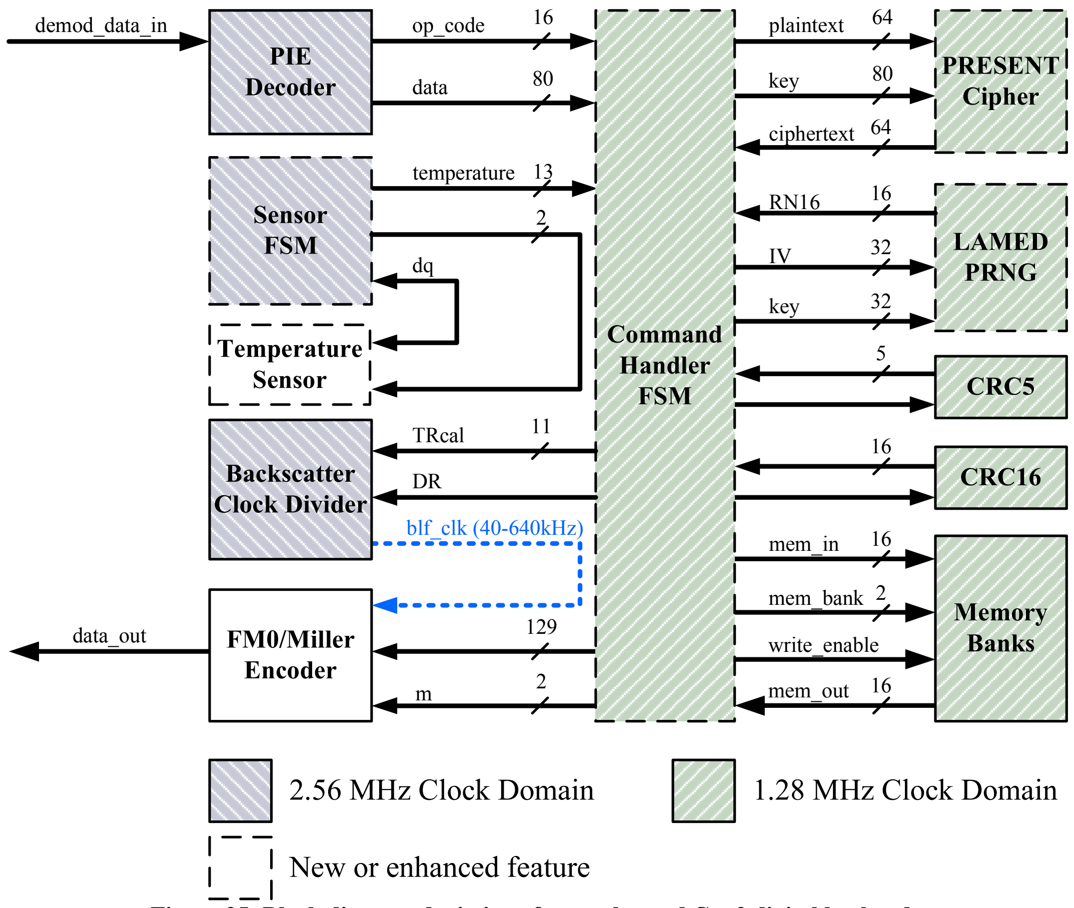


Figure 25: Block diagram depiction of our enhanced Gen2 digital backend.

4.2 PRESENT

The security protocol outlined in section 3.4.3 hinges upon the availability of a symmetric block cipher. PRESENT is a 64 bit block cipher with an 80 bit key, designed for low power, low gate count applications [AB07]. In order to evaluate their design, the creators of PRESENT normalized several area optimized designs to the NAND2 gate of the technology used, a technique known as gate equivalence (GE). The authors claim an approximate GE of 1.5k in their implementation of PRESENT. The comparison between PRESENT and other area optimized block ciphers is illustrated in Fig. 26.

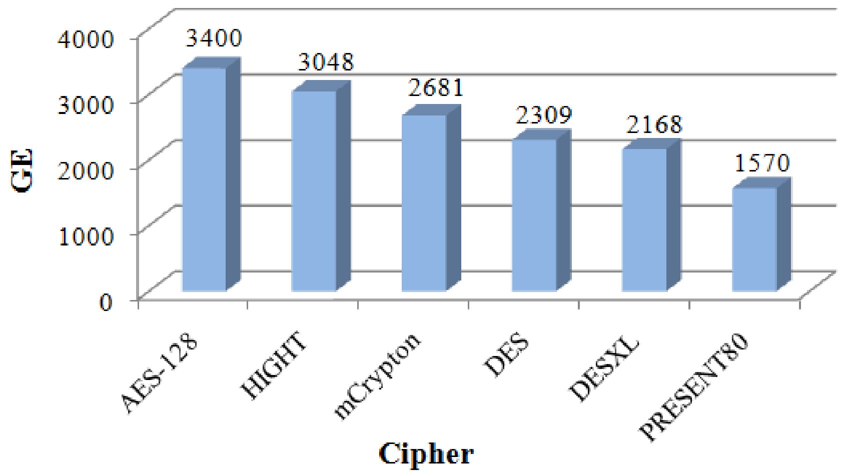


Figure 26: From [AB07] a gate equivalence comparison of several area optimized block ciphers. With area a primary design constraint, PRESENT is an excellent choice for our RFID sensor platform.

4.2.1 PRESENT Algorithm

The basic PRESENT algorithm is given in Fig. 27 from [AB07]. There are 32 rounds with each round consisting of a 64 bit round key XOR, sbox, and permutation layer. After each round, the next round key is generated by the key scheduler and is represented by the *update* box in Fig. 27. The sbox is the direct 4-4 bit substitution in Table 5. One of the powerful aspects of PRESENT is the simplistic manner by which the PRESENT sbox can be implemented in hardware using only 10's of gates, which compares favorably to AES 8-8 bit sbox that requires approximately 200 gates [DC05]. The permutation layer reallocates 62 of the 64 bits to different bit positions dictated by Table. 6. Naturally, this layer does not require any gates, only specific wire routing, and does not contribute to the GE of PRESENT.

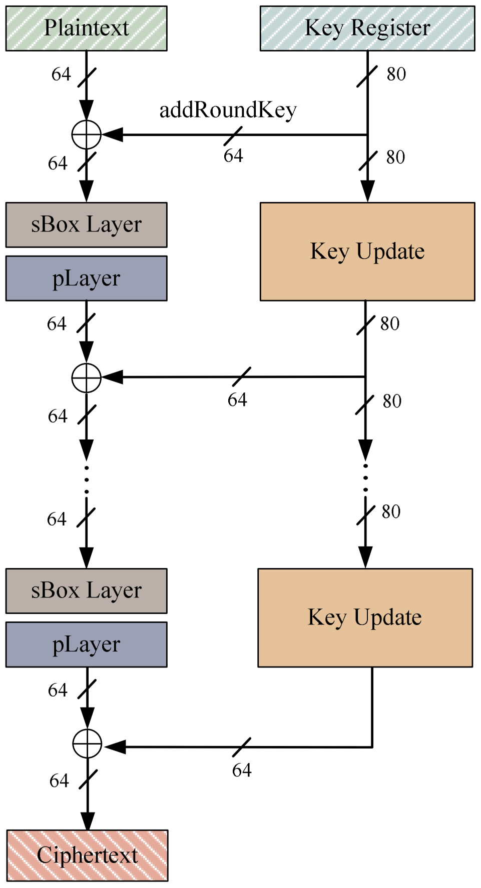


Figure 27: From [AB07] PRESENT algorithm block diagram.

Table 5: From [AB07] PRESENT sbox layer substitution mapping.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 6: From [AB07] PRESENT permutation layer

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P(i)	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P(i)	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
P(i)	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
P(i)	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

4.2.2 PRESENT Key Scheduler

Before each round of sbox and permutation, an XOR is done with the 64 most significant bits of the key, which is then updated with key scheduling algorithm. This is done a total of 32 times. The key scheduling algorithm is given in Alg. 2, which states that the key is rotated by 61 positions, the 4 most significant bits are put through the PRESENT sbox, and then an XOR of bits 19-15 and the round counter is done.

```
$[k_{79}k_{78}\ldots k_1k_0] = [k_{18}k_{17}\ldots k_{20}k_{19}]$
$[k_{79}k_{78}k_{77}k_{76}] = \mathrm{sbox}[k_{79}k_{78}k_{77}k_{76}]$
$[k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus \text{round counter}$
```

Algorithm 1: PRESENT key scheduling algorithm

4.3 LAMED

LAMED is a pseudo random number generator (PRNG) designed by [PP07] to conform to the Gen2 requirements of:

1. Probability of a single RN16: The probability that any RN16 drawn from the RNG has value RN16=j for any j, shall be bounded by Eq. 3.

$$\frac{.8}{2^{16}} < P(RN16 = j) < \frac{1.25}{2^{16}} \tag{3}$$

2. Probability of simultaneously identical sequences: For a tag population of up to 10,000 tags, the probability that any of two or more tags simultaneously generate the same sequence of RN16s shall be less than 0.1%, regardless of when the tags are energized.
3. Probability of predicting an RN16: An RN16 drawn from a tag's RNG 10 ms after the end of Tr, shall not be predictable with a probability greater than 0.025% if the outcomes of prior draws from the RNG, performed under identical conditions, are known.

The LAMED algorithm given in Alg. 2 is constructed from 32-bit XORs, 32-bit ADDs and variable length barrel shift operations. The simplicity of these operations should keep the design power requirements relatively low, though no direct power numbers are reported by the authors.

```
If n is odd, $a_0 = a_1 + iv$ & $a_1 = out \oplus s$
If n is even, $a_0 = a_1 \oplus iv$ & $a_1 = out + s$
$aux1 = a_0 + a_1$
$aux2 = a_0 \oplus a_1$
$aux3 = barrelshift(aux1, 5)$
$aux3 = aux3 + aux2$
$aux3 = barrelshift(aux3,3)$
$aux3 = aux3 \oplus aux1$
$aux3 = barrelshift(aux3, 4)$
$aux3 = a_1 + aux3$
$aux3 = barrelshift(aux3, 2)$
$aux3 = aux3 + aux1$
$aux3 = barrelshift(aux3, 2)$
$aux3 = aux3 \oplus aux1$
$aux3 = barrelshift(aux3, 3)$
$aux3 = aux3 + a_1$
$aux3 = barrelshift(aux3, 2)$
$aux3 = aux3 + aux1$
$aux3 = barrelshift(aux3, 4)$
$aux3 = aux3 \oplus a_1$
$aux3 = barrelshift(aux3, 1)$
$aux3 = aux3 + aux2$
$aux3 = barrelshift(aux3, 2)$
$out = aux1 \oplus aux3$
```

Algorithm 2: LAMED PRNG Algorithm. a0 is a stored unique key and a1 is an initial vector. Aux1, 2, & 3 are temporary registers used in the algorithm. All vectors are 32 bits in length.

Two 32 bit vectors, a_0 and a_1, are operated on to generate a 32 bit pseudo random number (PRN). An XOR of the 16 MSB and 16 LSB of the 32 bit output creates the 16 bit PRN. These vectors are then updated for each new PRN generation round according to Eq. 4, 5 where the initial values for a_0 and a_1 are a stored unique secret key, s, and an initial vector, IV, respectively. From these equations we can see that at the end of each round we must store the output and a_1 value from the previous round to prevent the PRNG from initializing to the same value each time. We have chosen to store these values in the User memory bank words 0-3.

$$a_o^n = \begin{cases} a_1^{n-1} + IV & n = odd \\ a_1^{n-1} \oplus IV & n = even \end{cases} \tag{4}$$

$$a_1^n = \begin{cases} out_{n-1} \oplus s & n = odd \\ out_{n-1} + s & n = even \end{cases} \tag{5}$$

The authors put their design through a battery of statistical tests to prove that it does in fact conform to the randomness requirements of Gen2. They also claim that their implementation is approximately 1.5k gates. This is an acceptably low gate count when compared to other similar LFSR based stream ciphers such as Trivium and Grain which require an extra 90% and 34% overhead compared to LAMED [PP07].

4.4 Dual Clocking Scheme

The authors of [QL09] make a compelling argument for the use of a separate clock to drive the PIE decoder and backscatter link frequency divider due to both lower bit error rate and power consumption of the tag. The standard clock frequency quoted for driving the digital logic in a Gen2 tag is 1.28MHz [AN07]. Looking at Fig. 28, we can see that certain symbols will have an error of ± 1 sample.

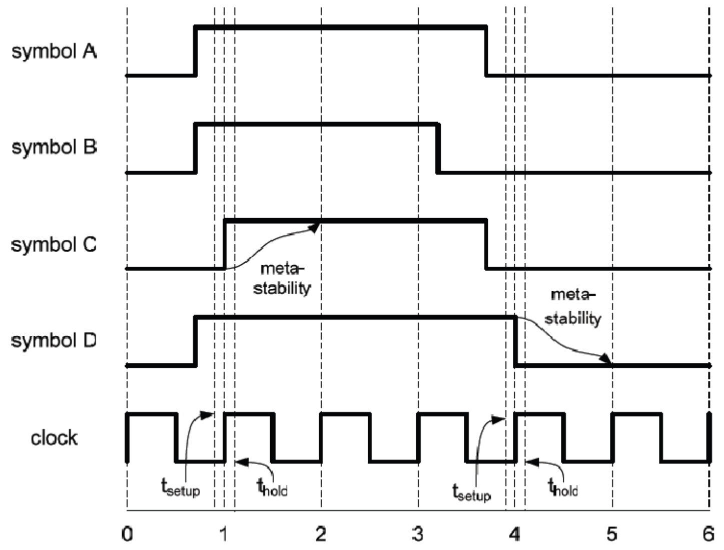


Figure 28: From [QL09] PIE sampling error. PIE decoding requires sampling the incoming signal and counting the number of samples to interpret the data. Depending on when the data transitions, a sampling error of +/- 1 sample is possible.

The authors are able to show that a single 1.28MHz clock is insufficient for two reasons. The first is that because of this sampling error data coming from the reader could be misinterpreted. The second is that by powering down the 2nd clock when the tag is neither receiving nor transmitting data, that a power savings of between 5-11% is possible. For these reasons we have designed our platform with two clock domains. In implementation we were only able to acquire a 10.24MHz clock; therefore we used two simple clock dividers, which we do not consider part of our design, to generate both clocks and dedicated one FPGA I/O pin to resetting the clock dividers.

4.5 Temperature Sensor

Several factors were considered in the choice of our temperature sensor. In order for our platform to realistically demonstrate its use in an environmental sensing capacity, we needed a low speed sensor with minimal external wires to keep the load capacitance, and therefore power, to a minimum. Also, early in the design process we were not sure of the I/O voltage requirements and were therefore looking for a sensor that would work over a range of voltages. For these reasons we selected the DS1620 from Maxim Integrated Products [DS1620].

This sensor is rated to work over a range of 2.7-5V at speeds below 1.75MHz. The sensor has 9 bits of precision and is accurate to ½ ºC, which is sufficient for our purposes. Data is transferred over a 3 wire communication bus using a protocol similar to the Serial Peripheral Interface protocol shown in Fig 29. Command sessions are initiated by driving the rst_n signal high while the clk_conv signal is high. Data on the dq pin is then clocked on the rising edge of each consecutive clk_conv signal LSB first. Fig. 29 represents a simulation of the commands 8'hEE and 8'hAA being clocked into the sensor, followed by 9'h1FF being received.

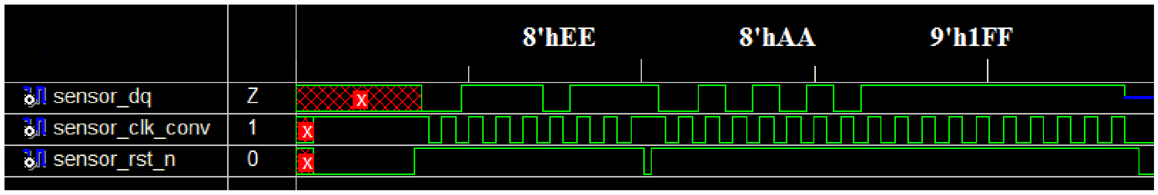


Figure 29: Simulated output of the temperature sensor FSM. The controller transmits 8'hEE and 8'hAA to prepare the sensor and 9'hFF represents the 9 bit temperature reading. All data is clocked a the rising edge of clk_conv.

We designed an FSM which queries the sensor upon power up or reset and then saves the sensor data to memory bank RES word 4. The sensor FSM then signals completion by raising the temp_ready flag. This allows our sensor to be replaced with minimal changes, keeping in line with our goal of reuse and modularity.

CHAPTER 5

DESIGN EMULATION & VALIDATION

5.1 Our Platform

This work combines a Xilinx Spartan 3a FPGA board, WISP RFID tag, and temperature sensor to create our hardware programmable RFID platform illustrated in Fig. 30. The idea is to use a portion of the analog circuitry from the WISP as a means for the FPGA to transmit data to and receive data from the RFID reader. To do this we will disable the MSP430 microcontroller, sample the receive pin, and toggle the transmit pin from the FPGA board. Unlike a standard RFID tag, power and clock will be supplied by the FPGA board. A photograph of our experimental platform is found in Fig. 31.

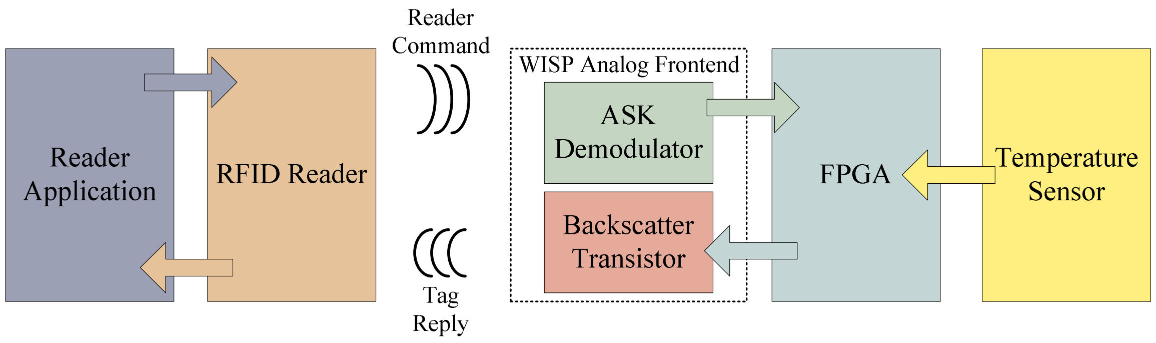


Figure 30: FPGA RFID sensing platform block diagram.

A schematic and photograph of the circuitry required to properly interface the WISP and Spartan FPGA is given in Fig. 32. Because the WISP I/O operates at 1.8V and FPGA operates at 3.3V, voltage dividers in the form of potentiometers were used on the transmit and receive enable pins. The WISP also inverts the demodulated PIE encoded data from the reader so the inverter not only reverses this operation, but shifts the data into the 3.3V domain.

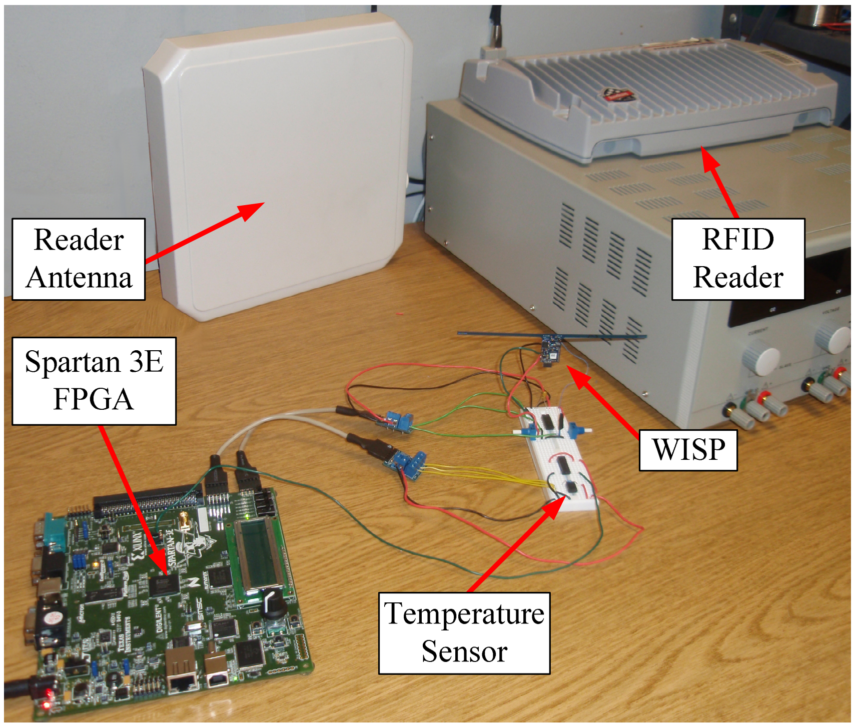


Figure 31: Photograph of our FPGA RFID sensing platform. Commands from the RFID reader are demodulated with the WISP circuitry and sampled by the FPGA. Commands are transmitted to the reader by toggling the WISP transmit I/O pin from the FPGA.

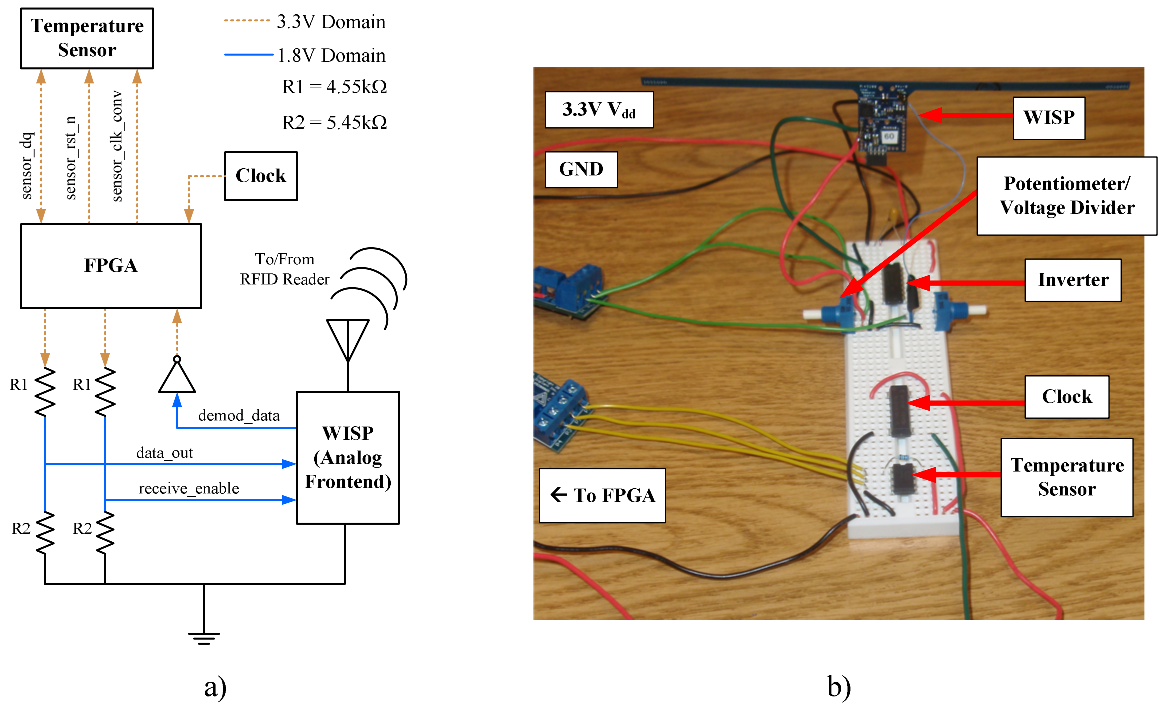


Figure 32: Circuitry required to integrate the FPGA with the WISP and sensor. a) schematic. b) photograph.

