Cryptographic Coprocessor for Wireless Sensors

ECE 493 Final Report

Faculty Supervisor:
Dr. Kaps

Team (in alphabetical order):
Supreet Kaur
James McCall
Brandon Thomson

December, 2008

Department of Electrical and Computer Engineering
George Mason University
Fairfax, VA 22030

Contains materials from `www.digilentinc.com` and `www.xbow.com`

Executive Summary

This design document specifies the details of our project: a cryptographic coprocessor, implemented in an FPGA, for a MICA2 wireless sensor node. The FPGA is be used to host an AES encryption core that was provided to us by Dr. Kaps in VHDL. The project is being continued by another group for their Fall 2008-Spring 2009 as we did not complete the full vision for the project.

The FPGA device is mainly intended for prototyping and experimentation with various encryption algorithms as ASIC components which can provide AES encryption in realtime are already available. On the other hand, the work done for this project could help kickstart the development of ASICs suitable for other encryption algorithms that would provide fast encryption with maximal power efficiency. Despite this situation, we used various techniques to make the FPGA implementation as power-efficient as possible.

Contents

List of Figures

Acknowledgments

We would like to thank our faculty supervisor, Dr. Kaps, for introducing us to such an interesting project, for all the late nights, for helping with the technical details where we were deficient, for letting us focus on the areas that were most interesting to us, and for working around our sometimes intolerable schedules.

We'd also like to thank Dr. Gaj for giving us some very useful comments that have changed and clarified some critical details of this project.

1 Approach

We'll begin with a gentle introduction to the various aspects of this project and then ease into the details of our specific design.

Wireless sensor nodes (WSNs) are small devices equipped with an RF transceiver, a battery, multiple sensors, and typically a small microcontroller. The transceiver enables communication with other (often identical) nodes and a base station, while the microcontroller controls the communication and gives the nodes local data-processing ability.

Hundreds or thousands of these sensor nodes can collectively form mesh networks to facilitate monitoring, tracking, and surveillance applications in commercial, industrial, and military environments. A few specific applications include detection of gas leaks and pollution levels at chemical plants, early detection of forest fires or volcanic activity, and position tracking for the armed forces [1].

It's desirable that each node use as little energy as possible because this saves labor and enables the nodes to be deployed in locations where its difficult to change a battery. Remote wilderness locations, toxic and extreme environments, and even human or animal bodies are examples of places where sensor nodes could be useful but it would also be very challenging to replace a battery.

To minimize power consumption, it's necessary to carefully budget microcontroller clock cycle use and transceiver activity. These components can be powered nearly all the way off on most node platforms when they're not in use which means the primary determinant of battery life is the percentage of time the various components are active, or their "duty cycle." Naturally, effectively balancing the use of these components in a wireless sensor application to achieve minimum energy consumption is challenging.

For example, it's obvious that in the vast majority of cases it would be wasteful of both power and bandwidth for a mote to transmit every bit of the data recorded to every other mote and to a mains-powered base station with no processing or local logic at all. What's less clear is exactly how much local processing should be done on each mote to keep the transmitter power consumption low without inadvertently wasting power by running the microcontroller for too long. Like most engineering problems, there are obvious tradeoffs involved: do you want to focus on having a high frequency of updates, maximizing battery life, or getting the software working as quickly as possible?

When you add an FPGA (a device that can calculate extremely quickly but also uses a large amount of power) into the mix, things only get more complex. It should be clear, though, that any device which could potentially reduce the energy consumption of a wireless mote for a particular application would be a useful addition. We'll examine the use of an FPGA for an encryption application and provide a detailed analysis of the problem in the following section.

1.1 Applications for FPGA add-on

For some applications, an FPGA may be a suitable addition to the onboard microcontroller for local data processing. The high power consumption of FPGAs can often be mitigated by the fact that they run many algorithms orders of magnitude more quickly than microcontrollers, and thus may use less total energy than a microcontroller to perform similar tasks. Some FPGAs even have a "suspend" mode that allows them to retain their configuration while operating in a low-power state so they don't need to be reconfigured every time operation is resumed.

1.2 Mote Software

TinyOS is a free and open source operating system and support platform developed specifically for wireless sensor networks. It's written in the specially-developed nesC programming language and provides a common set of commands and libraries that can be used on multiple hardware platforms. The wireless motes we'll be using for this project are supported by TinyOS 2.x, and so we'll be using it extensively. Details about nesC are discussed in §1.7.

1.3 Rationale for Encryption on WSNs

Many potential applications of wireless sensor nodes can benefit from message authentication and confidentiality. For instance, a sensor node deployed in a critical military network with a remote command interface should only accept commands from authorized users. Likewise, it may be undesirable for unauthorized users to be able to view data the sensor nodes are collecting. Encryption can assist with both of these goals.

There are two major classes of encryption: symmetric and asymmetric. In general, symmetric-key encryption is only effective if the symmetric key is kept secret by both parties involved in a communication. For this reason, it is not ideal for deployment in wireless sensor network scenarios where individual motes may be compromised and a private key recovered by an attacker.

Asymmetric (or public-key) encryption involves a pair of keys, public and private. Each public key is published while its corresponding private key is kept secret. Asymmetric encryption allows parties to disguise information they send to each other, to ensure data has not been modified in transit, to confirm a sender's identity, and to prevents a sender of information from claiming at a later date that the information was never sent. An additional benefit for WSNs is the fact that each mote need only contain the public keys of other motes and any base stations, which would limit security risks if some motes were collected and their contents analyzed by an attacker [2][3].

This has been only a brief discussion of the reasons for encryption and security in WSNs. A full analysis of the benefits and drawbacks of symmetric vs asymmetric cryptography and their applications for WSNs are outside the scope of this document. For more information about the implementations and their implications, see [2] and [4].

1.4 Project Overview

This project really contains two separate subprojects: one using the FPGA on a Digilent starter board, and one using the FPGA on a custom add-on board for the mote that we designed. While there are some differences, both are configured to handle encryption tasks in response to requests from the microcontroller. Due to size limitations, the FPGA module on the custom PCB can only use one encryption algorithim: AES/Rijndael. The FPGA is useful because it can encrypt data at a higher speed and with higher energy efficiency than the Atmega128L microcontroller on the mote. We note also that there are many other types of algorithms that are particularly suited to operation on FPGAs. Our implementation supports FPGA reconfiguration by JTAG so other uses will simply require a new FPGA bitfile and new code for the microcontroller.

Considering the nature of sensor nodes and the specified constraints, we determined that the best way to implement the FPGA expansion board was to purchase a low-power, discrete FPGA and design a PCB for direct connection to the sensor nodes through the Hirose 51-pin expansion connector on the motes.

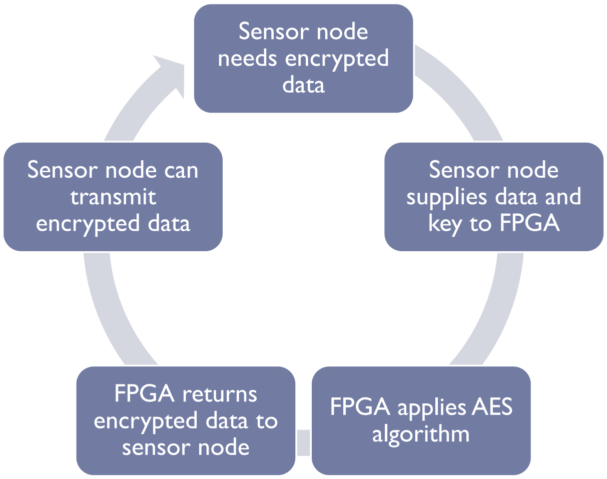

Sensor node needs encrypted data
Sensor node supplies data and key to FPGA
FPGA applies AES algorithm
FPGA returns encrypted data to sensor node
Sensor node can transmit encrypted data

Figure 1: A high-level overview of the communication process.

Initially we considered using Block RAMs on the FPGA to buffer incoming data so it was available locally when requested by the encryption code. After some discussion we concluded that it would make sense to send the data "just in time," especially since for AES the bytes for the key and the data are requested by the core in the same order each time. In this case discrete chunks of the data and the key will go across the bus alternately until all of the data has been transferred. We acknowledge that it is wasteful of both power and transmission bandwidth to repetitively send data over the serial bus in this way, but since bandwidth will not be a constraint and this will simply the overall design immensely it seems like a worthwhile tradeoff. A total of 256 bits will be transferred for AES; 128 bits for the key, and 128 bits for the data.

In figure 2, we show the dataflow for the system. Initially data is sampled from a sensor to memory in the microcontroller. Next, the data passes over a USART link from the microcontroller to the FPGA for encryption, and the encrypted data passes back over the USART to the microcontroller. The microcontroller then sends the data to the CC1000 for transmission, and the data is received by an identical CC1000 on a base station mote. The data finally goes through the mote's microcontroller and out over USB to a host PC where a custom Java application can be used to access and decrypt the data.

As mentioned in [2], there is some concern about having key data travelling over an open bus if the