5.2 Validation

5.2.1 Validation Overview

One significant challenge facing our design is verifying the system works to specification. For most of the smaller components, Verilog simulation testbenches should be sufficient. As the sub blocks are integrated into larger and more complex modules, constructing meaningful testbenches becomes more challenging and simulation time increases accordingly. On top of this, in order to verify that our entire design is compliant with the Gen2 standard in any meaningful manner, we must be able to test it with devices known to be compliant. To solve this challenge we will use a 3rd party RFID reader, which is known to be compliant, to test several aspects of our design.

The case can be made that the most critical component to the success of this thesis is the manner by which we validate our design. This is the primary function of our emulation platform. Using a 3rd party RFID reader known to be Gen2 compliant allows us a high level of confidence in the performance of our design. We were also able to use a Tektronix DPO7104 Digital Phosphor Oscilloscope to probe our design and capture I/O traces.

During debug we were only capable of probing four signals with the oscilloscope, leaving us partially blind to much of the inner state of the design during emulation. To circumvent this issue, we were able to design a perl script which was used to translate the I/O recorded by the scope into a Verilog testbench that we could then simulate in ISE. The input to this script is one channel of oscilloscope data in Waveform Text File format. The output is a pin name and verilog *initial* statement timing format. This process, illustrated in Fig. 33, allowed us to conduct functional simulations with actual stimuli provided by the RFID reader and compare expected against actual results, greatly enhancing our understanding of the protocol as well as reducing debug time. We have included this script in Appendix M, though it is in a format which requires the file and pin name to be specified in the script before execution.

5.2.2 RFID Reader

Through a joint effort with Professor Kevin Fu and the Computer Science department, we were able to verify our design with an Impinj Speedway Gen2 RFID reader. Using the Ethernet/web software interface shown in Fig. 34, we were able to verify of our design under varying conditions such as T_{ari}, backscatter link frequency, backscatter encoding, and PIE symbol ratio of the length of a Data1 symbol to a Data0 symbol. Table 7, lists the range of parameters we were able to verify.

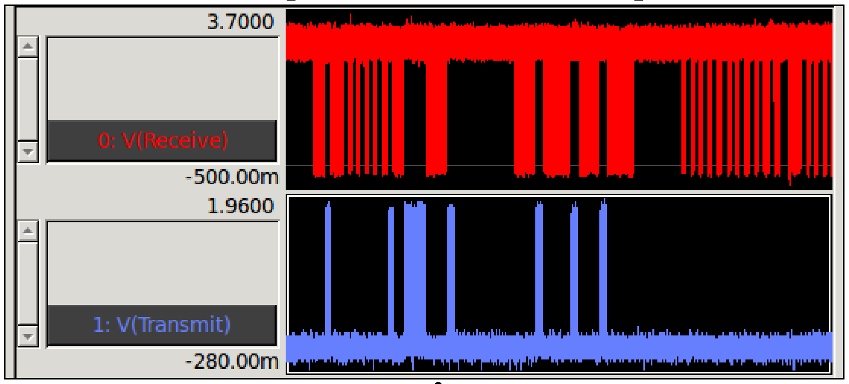
Table 7: Range of Gen2 parameters verified.

Reader Commands	Tari (us)	BLF (T-->R) (kHz)	Backscatter Encoding	PIE Ratio
Query	7.14	160	FM0	1.5:1
QueryRep	12.5	256	Miller4	2.0:1
QueryAdjust	25	640	Miller8	
Select				
Access				
ReqRN				
ACK				

5.2.3 Validation Milestones

During the planning of our project we were able to identify five major validation milestones. Each of these checkpoints involved integrating a greater number of modules as well as new I/O between the FPGA and reader. These milestones are listed in Table 8.

I/O Captured from Oscilloscope



Waveform Text File

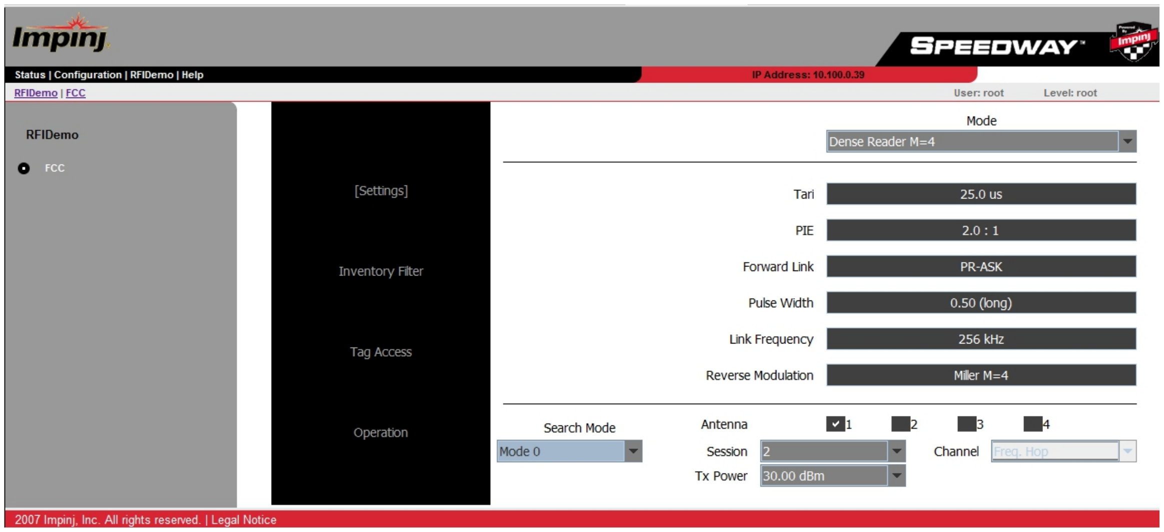
```
#Time    V(Receive)
-1.50280000e-003        3.05999986e+000
-1.50260000e-003        2.61999986e+000
-1.50240000e-003        3.13999986e+000
-1.50220000e-003        3.01999986e+000
-1.50200000e-003        2.89999986e+000
-1.50180000e-003        2.81999986e+000
-1.50160000e-003        2.93999986e+000
-1.50140000e-003        2.97999986e+000
-1.50120000e-003        2.93999986e+000
-1.50100000e-003        2.89999986e+000
-1.50080000e-003        2.81999986e+000
-1.50060000e-003        3.09999986e+000
-1.50040000e-003        2.85999986e+000
-1.50020000e-003        2.73999986e+000
```

Perl Script

Verilog Testbench

```
#0              demod_data = 1;
#1029600        demod_data = 0;
#11400          demod_data = 1;
#9600           demod_data = 0;
#2800           demod_data = 1;
#28600          demod_data = 0;
#3000           demod_data = 1;
#47000          demod_data = 0;
#2800           demod_data = 1;
#16000          demod_data = 0;
#3000           demod_data = 1;
#9400           demod_data = 0;
#3000           demod_data = 1;
#9600           demod_data = 0;
#3000           demod_data = 1;
```

Figure 33: Illustration of I/O capture to verilog testbench.



Mode: Dense Reader M=4

Tari: 25.0 us

PIE: 2.0 : 1

Forward Link: PR-ASK

Pulse Width: 0.50 (long)

Link Frequency: 256 kHz

Reverse Modulation: Miller M=4

Search Mode: Mode 0

Antenna: ✓ 1, 2, 3, 4

Session: 2

Tx Power: 30.00 dBm

Channel: Freq. Hop

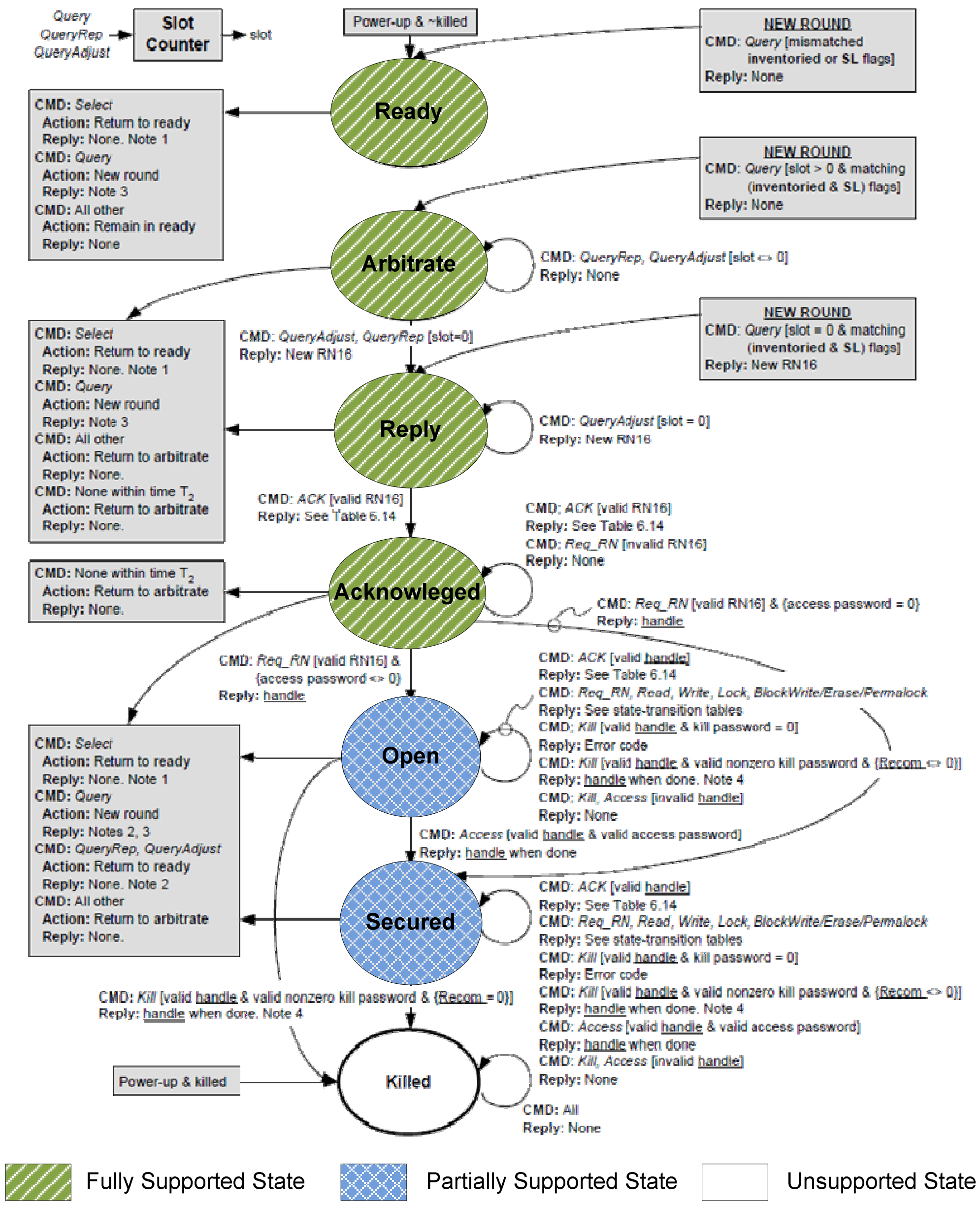
Figure 34: RFID reader software interface used in design validation.

Table 8: Validation milestones.

Milestone	Description	New Reader Commands	New Components Tested
M1	Generation and transmission of RN16 with Reader ACK	Query, QueryRep, QueryAdjust, ACK	PIE Decoder, CRC5, PRNG, FM0 & Miller Encoders, BLF Divider
M2	Transmission of EPCID	Select	CRC16
M3	Memory Reads	ReqRN, Read	Memory Banks & Controller
M4	Sensor data transmission	-	Sensor FSM
M5	Encrypted sensor data transmission	Access	PRESENT

5.3 Supported Gen2 Structure

Fig. 35 illustrates the extent of the Gen2 structure supported in our design. The only state not fully or partially supported is Killed in which the tag is fully disabled and does not respond to reader commands. Partial support for the Open and Secured states stems from the fact that we do not support the Kill or Lock commands which are required for full compliance. These commands restrict or disable tag response based on plaintext passwords and were not necessary for this project. Implementation of these commands using encryption would be an excellent example of future work using our platform.



Query
QueryRep
QueryAdjust
Slot Counter
slot
Power-up & ~killed
NEW ROUND
CMD: Query [mismatched inventoried or SL flags]
Reply: None
Ready
CMD: Select
Action: Return to ready
Reply: None. Note 1
CMD: Query
Action: New round
Reply: Note 3
CMD: All other
Action: Remain in ready
Reply: None
NEW ROUND
CMD: Query [slot > 0 & matching (inventoried & SL) flags]
Reply: None
Arbitrate
CMD: QueryRep, QueryAdjust [slot <> 0]
Reply: None
NEW ROUND
CMD: Query [slot = 0 & matching (inventoried & SL) flags]
Reply: New RN16
CMD: QueryAdjust, QueryRep [slot=0]
Reply: New RN16
CMD: Select
Action: Return to ready
Reply: None. Note 1
CMD: Query
Action: New round
Reply: Note 3
CMD: All other
Action: Return to arbitrate
Reply: None.
CMD: None within time T2
Action: Return to arbitrate
Reply: None.
Reply
CMD: QueryAdjust [slot = 0]
Reply: New RN16
CMD: ACK [valid RN16]
Reply: See Table 6.14
CMD: ACK [valid RN16]
Reply: See Table 6.14
CMD: Req_RN [invalid RN16]
Reply: None
CMD: None within time T2
Action: Return to arbitrate
Reply: None.
Acknowleged
CMD: Req_RN [valid RN16] & (access password = 0}
Reply: handle
CMD: Req_RN [valid RN16] & (access password <> 0}
Reply: handle
CMD: ACK [valid handle]
Reply: See Table 6.14
CMD: Req_RN, Read, Write, Lock, BlockWrite/Erase/Permalock
Reply: See state-transition tables
CMD: Kill [valid handle & kill password = 0]
Reply: Error code
CMD: Kill [valid handle & valid nonzero kill password & {Recom <> 0}]
Reply: handle when done. Note 4
CMD: Kill, Access [invalid handle]
Reply: None
Open
CMD: Select
Action: Return to ready
Reply: None. Note 1
CMD: Query
Action: New round
Reply: Notes 2, 3
CMD: QueryRep, QueryAdjust
Action: Return to ready
Reply: None. Note 2
CMD: All other
Action: Return to arbitrate
Reply: None.
CMD: Access [valid handle & valid access password]
Reply: handle when done
Secured
CMD: ACK [valid handle]
Reply: See Table 6.14
CMD: Req_RN, Read, Write, Lock, BlockWrite/Erase/Permalock
Reply: See state-transition tables
CMD: Kill [valid handle & kill password = 0]
Reply: Error code
CMD: Kill [valid handle & valid nonzero kill password & {Recom <> 0}]
Reply: handle when done. Note 4
CMD: Access [valid handle & valid access password]
Reply: handle when done
CMD: Kill, Access [invalid handle]
Reply: None
CMD: Kill [valid handle & valid nonzero kill password & {Recom = 0}]
Reply: handle when done. Note 4
Power-up & killed
Killed
CMD: All
Reply: None
▨ Fully Supported State
▦ Partially Supported State
☐ Unsupported State

Figure 35: Supported Gen2 FSM structure.

5.4 PRESENT Validation & Decryption

PRESENT is a relatively new and unknown cipher when compared to other well known ciphers such as AES or DES. Because of this, there is little literature available for reference when designing a PRESENT engine. In order to properly insure our design meets the PRESENT specifications we performed a round by round manual functional validation using test vectors found at [AP10]. We used this same web address to validate the decryption engine found in Appendix L. The decryption engine was not intended to be synthesized and was only used in decrypting actual sensor data.

5.5 Synthesis Results & Discussion

Although modularity and ease of reuse were our primary goals while coding this work, we wanted to make sure that our design was at least on the same order of magnitude of the 5k GE expected in a Gen2 tag [PP09]. Therefore, we synthesized our design using Design Compiler A-2007.I2-SP4 from Synopsys and the UMC 65nm standard cell library. During compilation we used Compile Ultra and specified all clocks appropriately for each module. Fig. 36, breaks down the synthesis results.

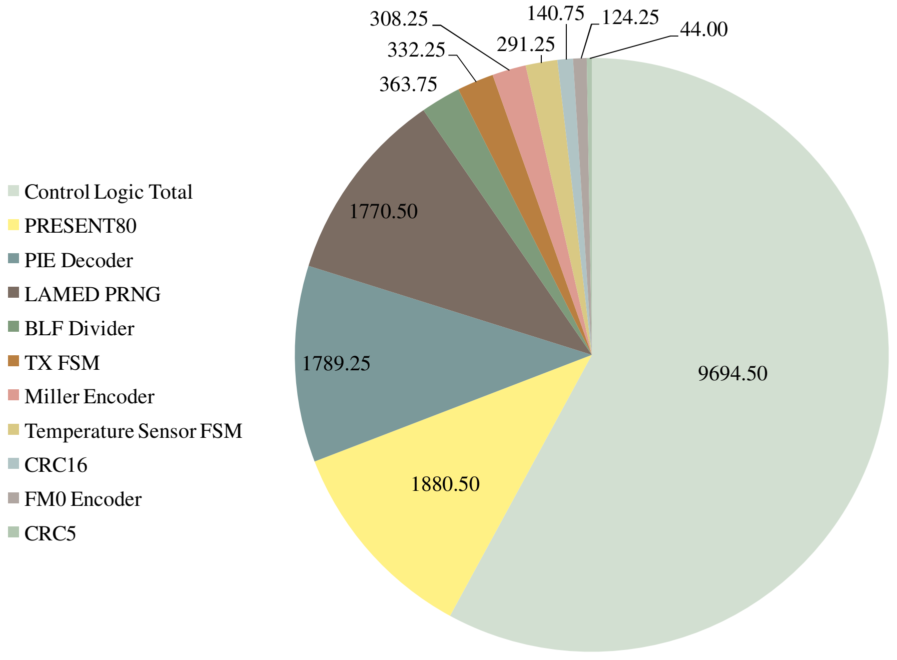


Figure 36: Gate equivalence synthesis results of our design.

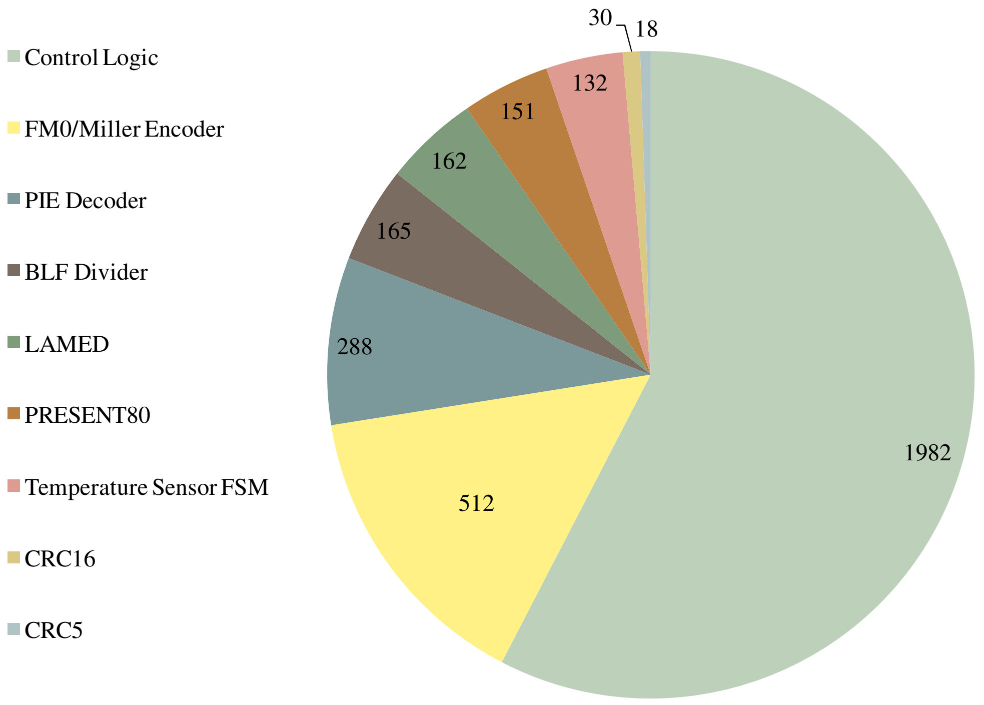


Figure 37: Thesis verilog lines of code breakdown.

From Fig. 36, we can show our design is approximately 14.7k GE without PRESENT or the Temperature Sensor FSM and 16.8k total. While this is approximately 3x the size of a Gen2 tag, we are on the same order of magnitude. Some of the extra logic can be attributed to the control logic needed to manage PRESENT and the sensor on top of the cipher and FSM itself, as well as redundant registers which were the result of the modularity built into the design. We believe that with significant effort put into optimizing the control logic and more tightly integrating the PIE decoder, the design could reach below the 10k GE mark.

Another indicator of where optimization effort should be placed is in the code size of the each module in the design. This is captured in Fig. 37, where, as expected, we can see a near 1 to 1 correlation between the code size and the synthesized area. The code size reported does not include comments or white space. All verilog code used in this thesis can be found in Appendices A-K.

5.6 Video Demonstration

We constructed a video demonstration of our platform performing in three distinct modes of operation. First we demonstrate basic ID queries with several different parameters. Next, we demonstrate unencrypted sensor data retrieval while we warm up the sensor to show the value increasing. Finally, we demonstrate encrypted sensor data retrieval and decryption. The timeline is as follows:

1. Introduction (42s)
2. ID query with multiple settings (1m 52s)
3. Unsecure sensor data query (1m 2s)
4. Encrypted sensor data query and decryption (1m 34s)

CHAPTER 6

CONCLUSIONS

6.1 Timeline

Table 9 outlines the timetable for this work. Initial ideas and specifications began in the summer of 2008 with most of the basic blocks simulated by summer of 2009. Early in 2010 we demonstrated a proof of concept for communication between the FPGA and RFID reader. By summer of 2010 the initial version of the code was complete and by the end of July the project and documentation were completed.

Table 9: Thesis project timeline.

Task	**Completion**
Initial project specs	Summer 2008
Ramp up on Gen2 protocol	Fall 2008
Ramp up on security and cryptography	Spring 2009
Verilog coding and synthesis of basic blocks	Summer 2009
Design security scheme	February 25, 2010
POC FPGA reader communication	March 2, 2010
Verilog coding and synthesis of basic Gen2 control logic	April 15, 2010
Verification and debug of basic Gen2 protocol with reader	May 15, 2010
Integrate temperature sensor	June 1, 2010
Integrate encryption/security protocol	June 15, 2010
Defend Thesis	July 6 & 7, 2010
Complete thesis document & code organization	August 10, 2010

6.2 Our Contributions

In this thesis, we were able to construct a platform capable of providing solutions and insight in the areas of RFID, security, and environmental sensing. The idea of low power, low cost, and secure environmental sensing is not in itself novel, but we were able to conceive a manner by which this could be accomplished through modifications to an existing solution to low power, low cost wireless identification. This work provides highly detailed information about the design, simulation, and prototyping such a platform.

Through our work, we have demonstrated the ability to decipher the information contained in the EPC Class 1 Generation 2 protocol and provide the basic components required to construct the digital backend of a modern RFID tag. We incorporated the work of several other researchers involving multiple clock domains and pseudo random number generation. We were also able to demonstrate two specific extensions to the basic components, a novel low power block cipher and environmental sensing both of which were added within the constraints of the current Gen2 protocol.

We were also able to prototype and validate the majority of our design with off-the-shelf 3rd party components to ensure compliance. We recognized that in undertaking such a large design from the ground up, detailed analysis would be required throughout the design process. We were able to devise a scheme to translate communications between the reader and tag into Verilog testbenches. This not only provided more in depth analysis, but reduced debug time and allowed us a higher degree of confidence as the complexity of the work increased.

6.3 Future Work

It is our belief that this work could serve as a stepping stone to several future works. A natural extension would involve ASIC synthesis (including layout) of the HDL code, which would allow studies in several areas including low power ASIC design, subthreshold RFID tags, and the effects of scaling on RFID tags. The modularity we intentionally built into our design will also allow for studies involving other ciphers such as AES, additional sensors, and optimizations to specific portions of the design.