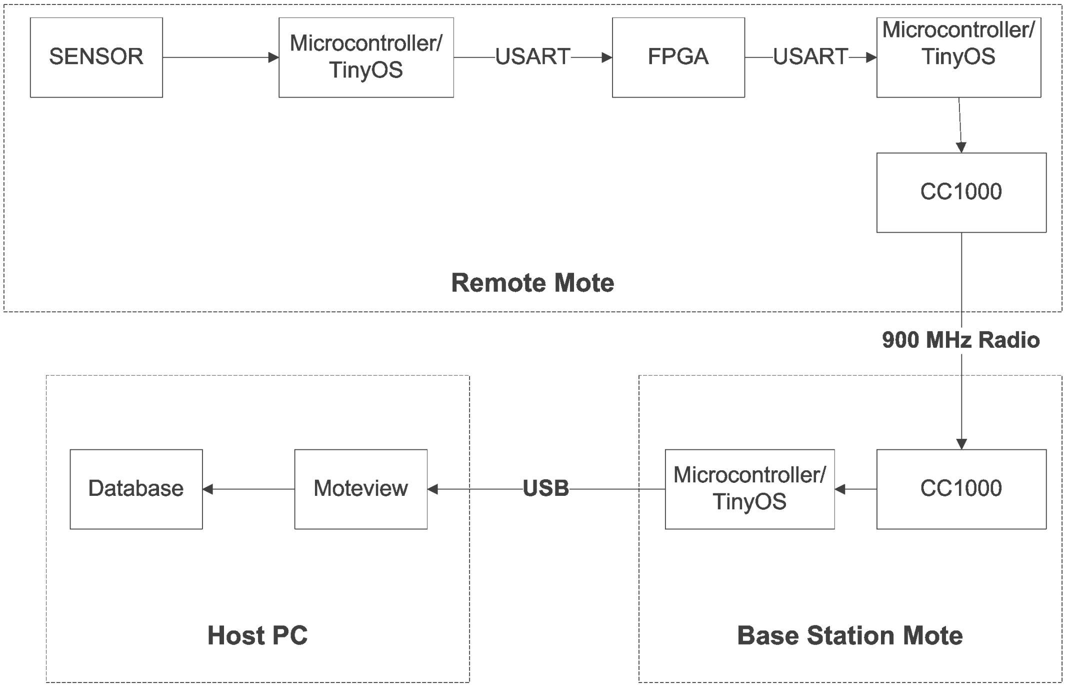


Figure 2: A representation of the dataflow during device operation.

nodes are to be deployed in an untrusted environment. Our design does not make any effort to protect the key data, and so using a public-key core like Rabin would be essential if good security were desired in this situation. Unfortunately the size of the FPGA used on the PCB is insufficient for an algorithm of this level of complexity.

1.5 Design Requirements

We identified some key requirements in the early design phase:

- Develop communication protocol between MICA mote and FPGA
- Enable AES encryption on MICA2 platform with FPGA
- Develop software platform to demonstrate encryption
- Test with S3E Starter Board
- Increase speed compared to calculation on CPU
- Decrease CPU memory utilization
- Design portable, production-quality FPGA support platform
- Cost under $100 in mass-production
- Appropriate size for deployment with sensor node
- Power consumption profile:
- Suitable for battery operation
- Superior to CPU for encryption task

1.6 Power Consumption

To determine whether it was really feasible to power an FPGA from two AA batteries, we examined the power requirements of a Spartan 3AN and it's necessary support circuitry (a voltage regulator and perhaps a DC–to–DC converter, depending on the number of battery cells used). We determined that this was possible, but barely.

Based on the FPGA used in the project, the XC3S50AN, an order-of magnitude estimate is that the device and its support circuitry will dissipate about 3W of power when operating in full active mode. To make the math simple, let's assume 3 battery cells in series, which means 1W of power would need to be provided by each cell.

Since Duracell alkaline AA batteries have characteristics similar to other alkaline batteries, we've used the Duracell AA alkaline datasheet to determine some general performance characteristics for AA batteries. Alkaline batteries operate over a fairly wide voltage range from 0.8V to 1.5V. If we average it out to assume that most of the time they are operating at 1.1V, then roughly 909mA of current will be needed from each battery on average. A constant 1 A drain would cause the batteries to be depleted extremely quickly, in less than one hour (total) of operation.

It looks like 3 AA batteries are not an appropriate power source for the FPGA, and that means the 2 that come with the mote are definitely not appropriate. We'll need to use a separate, higher capacity power supply for the FPGA. 3 D cell batteries would last longer; roughly 8 hours for 1 A of constant drain. We would not be honoring the requirements of keeping the size of the mote device small if we used multiple D cell batteries to power it, but this tradeoff may be unavoidable. 3 D cells are not likely to save much space vs 4 D cells, and including an extra cell would increase runtime significantly, so 4 D cells is probably the magic number.

Of course, the FPGA will not be operated at the full power consumption level for 3 straight hours: the idea is to only turn it on only very briefly when an encryption operation is desired and then turn it off again as quickly as possible. What follows then is a short analysis of the throughput of the device for AES and Rabin.

1.7 nesC Program Structure

The nesC language is essentially C with some added constructs for object-orientation and enabling concurrency. All nesC applications consist of one or more components assembled (or wired) together statically to form an executable image. The components provide and use *interfaces*. "Provided" interfaces represent the functionality the component provides to its user, while "used" interfaces represent the functionality the component needs to perform its job [5]. For more information about nesC and why we used it, see the proposal (starts on page 25).

In figure 3 we show an automatically generated flow diagram for our USART test code created using the free GraphViz tool. Since large, well-designed nesC applications tend to have a little bit of code spread out over many small text files, it can sometimes be hard to grasp the purpose of the various linkages all at once. The use of the GraphViz tool to create these component maps makes the code layout easier to understand.

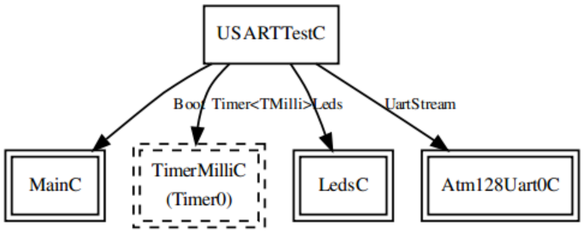


Figure 3: Automatically-generated flow diagram for our USARTTest application.

While we did use a 500kbit/s serial link, it was not the fastest serial implementation supported by the mote's microcontroller: 1Mbit/s is available. Due to some problems getting this baud rate supported on the FPGA side we stuck with the second highest speed.

There are also some small delays in between bytes the mote sends when the serial line is idle. Removing these delays could increase transmission speed by 10-20%.

2 Design

For the primary project we have used the Digilent Spartan 3E Starter board (provided by Dr. Kaps), shown in figure 4.