Our work will also allow more vigorous studies into the future of the Gen2 protocol, most specifically into the addition of security, but one could imagine work studying the effects of different encoding schemes, pseudo random number generators, and memory. It could also be used by analog designers to study best practices in RFID analog and RF frontend design. With our work, future research can perform in depth timing, power, and area analysis in any of the aforementioned areas.

A long term application of our work could be inclusion into a hybrid RFID device involving an FPGA and microcontroller where the protocol or cryptographic portions could be handled by the FPGA and computation or environmental sensing could be performed by the microcontroller. As technology scales, perhaps such a device could even be passively powered. Such a project would involve analog designers, RTL designers, and computer scientists. This work would serve as an excellent foundation for such a project.

APPENDIX A

PRESENT ENCRYPTION VERILOG

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    10:39:07 06/04/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    present80
// Project Name:   M.S. Thesis
// Revision:                1.0
//
// Additional Comments: PRESENT block cipher with an 80 bit key size. Designed
// according to publication PRESENT: An Ultra-Lightweight Block Cipher (CHES 07)
////////////////////////////////////////////////////////////////////////////////

module present80(clk, enable, reset, plaintext, key, ciphertext, complete);
//Note: This circuit is only capable of performing the PRESENT encryption, not decryption
input clk;
input enable;
input reset;
input [63:0] plaintext;//Input text to cipher
input [79:0] key;//80 bit unique secret key
output [63:0] ciphertext;//Ciphertext output of cipher
output reg complete;//Flag used to indicate encryption/decryption is complete
reg [79:0] roundkey_in;//Round key before update
wire[79:0] roundkey_out;//Round key after update
wire [63:0] key_text_xor;//XOR of roundkey and round output
wire [63:0] sbox_out;//Result of sbox layer
wire [63:0] permute;//Result of permutation layer
reg [4:0] roundcounter;//Important: For this to work, must be non-saturating counter
reg [63:0] temp;//Used to store intermidiate round values
always@ (posedge clk) begin
if (reset) begin
roundkey_in     <= key;
temp            <= plaintext;
roundcounter    <= 1;
complete        <= 0;
end
else if (enable) begin
case (roundcounter)
default: begin
roundkey_in     <= roundkey_out;
roundcounter    <= roundcounter + 1;
temp                        <= permute;
end
0: begin
complete                    <= 1;
end
endcase
end
end
```

```
keyscheduler k0(.enable(enable), .key_in(roundkey_in), .roundkey(roundkey_out),
.roundcounter(roundcounter));
//sbox layer
sbox s0(key_text_xor[63:60], sbox_out[63:60]);
sbox s1(key_text_xor[59:56], sbox_out[59:56]);
sbox s2(key_text_xor[55:52], sbox_out[55:52]);
sbox s3(key_text_xor[51:48], sbox_out[51:48]);
sbox s4(key_text_xor[47:44], sbox_out[47:44]);
sbox s5(key_text_xor[43:40], sbox_out[43:40]);
sbox s6(key_text_xor[39:36], sbox_out[39:36]);
sbox s7(key_text_xor[35:32], sbox_out[35:32]);
sbox s8(key_text_xor[31:28], sbox_out[31:28]);
sbox s9(key_text_xor[27:24], sbox_out[27:24]);
sbox s10(key_text_xor[23:20], sbox_out[23:20]);
sbox s11(key_text_xor[19:16], sbox_out[19:16]);
sbox s12(key_text_xor[15:12], sbox_out[15:12]);
sbox s13(key_text_xor[11:08], sbox_out[11:08]);
sbox s14(key_text_xor[07:04], sbox_out[07:04]);
sbox s15(key_text_xor[03:00], sbox_out[03:00]);
//permutation layer
assign permute[0] = sbox_out[0];
assign permute[1] = sbox_out[4];
assign permute[2] = sbox_out[8];
assign permute[3] = sbox_out[12];
assign permute[4] = sbox_out[16];
assign permute[5] = sbox_out[20];
assign permute[6] = sbox_out[24];
assign permute[7] = sbox_out[28];
assign permute[8] = sbox_out[32];
assign permute[9] = sbox_out[36];
assign permute[10] = sbox_out[40];
assign permute[11] = sbox_out[44];
assign permute[12] = sbox_out[48];
assign permute[13] = sbox_out[52];
assign permute[14] = sbox_out[56];
assign permute[15] = sbox_out[60];
assign permute[16] = sbox_out[1];
assign permute[17] = sbox_out[5];
assign permute[18] = sbox_out[9];
assign permute[19] = sbox_out[13];
assign permute[20] = sbox_out[17];
assign permute[21] = sbox_out[21];
assign permute[22] = sbox_out[25];
assign permute[23] = sbox_out[29];
assign permute[24] = sbox_out[33];
assign permute[25] = sbox_out[37];
assign permute[26] = sbox_out[41];
assign permute[27] = sbox_out[45];
assign permute[28] = sbox_out[49];
assign permute[29] = sbox_out[53];
assign permute[30] = sbox_out[57];
assign permute[31] = sbox_out[61];
assign permute[32] = sbox_out[2];
assign permute[33] = sbox_out[6];
assign permute[34] = sbox_out[10];
assign permute[35] = sbox_out[14];
assign permute[36] = sbox_out[18];
assign permute[37] = sbox_out[22];
assign permute[38] = sbox_out[26];
assign permute[39] = sbox_out[30];
assign permute[40] = sbox_out[34];
assign permute[41] = sbox_out[38];
assign permute[42] = sbox_out[42];
assign permute[43] = sbox_out[46];
assign permute[44] = sbox_out[50];
assign permute[45] = sbox_out[54];
assign permute[46] = sbox_out[58];
assign permute[47] = sbox_out[62];
assign permute[48] = sbox_out[3];
assign permute[49] = sbox_out[7];
assign permute[50] = sbox_out[11];
assign permute[51] = sbox_out[15];
assign permute[52] = sbox_out[19];
assign permute[53] = sbox_out[23];
assign permute[54] = sbox_out[27];
assign permute[55] = sbox_out[31];
assign permute[56] = sbox_out[35];
assign permute[57] = sbox_out[39];
assign permute[58] = sbox_out[43];
assign permute[59] = sbox_out[47];
assign permute[60] = sbox_out[51];
assign permute[61] = sbox_out[55];
assign permute[62] = sbox_out[59];
assign permute[63] = sbox_out[63];
assign key_text_xor = roundkey_in[79:16] ^ temp;
assign ciphertext = key_text_xor;
endmodule

module keyscheduler(enable, key_in, roundkey, roundcounter);
//This is the key scheduling circuit for PRESENT80
//Output is the round key used at the start of each PRESENT round
input enable;
input [79:0] key_in;//Unique key
input [4:0] roundcounter;
//bits [79:16] of key_out are the actual round keys
output [79:0] roundkey;
wire [79:0] shiftout, nextkey;
//Sbox 4 MSB of shifted data
sbox keysbox(shiftout[79:76], nextkey[79:76]);
//perform the 18 bit barrel shift operation
assign shiftout[79:0] = {key_in[18:0], key_in[79:19]};
//XOR bits 15-19 with round counter
assign nextkey[19] = shiftout[19] ^ roundcounter[4];
assign nextkey[18] = shiftout[18] ^ roundcounter[3];
assign nextkey[17] = shiftout[17] ^ roundcounter[2];
assign nextkey[16] = shiftout[16] ^ roundcounter[1];
assign nextkey[15] = shiftout[15] ^ roundcounter[0];
assign nextkey[14:0] = shiftout[14:0];
assign nextkey[75:20] = shiftout[75:20];
assign roundkey = (enable) ? nextkey : 0;
endmodule
```

```
module sbox(sbox_in, sbox_out);
// 4 bit to 4 bit substitution box used in the PRESENT algorithm and key scheduler
input [3:0] sbox_in;
output [3:0] sbox_out;
wire in0, in1, in2, in3;
wire out0, out1, out2, out3;
assign in0 = sbox_in[3];
assign in1 = sbox_in[2];
assign in2 = sbox_in[1];
assign in3 = sbox_in[0];
//Simplified Sbox Equations
assign out0 = (!in0 & (!(in2 ^ in3) | in1 & in2)) | (in0 & !in1 & (in2 | in3));
assign out1 = (!in0 & (!in1 & (!in2 | !in3) | in1 & in2 & in3)) | (in0 & (!in1 & in2 & !in3 | !in2 & (in1 |
in3)));
assign out2 = (!in0 & (in2 & (!in1 | !in3))) | (in0 & (in1 & in3 | !in1 & (!in2 | !in3)));
assign out3 = (!in0 & (in3 & (!in1 | in2) | in1 & !in2 & !in3)) | (in0 & (!in3 & (!in1 | in2) | in1 & !in2 &
in3));
assign sbox_out = {out0, out1, out2, out3};
endmodule
```

APPENDIX B

LAMED PRNG VERILOG

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    10:00:00 03/08/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    lamed
// Project Name:   M.S. Thesis
// Revision:                  1.0
//
// Additional Comments: Pseudo random number generator based on published paper
// LAMED - A PRNG for EPC Class-1 Generation-2 RFID specification
//////////////////////////////////////////////////////////////////////////////////
module lamed(clk, enable, reset, key, iv, out, data_ready);
input clk;
input enable;
input reset;
input [31:0] key;//stored secret key used to initialized PRNG
input [31:0] iv;//stored initial vector also used to initialized PRNG
output [31:0] out;//32 bit prng output. 16 bit comes from bitwise XOR of 16 MSB and 16 LSB
output reg data_ready;//Flag to indicate PRNG complete
//These three registers are acted upon to generate final result
//Three primary operations are 32 bit XOR, 32 bit ADD, and variable length barrelshift
reg [31:0] aux1, aux2, aux3;
reg [3:0] state;//FSM state register
//state values
parameter LOAD          = 4'h0;
parameter S2            = 4'h1;
parameter SHIFT5        = 4'h2;
parameter SHIFT3_1      = 4'h3;
parameter SHIFT4_1      = 4'h4;
parameter SHIFT2_1      = 4'h5;
parameter SHIFT2_2      = 4'h6;
parameter SHIFT3_2      = 4'h7;
parameter SHIFT2_3      = 4'h8;
parameter SHIFT4_2      = 4'h9;
parameter SHIFT1        = 4'hA;
parameter SHIFT2_4      = 4'hB;
parameter READY         = 4'hC;
reg [2:0] counter;//counter used to track how many bit positions we have moved during a barrelshift
operation
always @ (posedge clk) begin
if (reset) begin
state           <= LOAD;
data_ready      <= 0;
counter         <= 0;
end
else if (enable) begin
case (state)
LOAD: begin
```

```
aux1 <= key + iv;
aux2 <= key ^ iv;
data_ready <= 0;
state <= S2;
end//LOAD
S2: begin
aux3      <= aux1;
counter   <= 0;//reset counter value
state     <= SHIFT5;
end//S2
SHIFT5: begin//5 bit barrelshift & Add
if (counter < 5) begin
counter            <= counter + 1'b1;
aux3[31:0]         <= {aux3[30:0],aux3[31]};
end
else begin
state     <= SHIFT3_1;
aux3      <= aux3 + aux2;
counter   <= 0;
end
end//SHIFT5
SHIFT3_1: begin//3 bit barrelshift & XOR
if (counter < 3) begin
counter            <= counter + 1'b1;
aux3[31:0]         <= {aux3[30:0],aux3[31]};
end
else begin
state     <= SHIFT4_1;
aux3      <= aux3 ^ aux1;
counter   <= 0;
end
end//SHIFT3_1
SHIFT4_1: begin//4 bit barrelshift & Add
if (counter < 4) begin
counter            <= counter + 1'b1;
aux3[31:0]         <= {aux3[30:0],aux3[31]};
end
else begin
state     <= SHIFT2_1;
aux3      <= aux3 + iv;
counter   <= 0;
end
end//SHIFT4_1
SHIFT2_1: begin//2 bit barrelshift & Add
if (counter < 2) begin
counter            <= counter + 1'b1;
aux3[31:0]         <= {aux3[30:0],aux3[31]};
end
else begin
state     <= SHIFT2_2;
aux3      <= aux3 + aux1;
counter   <= 0;
end
end//SHIFT2_1
SHIFT2_2: begin//2 bit barrelshift & XOR
if (counter < 2) begin
```

```
counter         <= counter + 1'b1;
aux3[31:0]      <= {aux3[30:0],aux3[31]};
end
else begin
state    <= SHIFT3_2;
aux3     <= aux3 ^ aux1;
counter  <= 0;
end
end//SHIFT2_2
SHIFT3_2: begin//3 bit barrelshift & Add
if (counter < 3) begin
counter         <= counter + 1'b1;
aux3[31:0]      <= {aux3[30:0],aux3[31]};
end
else begin
state    <= SHIFT2_3;
aux3     <= aux3 + iv;
counter  <= 0;
end
end
SHIFT2_3: begin//2 bit barrelshift & Add
if (counter < 2) begin
counter         <= counter + 1'b1;
aux3[31:0]      <= {aux3[30:0],aux3[31]};
end
else begin
state    <= SHIFT4_2;
aux3     <= aux3 + aux1;
counter  <= 0;
end
end//SHIFT2_3
SHIFT4_2: begin//4 bit barrelshift & XOR
if (counter < 4) begin
counter <= counter + 1'b1;
aux3[31:0] <= {aux3[30:0],aux3[31]};
end
else begin
state    <= SHIFT1;
aux3     <= aux3 ^ iv;
counter  <= 0;
end
end//SHIFT4_2
SHIFT1: begin//1 bit barrelshift & Add
if (counter < 1) begin
counter         <= counter + 1'b1;
aux3[31:0]      <= {aux3[30:0],aux3[31]};
end
else begin
state    <= SHIFT2_4;
aux3     <= aux3 + aux2;
counter  <= 3'b000;
end
end//SHIFT1
SHIFT2_4: begin//2 bit barrelshift & Add
if (counter < 2) begin
counter         <= counter + 1'b1;
```

```
aux3[31:0]          <= {aux3[30:0],aux3[31]};
end
else begin
state       <= READY;
aux3        <= aux3 ^ aux1;
end
end//SHIFT2_4
READY: begin//Raise ready flag and hold value
if (!data_ready) begin
data_ready <= 1;
end
state <= READY;
end//READY
endcase
end
end
assign out = (enable) ? aux3 : 0;//aux3 contains final value
endmodule
```

APPENDIX C

CRC16 VERILOG

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    15:52:35 05/17/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    crc16
// Project Name:   M.S. Thesis
// Revision:                        1.0
//
// Additional Comments: 16 bit cyclic redundancy check from Appendix F of EPC
// Class-1 Generation 2 RFID Protocol
////////////////////////////////////////////////////////////////////////////////
module crc16(clk, enable, reset, data_in, q);
input reset;//Used to preload FFFF before clocking data
input data_in;//Data should be shifted in on the posedge of clk
output reg [15:0] q;//When calculating CRC16 (not checking) Q is actually !Q
always @ (negedge clk) begin
if (reset) begin
q <= 16'hFFFF;
end
else if (enable) begin
q[0] <= data_in ^ q[15];
q[1] <= q[0];
q[2] <= q[1];
q[3] <= q[2];
q[4] <= q[3];
q[5] <= (data_in ^ q[15]) ^ q[4];
q[6] <= q[5];
q[7] <= q[6];
q[8] <= q[7];
q[9] <= q[8];
q[10] <= q[9];
q[11] <= q[10];
q[12] <= (data_in ^ q[15]) ^ q[11];
q[13] <= q[12];
q[14] <= q[13];
q[15] <= q[14];
end
end
endmodule
```

APPENDIX D

CRC5 VERILOG

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    09:26:00 04/27/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    crc5
// Project Name:   M.S. Thesis
// Revision:                1.0
//
// Additional Comments: 5 bit cyclic redundency check circuit conforming to
// Appendix F of the EPC Class 1 Generation 2 Protocol
////////////////////////////////////////////////////////////////////////////////
module crc5(clk, enable, reset, data_in, q);
input clk;//Digital logic clock
input enable;
input reset;//Used to preset shift register with 01001
input data_in;
output reg [4:0] q;
always @ (posedge clk or posedge reset) begin
if (reset) begin
q <= 5'b01001;//Preset shift register
end
else if (enable) begin
q[0] <= data_in ^ q[4];
q[1] <= q[0];
q[2] <= q[1];
q[3] <= q[2] ^ (data_in ^ q[4]);
q[4] <= q[3];
end
end
endmodule
```

APPENDIX E

BACKSCATTER CLOCK DIVIDER VERILOG

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    08/25/2009
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    blf_divider
// Project Name:   M.S. Thesis
// Revision:                1.0
//
// Additional Comments: Backscatter Link Frequency Divider
//
//////////////////////////////////////////////////////////////////////////////////
module blf_divider(enable, clk, dr, trcal, blf_clk, reset);
input clk;//2.56MHz sample clock, divided to create the backscatter clk
input reset;
input enable;
input dr;//DR value received by FSM, 0 --> DR=8, 1 --> DR=64/3
input [10:0] trcal;//Tag to Reader calibration symbol
output reg blf_clk;//BLF = DR/trcal
reg [4:0] clk_counter;//64 samples is the largest clk divider, so 32 per half clk
reg [4:0] div_val;//used to store the clock divider value based on the trcal value
always @* begin
if (!dr) begin //DR = 8
if (trcal <= 10'b0000110110) begin //54
div_val = 5'h03;//426.6kHz
end
else if (trcal <= 10'b0001000111) begin //71
div_val = 5'h04;//320kHz
end
else if (trcal <= 10'b0001010111) begin //87
div_val = 5'h05;//256kHz
end
else if (trcal <= 10'b0001100111) begin //103
div_val = 5'h06;//213.3kHz
end
else if (trcal <= 10'b0001110111) begin //119
div_val = 5'h07;//182.9kHz
end
else if (trcal <= 10'b0010000111) begin //135
div_val = 5'h08;//160kHz
end
else if (trcal <= 10'b0010010111) begin //151
div_val = 5'h09;//142.2kHz
end
else if (trcal <= 10'b0010100111) begin //167
div_val = 5'h0a;//128kHz
end
else if (trcal <= 10'b0010110111) begin //183
div_val = 5'h0b;//116.4kHz
```

```
end
else if (trcal <= 10'b0011000111) begin //199
div_val = 5'h0c;//106.6kHz
end
else if (trcal <= 10'b0011010111) begin //215
div_val = 5'h0d;//98.5kHz
end
else if (trcal <= 10'b0011100111) begin //231
div_val = 5'h0e;//91.4kHz
end
else if (trcal <= 10'b0011110111) begin //247
div_val = 5'h0f;//85.3kHz
end
else if (trcal <= 10'b0100000111) begin //263
div_val = 5'h10;//80kHz
end
else if (trcal <= 10'b0100010111) begin //279
div_val = 5'h11;//75.3kHz
end
else if (trcal <= 10'b0100100111) begin //295
div_val = 5'h12;//71.1kHz
end
else if (trcal <= 10'b0100110111) begin //311
div_val = 5'h13;//67.4kHz
end
else if (trcal <= 10'b0101000111) begin //327
div_val = 5'h14;//64kHz
end
else if (trcal <= 10'b0101010111) begin //343
div_val = 5'h15;//61.0kHz
end
else if (trcal <= 10'b0101100111) begin //359
div_val = 5'h16;//58.2kHz
end
else if (trcal <= 10'b0101110111) begin //375
div_val = 5'h17;//55.7kHz
end
else if (trcal <= 10'b0110000111) begin //391
div_val = 5'h18;//53.3kHz
end
else if (trcal <= 10'b0110010111) begin //407
div_val = 5'h19;//51.2kHz
end
else if (trcal <= 10'b0110100111) begin //423
div_val = 5'h1a;//49.2kHz
end
else if (trcal <= 10'b0110110111) begin //439
div_val = 5'h1b;//47.4kHz
end
else if (trcal <= 10'b0111000111) begin //455
div_val = 5'h1c;//45.7kHz
end
else if (trcal <= 10'b0111010111) begin //471
div_val = 5'h1d;//44.1kHz
end
else if (trcal <= 10'b0111100111) begin //487
```

```
div_val = 5'h1e;//42.7kHz
end
else if (trcal <= 10'b0111110111) begin //503
div_val = 5'h1f;//41.3kHz
end
else if (trcal <= 10'b1000000000) begin //512
div_val = 5'h20;//40kHz
end
end//DR = 8
else begin //DR = 64/3
if (trcal <= 10'b0001100110) begin //102
div_val = 5'h02;//640kHz
end
else if (trcal <= 10'b0010010010) begin //146
div_val = 5'h03;//426.6kHz
end
else if (trcal <= 10'b0010111101) begin //189
div_val = 5'h04;//320kHz
end
else if (trcal <= 10'b0011101000) begin //232
div_val = 5'h05;//256kHz
end
else if (trcal <= 10'b0100010011) begin //275
div_val = 5'h06;//213.3kHz
end
else if (trcal <= 10'b0100111110) begin //318
div_val = 5'h07;//182.9kHz
end
else if (trcal <= 10'b0101101001) begin //361
div_val = 5'h08;//160kHz
end
else if (trcal <= 10'b0110010100) begin //404
div_val = 5'h09;//142.2kHz
end
else if (trcal <= 10'b0110111110) begin //446
div_val = 5'h0a;//128kHz
end
else if (trcal <= 10'b0111101001) begin //489
div_val = 5'h0b;//116.4kHz
end
else if (trcal <= 10'b1000010100) begin //532
div_val = 5'h0c;//106.6kHz
end
else if (trcal <= 10'b1000111111) begin //575
div_val = 5'h0d;//98.5kHz
end
else if (trcal <= 10'b1001000000) begin //576
div_val = 5'h0e;//91.4kHz
end
end//DR = 64/3
end //always
always @ (posedge clk) begin//NOTE, the 2.56 MHz clk MUST be activated before the enable signal.
if (reset) begin
clk_counter <= 0;//reset the counter
end
else if (enable) begin
```