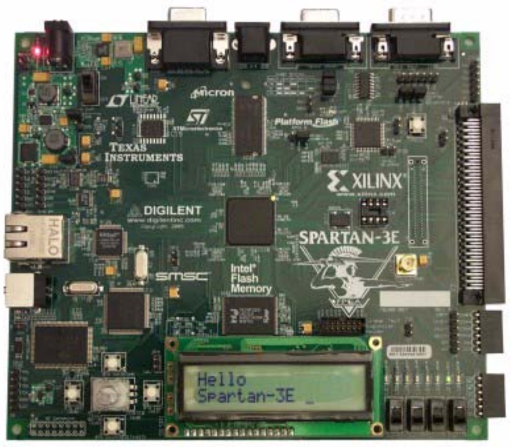


Figure 4: The Digilent Spartan 3E Starter Board (©2007 Digilent)

We used this board to ensure that our USART interconnections work, that the software developed for the motes is effective, and that the cores implemented on the FPGA return the expected results for the keys and data that are input. For more details about the board, see the proposal (starts on page 25).

2.1 VHDL Design

As we did decide to do serial transmission, we had to implement a UART on the FPGA. We did have a choice in implementing synchronous or asynchronous communication, but this was not a difficult decision, because using synchronous communication means sharing the same clock between the sensor node and our PCB. Therefore, the FPGA rate would have been slowed, because the mote runs on an 8 MHz oscillator, and the FPGA runs on a 50 MHz oscillator. In using asynchronous communication, all we needed to do was send start and stop bits attached to each byte for handshaking between the two components; therefore, we opted to use asynchronous communication.

We originally tried opening source code, however in all the designs we found, the baud rate was always set to one specific value. This ended up becoming a problem, because the baud rate was not easily modified for our application. After spending too much time trying to fix other peoples UART codes, we decided to design our own unit which would be tailored for our needs. We also wanted to make our unit portable in the fact that the baud rate could easily be changed, because we werent sure how fast we were going to be able to get the baud rate at the time, but also so it could be used in other applications. Therefore the UART we designed consists of 3 files which are a baud rate generator, a transmit unit and a receive unit.

The baud rate generator was not a very complex unit, because all that really needs to be done is divide the system clock by a certain amount to give the baud rate that is needed. However, since we wanted multiple rates and the mote could send rates up to 1 MHz, there is an initial operation which divides the system clock using a basic counter. The value the counter counts to can be changed in the package file which creates baud rates ranging from 1.2 KHz to 3.125 MHz. Without this initial divisor, the baud rate would be 3.125 MHz, so it was already needed to slow the rate down some to work with the sensor node. The second counter is used for sampling, so it just divides the system clock a second time after the divisor by 16, because each bit is sampled 16 times to check for a new start bit.

There is a third counter which is used for synchronizing. Since the baud rates of the two components dont match, timing errors develop after each bit is sent through. However, as long as the timing error is small enough, less than 50% per byte, the errors will not affect the operation of the unit. However, since the timing errors do accumulate, they need to be fixed, because they will accumulate and cause mistakes no matter how small it is. The third counter serves the purpose of re-synchronizing the baud rate generator after each byte is sent to ensure proper operation.

The receive and transmit units behave in much the same way; they turn bytes into bits and vice versa using a shift register. I originally made a separate file that implemented an 8 bit shift register, but then I realized I can just as easily shift in each bit by just implementing it in my state machine and just looping through the same state until the shift register is full. Therefore the receive unit is coded solely by using a state machine. It had a total of 6 states which included error checking, but this was not needed so I changed it back to 4 states. The receive unit state machine is shown in figure 5.

As can be shown from the figure, the receive unit will stay in the idle state until a start bit arrives from the sensor node. Once this happens, the operation goes to the start state where it checks to make sure the shift register is not full. If it is full, then the control instead goes to the done state, and if is not full then it goes to the shift state. The shift state shifts each new bit into the least significant bit position of the shift register. The RxRdy signal goes high when the shift register is full which tells the system that the receive unit is ready to send the byte out. Therefore when RxRdy = 1, the control will go to the done state. The control stops in this state for 1 cycle to send the byte out and set the ClrDiv bit which goes to the baud rate generator to re-synchronize the system.

The transmit unit was also implemented with a state machine which is shown below in figure 6. It basically has the same characteristics of the receive unit, the only difference being that it starts with a full shift register and each bit needs to be shifted out on the serial line to the mote.

As seen in the figure, the transmit unit stays in an idle state until it gets a load signal from the top level control. When load goes high, a byte is coming in from the RAM. The operation then goes to the load state for one cycle which loads the 10-bit shift register. The reason this is a 10-bit register is that