```
if (clk_counter < div_val) begin
clk_counter <= clk_counter + 1'b1;
blf_clk          <= blf_clk;
end
else begin
clk_counter <= 5'b00001;
blf_clk          <= !blf_clk;
end
end
else blf_clk              <= 0;
end
endmodule
```

APPENDIX F

PIE DECODER VERILOG

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    14:23:25 03/30/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    pie_decoder
// Project Name:   M.S. Thesis
// Revision:                    1.0
//
// Additional Comments: PIE Decoding FSM, with output flags indicating when both
// opcode and data have been received
////////////////////////////////////////////////////////////////////////////////
module pie_decoder_new(clk256, data_in, enable, reset, op_code_ready, op_code, data_ready, data,
query_received, rtcal, trcal, data_bits);
input clk256;//2.56 MHz Clock used to sample incoming data
input data_in;//Demodualted data input
//When an instruction is received, the op code ready flag will signal the FSM that a command has been
recieved
//and will place the instruction opcode into the op_code register while it continues to sample the incoming
data
//When the data from a reader command has been read, the data_ready flag will signal
//FIXME This might need to be further divided to give the FSM more time
//FIXME may also need to tell the FSM how many bits of data have been received
output reg query_received;
reg [3:0] state;
output reg [7:0] rtcal;//Reader --> Tag calibration symbol, Gen2 section 6.3.1.2.11
//Used to sample RTcal & Delimiter max value of RTCal is 75us or 192 samples @ 2.56MHz
reg [11:0] counter;
reg [6:0] pivot;//1/2 RTcal , all symbols longer = data1, shorter = data0
output reg [10:0] trcal;//Tag --> Reader calibration symbol, Gen2 section 6.3.1.2.11
//counter to track the number of data bits received, output used for variable length data
output reg [6:0] data_bits;
reg [11:0] timeout;//Counter used to reset decoder after a certain amount of time in WAIT state to avoid
endless looping
//State machine state parameters
parameter READY             = 4'h0;
parameter DELIMITER         = 4'h1;
parameter INITIALDATA0      = 4'h2;
parameter INITIALPW         = 4'h3;
parameter SAMPLERTCAL       = 4'h4;
parameter INITIALPW2        = 4'h5;
parameter DECIDETRCAL       = 4'h6;
parameter DECIDEOPCODE      = 4'h7;
parameter OPCODEPW          = 4'h8;
parameter DATADECODE        = 4'h9;
parameter DATAPW            = 4'hA;
parameter WAIT              = 4'hB;
//Total number of commands supported
parameter NUMCMDS = 4'd11;
parameter MAX_OPCODE_LENGTH = 5'd16;
//max num of bits to represent length of opcode
//ln(MAX_OPCODE_LENGTH)/ln(2)
parameter OPCODE_LENGTH = 3'd4;
//ln(Max fixed length command - opcode length)/ln(2)
parameter MAX_DATA_LENGTH = 3'd6;
parameter CMD_TABLE_WORD = MAX_OPCODE_LENGTH + OPCODE_LENGTH + 1 +
MAX_DATA_LENGTH;
//Opcodes from Gen2 Table 6.18
parameter QUERYREP          = 2'b00;
parameter ACK               = 2'b01;
parameter QUERY             = 4'b1000;
parameter QUERYADJUST       = 4'b1001;
parameter SELECT            = 4'b1010;
parameter NAK               = 8'b11000000;
parameter REQ_RN            = 8'b11000001;
parameter READ              = 8'b11000010;
parameter WRITE             = 8'b11000011;
parameter KILL              = 8'b11000100;
parameter LOCK              = 8'b11000101;
parameter ACCESS            = 8'b11000110;
//This 2d array will contain the opcode, opcode length, whether it is fixed length, how many bits of data if it
is fixed length
reg  [CMD_TABLE_WORD - 1:0] cmdreg [NUMCMDS-1 : 0];
//Used to count the number of bits in the opcode
reg [4:0] op_code_counter;
//Counter
integer i;
always @ (posedge clk256) begin
if (reset) begin
state                       <= READY;
op_code_ready               <= 0;
data_ready                  <= 0;
op_code                     <= 0;
data                        <= 0;
counter                     <= 0;
query_received              <= 0;
data_bits                   <= 0;
op_code_counter             <= 0;
timeout                     <= 0;
rtcal                       <= 0;
//Command Information Array
//Array contains all supported Opcodes and the length of the data
//16 Bits for Opcode, 4 bits for opcode length, 1 bit for fixed/non fixed length
//6 bits for data length (if fixed)
cmdreg[0] <= {14'h0,     QUERYREP,        4'd2, 1'b1, 6'd02};
cmdreg[1] <= {14'h0,     ACK,             4'd2, 1'b1, 6'd16};
cmdreg[2] <= {12'h0,     QUERY,           4'd4, 1'b1, 6'd18};
cmdreg[3] <= {12'h0,     QUERYADJUST,     4'd4, 1'b1, 6'd5};
cmdreg[4] <= {12'h0,     SELECT,          4'd4, 1'b0, 6'd0};
cmdreg[5] <= {8'h0,      NAK,             4'd8, 1'b1, 6'd0};
cmdreg[6] <= {8'h0,      REQ_RN,          4'd8, 1'b1, 6'd32};
cmdreg[7] <= {8'h0,      READ,            4'd8, 1'b0, 6'd0};
cmdreg[8] <= {8'h0,      WRITE,           4'd8, 1'b0, 6'd0};
cmdreg[9] <= {8'h0,      KILL,            4'd8, 1'b1, 6'd52};
cmdreg[10] <= {8'h0,     ACCESS,          4'd8, 1'b1, 6'd48};
```

```
end
else if (enable) begin
case (state)
READY: begin//Reset counters and prepare for incoming data
if (data_in) begin
state                   <= READY;
op_code_ready           <= 0;
data_ready              <= 0;
op_code                 <= 0;
data                    <= 0;
counter                 <= 0;
query_received          <= 0;
data_bits               <= 0;
op_code_counter         <= 0;
timeout                 <= 0;
rtcal                   <= 0;
end
else begin//When data in = 0 begin sampling delimiter
state <= DELIMITER;
counter <= counter + 1;
end
end //READY
DELIMITER: begin
//Delimiter of 12.5 us +/- 5% means between 30 and 34 samples is a valid delimiter (at 2.56MHz)
if (counter > 50) begin//FIXME, RFID reader generates long delimter values
state <= READY;
end
else if (!data_in) begin
state <= DELIMITER;
counter <= counter + 1;
end
else if (data_in) begin
if (counter < 20) begin//FIXME counter value too small technically, but RFID reader generating small
demlimiter
state <= READY;
end
else begin
state <= INITIALDATA0;
end
end
end //DELIMITER
INITIALDATA0: begin//all commands begin with data0
if (data_in) begin
state <= INITIALDATA0;
end
else begin
state <= INITIALPW;
end
end //INITIALDATA0
INITIALPW: begin//First PW value, could check length is correct
if (!data_in) begin
state <= INITIALPW;
counter <= 0;
end
else begin
state <= SAMPLERTCAL;
```

```
counter <= counter + 1;
end
end //INITIALPW
SAMPLERTCAL: begin//RTcal always follows initial data0
if (data_in) begin
state <= SAMPLERTCAL;
counter <= counter + 1;
end
else begin
state <= INITIALPW2;
pivot <= counter / 2;
trcal <= 0;//reset TRcal counter
rtcal <= counter;//store RTcal
end
end //SAMPLERTCAL
INITIALPW2: begin//Pw after RTcal symbol
if (!data_in) begin
state <= INITIALPW2;
counter <= counter + 1;
end
else begin
state <= DECIDETRCAL;
trcal <= trcal + 1;
op_code_counter <= 0;
end
end //INITIALPW2
DECIDETRCAL: begin
//At this point we could either be receiving a command or a TRcal value
//If we do get a TRcal value then we know we have a Query command
if (data_in) begin
trcal <= trcal + 1;
if (trcal < 8*pivot) begin//Timeout
state <= DECIDETRCAL;
end
else begin
state <= READY;
end
end
else begin
if (trcal < pivot) begin //We got a Data0
op_code[15:0] <= {op_code[14:0], 1'b0};
op_code_counter <= op_code_counter + 1;
end
else begin //Don't know if its Data1 or TRCal
if (trcal < rtcal) begin //TRcal is always greater than RTcal
//We got a Data1
op_code[15:0] <= {op_code[14:0], 1'b1};
op_code_counter <= op_code_counter + 1;
end
end
state <= OPCODEPW;
counter <= 0;
end
end //DECIDETRCAL
OPCODEPW: begin
//Check the cmdreg table to determine if a valid opcode has been received
```

```
for (i = 0; i < NUMCMDS; i = i + 1) begin
if (op_code == cmdreg[i][26:11] & op_code_counter == cmdreg[i][10:7]) begin
op_code_ready <= 1;
end
end
if (!data_in) begin
state <= OPCODEPW;
counter <= 0;
end
else begin
if (!op_code_ready) begin
state <= DECIDEOPCODE;
end
else begin
state <= DATADECODE;
end
counter <= counter + 1;
end
end //OPCODEPW
DECIDEOPCODE: begin
if (data_in) begin
counter <= counter + 1;
if (counter < 8*pivot) begin
state <= DECIDEOPCODE;
end
else begin
state <= READY;//Timeout
end
end
else begin
if (counter < pivot) begin
//Data0 received
op_code[15:0] <= {op_code[14:0], 1'b0};
op_code_counter <= op_code_counter + 1;
state <= OPCODEPW;
end
else if (counter > pivot) begin
//Data1 received
op_code[15:0] <= {op_code[14:0], 1'b1};
op_code_counter <= op_code_counter + 1;
state <= OPCODEPW;
end
end
end //DECIDEOPCODE
DATADECODE: begin
if (data_in) begin
if (counter < rtcal) begin
state <= DATADECODE;
counter <= counter + 1;
end
else begin
//long symbol means we have received all the data
//should only happen for non fixed length data
data_ready <= 1;
state <= WAIT;
end
```

```
end
else begin
if (counter < pivot) begin
//Data0 received
data[79:0] <= {data[78:0], 1'b0};
end
else if (counter > pivot) begin
//Data1 received
data[79:0] <= {data[78:0], 1'b1};
end
state <= DATAPW;
counter <= 0;
data_bits <= data_bits + 1;
end
end //DATADECODE
DATAPW: begin
//Based on the opcode we can tell by the number of data bits if we have received all the data
//Only works for fixed length instructions
for(i = 0; i < NUMCMDS; i = i + 1) begin
if (data_bits == cmdreg[i][5:0] & op_code == cmdreg[i][26:11] & cmdreg[i][6] == 1) begin
data_ready          <= 1;
timeout             <= 0;
end
//Need to raise flag faster on a READ command
else if (op_code == 16'h00C2) begin
case (data_bits)
//EBV formatting dictates how long the READ command is
//We will only be supporting 50 bit read commands
50: begin
if (data[47] == 0) begin
data_ready          <= 1;
timeout             <= 0;
end
end
endcase
end
end
if (!data_in) begin
state <= DATAPW;
end
else begin
if (data_ready) begin
state <= WAIT;
end
else begin
state <= DATADECODE;
counter <= counter + 1;
end
end
end //DATAPW
WAIT: begin
//THIS state will hold the output until the decoder is reset after all data has been received
//Or timeout = 6*RTcal
if (timeout < 12*pivot) begin
state <= WAIT;
timeout <= timeout + 1;
```

```
end
else begin
state <= READY;//Timeout has occured
end
end //WAIT
endcase //state
end
end
endmodule
```

APPENDIX G

TPRI VERILOG

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    10:22:52 05/24/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    tpri_logic
// Project Name:   M.S. Thesis
// Revision:                        1.0
//
// Additional Comments: This module takes in DR and TRcal and outputs Tpri. Tpri
// is used in calculation of T1
//////////////////////////////////////////////////////////////////////////////////
module tpri_logic(dr, trcal_value, tpri);
//For more information on T1 see Gen2 Figure 6.13 Link Timing Parameters
//Discrete values of tpri
//tpri = value / Dig Clk
//Keep in mind that trcal_value is sampled with the sample (NOT DIGITAL) clk
always @ (dr, trcal_value) begin
case (dr)
0: begin//DR=8
if (trcal_value < 56) begin
tpri <= 3;
end
else if (trcal_value < 72) begin
tpri <= 4;
end
else if (trcal_value < 88) begin
tpri <= 5;
end
else if (trcal_value < 104) begin
tpri <= 6;
end
else if (trcal_value < 120) begin
tpri <= 7;
end
else if (trcal_value < 136) begin
tpri <= 8;
end
else if (trcal_value < 152) begin
tpri <= 9;
end
else if (trcal_value < 168) begin
tpri <= 10;
end
else if (trcal_value < 184) begin
tpri <= 11;
end
else if (trcal_value < 200) begin
tpri <= 12;
```

```
end
else if (trcal_value < 216) begin
tpri <= 13;
end
else if (trcal_value < 232) begin
tpri <= 14;
end
else if (trcal_value < 248) begin
tpri <= 15;
end
else if (trcal_value < 264) begin
tpri <= 16;
end
else if (trcal_value < 280) begin
tpri <= 17;
end
else if (trcal_value < 296) begin
tpri <= 18;
end
else if (trcal_value < 312) begin
tpri <= 19;
end
else if (trcal_value < 328) begin
tpri <= 20;
end
else if (trcal_value < 344) begin
tpri <= 21;
end
else if (trcal_value < 360) begin
tpri <= 22;
end
else if (trcal_value < 376) begin
tpri <= 23;
end
else if (trcal_value < 392) begin
tpri <= 24;
end
else if (trcal_value < 408) begin
tpri <= 25;
end
else if (trcal_value < 424) begin
tpri <= 26;
end
else if (trcal_value < 440) begin
tpri <= 27;
end
else if (trcal_value < 456) begin
tpri <= 28;
end
else if (trcal_value < 472) begin
tpri <= 29;
end
else if (trcal_value < 488) begin
tpri <= 30;
end
else if (trcal_value < 504) begin
```

```
tpri <= 31;
end
else if (trcal_value < 520) begin//Actual highest should only be 513
tpri <= 32;
end
end
1: begin//DR=64/3
if (trcal_value < 107) begin
tpri <= 2;
end
else if (trcal_value < 150) begin
tpri <= 3;
end
else if (trcal_value < 192) begin
tpri <= 4;
end
else if (trcal_value < 235) begin
tpri <= 5;
end
else if (trcal_value < 278) begin
tpri <= 6;
end
else if (trcal_value < 320) begin
tpri <= 7;
end
else if (trcal_value < 363) begin
tpri <= 8;
end
else if (trcal_value < 406) begin
tpri <= 9;
end
else if (trcal_value < 448) begin
tpri <= 10;
end
else if (trcal_value < 491) begin
tpri <= 11;
end
else if (trcal_value < 534) begin
tpri <= 12;
end
else if (trcal_value < 580) begin
tpri <= 13;
end
end
default: tpri <= 0;
endcase
end
endmodule
```

APPENDIX H

DATA TRANSMISSION VERILOG

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    13:25:38 05/11/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    tx_fsm
// Project Name:   M.S. Thesis
// Revision:                   1.0
//
// Additional Comments: Circuit designed to select the correct encoder and data
// when transmitting data to RFID reader
////////////////////////////////////////////////////////////////////////////////
module tx_fsm(blf_clk, enable, reset, num_bits, trext, tx_data, tx_complete, data_out, m);
input enable;
input [7:0] num_bits;//Number of data bits to be transmitted
input [1:0] m;//m value from Gen2 6.3.1.3.2.3 used to select encoding type
output tx_complete;//Flag indiating data has been transmitted
output data_out;//output on transmit pin
reg encoder_data_in;//Input to encoders, output of MUX of input data, selected by counter
//encoder reset and enable
reg fm0_reset, miller_reset;
wire fm0_enable, miller_enable;
wire fm0_preamble, miller_preamble;
//encoder outputs
wire fm0_out, miller_out;
//used to track number of clock cycles for different miller encoding types
wire [2:0] miller_clk_count;
reg [7:0] i;//counter to select data to transmit
//Output data Mux 129 to 1
8'h08: encoder_data_in = tx_data[8];
8'h09: encoder_data_in = tx_data[9];
8'h0a: encoder_data_in = tx_data[10];
8'h0b: encoder_data_in = tx_data[11];
8'h0c: encoder_data_in = tx_data[12];
8'h0d: encoder_data_in = tx_data[13];
8'h0e: encoder_data_in = tx_data[14];
8'h0f: encoder_data_in = tx_data[15];
8'h10: encoder_data_in = tx_data[16];
8'h11: encoder_data_in = tx_data[17];
8'h12: encoder_data_in = tx_data[18];
8'h13: encoder_data_in = tx_data[19];
8'h14: encoder_data_in = tx_data[20];
8'h15: encoder_data_in = tx_data[21];
8'h16: encoder_data_in = tx_data[22];
8'h17: encoder_data_in = tx_data[23];
8'h18: encoder_data_in = tx_data[24];
8'h19: encoder_data_in = tx_data[25];
8'h1a: encoder_data_in = tx_data[26];
8'h1b: encoder_data_in = tx_data[27];
```

```
8'h1c: encoder_data_in = tx_data[28];
8'h1d: encoder_data_in = tx_data[29];
8'h1e: encoder_data_in = tx_data[30];
8'h1f: encoder_data_in = tx_data[31];
8'h20: encoder_data_in = tx_data[32];
8'h21: encoder_data_in = tx_data[33];
8'h22: encoder_data_in = tx_data[34];
8'h23: encoder_data_in = tx_data[35];
8'h24: encoder_data_in = tx_data[36];
8'h25: encoder_data_in = tx_data[37];
8'h26: encoder_data_in = tx_data[38];
8'h27: encoder_data_in = tx_data[39];
8'h28: encoder_data_in = tx_data[40];
8'h29: encoder_data_in = tx_data[41];
8'h2a: encoder_data_in = tx_data[42];
8'h2b: encoder_data_in = tx_data[43];
8'h2c: encoder_data_in = tx_data[44];
8'h2d: encoder_data_in = tx_data[45];
8'h2e: encoder_data_in = tx_data[46];
8'h2f: encoder_data_in = tx_data[47];
8'h30: encoder_data_in = tx_data[48];
8'h31: encoder_data_in = tx_data[49];
8'h32: encoder_data_in = tx_data[50];
8'h33: encoder_data_in = tx_data[51];
8'h34: encoder_data_in = tx_data[52];
8'h35: encoder_data_in = tx_data[53];
8'h36: encoder_data_in = tx_data[54];
8'h37: encoder_data_in = tx_data[55];
8'h38: encoder_data_in = tx_data[56];
8'h39: encoder_data_in = tx_data[57];
8'h3a: encoder_data_in = tx_data[58];
8'h3b: encoder_data_in = tx_data[59];
8'h3c: encoder_data_in = tx_data[60];
8'h3d: encoder_data_in = tx_data[61];
8'h3e: encoder_data_in = tx_data[62];
8'h3f: encoder_data_in = tx_data[63];
8'h40: encoder_data_in = tx_data[64];
8'h41: encoder_data_in = tx_data[65];
8'h42: encoder_data_in = tx_data[66];
8'h43: encoder_data_in = tx_data[67];
8'h44: encoder_data_in = tx_data[68];
8'h45: encoder_data_in = tx_data[69];
8'h46: encoder_data_in = tx_data[70];
8'h47: encoder_data_in = tx_data[71];
8'h48: encoder_data_in = tx_data[72];
8'h49: encoder_data_in = tx_data[73];
8'h4a: encoder_data_in = tx_data[74];
8'h4b: encoder_data_in = tx_data[75];
8'h4c: encoder_data_in = tx_data[76];
8'h4d: encoder_data_in = tx_data[77];
8'h4e: encoder_data_in = tx_data[78];
8'h4f: encoder_data_in = tx_data[79];
8'h50: encoder_data_in = tx_data[80];
8'h51: encoder_data_in = tx_data[81];
8'h52: encoder_data_in = tx_data[82];
8'h53: encoder_data_in = tx_data[83];
```

```
8'h54: encoder_data_in = tx_data[84];
8'h55: encoder_data_in = tx_data[85];
8'h56: encoder_data_in = tx_data[86];
8'h57: encoder_data_in = tx_data[87];
8'h58: encoder_data_in = tx_data[88];
8'h59: encoder_data_in = tx_data[89];
8'h5a: encoder_data_in = tx_data[90];
8'h5b: encoder_data_in = tx_data[91];
8'h5c: encoder_data_in = tx_data[92];
8'h5d: encoder_data_in = tx_data[93];
8'h5e: encoder_data_in = tx_data[94];
8'h5f: encoder_data_in = tx_data[95];
8'h60: encoder_data_in = tx_data[96];
8'h61: encoder_data_in = tx_data[97];
8'h62: encoder_data_in = tx_data[98];
8'h63: encoder_data_in = tx_data[99];
8'h64: encoder_data_in = tx_data[100];
8'h65: encoder_data_in = tx_data[101];
8'h66: encoder_data_in = tx_data[102];
8'h67: encoder_data_in = tx_data[103];
8'h68: encoder_data_in = tx_data[104];
8'h69: encoder_data_in = tx_data[105];
8'h6a: encoder_data_in = tx_data[106];
8'h6b: encoder_data_in = tx_data[107];
8'h6c: encoder_data_in = tx_data[108];
8'h6d: encoder_data_in = tx_data[109];
8'h6e: encoder_data_in = tx_data[110];
8'h6f: encoder_data_in = tx_data[111];
8'h70: encoder_data_in = tx_data[112];
8'h71: encoder_data_in = tx_data[113];
8'h72: encoder_data_in = tx_data[114];
8'h73: encoder_data_in = tx_data[115];
8'h74: encoder_data_in = tx_data[116];
8'h75: encoder_data_in = tx_data[117];
8'h76: encoder_data_in = tx_data[118];
8'h77: encoder_data_in = tx_data[119];
8'h78: encoder_data_in = tx_data[120];
8'h79: encoder_data_in = tx_data[121];
8'h7a: encoder_data_in = tx_data[122];
8'h7b: encoder_data_in = tx_data[123];
8'h7c: encoder_data_in = tx_data[124];
8'h7d: encoder_data_in = tx_data[125];
8'h7e: encoder_data_in = tx_data[126];
8'h7f: encoder_data_in = tx_data[127];
8'h80: encoder_data_in = tx_data[128];
default: encoder_data_in = 1'b0;
end
//foward reset to encoders
always @* begin
if (reset) begin
fm0_reset               <= 1;
miller_reset      <= 1;
end
else begin
fm0_reset               <= 0;
miller_reset      <= 0;
```

```
end
end
//Counter is incremented and therefore data is shifted to encoders
always @ (negedge blf_clk or posedge reset) begin
if (reset) begin
i           <= 0;
end
else if (enable & !tx_complete) begin
//Data is shift into the encoders on the negative edge of the clk
if (fm0_preamble | miller_preamble) begin
case (m)
2'b00: begin
if (i < num_bits) begin
i <= i + 1;
end
end
2'b01: begin
if (i < num_bits & miller_clk_count == 1) begin
i <= i + 1;
end
end
2'b10: begin
if (i < num_bits & miller_clk_count == 3) begin
i <= i + 1;
end
end
2'b11: begin
if (i < num_bits & miller_clk_count == 7) begin
i <= i + 1;
end
end
endcase
end
else begin
i <= 0;
end
end
end
fm0_encoder       fm0encoder( .enable(fm0_enable),
.reset(fm0_reset),
.blf_clk(blf_clk),
.trext(trext),
.data_in(encoder_data_in),
.data_out(fm0_out),
.preamble_complete(fm0_preamble)
);
miller_encoder millerencoder(
.enable(miller_enable),
.reset(miller_reset),
.data_in(encoder_data_in),
.data_out(miller_out),
.m(m),
.blf_clk(blf_clk),
.trext(trext),
.preamble_complete(miller_preamble),
.clk_count(miller_clk_count)
);
assign tx_complete = (reset) ? 1'b0 : (enable & i == num_bits & (!data_out) & (miller_enable |
fm0_enable)) ? 1'b1 : 1'b0;
//Drive output to 0 if reset, or select the correct encoder output
assign data_out = (reset) ? 1'b0 : (fm0_enable) ? fm0_out : (miller_enable) ? miller_out : 1'b0;
//enable encoders based on m value and enable signal
assign fm0_enable = (enable) ? (m == 2'b00) ? 1'b1 : 1'b0 : 1'b0;
assign miller_enable = (enable) ? (m == 2'b01 | m == 2'b10 | m == 2'b11) ? 1'b1 : 1'b0 : 1'b0;
endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    11:07:16 05/17/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    miller_encoder
// Project Name:   M.S. Thesis
// Revision:                        1.0
//
// Additional Comments: Miller encoder in compliance with Gen2 section 6.3.1.3.2.3
//
////////////////////////////////////////////////////////////////////////////////
module miller_encoder(enable, reset, data_in, data_out, m, blf_clk, trext, preamble_complete, clk_count);
input enable;
input reset;
input blf_clk;//Output clk of the Backscatter link frequency generator
//Data to be shifted out of the miller encoder, max of 129 bits in this version
//Final bit is always 1
input data_in;
//Miller sub carrier value (2, 4, or 8)
//01 = 2, 10 = 4, 11 = 8 (Gen 2 Table 6.21)
input [1:0] m;
input trext;//TRext value 0 = short preamble, 1 = long preamble
output data_out;//Resulting encoded output
output reg [2:0] clk_count;//Count clk ticks to determine when to change phase by counting clk ticks
reg [2:0] m_count;//Tells encoder how many clk ticks/bit based on m value
output reg preamble_complete;//Flag indicating preamble is complete

//FSM States:
//1. PREAMBLE_INIT when 16 or 4 sets of 0 w/o phase change are transmitted
//2. DATA_TX where first 010111 then actual data are transmitted
reg state;
parameter PREAMBLE_INIT = 1'b0;
parameter DATA_TX = 1'b1;
wire blf_clk_n;
//Signal indicating the intitial part of the preamble is complete
reg preamble_init_complete;
//Register that stores constant value of 010111 sent during preamble
reg [5:0] preamble;
//used to track the 6 bits 010111 in the preamble
reg [2:0] preamble_six_count;
//count the number of preamble bits transmitted
reg [7:0] preamble_count;
//XOR of these values used to generate output capable of changing signal phase
```

```
reg data1, data2;
//Used to track the last bit transmitted
reg prev_bit;
//Used to ensure data transmission begins on posedge of clk
reg start;
//Actual data about to be transmitted
wire tx_data;
always @ (posedge blf_clk or posedge reset) begin
if (reset) begin
data1 <= 0;
end
else if (enable & start) begin
case (state)
PREAMBLE_INIT: begin
//Always toggle during preamble initial phase
data1 <= !data1;
end
DATA_TX: begin
//data will always toggle unless prev_bit & data_in == 0,
//or if transmitting a 1 on a specific clk tick determined by m value
if (clk_count == 0 & !prev_bit & !tx_data) begin
data1 <= data1;
end
else begin
case (m)
2'b01: begin
if (clk_count == 1 & tx_data) begin
data1 <= data1;
end
else begin
data1 <= !data1;
end
end
2'b10: begin
if (clk_count == 2 & tx_data) begin
data1 <= data1;
end
else begin
data1 <= !data1;
end
end
2'b11: begin
if (clk_count == 4 & tx_data) begin
data1 <= data1;
end
else begin
data1 <= !data1;
end
end
default: data1 <= 0;
endcase
end
end
endcase
end
end
```

```
always @ (posedge blf_clk_n or posedge reset) begin
if (reset) begin
data2 <= 0;
start <= 0;
end
else if (start) begin
data2 <= !data2;
end
else if (enable) begin
start <= 1;
end
end
//Tracks number of clk ticks based on m value
always @ (posedge blf_clk_n or posedge reset) begin
if (reset) begin
clk_count <= 0;
preamble <= 6'b010111;
prev_bit <= 1;
end
else if (enable & start & preamble_init_complete) begin
case (m)
2'b01: begin
if(clk_count < 1) begin
clk_count <= clk_count + 1;
end
else begin
clk_count <= 0;
//Previous bit is used to calculate when to change phase
if (preamble_complete) begin
prev_bit <= tx_data;
end
else begin
preamble <= preamble << 1;
end
end
end
2'b10: begin
if(clk_count < 3) begin
clk_count <= clk_count + 1;
end
else begin
clk_count <= 0;
//Previous bit is used to calculate when to change phase
if (preamble_complete) begin
prev_bit <= tx_data;
end
else begin
preamble <= preamble << 1;
end
end
end
2'b11: begin
if(clk_count < 7) begin
clk_count <= clk_count + 1;
end
else begin
```

```
clk_count <= 0;
//Previous bit is used to calculate when to change phase
if (preamble_complete) begin
prev_bit <= tx_data;
end
else begin
preamble <= preamble << 1;
end
end
end
endcase
end
end
always @ (posedge blf_clk_n or posedge reset) begin
if (reset) begin
preamble_six_count <= 0;
preamble_complete <= 0;
end
else if (preamble_init_complete) begin
if (clk_count == 0) begin
if (preamble_six_count < 6) begin
preamble_six_count <= preamble_six_count + 1;
end
else begin
preamble_complete <= 1;
end
end
end
end
always @ (posedge data_out or posedge reset) begin
if (reset) begin
preamble_count <= 0;
end
else if (enable & state == PREAMBLE_INIT) begin
preamble_count <= preamble_count + 1;
end
end
always @ (posedge data_out or posedge reset) begin
if (reset) begin
state <= PREAMBLE_INIT;
preamble_init_complete <= 0;
end
//16*m_value - 1 = number of data_out ticks (long preamble)
else if (trext) begin
case (m)
2'b01: begin
if (preamble_count == 32) begin
preamble_init_complete <= 1;
state <= DATA_TX;
end
end
2'b10: begin
if (preamble_count == 64) begin
preamble_init_complete <= 1;
state <= DATA_TX;
end
```

```
end
2'b11: begin
if (preamble_count == 128) begin
preamble_init_complete <= 1;
state <= DATA_TX;
end
end
endcase
end
//4*m_value = number of data_out ticks (short preamble)
else begin
case (m)
2'b01: begin
if (preamble_count == 8) begin
preamble_init_complete <= 1;
state <= DATA_TX;
end
end
2'b10: begin
if (preamble_count == 16) begin
preamble_init_complete <= 1;
state <= DATA_TX;
end
end
2'b11: begin
if (preamble_count == 32) begin
preamble_init_complete <= 1;
state <= DATA_TX;
end
end
endcase
end
end
assign tx_data = (preamble_complete) ? (data_in) : (preamble[5]);
assign blf_clk_n = !blf_clk;
assign data_out = data1 ^ data2;
endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    13:52:37 08/27/2009
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    fm0_encoder
// Project Name:   M.S. Thesis
// Revision:                         1.0
//
// Additional Comments: FM0 encoder in compliance with Gen2 section 6.3.1.3.2.2
//
////////////////////////////////////////////////////////////////////////////////
module fm0_encoder(enable, blf_clk, trext, data_in, data_out, preamble_complete, reset);
input blf_clk;//Backscatter Link Freqency
input enable;
input reset;
```

```
//Bit used to decided on which preamble (extended or not) from (6.3.1.3.2.2)
//1 = extended, 0 = short
input trext;
input data_in;//Data to be encoded, shifted in serially
output data_out;//Encoded data output
//Indicates preamble is complete, do not want to shift data in until preamble is complete
output reg preamble_complete;
wire blf_clk_n;//Used for FM0 biphase sampling
reg [4:0] preamble_counter;//Used to track how many preamble bits have been completed
reg data_in_buf;
reg data1, data2;//Xor of these values creates data out
reg state;//Simple 1 bit state machine to differentiate between preamble and data transmit
parameter PREAMBLE = 0;
parameter DATA_TX = 1;
//used to ensure that posedge of blf clk start data transfer
reg start;
always @ (posedge blf_clk or posedge reset) begin
if (reset) begin
data1 <= 0;
start <= 0;
end
else if (enable) begin
//always toggle on posedge of sample clock except during 1 bit of preamble
if (trext & preamble_counter == 16 | !trext & preamble_counter == 4) begin
data1 <= data1;
end
else data1 <= !data1;
//shift in data
data_in_buf <= data_in;
if (!start) start <= 1;
end
end
always @ (posedge blf_clk_n or posedge reset) begin
if (reset) begin
data2 <= 0;
preamble_complete <= 0;
preamble_counter <= 0;
state <= PREAMBLE;
end
else if (enable & start) begin
case (state)
//0 is the preamble state
PREAMBLE: begin
if (!trext) begin
//Short Preamble
case (preamble_counter)
0: data2 <= data2;
1: data2 <= !data2;
2: data2 <= data2;
3: data2 <= !data2;
4: data2 <= data2;
5: begin
data2 <= data2;
preamble_complete <= 1;
state <= DATA_TX;
end
```

```
endcase
end
else begin
//Long Preamble
case (preamble_counter)
12: data2 <= data2;
14: data2 <= data2;
16: data2 <= data2;
17: begin
data2 <= data2;
preamble_complete <= 1;
state <= DATA_TX;
end
default: data2 <= !data2;
endcase
end
//Be sure to always reset the encoder or the counter value will be incorrect
preamble_counter <= preamble_counter + 1;
end//PREAMBLE
DATA_TX: begin
//Toggle if transmitting a 0
if (!data_in_buf) begin
data2 <= !data2;
end
else data2 <= data2;
end//DATA_TX
endcase
end
end//always
assign blf_clk_n = !blf_clk;
assign data_out = data1 ^ data2;
endmodule
```