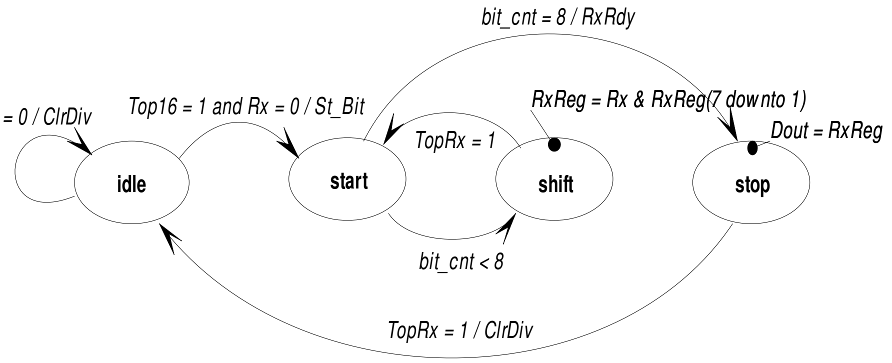


Figure 5: Receive unit state machine

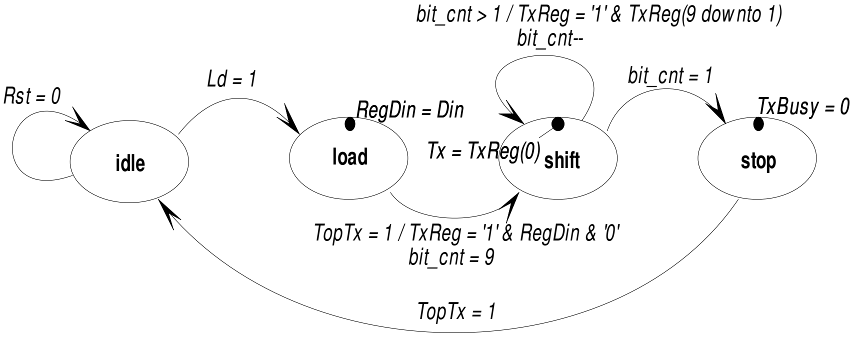


Figure 6: Receive unit state machine

the start and stop bit needed to be added to the byte so the mote can detect when data is coming in. The load state also sets the TxBusy signal which tells the system that the transmit unit is busy and it cannot accept new data yet. The control then goes to the shift state where the least significant bit of the shift register is sent out on the serial line. The transmit unit loops through these states until the last bit is shifted out, and it then the control goes to the done state. This state turns off TxBusy which notifies the system that transmit can accept a new byte, and the state machine returns to idle.

The purpose of this portion of the project was to get the AES encryption design and the UART working properly together on the FPGA. This constituted a significantly difficult portion of the project to implement properly. The block diagram of the top level system that was downloaded to the FPGA is shown below in figure 8 and the state machine for the control is shown below in figure 9.

As seen from the block diagram, we treated the AES and mote as black boxes with inputs and outputs, and the RAM, control unit, and the UART are also included. We did need to learn a little about the AES to ensure that it sends and receives data properly and communicates with the rest of the system, and the RAM was included to ensure that the data was received and sent to the mote with correct bytes and in correct order.

Referring to the state machine in figure 9, the system stays in a pause state until a start bit is detected from the sensor node. Once this happens the system goes to the receive state which controls the behavior of the receive unit shown above in figure 5. Each time a byte is ready in the receive unit which is signaled by the RXRdy signal, the system will go to the UartRam stage which loads the byte from the receive unit into the RAM. Since the mote sends 32 bytes to the FPGA, which constitutes 16 bytes for data and 16 bytes for the key, the system loops through the receive and UartRam states 32 times collecting all the bytes from the mote. Once the RAM is full, the system will go to the compute state which sends an enable signal to the AES unit turning it on to encrypt. All the bytes are sent into the AES, and the AES processes the data. It takes approximately 514 clock cycles to perform the encryption.

Using a counter to count the encryption cycles, the system then goes to the AesRam state which controls the process of returning the encrypted bytes to the RAM. The key is not needed anymore so only the 16 bytes of encrypted data is returned. The counter in the AesRam state does go to 18 which can be a bit confusing, but it was needed to ensure correct operation. This is because a write signal from the control of the AES unit is asserted a couple cycles early, and rather than modify the AES, I just made the counter count to 18 to receive the correct data.

Once the RAM has the 16 bytes of encrypted data, the AES enable signal is turned off and the system then moves to the RamUart state. This state loads each byte from the RAM to the transmit unit, so it only stays in this state for one cycle, then it goes to the transmit state. The RamUart and transmit states controls the behavior of transmit unit shown above in figure 6. Therefore the system stays in the transmit unit until TxBusy goes low, which tells the system to return to the RamUart state and load another byte. However, the transmit state also detects if this is the 16th byte, and if it is, the system returns to pause because that was the last of the data. Once the mote receives the last byte from the PCB, it sends a done signal which returns the FPGA to suspend mode to save power. The system is once again in sleep mode until a new start bit signals new data coming in.

2.2 Software

The software is designed to provide communication interface between all the hardware components. One of the sensor node is placed remotely connected with FPGA and the second sensor node is connected to the host PC via USB port. Mainly for the purpose of testing the successful operation of all the components, three programs are written:

2.2.1 UART and Data Gathering application

The remote sensor node, connected with the FPGA add-on module is programmed with this application. The flowchart of the program execution is given in figure 10. The sensor node waits for a command from the base station. When it receives command, the code executes to take data samples from the photocell on the sensor board connected with the remote FPGA. Sixteen samples are taken by the sensor node (the number of samples being taken can be changed according to the user demand). Once the data is collected, it is placed in an array. According to the command data, for just retrieving data from sensor node, or aes, for obtaining data in encrypted form-the code transfers the collected data to the proper medium of communication.