APPENDIX I

PRIMARY FSM VERILOG

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    16:07:30 04/01/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    primary_fsm
// Project Name:   M.S. Thesis
// Revision:                    1.0
//
// Additional Comments: Primary FSM for Class 1 Generation 2 EPC RFID Tag with
// several modifications such as an external temperature sensor
//////////////////////////////////////////////////////////////////////////////////
module primary_fsm(sensor_rst_n, sensor_clk_conv, sensor_dq, por, demod_data, clk128, clk256,
clk256_enable, receive_enable, transmit, mem_reset);
input por;//Power on Reset
input demod_data;//Demodulated data input from Analog Frontend
input clk128;//1.28 MHz clock which drives most of the digital logic
input clk256;//2.56 MHz clock used for PIE decoder, BLF divider, & temperature sensor
input mem_reset;//Want to be independant of FSM
output reg clk256_enable;//2.56 MHz Sample & backscatter clock enable
output reg receive_enable;//Enables WISP to receive data
output transmit;
//Memory register control signals
reg mem_enable;//Allow read/write to memory
reg mem_write_enable;//1 = mem write, 0 = mem read
reg [1:0] mem_bank;//Memory bank (Figure 6.17)
reg [15:0] mem_in;//Memory input on a write
reg [3:0] mem_word;//Word # being read/written to memory
reg [7:0] mem_max;//Last word # to be read/written to in memory
reg [2:0] mem_count;//# of words read/written to memory
wire [15:0] mem_out;//Memory output on a read
//PIE decoder signals
reg pie_enable;
reg pie_reset;
wire [79:0] pie_data;//Max of 80 bits of data from each reader command
wire pie_data_ready;//Flag indicating command data has been decoded
wire [15:0] pie_opcode;//Reader command opcode
wire pie_opcode_ready;//Flag indicating reader command opcode has been decoded
wire pie_query_received;
wire [6:0] pie_data_bits;//Commands can be variable length, tracks number of bits received
//LAMED PRNG Control Signals
reg prng_enable;
reg prng_reset;
reg [31:0] prng_key, prng_iv;
reg [15:0] prng_16_out;//16 bit PRNG value, result of XOR between 16 MSB and 16 LSB of 32 bit output
reg n;//Indicates odd or even round of PRN generation
reg [31:0] a0, a1;
wire prng_ready;//Flag indicating PRNG has completed generating a new result
wire [31:0] prng_32_out;//32 bit output of PRNG
reg [15:0] prng_16_prev;//Previous value of prng used to XOR with password on Access Command
//CRC5 Control Signals
reg crc5_reset;
reg crc5_enable;
reg [21:0] crc5_data;
reg crc5_valid;
wire [4:0] crc5_result;
reg [5:0] crc5_counter;
//CRC16 Control Signals
reg crc16_reset;
reg crc16_enable;
reg [6:0] crc16_count;//Used to count clk cycles when computing crc16
reg [111:0] crc16_data_in;//Preload this reg with all data to be sent to crc16 [111] = MSB
wire [15:0] crc16_out;
reg crc16_valid;
//Temperature Sensor Control Signals
reg sensor_reset;
reg sensor_enable;
wire sensor_complete;
wire [8:0] sensor_data;
output sensor_rst_n;
output sensor_clk_conv;
inout sensor_dq;
//PRESENT Block Cipher Control Signals
reg present_enable;
reg present_reset;
reg [63:0] present_plaintext;
reg [79:0] present_key;
wire present_complete;
wire [63:0] present_ciphertext;
//Various Control Related Signals
reg [15:0] curr_cmd;//Last received opcode, required because PIE decoder resets
reg [15:0] slot_counter;//Tag position in response Queue, Gen2 section 6.3.2.4
reg slot_counter_ready;//Flag indicating slot counter has been generated
reg epc_retrieved;//Flag used to indicated we have retrieved EPCID during powerup, so we can move on to
retrieving the PRESENT key
wire[10:0] trcal;//Tag --> Reader calibration symbol, Gen2 section 6.3.1.2.8
reg [10:0] trcal_value;
wire[7:0] rtcal;//Reader --> Tag calibration symbol, Gen2 section 6.3.1.2.8
reg [7:0] rtcal_value;
reg dr;//Divide ratio, Gen2 section 6.3.1.2.8
reg [1:0] m;//M value selects encoding type (FM0, Miller2, Miller4, Miller8)
reg trext;//Select between long and short preamble
reg [31:0] access_password;//32 bit access password, arrives from 2 16 bit ACCESS commands
(6.3.2.11.3.6)
reg access_req;//Indicates 1st of 2 access commands have been recieved
//Backscatter control signals
reg blf_enable;//Signal to enable the backscatter clock
wire blf_clk;//Backscatter clock signal, Gen2 section 6.3.1.3.5
reg blf_reset;
reg [128:0] data_out;//Data to be transmitted to reader
reg [7:0] num_tx_bits;//Number of data bits to be transmitted to the reader
reg tx_enable;
reg tx_reset;
wire tx_complete;//Flag indicating data transmission complete
```

```
//In reply state, after RN16 is backscattered, raise this flag. Indicates that we are waiting for an ACK (or
other command)
reg waiting_ack;
reg t1_passed;//Flag indicating that t1 min reponse delay has been satisfied, Gen2 table 6.13
reg [7:0] t1_count;//Clock cycles since last command was decoded, used to calculate T1
wire [5:0] tpri;//TRcal/DR, backscatter-link pulse-repetition interval
//StoredPC is defined in section 6.3.2.1.2.2 in Gen2. It is used to indicate the length of EPCID,
//as well as several other paramaters
reg [15:0] stored_pc;
reg [15:0] stored_crc;//Gen2 section 6.3.2.1.2.1
reg [95:0] epc_id;
//Used to verify Read/Write Commands
reg [15:0] handle;
//If no state changes for a certain amount of time, assume frozen and reset
//When all bugs fixed, should not be needed
reg [15:0] global_reset_count;
reg global_reset_sig;
reg [3:0] prev_state;
integer i;//counter, used to load data in for loops
reg [3:0] state;//FSM state register (Figure 6.19)
//Gen2 state values
parameter POWERUP               = 4'h0;
parameter READY                         = 4'h1;
parameter ARBITRATE             = 4'h2;
parameter REPLY                         = 4'h3;
parameter ACKNOWLEDGED          = 4'h4;
parameter OPEN                  = 4'h5;
parameter SECURED               = 4'h6;
parameter KILLED                        = 4'h7;
//Opcodes from Gen2 Table 6.18
parameter QUERYREP              = 16'h0000;
parameter ACK                           = 16'h0001;
parameter QUERY                         = 16'h0008;
parameter QUERYADJUST           = 16'h0009;
parameter SELECT                        = 16'h000A;
parameter NAK                           = 16'h00C0;
parameter REQ_RN                        = 16'h00C1;
parameter READ                          = 16'h00C2;
parameter WRITE                         = 16'h00C3;
parameter KILL                  = 16'h00C4;
parameter LOCK                          = 16'h00C5;
parameter ACCESS                        = 16'h00C6;
always@(posedge clk128) begin
if (!global_reset_sig) begin
//FIXME, need to verify not killed
state                                   <= POWERUP;
pie_reset                               <= 1;
//Reset PRNG Controls
prng_enable                     <= 0;
prng_reset                              <= 1;
n                                                       <= 0;
slot_counter                    <= 0;
slot_counter_ready      <= 0;
crc5_reset                              <= 0;
crc5_enable                     <= 0;
receive_enable          <= 0;
```

```
//Encoder Controls
num_tx_bits                     <= 0;
tx_enable                               <= 0;
waiting_ack                             <= 0;
//Reset Memory Controls
mem_enable                              <= 1;
mem_write_enable                <= 0;
mem_in                                  <= 0;
mem_word                                <= 0;
crc16_valid                             <= 0;
//Temperature Sensor Controls
sensor_reset                    <= 1;
sensor_enable                   <= 0;
//PRESENT Block Cipher Controls
present_reset                   <= 1;
present_enable                  <= 0;
epc_id                                  <= 0;
access_req                              <= 0;
//access_password               <= 0;
epc_retrieved                   <= 0;
end
else begin
case (state)
POWERUP: begin
prng_reset              <= 1;
prng_enable     <= 0;
crc5_reset              <= 0;
crc5_enable     <= 0;
//FIXME need to make sure not killed
pie_reset               <= 1;
pie_enable              <= 0;
clk256_enable   <= 0;
receive_enable <= 0;
blf_enable              <= 0;
blf_reset               <= 1;
//Encoder Controls
num_tx_bits     <= 0;
waiting_ack             <= 0;
if (!sensor_complete) begin
//Activate Temperature Sensor
sensor_reset                    <= 0;
sensor_enable                   <= 1;
end
else begin
sensor_enable                   <= 0;
end
if (!epc_retrieved) begin
//Retreive EPCID from memory
//I am simplifing this process. Please see section 6.3.2.1.2.2 for proper implementation of storedpc & epcid
//After data has been retrieved, calculate and store StoredCRC
case (mem_word)
0: begin
mem_enable                      <= 1;
mem_bank                        <= 2'b01;//EPC Bank
mem_write_enable        <= 0;//Read Mem
mem_word <= mem_word + 1;//StoredPC in word 1 of EPC Bank
```

```
end
1: begin
stored_pc <= mem_out;
mem_word <= mem_word + 1;//epcid is stored in words 2-6 of EPC Bank
end
2: begin
epc_id[15:0] <= mem_out;
mem_word <= mem_word + 1;
end
3: begin
epc_id[31:16] <= mem_out;
mem_word <= mem_word + 1;
end
4: begin
epc_id[47:32] <= mem_out;
mem_word <= mem_word + 1;
end
5: begin
epc_id[63:48] <= mem_out;
mem_word <= mem_word + 1;
end
6: begin
epc_id[79:64] <= mem_out;
mem_word <= mem_word + 1;
//Reset CRC16 to calculate StoredCRC
crc16_reset <= 1;
end
7: begin
epc_id[95:80] <= mem_out;
crc16_reset         <= 0;
crc16_count         <= 111;//should always be #bits -1
mem_word <= mem_word + 1;
end
8: begin
//Load CRC16 DataIn
for (i = 0; i < 96; i = i + 1) begin
crc16_data_in[i] <= epc_id[i];
end
for (i = 0; i < 16; i = i + 1) begin
crc16_data_in[i+96] <= stored_pc[i];
end
mem_word <= mem_word + 1;
crc16_enable        <= 1;
end
9: begin
//Calculate StoredCRC according to Appendix F
if (crc16_count > 0) begin
crc16_data_in <= crc16_data_in << 1;
crc16_count <= crc16_count - 1;
end
else begin
//Inverse of Q = StoredCRC
//Technically this should be stored in EPC Bank Word 0
stored_crc[0] <= !crc16_out[0];
stored_crc[1] <= !crc16_out[1];
stored_crc[2] <= !crc16_out[2];
```

```
stored_crc[3] <= !crc16_out[3];
stored_crc[4] <= !crc16_out[4];
stored_crc[5] <= !crc16_out[5];
stored_crc[6] <= !crc16_out[6];
stored_crc[7] <= !crc16_out[7];
stored_crc[8] <= !crc16_out[8];
stored_crc[9] <= !crc16_out[9];
stored_crc[10] <= !crc16_out[10];
stored_crc[11] <= !crc16_out[11];
stored_crc[12] <= !crc16_out[12];
stored_crc[13] <= !crc16_out[13];
stored_crc[14] <= !crc16_out[14];
stored_crc[15] <= !crc16_out[15];
crc16_enable <= 0;
epc_retrieved <= 1'b1;
mem_word <= 0;
mem_enable <= 0;
end
end
default: begin
state <= POWERUP;
end
endcase
end
else begin
//EPCID has been read from memory, now fetch PRESENT key
case (mem_word)
0: begin
mem_enable                  <= 1;
mem_bank                    <= 2'b00;//RES bank
mem_write_enable      <= 0;//Read Mem
mem_word                    <= 8;//PRESENT key is stored in words 8-12
end
8: begin
present_key[15:00]    <= mem_out;
mem_word                         <= mem_word + 1;
end
9: begin
present_key[31:16]    <= mem_out;
mem_word                         <= mem_word + 1;
end
10: begin
present_key[47:32]    <= mem_out;
mem_word                         <= mem_word + 1;
end
11: begin
present_key[63:48]    <= mem_out;
mem_word                         <= mem_word + 1;
end
12: begin
present_key[79:64]    <= mem_out;
mem_word                         <= mem_word + 1;
end
13: begin
mem_word                         <= 0;
mem_enable                       <= 0;
```

```
state                                   <= READY;
end
endcase
end
end //POWERUP
READY: begin
//Activate Decoder
if (pie_reset) begin
pie_reset               <= 0;
pie_enable              <= 1;
clk256_enable     <= 1;
receive_enable <= 1;
end
else if (pie_opcode_ready) begin
case (curr_cmd)
QUERY: begin
state                                   <= ARBITRATE;
slot_counter_ready      <= 0;
mem_enable                              <= 0;
end
default: begin
//Don't reset until the entire command has been received
if (pie_data_ready) begin
state <= READY;
pie_reset <= 1;//FIXME SELECT Command should have a different behavior
end
end
endcase
end
//Fetch seed, iv, a0, & a1 values from memory and activate prng
if (!prng_ready) begin
if (!mem_enable) begin
//Setup Memory Bank for Read
mem_bank                        <= 2'b11;//User Bank
mem_write_enable        <= 0;//Read Mem
mem_word                                <= 0;
mem_enable                      <= 1;
end
else if (!prng_enable) begin
//Fetch Seed and IV values for PRNG
case (mem_word)
0: begin
prng_key[31:16]  <= mem_out;
mem_word                        <= mem_word + 1;
end
1: begin
prng_key[15:0]   <= mem_out;
mem_word                        <= mem_word + 1;
end
2: begin
prng_iv[31:16]   <= mem_out;
mem_word                        <= mem_word + 1;
end
3: begin
prng_iv[15:0]           <= mem_out;
mem_word                        <= mem_word + 1;
```

```
end
4: begin
a0[31:16]                    <= mem_out;
mem_word                     <= mem_word + 1;
end
5: begin
a0[15:0]                     <= mem_out;
mem_word                     <= mem_word + 1;
end
6: begin
a1[31:16]                    <= mem_out;
mem_word                     <= mem_word + 1;
end
7: begin
a1[15:0]                     <= mem_out;
mem_word                     <= mem_word + 1;
end
8: begin
//Activate PRNG
prng_reset              <= 0;
prng_enable        <= 1;
mem_word                <= 2;
end
endcase
end
end
//prng ready
else begin
//Store PRN16, and store a0 & a1 in memory
if (mem_enable) begin
prng_enable <= 0;
//Store Updated a0 & a1, see LAMED algorithm for more information
case (mem_word)
2: begin
//Part of LAMED algorithm, 16 bit output = XOR of 32 bit MSB & LSB
prng_16_out <= prng_32_out[31:16] ^ prng_32_out[15:0];
//Update a0 & a1
if (!n) begin
//Even N
a0 <= a1 ^ prng_iv;
a1 <= prng_32_out + prng_key;
end
else begin
//Odd N
a0 <= a1 + prng_iv;
a1 <= prng_32_out + prng_key;
end
mem_word                     <= mem_word + 1;
n                                     <= !n;
mem_bank                     <= 2'b11;//User Bank
end
3: begin
mem_write_enable        <= 1;//Write Mem
mem_in                       <= a0[31:16];
mem_word                     <= mem_word + 1;
end
```

```
4: begin
mem_in                          <= a0[15:0];
mem_word                        <= mem_word + 1;
end
5: begin
mem_in                          <= a1[31:16];
mem_word                        <= mem_word + 1;
end
6: begin
mem_in                          <= a1[15:0];
mem_word                        <= mem_word + 1;
end
7: begin
mem_enable                      <= 0;
mem_bank                        <= 2'b01;
end
endcase
end
end
end//READY
ARBITRATE: begin
//Upon Entering Arbitrate (meaning we received a Query Command), the following must be done:
//Wait for PIE Decoder to finish gather data (pie_data_ready flag)
//Wait for PRNG to finish
//Get Query data (DR, M, Q), includes performing CRC5 check
//Generate slot counter
//After these have been done, reset the PIE decoder and wait for the next command
//Wait for decoding of current command
if (!pie_data_ready) begin
state <= ARBITRATE;
//This indicates that we have reset the decoder after receiving a Query Command
if (pie_reset) begin
pie_reset <= 0;
end
end
else if (slot_counter_ready & slot_counter == 0) begin
//Slot counter = 0 means reply
state                       <= REPLY;
//Setup transmit FSM
for(i = 0; i < 16; i = i + 1) begin
data_out[i] <= prng_16_out[15 - i];
end
data_out[16]        <= 1;//In both encoding, last symbol is always 1
num_tx_bits         <= 17;
blf_enable                  <= 1;
blf_reset                   <= 0;
tx_reset            <= 1;
waiting_ack                  <= 0;
slot_counter_ready          <= 0;
pie_reset <= 1;
crc5_reset <= 1;
end
else begin
//QUERY command data contains several useful values including DR, M, & Q
case (curr_cmd)
QUERY: begin
```