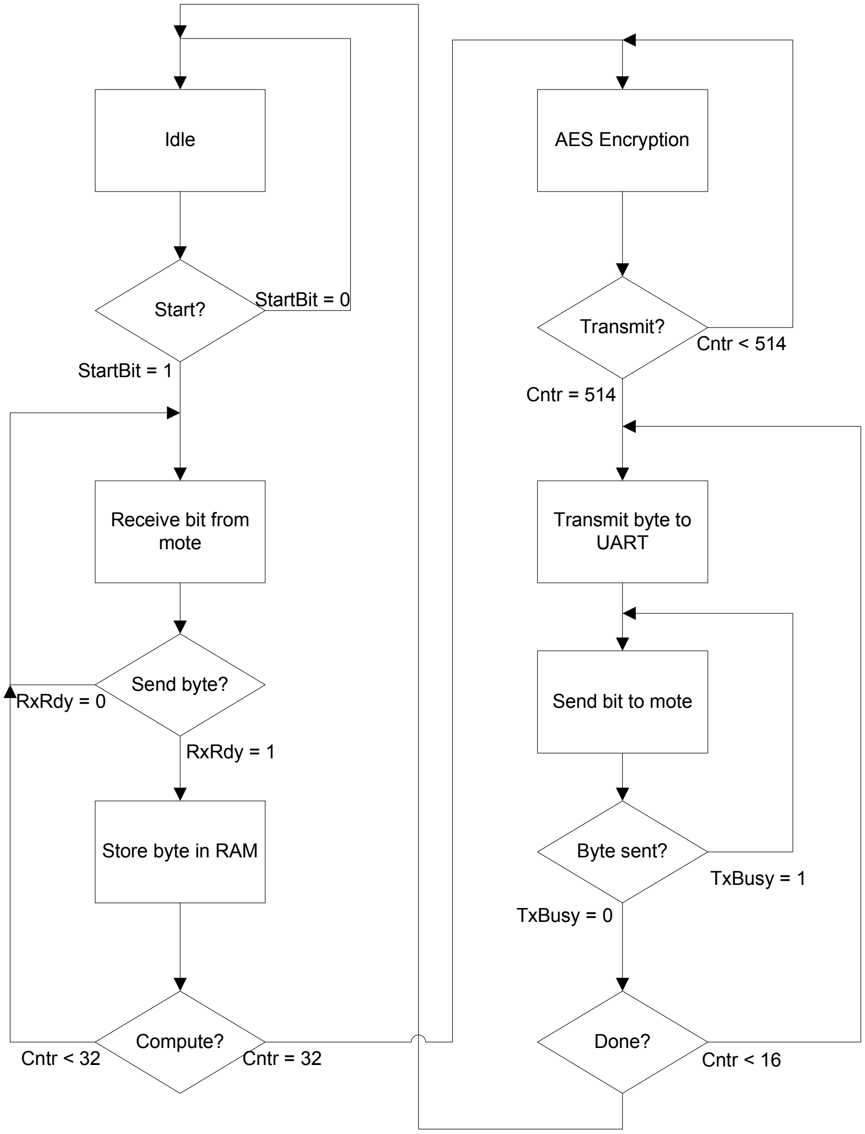


Figure 7: ASM chart for the top-level state machine.