```
//Perform a CRC5 Check
if (!crc5_reset & !crc5_enable) begin
crc5_reset        <= 1;
end
else if (crc5_reset & !crc5_enable) begin
crc5_reset        <= 0;
crc5_enable <= 1;
end
else if (crc5_enable & !crc5_valid) begin
//FIXME, this should timeout based on T2
state <= ARBITRATE;
end
//Means we got a valid CRC for the Query Command
else if (crc5_valid) begin
if (!slot_counter_ready) begin
//Q value bits will determine slot counter value
case (pie_data[8:5])
0: slot_counter   <= {15'h0000, prng_16_out[0]};
1: slot_counter   <= {14'h0000, prng_16_out[1:0]};
2: slot_counter   <= {13'h0000, prng_16_out[2:0]};
3: slot_counter   <= {12'h000, prng_16_out[3:0]};
4: slot_counter   <= {11'h000, prng_16_out[4:0]};
5: slot_counter   <= {10'h000, prng_16_out[5:0]};
6: slot_counter   <= {9'h000, prng_16_out[6:0]};
7: slot_counter   <= {8'h00, prng_16_out[7:0]};
8: slot_counter   <= {7'h00, prng_16_out[8:0]};
9: slot_counter   <= {6'h00, prng_16_out[9:0]};
10: slot_counter  <= {5'h00, prng_16_out[10:0]};
11: slot_counter  <= {4'h0, prng_16_out[11:0]};
12: slot_counter  <= {3'h0, prng_16_out[12:0]};
13: slot_counter  <= {2'h0, prng_16_out[13:0]};
14: slot_counter  <= {1'h0, prng_16_out[14:0]};
15: slot_counter  <= prng_16_out[15:0];
endcase
dr        <= pie_data[17];
m                 <= pie_data[16:15];
trext <= pie_data[14];
//Sel, Session, & Target bits have not been accounted for. At present we are assuming we are always being
communicated with
slot_counter_ready <= 1;
trcal_value <= trcal;
crc5_enable <= 0;//CRC5 passed, disable CRC5
crc5_reset <= 1;
end
end
//With each QUERY command we will store the value from the temperature sensor in the RES memory
bank word 4
if (sensor_complete) begin
if (!mem_enable) begin
mem_bank          <= 2'b00;//Reserved bank
mem_word          <= 4;
mem_in            <= {5'h00, sensor_data};//FIXME, data may be backwards
mem_enable        <= 1;
end
else begin
mem_enable                    <= 0;
```

```
sensor_reset        <= 1;//Lower sensor_complete flag
sensor_enable       <= 0;
end
end
else begin
end
end//QUERY
QUERYREP: begin
slot_counter <= slot_counter - 1;
pie_reset <= 1; //Reset the PIE decoder
end
QUERYADJUST: begin
slot_counter <= slot_counter - 1;
pie_reset <= 1; //Reset the PIE decoder
end
SELECT: begin
state <= READY;
pie_reset <= 1; //Reset the PIE decoder
end
default: begin
//All other commands just return to arbitrate
state <= ARBITRATE;
pie_reset <= 1;
end
endcase
end
end//ARBITRATE
REPLY: begin
//Indicates we have backscattered RN16 and are waiting for ACK (or other command)
if (waiting_ack) begin
//Any operations that can be executed by only knowing the opcode is placed here (such as PRN generation)
if (pie_opcode_ready) begin
case (curr_cmd)
ACK: begin
//Must wait for data to be received to validate the RN16 value
//Reset TX FSM
state               <= REPLY;
end
QUERY: begin
//Receiving a Query Cmd starts a new round, generate new RN16 & Slot Value
//Since handling this is already built into the Arbitrate command, we will return there
//FIXME, generate a new RN16 value **COULD PROBABLY JUST BUILD THIS INTO THE
PIE_OPCODE_READY SINCE QUERY AND QUERYADJUST ALWAYS DO IT***
state                           <= ARBITRATE;
waiting_ack         <= 0;
slot_counter_ready <= 0;
end
QUERYADJUST: begin
//FIXME, generate a new RN16 value
state                           <= REPLY; //FIXME, should generate a new rn16 value, change q and if slot =
0 then backscatter rn16, if not return to reply
waiting_ack         <= 0;
end
default: begin
//All other commands should send the tag to ARBITRATE
state <= ARBITRATE;
```

```
end
endcase
end
if (pie_data_ready) begin
//Means we must have received an ACK command, so we need to verify the RN16 value
if (pie_data[15:0] == prng_16_out) begin
//Valid RN16, backscatter StoredPC, EPCID, and StoredCRC, then transition to ACK state
if (!tx_complete) begin
if (tx_reset & !tx_enable) begin
num_tx_bits <= 128;
blf_enable              <= 1;
blf_reset               <= 0;
//Load StoredPC, EPCID, & StoredCRC for transmission
for(i = 0; i < 16; i = i + 1) begin
data_out[i] <= stored_pc[15 - i];
end
for(i = 0; i < 96; i = i + 1) begin
data_out[i+16] <= epc_id[95 - i];
end
for(i = 0; i < 16; i = i + 1) begin
data_out[i+112] <= stored_crc[15 - i];
end
//Last bit always 1
data_out[128] <= 1'b1;
tx_reset          <= 0;
end
else if (t1_passed) begin
tx_enable <= 1;//begin transmission after t1
pie_enable <= 0;//disable receive during transmission
end
state <= REPLY;
end
else begin
tx_reset  <= 1;
tx_enable         <= 0;
pie_reset <= 1;
//Data has been backscattered, transition state
state             <= ACKNOWLEDGED;
prng_reset        <= 1;
crc16_valid <= 0;
end
end
else begin
//Invalid RN16
state             <= ARBITRATE;
end
end
end
else begin
//Cannot Reply before T1, indicated by t1_passed flag
if (tx_reset & !tx_enable) begin
//Reset and enable the transmit FSM
tx_reset <= 0;
pie_reset <= 0;
end
else if (tx_complete) begin
```

```
waiting_ack         <= 1;
tx_reset            <= 1;
tx_enable                       <= 0;
state                           <= REPLY;
pie_enable                      <= 1;
end
else if (t1_passed) begin
tx_enable <= 1;//begin transmission after t1
pie_enable <= 0;//disable receive during transmission
end
end
end//REPLY
ACKNOWLEDGED : begin
//Enable PIE Decoder
if (pie_reset) begin
pie_reset <= 0;
pie_enable <= 1;
end
if (pie_opcode_ready) begin
case (curr_cmd)
QUERY: begin
state <= ARBITRATE;
slot_counter_ready <= 0;
end
REQ_RN: begin
prng_reset <= 0;
prng_enable <= 1;
end
default: begin
state <= ACKNOWLEDGED;
end
endcase
end
if (pie_data_ready) begin
//See state transition table B.4 for more information
case (curr_cmd)
QUERYREP: begin
state <= READY;
end
QUERYADJUST: begin
state <= READY;
end
ACK: begin
//Means we must have received an ACK command, so we need to verify the RN16 value
if (pie_data[15:0] == prng_16_out) begin
//Valid RN16
state                           <= ACKNOWLEDGED;
num_tx_bits         <= 128;
blf_enable                      <= 1;
blf_reset                       <= 0;
tx_reset            <= 1;
//Load StoredPC, EPCID, & StoredCRC for transmission
for(i = 0; i < 16; i = i + 1) begin
data_out[i] <= stored_pc[15 - i];
end
for(i = 0; i < 96; i = i + 1) begin
```

```
data_out[i+16] <= epc_id[95 - i];
end
for(i = 0; i < 16; i = i + 1) begin
data_out[i+112] <= stored_crc[15 - i];
end
data_out[128] <= 1'b1;
end
else begin
//Invalid RN16
state <= ARBITRATE;
end
end
REQ_RN: begin
if (!crc16_valid) begin
//Must first check that RN16 is the same as the last RN16, then check the CRC16 value
if (pie_data[31:16] == prng_16_out) begin
//Valid RN16
if (!crc16_enable & !crc16_reset) begin
//Load CRC16 with REQ_RN opcode & RN16
for (i = 0; i < 8; i = i + 1) begin
crc16_data_in[i+104] <= curr_cmd[i];
end
for (i = 0; i < 32; i = i + 1) begin
crc16_data_in[i+72] <= pie_data[i];
end
crc16_count <= 39;
crc16_reset <= 1;//Preload FFFF
end
else if (crc16_reset & !crc16_enable) begin
crc16_reset <= 0;
crc16_enable <= 1;
end
else begin
if (rtcal_value == 0) begin
//We have disabled the CRC16 check, as it is not specifically required by Gen2
//And at high speed we cannot meet T1
//This code does perform the check, and could be used
if (crc16_count > 0) begin
crc16_count <= crc16_count - 1;
crc16_data_in <= crc16_data_in << 1;
if (crc16_count == 1) begin
//Prepare memory banks for updating
mem_word                        <= 2;
mem_enable                      <= 0;
mem_write_enable        <= 0;
mem_bank                                        <= 2'b11;//User bank
end
state <= ACKNOWLEDGED;
end
else begin
crc16_enable <= 0;
//Method 1 Appendix F
if (crc16_out == 16'h1D0F) begin
//CRC16 passed
crc16_valid <= 1;
crc16_count <= 15;
```

```
end
else begin//CRC Does not Match
state <= ARBITRATE;
end
end
end
else begin
mem_word                    <= 2;
mem_enable                  <= 0;
mem_write_enable        <= 0;
mem_bank                            <= 2'b11;//User bank
crc16_enable <= 0;
crc16_valid <= 1;
crc16_count <= 15;
end
end
end
else begin
//Invalid RN16
state <= ACKNOWLEDGED;
end
end//!crc16_valid
else begin
if (prng_ready) begin
if (mem_word == 2) begin
//Update RN16 for backscatter
prng_16_out <= prng_32_out[31:16] ^ prng_32_out[15:0];
end
//crc16_count indicates we have the crc16 value
else if (mem_word == 3 & crc16_count == 0) begin
//Setup transmit FSM
for(i = 0; i < 16; i = i + 1) begin
data_out[i] <= prng_16_out[15 - i];
end
for(i = 0; i < 16; i = i + 1) begin
data_out[i+16] <= !crc16_out[15 - i];
end
data_out[32]     <= 1;//In both encoding, last symbol is always 1
num_tx_bits      <= 33;
blf_enable               <= 1;
blf_reset                <= 0;
//Used to verify Read/Write Commands
handle                   <= prng_16_out;
end
else if (mem_word > 3) begin
//Cannot Reply before T1, indicated by t1_passed flag
if (tx_reset & !tx_enable) begin
//Reset and enable the transmit FSM
tx_reset <= 0;
end
else if (tx_complete) begin
tx_reset         <= 1;
tx_enable                <= 0;
state                    <= OPEN;
crc16_valid      <= 0;
pie_enable               <= 1;
```

```
pie_reset                    <= 1;
prng_reset                   <= 1;
end
else if (t1_passed) begin
tx_enable <= 1;//begin transmission after t1
pie_enable <= 0;//disable receive during transmission
end
end
//Update a0 & a1 and store in mem, see LAMED algorithm for more information
case (mem_word)
2: begin
if (!n) begin
a0 <= a1 ^ prng_iv;
a1 <= prng_32_out + prng_key;
end
else begin
a0 <= a1 + prng_iv;
a1 <= prng_32_out + prng_key;
end
mem_word                          <= mem_word + 1;
n                                                  <= !n;
end
3: begin
mem_write_enable             <= 1;
mem_enable                        <= 1;
mem_in                            <= a0[31:16];
if (crc16_count > 0) begin
//Calculate CRC16 value for New RN16 and opcode
if (!crc16_enable & !crc16_reset) begin
//Load CRC16 with REQ_RN opcode & New RN16
for (i = 0; i < 16; i = i + 1) begin
crc16_data_in[i+96] <= prng_16_out[i];
end
crc16_reset <= 1;//Preload FFFF
end
else if (crc16_reset & !crc16_enable) begin
crc16_reset <= 0;
crc16_enable <= 1;
end
else begin
crc16_count <= crc16_count - 1;
crc16_data_in <= crc16_data_in << 1;
end
end
else begin
mem_word                          <= mem_word + 1;
end
end
4: begin
mem_in                            <= a0[15:0];
mem_word                          <= mem_word + 1;
end
5: begin
mem_in                            <= a1[31:16];
mem_word                          <= mem_word + 1;
end
```

```
6: begin
mem_in                          <= a1[15:0];
mem_word                        <= mem_word + 1;
end
7: begin
mem_enable                      <= 0;
end
endcase
end
end
end//REQ_RN
SELECT: begin
state <= READY;
end
default: begin
state <= ARBITRATE;
end
endcase
end
end//ACKOWLEDGED
OPEN: begin
//Enable PIE Decoder
if (pie_reset) begin
pie_reset <= 0;
pie_enable <= 1;
end
if (pie_opcode_ready & !pie_data_ready) begin
case (curr_cmd)
ACCESS: begin
//Activate PRNG
prng_reset          <= 0;
prng_enable <= 1;
//Prepare Memory for updating a0 & a1
mem_word                        <= 2;
mem_enable                      <= 0;
mem_write_enable            <= 0;
mem_bank                                <= 2'b11;//User bank
//Prepare CRC16
crc16_count                     <= 15;
crc16_valid                     <= 0;
crc16_reset                             <= 0;
crc16_enable                    <= 0;
//Reset TX FSM
tx_reset                        <= 1;
tx_enable                               <= 0;
end
REQ_RN: begin
//Activate PRNG
prng_reset          <= 0;
prng_enable <= 1;
//Prepare Memory for updating a0 & a1
mem_word                        <= 2;
mem_enable                      <= 0;
mem_write_enable            <= 0;
mem_bank                                <= 2'b11;//User bank
//Prepare CRC16
```

```
crc16_count                        <= 15;
crc16_valid                        <= 0;
crc16_reset                                  <= 0;
crc16_enable                       <= 0;
//Reset TX FSM
tx_reset                           <= 1;
tx_enable                                    <= 0;
end
default: begin
state <= OPEN;
end
endcase
end
if (pie_data_ready) begin
case (curr_cmd)
ACCESS: begin
//Receiving an ACCESS command in OPEN state gives 1st half of access password, backscatter new
RN16, and moves to SECURED state
//Access password = password XOR rn16
access_password[31:16] <= pie_data[47:32] ^ prng_16_prev;
//Once PRNG is ready, update RN16, calculate CRC16, backscatter both, and update a0, a1
if (prng_ready) begin
//Update RN16
prng_16_out <= prng_32_out[31:16] ^ prng_32_out[15:0];
//Calculate crc16
if (!crc16_valid) begin
if (crc16_count > 0) begin
//Calculate CRC16 value for New RN16 and opcode
if (!crc16_reset & !crc16_enable) begin
crc16_reset <= 1;//Preload FFFF
end
else if (!crc16_enable) begin
//Load CRC16 with new RN16
for (i = 0; i < 16; i = i + 1) begin
crc16_data_in[i+96] <= prng_16_out[i];
end
crc16_reset        <= 0;
crc16_enable       <= 1;
end
else begin
crc16_count        <= crc16_count - 1;
crc16_data_in      <= crc16_data_in << 1;
end
end
else begin
crc16_enable       <= 0;
crc16_valid        <= 1;
end
end
else begin
//Setup TX FSM
if (tx_reset & !tx_enable) begin
//Prepare Data
for(i = 0; i < 16; i = i + 1) begin
data_out[i]        <= prng_16_out[15 - i];
end
```

```
for(i = 0; i < 16; i = i + 1) begin
data_out[i+16] <= !crc16_out[15 - i];
end
data_out[32]        <= 1;//In both encoding, last symbol is always 1
num_tx_bits         <= 33;
blf_enable                    <= 1;
blf_reset                     <= 0;
tx_reset <= 0;
end
else if (tx_complete) begin
tx_reset            <= 1;
tx_enable                     <= 0;
state                         <= SECURED;
crc16_valid         <= 0;
pie_enable                    <= 1;
pie_reset                     <= 1;
access_req                    <= 1;
prng_reset                    <= 1;
end
//Begin transmission after t1
else if (t1_passed) begin
tx_enable           <= 1;
//Disable receive during transmission
pie_enable          <= 0;
end
end
//Update a0 & a1 and store in mem
case (mem_word)
2: begin
if (!n) begin
a0 <= a1 ^ prng_iv;
a1 <= prng_32_out + prng_key;
end
else begin
a0 <= a1 + prng_iv;
a1 <= prng_32_out + prng_key;
end
mem_word                                <= mem_word + 1;
n                                                       <= !n;
end
3: begin
mem_write_enable              <= 1;
mem_enable                              <= 1;
mem_in                                  <= a0[31:16];
mem_word                                          <= mem_word + 1;
end
4: begin
mem_in                                  <= a0[15:0];
mem_word                                <= mem_word + 1;
end
5: begin
mem_in                                  <= a1[31:16];
mem_word                                <= mem_word + 1;
end
6: begin
mem_in                                  <= a1[15:0];
```

```
mem_word                    <= mem_word + 1;
end
7: begin
mem_enable                  <= 0;
end
endcase
end
end//ACCESS
REQ_RN: begin
//REQ_RN requires us to generate a new RN16, calculate a CRC16, and backscatter RN16 and CRC16
//FIXME, should verify RN16 is what we sent
//Once PRNG is ready, update RN16, calculate CRC16, backscatter both, and update a0, a1
if (prng_ready) begin
//Update RN16
prng_16_out       <= prng_32_out[31:16] ^ prng_32_out[15:0];
prng_16_prev      <= prng_32_out[31:16] ^ prng_32_out[15:0];
//Calculate crc16
if (!crc16_valid) begin
if (crc16_count > 0) begin
//Calculate CRC16 value for New RN16 and opcode
if (!crc16_reset & !crc16_enable) begin
crc16_reset <= 1;//Preload FFFF
end
else if (!crc16_enable) begin
//Load CRC16 with new RN16
for (i = 0; i < 16; i = i + 1) begin
crc16_data_in[i+96] <= prng_16_out[i];
end
crc16_reset       <= 0;
crc16_enable      <= 1;
end
else begin
crc16_count       <= crc16_count - 1;
crc16_data_in     <= crc16_data_in << 1;
end
end
else begin
crc16_enable      <= 0;
crc16_valid       <= 1;
end
end
else begin
//Setup TX FSM
if (tx_reset & !tx_enable) begin
//Prepare Data
for(i = 0; i < 16; i = i + 1) begin
data_out[i]       <= prng_16_out[15 - i];
end
for(i = 0; i < 16; i = i + 1) begin
data_out[i+16] <= !crc16_out[15 - i];
end
data_out[32]      <= 1;//In both encoding, last symbol is always 1
num_tx_bits       <= 33;
blf_enable              <= 1;
blf_reset               <= 0;
tx_reset <= 0;
```

```
end
else if (tx_complete) begin
tx_reset            <= 1;
tx_enable                   <= 0;
state                       <= OPEN;
crc16_valid         <= 0;
pie_enable                  <= 1;
pie_reset                   <= 1;
end
//Begin transmission after t1
else if (t1_passed) begin
tx_enable           <= 1;
//Disable receive during transmission
pie_enable          <= 0;
end
end
//Update a0 & a1 and store in mem
case (mem_word)
2: begin
if (!n) begin
a0 <= a1 ^ prng_iv;
a1 <= prng_32_out + prng_key;
end
else begin
a0 <= a1 + prng_iv;
a1 <= prng_32_out + prng_key;
end
mem_word                            <= mem_word + 1;
n                                                   <= !n;
end
3: begin
mem_write_enable            <= 1;
mem_enable                          <= 1;
mem_in                              <= a0[31:16];
mem_word                                    <= mem_word + 1;
end
4: begin
mem_in                              <= a0[15:0];
mem_word                            <= mem_word + 1;
end
5: begin
mem_in                              <= a1[31:16];
mem_word                            <= mem_word + 1;
end
6: begin
mem_in                              <= a1[15:0];
mem_word                            <= mem_word + 1;
end
7: begin
mem_enable                          <= 0;
end
endcase
end
end//REQ_RN
READ: begin
//First validate RN16 (handle) Matches
```

```
if (pie_data[31:16] == handle) begin
//Valid Handle
//READ Command is variable length command so different bits are required
//For now we are assumming only 50 bits of READ data
if (!mem_enable) begin
case(pie_data_bits)
50: begin
//FIXME, need to verify this address exists using starting and max
mem_bank <= pie_data[49:48];//MemBank
mem_word <= pie_data[47:40];//WordPtr
mem_max         <= pie_data[47:40] + pie_data[39:32];//WordCount
mem_count <= 0;
mem_enable                          <= 1;
mem_write_enable            <= 0;
data_out                            <= 0;//clear tx data
end
endcase
end
else begin
//Load Memory and Handle into CRC16 data in
if (mem_word < mem_max) begin
//Load CRC16
case (mem_count)
0: begin
crc16_data_in[111]                  <= 1'b0;//Reply starts with 0
crc16_data_in[110:95]       <= mem_out;
end
1: begin
crc16_data_in[94:79]        <= mem_out;
end
2: begin
crc16_data_in[78:63]        <= mem_out;
end
3: begin
crc16_data_in[62:47]        <= mem_out;
end
4: begin
crc16_data_in[46:31]        <= mem_out;
end
//FIXME, because of CRC16 data_in length, max of 4 words can be read
endcase
mem_word <= mem_word + 1;
mem_count <= mem_count + 1;
crc16_enable        <= 0;
crc16_reset         <= 1;
crc16_valid         <= 0;
end
else begin
if (!crc16_enable) begin
//Load Handle
case (mem_count)
1: begin
crc16_data_in[94:79]        <= handle;
crc16_count                                 <= 32;
crc16_enable                                <= 1;
crc16_reset                                 <= 0;
```

```
//Load data fetched from memory
for (i = 0; i < 17; i = i + 1) begin
data_out[i] <= crc16_data_in[111-i];
end
end
2: begin
crc16_data_in[78:63]        <= handle;
crc16_count                                    <= 48;
crc16_enable                                   <= 1;
crc16_reset                                    <= 0;
//Load data fetched from memory
for (i = 0; i < 33; i = i + 1) begin
data_out[i] <= crc16_data_in[111-i];
end
end
3: begin
crc16_data_in[62:47]        <= handle;
crc16_count                                    <= 64;
crc16_enable                                   <= 1;
crc16_reset                                    <= 0;
for (i = 0; i < 49; i = i + 1) begin
data_out[i] <= crc16_data_in[111-i];
end
end
4: begin
crc16_data_in[46:31]        <= handle;
crc16_count                                    <= 80;
crc16_enable                                   <= 1;
crc16_reset                                    <= 0;
//Load data fetched from memory
for (i = 0; i < 65; i = i + 1) begin
data_out[i] <= crc16_data_in[111-i];
end
end
5: begin
crc16_data_in[30:15]        <= handle;
crc16_count                                    <= 96;
crc16_enable                                   <= 1;
crc16_reset                                    <= 0;
//Load data fetched from memory
for (i = 0; i < 81; i = i + 1) begin
data_out[i] <= crc16_data_in[111-i];
end
end
endcase
end
else begin
if (!crc16_valid) begin
if (crc16_count > 0) begin
crc16_count <= crc16_count - 1;
crc16_data_in <= crc16_data_in << 1;
end
else begin
crc16_valid <= 1;
//May be redundant
tx_reset  <= 1;
```