•

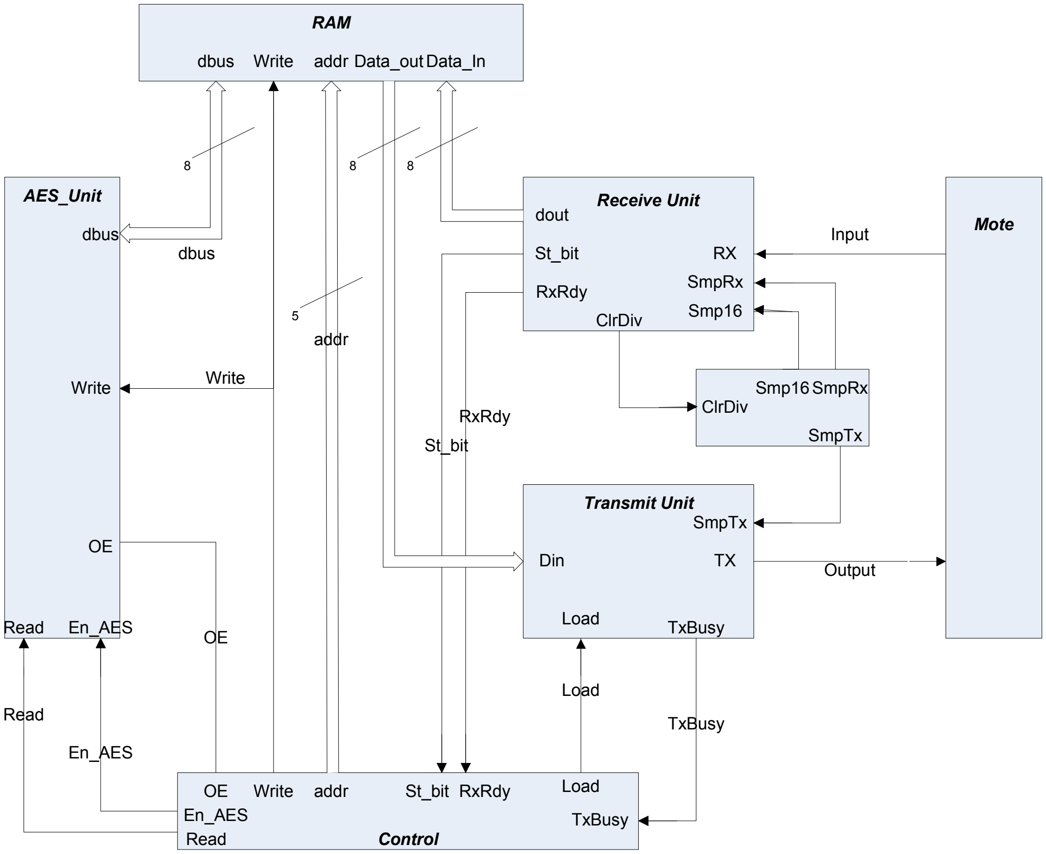


Figure 8: Block diagram of the FPGA

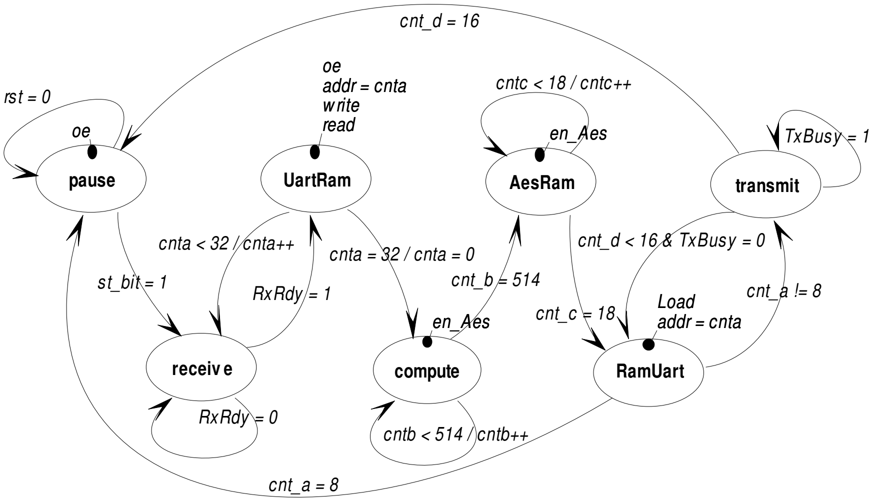


Figure 9: Top level state machine.

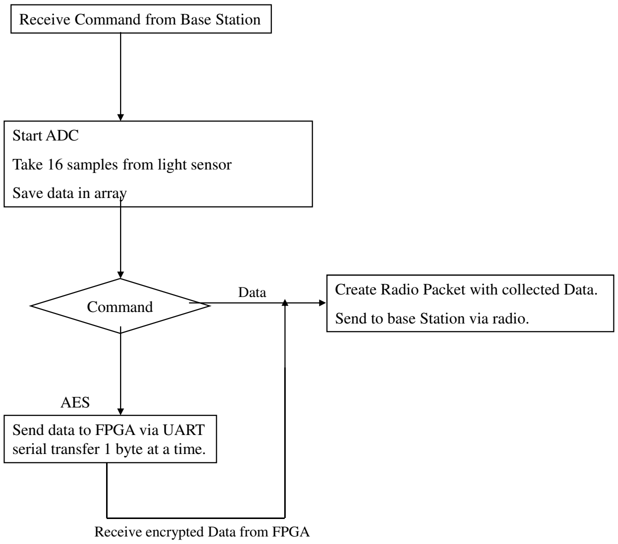

Receive Command from Base Station
Start ADC
Take 16 samples from light sensor
Save data in array
Command
Data
Create Radio Packet with collected Data.
Send to base Station via radio.
AES
Send data to FPGA via UART serial transfer 1 byte at a time.
Receive encrypted Data from FPGA

Figure 10: Flow chart for UART-DATA Application.

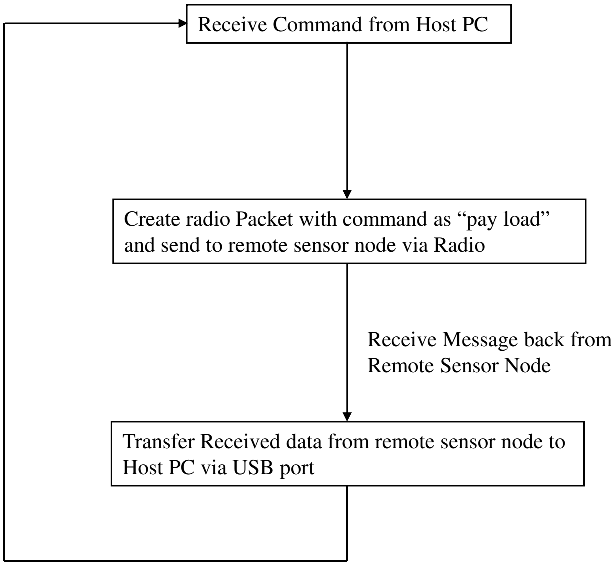

Receive Command from Host PC
Create radio Packet with command as “pay load” and send to remote sensor node via Radio
Receive Message back from Remote Sensor Node
Transfer Received data from remote sensor node to Host PC via USB port

Figure 11: Flow chart for base station.