```
tx_enable             <= 0;
//Load Handle & CRC16
case (mem_count)
1: begin
for (i = 0; i < 16; i = i + 1) begin
data_out[i+17] <= handle[15-i];
end
for (i = 0; i < 16; i = i + 1) begin
data_out[i+33] <= !crc16_out[15-i];
end
//Last bit is always 1
data_out[49] <= 1'b1;
num_tx_bits <= 50;
end
2: begin
for (i = 0; i < 16; i = i + 1) begin
data_out[i+33] <= handle[15-i];
end
for (i = 0; i < 16; i = i + 1) begin
data_out[i+49] <= !crc16_out[15-i];
end
//Last bit is always 1
data_out[65] <= 1'b1;
num_tx_bits <= 66;
end
3: begin
for (i = 0; i < 16; i = i + 1) begin
data_out[i+49] <= handle[15-i];
end
for (i = 0; i < 16; i = i + 1) begin
data_out[i+65] <= !crc16_out[15-i];
end
//Last bit is always 1
data_out[81] <= 1'b1;
num_tx_bits <= 82;
end
4: begin
for (i = 0; i < 16; i = i + 1) begin
data_out[i+65] <= handle[15-i];
end
for (i = 0; i < 16; i = i + 1) begin
data_out[i+81] <= !crc16_out[15-i];
end
//Last bit is always 1
data_out[97] <= 1'b1;
num_tx_bits <= 98;
end
5: begin
for (i = 0; i < 16; i = i + 1) begin
data_out[i+81] <= handle[15-i];
end
for (i = 0; i < 16; i = i + 1) begin
data_out[i+97] <= !crc16_out[15-i];
end
//Last bit is always 1
data_out[113] <= 1'b1;
```

```
num_tx_bits <= 114;
end
endcase
end
end//if !crc16_valid
else begin
if (!tx_complete) begin
if (tx_reset & !tx_enable) begin
blf_enable                    <= 1;
blf_reset                     <= 0;
tx_reset            <= 0;
end
else if (t1_passed) begin
tx_enable <= 1;//begin transmission after t1
pie_enable <= 0;//disable receive during transmission
end
end
else begin
//Reset tx fsm
tx_reset  <= 1;
tx_enable           <= 0;
//Reset decoder
if (!pie_reset & !pie_enable) begin
pie_reset <= 1;
end
else begin
pie_reset <= 1;
pie_enable <= 1;
end
//Clean up additional signals
crc16_valid <= 0;
mem_enable  <= 0;
blf_enable  <= 0;
blf_reset           <= 1;
end
end
end
end
end
end
else begin
//Invalid Handle
state <= OPEN;
end
end //READ
endcase
end//pie_data_ready
else begin
state <= OPEN;
end
end//OPEN
SECURED: begin
//Enable PIE Decoder
if (pie_reset) begin
pie_reset <= 0;
pie_enable <= 1;
```

```
end
if (pie_opcode_ready & !pie_data_ready) begin
case (curr_cmd)
ACCESS: begin
//Activate PRNG
prng_reset          <= 0;
prng_enable <= 1;
//Prepare Memory for updating a0 & a1
mem_word                    <= 2;
mem_enable                  <= 0;
mem_write_enable        <= 0;
mem_bank                            <= 2'b11;//User bank
//Prepare CRC16
crc16_count                 <= 15;
crc16_valid                 <= 0;
crc16_reset                         <= 0;
crc16_enable                <= 0;
//Reset TX FSM
tx_reset                    <= 1;
tx_enable                           <= 0;
end
REQ_RN: begin
//Activate PRNG
prng_reset          <= 0;
prng_enable <= 1;
//Prepare Memory for updating a0 & a1
mem_word                    <= 2;
mem_enable                  <= 0;
mem_write_enable        <= 0;
mem_bank                            <= 2'b11;//User bank
//Prepare CRC16
crc16_count                 <= 15;
crc16_valid                 <= 0;
crc16_reset                         <= 0;
crc16_enable                <= 0;
//Reset TX FSM
tx_reset                    <= 1;
tx_enable                           <= 0;
end
endcase
end
if (pie_data_ready) begin
case (curr_cmd)
ACCESS: begin
//Receiving an ACCESS command in SECURED state could give either 16 MSB or 16 LSB of access
password depending on how we got here
//Need to backscatter new RN16 & CRC16
//FIXME, should verify RN16 is what we sent
if (!access_req) begin
//Access password = password XOR rn16
access_password[31:16]    <= pie_data[47:32] ^ prng_16_prev;
end
else begin
access_password[15:0]     <= pie_data[47:32] ^ prng_16_prev;
end
//Once PRNG is ready, update RN16, calculate CRC16, backscatter both, and update a0, a1
```

```
if (prng_ready) begin
//Update RN16
prng_16_out <= prng_32_out[31:16] ^ prng_32_out[15:0];
//Calculate crc16
if (!crc16_valid) begin
if (crc16_count > 0) begin
//Calculate CRC16 value for New RN16 and opcode
if (!crc16_reset & !crc16_enable) begin
crc16_reset <= 1;//Preload FFFF
end
else if (!crc16_enable) begin
//Load CRC16 with new RN16
for (i = 0; i < 16; i = i + 1) begin
crc16_data_in[i+96] <= prng_16_out[i];
end
crc16_reset         <= 0;
crc16_enable        <= 1;
end
else begin
crc16_count         <= crc16_count - 1;
crc16_data_in       <= crc16_data_in << 1;
end
end
else begin
crc16_enable        <= 0;
crc16_valid         <= 1;
end
end
else begin
//Setup TX FSM
if (tx_reset & !tx_enable) begin
//Prepare Data
for(i = 0; i < 16; i = i + 1) begin
data_out[i]         <= prng_16_out[15 - i];
end
for(i = 0; i < 16; i = i + 1) begin
data_out[i+16] <= !crc16_out[15 - i];
end
data_out[32]        <= 1;//In both encoding, last symbol is always 1
num_tx_bits         <= 33;
blf_enable                      <= 1;
blf_reset                       <= 0;
tx_reset <= 0;
end
else if (tx_complete) begin
tx_reset            <= 1;
tx_enable                       <= 0;
state                           <= SECURED;
crc16_valid         <= 0;
pie_enable                      <= 1;
pie_reset                       <= 1;
prng_reset                      <= 1;
access_req                      <= 1;
end
//Begin transmission after t1
else if (t1_passed) begin
```

```
tx_enable            <= 1;
//Disable receive during transmission
pie_enable           <= 0;
end
end
//Update a0 & a1 and store in mem
case (mem_word)
2: begin
if (!n) begin
a0 <= a1 ^ prng_iv;
a1 <= prng_32_out + prng_key;
end
else begin
a0 <= a1 + prng_iv;
a1 <= prng_32_out + prng_key;
end
mem_word                              <= mem_word + 1;
n                                                  <= !n;
end
3: begin
mem_write_enable           <= 1;
mem_enable                            <= 1;
mem_in                                <= a0[31:16];
mem_word                                        <= mem_word + 1;
end
4: begin
mem_in                                <= a0[15:0];
mem_word                              <= mem_word + 1;
end
5: begin
mem_in                                <= a1[31:16];
mem_word                              <= mem_word + 1;
end
6: begin
mem_in                                <= a1[15:0];
mem_word                              <= mem_word + 1;
end
7: begin
mem_enable                            <= 0;
end
endcase
end
end//ACCESS
REQ_RN: begin
//REQ_RN requires us to generate a new RN16, calculate a CRC16, and backscatter RN16 and CRC16
//FIXME, should verify RN16 is what we sent
//Once PRNG is ready, update RN16, calculate CRC16, backscatter both, and update a0, a1
if (prng_ready) begin
//Update RN16
prng_16_out        <= prng_32_out[31:16] ^ prng_32_out[15:0];
prng_16_prev       <= prng_32_out[31:16] ^ prng_32_out[15:0];
//Calculate crc16
if (!crc16_valid) begin
if (crc16_count > 0) begin
//Calculate CRC16 value for New RN16 and opcode
if (!crc16_reset & !crc16_enable) begin
```

```
crc16_reset <= 1;//Preload FFFF
end
else if (!crc16_enable) begin
//Load CRC16 with new RN16
for (i = 0; i < 16; i = i + 1) begin
crc16_data_in[i+96] <= prng_16_out[i];
end
crc16_reset       <= 0;
crc16_enable      <= 1;
end
else begin
crc16_count       <= crc16_count - 1;
crc16_data_in     <= crc16_data_in << 1;
end
end
else begin
crc16_enable      <= 0;
crc16_valid       <= 1;
end
end
else begin
//Setup TX FSM
if (tx_reset & !tx_enable) begin
//Prepare Data
for(i = 0; i < 16; i = i + 1) begin
data_out[i]       <= prng_16_out[15 - i];
end
for(i = 0; i < 16; i = i + 1) begin
data_out[i+16] <= !crc16_out[15 - i];
end
data_out[32]      <= 1;//In both encoding, last symbol is always 1
num_tx_bits       <= 33;
blf_enable                  <= 1;
blf_reset                   <= 0;
tx_reset <= 0;
end
else if (tx_complete) begin
tx_reset          <= 1;
tx_enable                   <= 0;
state                       <= SECURED;
crc16_valid       <= 0;
pie_enable                  <= 1;
pie_reset                   <= 1;
prng_reset                  <= 1;
end
//Begin transmission after t1
else if (t1_passed) begin
tx_enable         <= 1;
//Disable receive during transmission
pie_enable        <= 0;
end
end
//Update a0 & a1 and store in mem
case (mem_word)
2: begin
if (!n) begin
```

```
a0 <= a1 ^ prng_iv;
a1 <= prng_32_out + prng_key;
end
else begin
a0 <= a1 + prng_iv;
a1 <= prng_32_out + prng_key;
end
mem_word                    <= mem_word + 1;
n                                    <= !n;
end
3: begin
mem_write_enable       <= 1;
mem_enable                  <= 1;
mem_in                      <= a0[31:16];
mem_word                         <= mem_word + 1;
end
4: begin
mem_in                      <= a0[15:0];
mem_word                    <= mem_word + 1;
end
5: begin
mem_in                      <= a1[31:16];
mem_word                    <= mem_word + 1;
end
6: begin
mem_in                      <= a1[15:0];
mem_word                    <= mem_word + 1;
end
7: begin
mem_enable                  <= 0;
end
endcase
end
end//REQ_RN
READ: begin
//First validate RN16 (handle) Matches
if (pie_data[31:16] == handle) begin
//Valid Handle
//READ Command is variable length command so different bits are required
//For now we are assumming only 50 bits of READ data
if (!mem_enable) begin
case(pie_data_bits)
50: begin
//FIXME, need to verify this address exists using starting and max
mem_bank <= pie_data[49:48];//MemBank
mem_word <= pie_data[47:40];//WordPtr
mem_max         <= pie_data[47:40] + pie_data[39:32];//WordCount
mem_count <= 0;
mem_enable                  <= 1;
mem_write_enable       <= 0;
data_out                    <= 0;//clear tx data
end
endcase
end
else begin
//Load Memory and Handle into CRC16 data in
```

```
if (mem_word < mem_max) begin
//Load CRC16
case (mem_count)
0: begin
crc16_data_in[111]                  <= 1'b0;//Reply starts with 0
crc16_data_in[110:95]       <= mem_out;
present_reset                               <= 1;//PRESENT may be used, so premptively reset and
disable
present_enable                              <= 0;
end
1: begin
crc16_data_in[94:79]        <= mem_out;
end
2: begin
crc16_data_in[78:63]        <= mem_out;
end
3: begin
crc16_data_in[62:47]                        <= mem_out;
//Load sensor data into PRESENT
present_plaintext[63:16] <= crc16_data_in[110:63];
present_plaintext[15:00] <= mem_out;
//This represents the XOR of the present key and TRNG from the reader
//Due to limitations of the reader software we must emulate this using
//the 32 bit access password, expanded to 80 bits through duplication
present_key[79:48] <= present_key[79:48] ^ access_password;
present_key[47:16] <= present_key[47:16] ^ access_password;
present_key[15:00] <= present_key[15:00] ^ access_password[31:16];
end
4: begin
crc16_data_in[46:31]        <= mem_out;
end
//FIXME, because of CRC16 data_in length, max of 4 words can be read
endcase
mem_word                    <= mem_word + 1;
mem_count                   <= mem_count + 1;
crc16_enable        <= 0;
crc16_reset         <= 1;
crc16_valid         <= 0;
end
else begin
if (!crc16_enable) begin
//Load Handle
case (mem_count)
1: begin
crc16_data_in[94:79]        <= handle;
crc16_count                                 <= 32;
crc16_enable                                <= 1;
crc16_reset                                 <= 0;
//Load data fetched from memory
for (i = 0; i < 17; i = i + 1) begin
data_out[i] <= crc16_data_in[111-i];
end
end
2: begin
crc16_data_in[78:63]        <= handle;
crc16_count                                 <= 48;
```

```
crc16_enable                        <= 1;
crc16_reset                         <= 0;
//Load data fetched from memory
for (i = 0; i < 33; i = i + 1) begin
data_out[i] <= crc16_data_in[111-i];
end
end
3: begin
crc16_data_in[62:47]        <= handle;
crc16_count                         <= 64;
crc16_enable                        <= 1;
crc16_reset                         <= 0;
for (i = 0; i < 49; i = i + 1) begin
data_out[i] <= crc16_data_in[111-i];
end
end
4: begin
//If we are reading sensor data from the secure state we need to encrypt the data
if (mem_bank == 2'b00 & mem_max == 8) begin
if (present_complete) begin
crc16_data_in[110:47] <= present_ciphertext;
data_out[0] <= 0;
//Load PRESENT data for transmit
for (i = 1; i < 65; i = i + 1) begin
data_out[i] <= present_ciphertext[64-i];
end
crc16_count                         <= 80;
crc16_enable                        <= 1;
crc16_reset                         <= 0;
end
else begin
present_reset                       <= 0;
present_enable                      <= 1;
end
end
else begin
crc16_count                         <= 80;
crc16_enable                        <= 1;
crc16_reset                         <= 0;
//Load data fetched from memory
for (i = 0; i < 65; i = i + 1) begin
data_out[i] <= crc16_data_in[111-i];
end
end
crc16_data_in[46:31]        <= handle;
end
5: begin
crc16_data_in[30:15]        <= handle;
crc16_count                         <= 96;
crc16_enable                        <= 1;
crc16_reset                         <= 0;
//Load data fetched from memory
for (i = 0; i < 81; i = i + 1) begin
data_out[i] <= crc16_data_in[111-i];
end
end
```

```
endcase
end
else begin
if (!crc16_valid) begin
if (crc16_count > 0) begin
crc16_count <= crc16_count - 1;
crc16_data_in <= crc16_data_in << 1;
end
else begin
crc16_valid <= 1;
//May be redundant
tx_reset  <= 1;
tx_enable             <= 0;
//Load Handle & CRC16
case (mem_count)
1: begin
for (i = 0; i < 16; i = i + 1) begin
data_out[i+17] <= handle[15-i];
end
for (i = 0; i < 16; i = i + 1) begin
data_out[i+33] <= !crc16_out[15-i];
end
//Last bit is always 1
data_out[49] <= 1'b1;
num_tx_bits <= 50;
end
2: begin
for (i = 0; i < 16; i = i + 1) begin
data_out[i+33] <= handle[15-i];
end
for (i = 0; i < 16; i = i + 1) begin
data_out[i+49] <= !crc16_out[15-i];
end
//Last bit is always 1
data_out[65] <= 1'b1;
num_tx_bits <= 66;
end
3: begin
for (i = 0; i < 16; i = i + 1) begin
data_out[i+49] <= handle[15-i];
end
for (i = 0; i < 16; i = i + 1) begin
data_out[i+65] <= !crc16_out[15-i];
end
//Last bit is always 1
data_out[81] <= 1'b1;
num_tx_bits <= 82;
end
4: begin
for (i = 0; i < 16; i = i + 1) begin
data_out[i+65] <= handle[15-i];
end
for (i = 0; i < 16; i = i + 1) begin
data_out[i+81] <= !crc16_out[15-i];
end
//Last bit is always 1
```

```
data_out[97] <= 1'b1;
num_tx_bits <= 98;
end
5: begin
for (i = 0; i < 16; i = i + 1) begin
data_out[i+81] <= handle[15-i];
end
for (i = 0; i < 16; i = i + 1) begin
data_out[i+97] <= !crc16_out[15-i];
end
//Last bit is always 1
data_out[113] <= 1'b1;
num_tx_bits <= 114;
end
endcase
end
end//if !crc16_valid
else begin
if (!tx_complete) begin
if (tx_reset & !tx_enable) begin
blf_enable                  <= 1;
blf_reset                   <= 0;
tx_reset            <= 0;
end
else if (t1_passed) begin
tx_enable <= 1;//begin transmission after t1
pie_enable <= 0;//disable receive during transmission
end
end
else begin
//Reset tx fsm
tx_reset  <= 1;
tx_enable           <= 0;
//Reset decoder
if (!pie_reset & !pie_enable) begin
pie_reset <= 1;
end
else begin
pie_reset <= 1;
pie_enable <= 1;
end
//Clean up additional signals
crc16_valid <= 0;
mem_enable  <= 0;
blf_enable  <= 0;
blf_reset           <= 1;
//Reset the key to the original value
present_key[79:48] <= present_key[79:48] ^ access_password;
present_key[47:16] <= present_key[47:16] ^ access_password;
present_key[15:00] <= present_key[15:00] ^ access_password[31:16];
end
end
end
end
end
end
```

```
else begin
//Invalid Handle
state <= SECURED;
end
end //READ
endcase
end//pie_data_ready
else begin
state <= SECURED;
end
end
default: begin
state <= POWERUP;
mem_bank <= 2'b01;
end
endcase
end
end//always
always @(posedge pie_opcode_ready) begin
//After receiving opcode from PIE decoder load into curr_cmd reg
curr_cmd <= pie_opcode;
rtcal_value <= rtcal;
end
//CRC5 Control Logic
always @ (posedge clk128) begin
if (crc5_reset) begin
//Load Query Command and data
crc5_data           <= {4'b1000, pie_data[17:0]};
crc5_counter <= 0;
crc5_valid          <= 0;
end
else if (crc5_enable & crc5_counter < 22) begin
crc5_data           <= crc5_data << 1;
crc5_counter <= crc5_counter + 1;
end
else if (crc5_counter == 22) begin
if (crc5_result == 0) begin
crc5_valid <= 1;
end
end
end
//T1 control logic
always @ (posedge clk128) begin
if (!por) begin
t1_count <= 0;
end
else if (t1_count == 0 & pie_data_ready) begin //Start counting T1 from the start of the data_ready flag
t1_count <= t1_count + 1;
end
else if (t1_count > 0 & !t1_passed) begin //Keep counting until t1 is reached (based on t1_passed flag)
t1_count <= t1_count + 1;
end
else if (t1_passed) begin
if (pie_opcode_ready) begin
t1_count <= t1_count;
end
```

```
else begin
t1_count <= 0;
end
end
//rtcal_value must be divided by 2 to convert from sampleclk to dig clk
if (rtcal_value/2 > 10*tpri) begin
if (t1_count >= rtcal_value/2) begin
t1_passed <= 1;
end
else begin
t1_passed <= 0;
end
end
else begin
if (t1_count >= 10*tpri) begin
t1_passed <= 1;
end
else begin
t1_passed <= 0;
end
end
end
//Global Reset Controls
//Basically this is a counter that waits and if our fsm state does not change
//it will time out and reset the FSM, we are assuming that we are frozen.
always @ (posedge clk128) begin
if (!por) begin
global_reset_sig <= 0;
global_reset_count <= 0;
prev_state <= POWERUP;
end
else if (por & state == POWERUP) begin
global_reset_sig <= 1;
end
else if (por & global_reset_count < 32767) begin
if (prev_state == state) begin
global_reset_count <= global_reset_count + 1;
end
else begin
global_reset_count <= 0;
end
prev_state <= state;
end
else begin
global_reset_count <= 0;
global_reset_sig <= 0;//Timeout
end
end
//Instantiate all required subblocks
tpri_logic tpri_logic_0(.dr(dr),
.trcal_value(trcal_value),
.tpri(tpri)
);
mem_banks memory (                    .reset(mem_reset),
.enable(mem_enable),
.data_out(mem_out),
.write_enable(mem_write_enable),
.bank(mem_bank),
.data_in(mem_in),
.word(mem_word),
.clk(clk128)
);
.data_in(demod_data),
.enable(pie_enable),
.reset(pie_reset),
.op_code_ready(pie_opcode_ready),
.op_code(pie_opcode),
.data_ready(pie_data_ready),
.data(pie_data),
.query_received(pie_query_received),
.trcal(trcal),
.rtcal(rtcal),
.data_bits(pie_data_bits)
);
lamed prng_0(                              .clk(clk128),
.enable(prng_enable),
.reset(prng_reset),
.key(a0),
.iv(a1),
.out(prng_32_out),
.data_ready(prng_ready)
);
crc5 crc5_0(                               .clk(clk128),
.enable(crc5_enable),
.reset(crc5_reset),
.data_in(crc5_data[21]),
.q(crc5_result)
);
crc16 crc16_0(                             .clk(clk128),
.enable(crc16_enable),
.reset(crc16_reset),
.data_in(crc16_data_in[111]),
.q(crc16_out)
);
blf_divider        blf_div_0(        .clk(clk256),
.enable(blf_enable),
.dr(dr),
.trcal(trcal_value),
.blf_clk(blf_clk),
.reset(blf_reset)
);
.num_bits(num_tx_bits),
.trext(trext),
.tx_data(data_out),
.tx_complete(tx_complete),
.enable(tx_enable),
.data_out(transmit),
.m(m),
.reset(tx_reset)
);
ds1620 temp_sensor_0 (    .enable(sensor_enable),
.reset(sensor_reset),
.clk256(clk256),
.temperature_reading(sensor_data),
.rst_n(sensor_rst_n),
.dq(sensor_dq),
.clk_conv(sensor_clk_conv),
.temp_ready(sensor_complete)
);
present80 present_0 (        .enable(present_enable),
.reset(present_reset),
.clk(clk128),
.plaintext(present_plaintext),
.key(present_key),
.ciphertext(present_ciphertext),
.complete(present_complete)
);
endmodule
///////IMPLEMENTATION NOTES///////////////////////////////////////////////////////////////////
// Commands not supported : Kill, Write
// Commands partially supported: Read, QueryAdjust
// T2 is not supported, Gen2 table 6.13. Should not be a problem if the reader is compliant
//////////////////////////////////////////////////////////////////////////////////////////////
```

APPENDIX J

DS1620 TEMPERATURE SENSOR VERILOG

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    14:00:41 06/02/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    ds1620
// Project Name:   M.S. Thesis
// Revision:                1.0
//
// Additional Comments: Code used to retrieve a single 9 bit temperature reading
// from a DS1620 temperature sensor. See datasheet for more information
////////////////////////////////////////////////////////////////////////////////
module ds1620(temperature_reading, enable, reset, clk256, clk_conv, dq, rst_n, temp_ready);
input enable;
input reset;
inout dq;//Data I/O, all data sent LSB first
output reg clk_conv;//Clock used to drive sensor
output reg rst_n;
output reg [8:0] temperature_reading;//Temperature output from sensor
output reg temp_ready;//Flag used to indicate temperature has been read
reg [7:0] data_out;//Data to be sent to sensor
reg [2:0] state;
reg byte_count;//Tracks which command has been sent between START_CONVERT_T (!byte_count) and
READ_TEMPERATURE (byte_count)
reg [3:0] bit_count;//Tracks bits sent & received
reg [2:0] clk_count;//Used to ensure data is shifted before clk_conv is raised to latch data
reg mode;//Determines TX (transmit) or RX (receive) mode
parameter TX = 1'b1;
parameter RX = 1'b0;
//FSM state values
parameter [2:0] START           = 3'b000;
parameter [2:0] LOAD_DATA       = 3'b001;
parameter [2:0] TX_BYTE         = 3'b010;
parameter [2:0] RX_TEMP         = 3'b011;
parameter [2:0] STOP                    = 3'b100;
//Sensor command values from DS1620 datasheet
parameter [7:0] READ_TEMPERATURE = 8'hAA;
parameter [7:0] START_CONVERT_T       = 8'hEE;
always @ (posedge clk256 or posedge reset) begin
if (reset) begin
rst_n                           <= 0;
clk_conv                        <= 1;
byte_count                      <= 0;
bit_count                       <= 0;
clk_count                       <= 0;
temp_ready                      <= 0;
state                           <= START;
temperature_reading        <= 0;
end
```

```
else if (enable) begin
case (state)
START: begin
rst_n <= 1;//Raise rst_n to begin data transfer
mode     <= TX;
state <= LOAD_DATA;
end//START
LOAD_DATA: begin//First START_CONVERT_T is sent, reset is toggled, and READ_TEMPERATURE
is sent, then data is received
case(byte_count)
0: begin
data_out <= START_CONVERT_T;
end
1: begin
data_out <= READ_TEMPERATURE;
end
endcase
state <= TX_BYTE;
end//LOAD_DATA
TX_BYTE: begin//Shift out data over DQ pin
case(clk_count)
0: begin
clk_conv            <= 0;//lower output clk
clk_count           <= clk_count + 1;
end
1: begin
if (bit_count > 0) begin
data_out <= {1'b0, data_out[7:1]};//Right shift
end
clk_count           <= clk_count + 1;
end
2: begin
clk_conv            <= 1;//Raise output clk to latch data in sensor
clk_count           <= clk_count + 1;
end
3: begin
if (bit_count < 7) begin
state               <= TX_BYTE;
bit_count           <= bit_count + 1;
clk_count           <= 0;
end
else if (!byte_count) begin
state               <= STOP;//Toggle rst_n after sending START_CONVERT_T command
bit_count           <= 0;
clk_count           <= 0;
end
else begin
state               <= RX_TEMP;//Receive temperature value after sending READ_TEMPERATURE
command
bit_count           <= 0;
mode                <= RX;
clk_count           <= 2;
end
end
endcase
end//TX_BYTE
```

```
RX_TEMP: begin//Receive 9 bit temperature reading
case(clk_count)
0: begin
clk_conv                <= 1;
temperature_reading     <= {dq, temperature_reading[8:1]};//shift in value
clk_count               <= clk_count + 1;
end
1: begin
clk_count               <= clk_count + 1;
end
2: begin
clk_conv                <= 0;
clk_count               <= clk_count + 1;
end
3: begin
clk_count               <= 0;
if (bit_count < 9) begin
state           <= RX_TEMP;//Continue sampling until all 9 bits have been clocked
bit_count       <= bit_count + 1;
end
else begin
state           <= STOP;//After all bits have been received hold value
bit_count       <= 0;
end
end
endcase
end//RX_TEMP
STOP: begin//First time we enter STOP, toggle reset, 2nd time hold value and raise temp_ready flag
rst_n                   <= 0;
if (!byte_count) begin
state           <= START;
byte_count      <= byte_count + 1;
end
else begin
state           <= STOP;
temp_ready      <= 1;
end
end//STOP
endcase
end
end
assign dq = (mode) ? data_out[0] : 1'bz;//data_out[0] is the bit currently being transmitted to the sensor
endmodule
```

APPENDIX K

MEMORY BANKS VERILOG

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    15:53:38 05/20/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    mem_banks
// Project Name:   M.S. Thesis
// Revision:                 1.0
//
// Additional Comments: 4 Memory Banks in compliance with Gen2 section 6.3.2.1
// This module is only intended for prototyping, not synthesis
////////////////////////////////////////////////////////////////////////////////
module mem_banks(reset, enable, data_out, write_enable, bank, data_in, word, clk);
//We will use reset to preload the EPCID, and other information. If this memory is reset, all written data is
lost
//Number of words in each bank
parameter USER_MEM_WORDS  = 4'd8;
parameter TID_MEM_WORDS     = 4'd2;
parameter EPC_MEM_WORDS     = 4'd8;
parameter RES_MEM_WORDS     = 4'd13;
//largest number of bits required to address each bank
parameter MAX_NUM_BITS                    = 4'd4;
input enable;
input reset;
input clk;
//Word to be written/read
input [MAX_NUM_BITS - 1 : 0] word;
input [15:0] data_in;
input write_enable;
input [1:0] bank;
output reg [15:0] data_out;
//Counter used for reset
integer i;
//4 banks of 16 bit word memory
reg [15:0] user_mem [(USER_MEM_WORDS - 1) : 0];
reg [15:0] tid_mem  [(TID_MEM_WORDS - 1)  : 0];
reg [15:0] epc_mem  [(EPC_MEM_WORDS - 1)  : 0];
reg [15:0] res_mem  [(RES_MEM_WORDS - 1)  : 0];
always @(negedge clk) begin
if (reset) begin
//load epcid
epc_mem[7] <= 16'h1111;
epc_mem[6] <= 16'h2222;
epc_mem[5] <= 16'h3333;
epc_mem[4] <= 16'h4444;
epc_mem[3] <= 16'h5555;
epc_mem[2] <= 16'h6666;
//load StoredPC
epc_mem[1] <= 16'h3000;
```

```
//Stored CRC goes here
epc_mem[0] <= 16'h0000;
//User memory bank word 0-1 & 2-3 contain the seed and iv values for the prng
//After the initial values are retrieved, updated values will be stored in 4-7
user_mem[0] <= 16'hFF1E;
user_mem[1] <= 16'hFFAA;
user_mem[2] <= 16'hAABB;
user_mem[3] <= 16'hAAB8;
user_mem[4] <= 16'hFF1E;
user_mem[5] <= 16'hFFAA;
user_mem[6] <= 16'hAABB;
user_mem[7] <= 16'hAAB8;
//Store TID values
tid_mem[0]        <= 16'hF0F0;
tid_mem[1]        <= 16'hF0E0;
//RES mem locations 4-7 are where sensor data is placed
//Access & Kill Password should be in 0-3
//Clear RES mem 0-7
for (i = 0; i < 8; i = i + 1) begin
res_mem[i] <= 16'h0000;
end
//RES mem locations 8-12 store the 80 bit PRESENT key
res_mem[08] <= 16'h1111;
res_mem[09] <= 16'h2222;
res_mem[10] <= 16'h3333;
res_mem[11] <= 16'h4444;
res_mem[12] <= 16'h5555;
end
//Check if word is too large will handled by either mem controller or fsm
case (bank)
2'b00: begin
res_mem[word] <= data_in;
end
2'b01: begin
epc_mem[word] <= data_in;
end
2'b10: begin
tid_mem[word] <= data_in;
end
2'b11: begin
user_mem[word] <= data_in;
end
endcase
end
case (bank)
2'b00: begin
data_out[15:0] <= res_mem[word];
end
2'b01: begin
data_out[15:0] <= epc_mem[word];
end
2'b10: begin
data_out[15:0] <= tid_mem[word];
end
2'b11: begin
data_out[15:0] <= user_mem[word];
```

```
end
endcase
end
end
endmodule
```

APPENDIX L

PRESENT DECRYPTION VERILOG

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: University of Massachusetts Amherst
// Engineer: Michael Todd
//
// Create Date:    10:39:07 06/04/2010
// Design Name:    Class 1 Generatation 2 RFID
// Module Name:    present80_decrypt
// Project Name:   M.S. Thesis
// Revision:                    1.0
//
// Additional Comments: Module designed to decrypt PRESENT data with an 80 bit
// key size. Designed only for testing, not synthesis
//////////////////////////////////////////////////////////////////////////////////
module present80_with_decryption(clk, enable, reset, plaintext, key, ciphertext, complete);
input clk;
input enable;
input reset;
input [63:0] ciphertext;//Ciphertext to be decrypted
input [79:0] key;//secret 80 bit key
output [63:0] plaintext;//Result of decryption
output reg complete;//Flag used to indicate decryption is complete
reg [79:0] roundkey_in;//Round key before update
wire[79:0] roundkey_out;//Round key after update
wire [63:0] key_text_xor;//XOR of roundkey and round output
wire [63:0] sbox_out;//Result of sbox layer
wire [63:0] permute;//Result of permutation layer
reg [4:0] roundcounter;//Important: For this to work, must be non-saturating counter
reg [63:0] temp;//Used to store intermidiate round values
reg [79:0] keys [31:0];//Precomputed round keys used in decryption
reg keys_computed;//Flag indicating round keys have been computed
always@ (posedge clk) begin
if (reset) begin
roundkey_in     <= key;
temp                        <= ciphertext;
roundcounter    <= 1;
complete                    <= 0;
keys_computed   <= 0;
end
else if (enable) begin
if (keys_computed) begin
case (roundcounter)
31: begin
complete <= 1;
end
default: begin
roundkey_in     <= keys[roundcounter];
roundcounter    <= roundcounter - 1;
temp            <= sbox_out;
end
0: begin
```

```
temp                <= sbox_out;
roundkey_in         <= key;
roundcounter        <= 31;
end
endcase
end
else begin
//Precompute round keys for decryption
case (roundcounter)
default: begin
roundkey_in                 <= roundkey_out;
roundcounter                <= roundcounter + 1;
keys[roundcounter]          <= roundkey_out;
end
0: begin
keys_computed   <= 1;
roundcounter        <= 30;
roundkey_in         <= keys[31];
temp                <= ciphertext;
end
endcase
end
end
end
keyscheduler_dec k0(.enable(enable), .key_in(roundkey_in), .roundkey(roundkey_out),
.roundcounter(roundcounter));
//sbox layer
sbox_dec s0(permute[63:60], sbox_out[63:60]);
sbox_dec s1(permute[59:56], sbox_out[59:56]);
sbox_dec s2(permute[55:52], sbox_out[55:52]);
sbox_dec s3(permute[51:48], sbox_out[51:48]);
sbox_dec s4(permute[47:44], sbox_out[47:44]);
sbox_dec s5(permute[43:40], sbox_out[43:40]);
sbox_dec s6(permute[39:36], sbox_out[39:36]);
sbox_dec s7(permute[35:32], sbox_out[35:32]);
sbox_dec s8(permute[31:28], sbox_out[31:28]);
sbox_dec s9(permute[27:24], sbox_out[27:24]);
sbox_dec s10(permute[23:20], sbox_out[23:20]);
sbox_dec s11(permute[19:16], sbox_out[19:16]);
sbox_dec s12(permute[15:12], sbox_out[15:12]);
sbox_dec s13(permute[11:08], sbox_out[11:08]);
sbox_dec s14(permute[07:04], sbox_out[07:04]);
sbox_dec s15(permute[03:00], sbox_out[03:00]);
//permutation layer
assign permute[00] = key_text_xor[00];
assign permute[01] = key_text_xor[16];
assign permute[02] = key_text_xor[32];
assign permute[03] = key_text_xor[48];
assign permute[04] = key_text_xor[01];
assign permute[05] = key_text_xor[17];
assign permute[06] = key_text_xor[33];
assign permute[07] = key_text_xor[49];
assign permute[08] = key_text_xor[02];
assign permute[09] = key_text_xor[18];
assign permute[10] = key_text_xor[34];
assign permute[11] = key_text_xor[50];
assign permute[12] = key_text_xor[03];
assign permute[13] = key_text_xor[19];
assign permute[14] = key_text_xor[35];
assign permute[15] = key_text_xor[51];
assign permute[16] = key_text_xor[04];
assign permute[17] = key_text_xor[20];
assign permute[18] = key_text_xor[36];
assign permute[19] = key_text_xor[52];
assign permute[20] = key_text_xor[05];
assign permute[21] = key_text_xor[21];
assign permute[22] = key_text_xor[37];
assign permute[23] = key_text_xor[53];
assign permute[24] = key_text_xor[06];
assign permute[25] = key_text_xor[22];
assign permute[26] = key_text_xor[38];
assign permute[27] = key_text_xor[54];
assign permute[28] = key_text_xor[07];
assign permute[29] = key_text_xor[23];
assign permute[30] = key_text_xor[39];
assign permute[31] = key_text_xor[55];
assign permute[32] = key_text_xor[08];
assign permute[33] = key_text_xor[24];
assign permute[34] = key_text_xor[40];
assign permute[35] = key_text_xor[56];
assign permute[36] = key_text_xor[09];
assign permute[37] = key_text_xor[25];
assign permute[38] = key_text_xor[41];
assign permute[39] = key_text_xor[57];
assign permute[40] = key_text_xor[10];
assign permute[41] = key_text_xor[26];
assign permute[42] = key_text_xor[42];
assign permute[43] = key_text_xor[58];
assign permute[44] = key_text_xor[11];
assign permute[45] = key_text_xor[27];
assign permute[46] = key_text_xor[43];
assign permute[47] = key_text_xor[59];
assign permute[48] = key_text_xor[12];
assign permute[49] = key_text_xor[28];
assign permute[50] = key_text_xor[44];
assign permute[51] = key_text_xor[60];
assign permute[52] = key_text_xor[13];
assign permute[53] = key_text_xor[29];
assign permute[54] = key_text_xor[45];
assign permute[55] = key_text_xor[61];
assign permute[56] = key_text_xor[14];
assign permute[57] = key_text_xor[30];
assign permute[58] = key_text_xor[46];
assign permute[59] = key_text_xor[62];
assign permute[60] = key_text_xor[15];
assign permute[61] = key_text_xor[31];
assign permute[62] = key_text_xor[47];
assign permute[63] = key_text_xor[63];
assign key_text_xor = roundkey_in[79:16] ^ temp;
assign plaintext = key_text_xor;
endmodule
```

```
module keyscheduler_dec(enable, key_in, roundkey, roundcounter);
//This is the key scheduling circuit for PRESENT
input [79:0] key_in;
input enable;
input [4:0] roundcounter;
//bits [79:16] of key_out are the actual round keys
output [79:0] roundkey;
wire [79:0] shiftout, nextkey;
//Sbox 4 MSB of shifted data
sbox keysbox(shiftout[79:76], nextkey[79:76]);
//perform the 18 bit barrel shift operation
assign shiftout[79:0] = {key_in[18:0], key_in[79:19]};
//XOR bits 15-19 with round counter
assign nextkey[19] = shiftout[19] ^ roundcounter[4];
assign nextkey[18] = shiftout[18] ^ roundcounter[3];
assign nextkey[17] = shiftout[17] ^ roundcounter[2];
assign nextkey[16] = shiftout[16] ^ roundcounter[1];
assign nextkey[15] = shiftout[15] ^ roundcounter[0];
assign nextkey[14:0] = shiftout[14:0];
assign nextkey[75:20] = shiftout[75:20];
assign roundkey = (enable) ? nextkey : 0;
endmodule

module sbox_dec(sbox_in, sbox_out);
//This module is the circuit used for non-linear permutation in the
//Present block cipher
input [3:0] sbox_in;
output reg [3:0] sbox_out;
always @* begin
case (sbox_in)
4'h0: sbox_out <= 4'h5;
4'h1: sbox_out <= 4'hE;
4'h2: sbox_out <= 4'hF;
4'h3: sbox_out <= 4'h8;
4'h4: sbox_out <= 4'hC;
4'h5: sbox_out <= 4'h1;
4'h6: sbox_out <= 4'h2;
4'h7: sbox_out <= 4'hD;
4'h8: sbox_out <= 4'hB;
4'h9: sbox_out <= 4'h4;
4'hA: sbox_out <= 4'h6;
4'hB: sbox_out <= 4'h3;
4'hC: sbox_out <= 4'h0;
4'hD: sbox_out <= 4'h7;
4'hE: sbox_out <= 4'h9;
4'hF: sbox_out <= 4'hA;
endcase
end
endmodule
```

APPENDIX M

WAVEFORM TEXT FILE TO VERILOG TESTBENCH SCRIPT

```
#!/usr/bin/perl

#Waveform Text file to be parsed
my $file1 = 'access_password_test_short_Ch1.txt';

#Open file handler for both result files
open(FILE1, $file1);

#Load file contents into an array for checking
my @file1_contents = <FILE1>;

#Close the file handler
close(FILE1);

#output signal name
my $sig_name = 'temp2';

#Counters
my $j;
my $k;

my $result;

my $exponent;

#if simulation starts with negative value add offset to all numbers
my $offset = 0;

my $tempresult;

$j = 0;

$k = 0;

#when these values differ, append output
my $current_voltage;
my $prev_voltage;

my $temp_voltage;

my $current_time = 0;
my $prev_time = 0;


#my $sig_name = 'demod_data';

for $i (@file1_contents) {
   chomp $i;

   #check if first number is negative to apply offset
   if($i =~ /^\s+(-?)(\d)\.(\d+)e(-?)(\d+)\s+(-?)(\d)\.(\d+)e(-|\+)(\d+)/) {
```

```
if ($j == 0) {
   $j = 1;
   #calulate offset
   $result = "$1$2.$3";
   $exponent = "$4$5";
   $exponent = 10**$exponent;
   $offset = -1*$result*$exponent*10**9; #in ns
   #calculate starting voltage
   $result = "$6$7.$8";
   $exponent = "$9$10";
   $exponent = 10**$exponent;

   #only care about 0 and non 0 values
   $temp_voltage = int($result*$exponent);

   if ($temp_voltage > 0) {
      $prev_voltage = 1;

      print "#0 $sig_name = 1\;\n";
   }
   else {
      $prev_voltage = 0;
      print "#0 $sig_name = 0\;\n";
   }
}
else {
   #calulate current time (in ns)
   $result = "$1$2.$3";
   $exponent = "$4$5";
   $exponent = 10**$exponent;
   $tempresult = $result*$exponent*10**9 + $offset; #in ns


   #calculate current voltage value
   $result = "$6$7.$8";
   $exponent = "$9$10";
   $exponent = 10**$exponent;

   #only care about 0 and non 0 values
   $current_voltage = int($result*$exponent);

   #if ($k < 100000) {
   #    print "$current_voltage\n";
   #    $k++;
   #}

   if ($current_voltage > 0) {
      $current_voltage = 1;
   }
   else {
      #could be a negative voltage
      $current_voltage = 0;
   }

   #print "#$tempresult $sig_name = $current_voltage\n";
```

```
            if ($current_voltage == $prev_voltage) {
                #do nothing if the value did not change... may want to switch this around
            }
            else {
                #just want how many ns
                $current_time = $tempresult - $prev_time;

                print "#$current_time $sig_name = $current_voltage\;\n";
                $prev_voltage = $current_voltage;
                $prev_time = $current_time + $prev_time;
            }

        }
    }

};
```

REFERENCES

[EPC08] EPCglobal IncTM, EPCTM Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860MHz -960MHz Version 1.2.0 , October 2008.

[QL09] Qiasi Luo, et al. University of Science & Technology, Fudan University "A Low-Power Dual-Clock Strategy for Digital Circuits of EPC Gen2 RFID tag." Proc. of IEEE International Conference on RFID, USA, Orlando. April 2009.

[PP07] P. Peris-Lopez, J.C. Hernandz-Castro, J.M. Estevex-Tapiador and A. Ribagord, LAMED-A PRNG for EPC Class-1 Generation-2 RFID specification, Computer Standards and Interfaces, Elsevier (2007).

[AB07] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. "PRESENT: An Ultra-Lightweight Block Cipher." Lecture Notes in Computer Science 4727/2007 (2007).

[CT08] Ching-Cheng Tien, Chih-Hu Wang, Yi-Cheng Hong, Chih-Hao Chen, and Hsuan-Chih Lu. "Implementation of Novel EPC UHF RFID Tag with Sensor Data Transmission Capability." *Chung Hwa Journal of Science and Engineering* 6.4 (2008): 37-42. Print.

[DD08] Daniel Mark Dobkin. *The RF in RFID: Passive UHF RFID in Practice*. Amsterdam: Elsevier / Newnes, 2008. Print.

[PP09] Pedro Peris-Lopez, Julio Cesar Hernandez-Castro, Juan M.Estevez-Tapiador, and Arturo Ribagorda. An Ultra Light Authentication Protocol Suitable for Resource-limited Gen-2 RFID Tags. Journal of Information Science and Engineering, Vol. 25 No. 1, pp. 33-57, January 2009.

[AN07] Ansoft. RFID Radio Circuit Design in CMOS. 2007. <http://www.ansoft.com/RFID_whitepaper.cfm>.

[WI10] *WISP - Home*. Web. 14 Mar. 2010. <http://wisp.wikispaces.com/>.

[DB06] Daniel V. Bailey, Ari Juels. *Shoehorning Security into the EPC Standard*. RSA Laboratories. 23 January 2006.

[AM01] Alfred J. Menezes, Paul C. Van. Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. Boca Raton [u.a.]: CRC, 2001. Print.

[TF06] "Feds Ignore Claims That RFID Is More Harmful than Helpful | TechFreep World." *TechFreep | Daily Tech News and Free Press*. Web. 29 Mar. 2010. <http://techfreep.com/feds-ignore-claims-that-rfid-is-more-harmful-than-helpful.htm>. Photo Only

[WI10] "Radio-frequency Identification -." *Wikipedia, the Free Encyclopedia*. Web. 29 Mar. 2010. <http://en.wikipedia.org/wiki/Rfid>. Photo Only

[SA08] Syed Ahson, and Mohammad Ilyas. "Wisp: A Passively Powered UHF RFID Tag with Sensing and Computation." *RFID Handbook: Applications, Technology, Security, and Privacy*. Boca Raton: CRC, 2008. Print.

[HC07] Hee-Jin Chae, Daniel J. Yeager, Joshua R. Smith, and Kevin Fu. Maximalist cryptography and computation on the WISP UHF RFID tag. In Proceedings of the Conference on RFID Security, July 2007.

[DH08] Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. *Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses*. In Proceedings of the 29th Annual IEEE Symposium on Security and Privacy, May 2008.

[SC09] Shane S. Clark, Jeremy Gummeson, Kevin Fu, and Deepak Ganesan. *Towards autonomously-powered CRFIDs*. Workshop on Power Aware Computing and Systems (HotPower 2009), October 2009.

[MB09] Michael Buettner, Richa Prasad, Matthai Philipose, David Wetherall. *Recognizing Daily Activities with RFID-Based Sensors*. 11th International Conference on Ubiquitous Computing (UbiComp), 2009.

[RS06] R. Smith, Alanson Sample, Pauline Powledge, Alexander Mamishev, Sumit Roy. *A wirelessly powered platform for sensing and computation*. Joshua Proceedings of Ubicomp 2006: 8th International Conference on Ubiquitous Computing. Orange Country, CA, USA, September 17-21 2006, pp. 495-506.

[AM07] Adam S.W. Man, et al. Low Power VLSI Design for a RFID Passive Tag baseband System Enhanced with an AES Cryptography Engine. The Hong Kong University of Science and Technology. RFID Eurasia, 2007 1st Annual, 2007

[DB06] Bailey, Daniel V., and Ari Juels. "Shoehorning Security into the EPC Standard." *Security and Cryptography for Networks*. Vol. 4116. New York: Springer-Verlag Berlin/Heidelberg, 2006. 303-20. Print. Lecture Notes in Computer Science.

[DC05] Canright, D. "A Very Compact S-Box for AES." *Cryptographic Hardware and Embedded Systems - CHES 2005: 7th International Workshop : Proceedings*. Vol. 3659. Berlin: Springer, 2005. 441-45. Print. Lecture Notes in Computer Science.

[DS1620] *DS1620 Digital Thermometer and Thermostat*. Datasheet. Dallas Semiconductor/Maxim. Web. <http://pdfserv.maxim-ic.com/en/ds/DS1620.pdf>.

[AP10] Poschmann, Axel. "Present." *PRESENT*. Web. 01 July 2010. <http://www.lightweightcrypto.org/present/>.

[FT09] Truta, Filip. Photograph. *Apple Equips Prototype iPhones with RFID-Capabilities*. Softpedia, 6 Nov. 2009. Web. <http://news.softpedia.com/images/news2/Apple-Equips-Prototype-iPhones-with-RFID-Capabilities-Rumor-2.jpg>. Image Only

[IPJ08] *Speedway Reader.* Impinj. 2008. Brochure. Image Only