Command = data The collected data is placed in a packet (packet is predefined by the user) and sent to the sensor node connected at the base station. The radio communication utilizes 916.7 MHz frequency. The wireless sensor nodes manufactured by Crossbow use this radio frequency for wireless communication. The communication via radio is unreliable but acknowledgements are sent for each packet sent.

Command = aes The collected data is transferred to the FPGA via UART on the sensor node. The URAT transfers data serially, 1-byte at a time. The start and the end of each byte is denoted by start bit and stop bit. A total of 32-bytes, which include the data and the AES encryption key, are transferred to the FPGA. FPGA runs the AES algorithm and returns a 16-byte encrypted data. Once the data is received from the FPGA, sensor node sends the encrypted data to the base station sensor node via radio communication.

2.2.2 Base Station Application

The sensor node connected with the base station is programmed to provide interface for both radio and serial communication. The application acts as a bridge between the remote sensor node and the host PC. The flow of the code is given in figure 11.

Serial Communication The sensor node communicates with host PC via serial ports. The serial communication is slow with the baud rate for Mica2 of 57600. The sensor node is programmed to receive and send serial packets to the host PC via USB port on the Host PC. The sensor node is connected to one of the USB ports using which the sensor node is able to send packets to the PC. The sensor node, however, receives packets on the next USB port on the PC (as defined by the PC).

Send packets to PC The code uses user defined packet definition which allows sensor node to send one byte per packet to the PC. The data received by the base station sensor node from the remote sensor node is an array of size sixteen (decided by the user). When base station mote receives the data, it sends the data to the PC by sending each element of the data array one at a time. This implementation solved the speed conflict between the radio communication and the serial communication.

Receive packets from PC When sensor node receives a serial packet from PC via USB, it sends it as a radio message to the remote sensor node. According to the application design, the received data from the serial port is always a command entered by the user for the remote sensor node.

Radio Communication The base station requires radio communication to transfer messages between the PC and the remote sensor node.

Send packets to radio The base station mote, upon receiving command from the PC (in the serial packet), creates a radio packet (defined by the user) and sends to the remote sensor node.

Receiving packets from radio The base station receives radio packets from the remote sensor node which contain the data demanded by the user at the host PC. The mote receives the data and stores in an array, which is then returned to the PC via serial communication.

2.2.3 User Application

The user application is a simple java application running on the host PC which allows user to command the remote sensor node via base station sensor node. The command entered by the user (data or aes) is sent to the base station via USB port at which the sensor node is connected with the MIB500 programming board and then data is received, after execution cycle, from the base station sensor node via another USB port on the PC, which is decided by the hardware on the PC.

The UART and data application and the radio and serial communication applications are programmed using nesC. nesC is an event based programming language, which is an extension to C programming language. The nesC code links the hardware components being used for the application with the interfaces.

2.3 Custom PCB

We initially envisioned a fully-custom PCB, approximately the same size as the mote, that would be used to host the encryption core. The benefits of designing a PCB for a discrete FPGA are two-fold: power consumption is much lower than the demo board, and the size is more appropriate. Both are important for wireless sensor nodes which are small and are expected to be deployed for long periods of time without recharging.

While we did successfully design and prototype a PCB, unfortunately we began to run short of time just as we entered the testing phase, and there certainly was no time for a second revision. Despite the time shortage, we believe that the design is solid and that it would be a very effective add-on module for the FPGA with only a few minor tweaks and revisions. Some details about the finished PCB follow.

Texas Instruments developed a triple voltage regulator specifically for the Spartan 3 series which supplies the correct voltages to the FPGA and takes care of many complex regulation issues. While soldering this tiny, tiny chip turned out to be somewhat of a nightmare, the time and energy it saved during the design process was well worth any hassle. The finished circuit design for the regulation circuit is shown in figure 12 and is based heavily on work by Andrew Greensted[6]. The parts used were of course completely different since we did not use a UK-based parts supplier, and this necessitated using some parts from existing libraries and CADding out some parts from scratch based on manufacturers datasheets and other sources. One thing we learned during this process was that product datasheets are not always accurate or readable!

The expansion board uses the same 51-pin expansion connector as the mote so that it can be plugged directly into the mote. This adds to the convenience, appearance, and sturdiness of the device as there are not a lot of unsightly wires and volatile connections to constantly worry about.

To reprogram the FPGA, the user needs to load a special program on the mote which sets the configuration pins on the FPGA correctly and then powers on the device. A standard JTAG cable can then be used to program the nonvolatile memory included in the FPGA package. Once configuration is complete, the program on the mote can be changed back to the standard program used for normal operation.

We considered including configuration jumpers on the board such that the user would not need to load a program on the mote in order to reconfigure the FPGA. The problem was that three configuration jumpers would be required and there was not any room to add these jumpers on the board without increasing the physical dimensions of the board. Engineering is always about tradeoffs, and in this case we decided that the size of the board was more important than easy configuration ability. Besides, for our application the image *never* needs to be changed after it is loaded for the first time!

Choosing the best size for the FPGA was challenging as well. Larger sizes offer more flexibility for simulating larger cores (like the public-key Rabin, and any others that Dr. Kaps might like to try) but also consume significantly more power. Another big issue is that most large FPGAs are only available in a ball-grid array pattern which is impossible to solder by hand. Some “breakout”–type boards for ball grid array packages do exist, but they are large, unwieldy, and expensive. After much consideration we

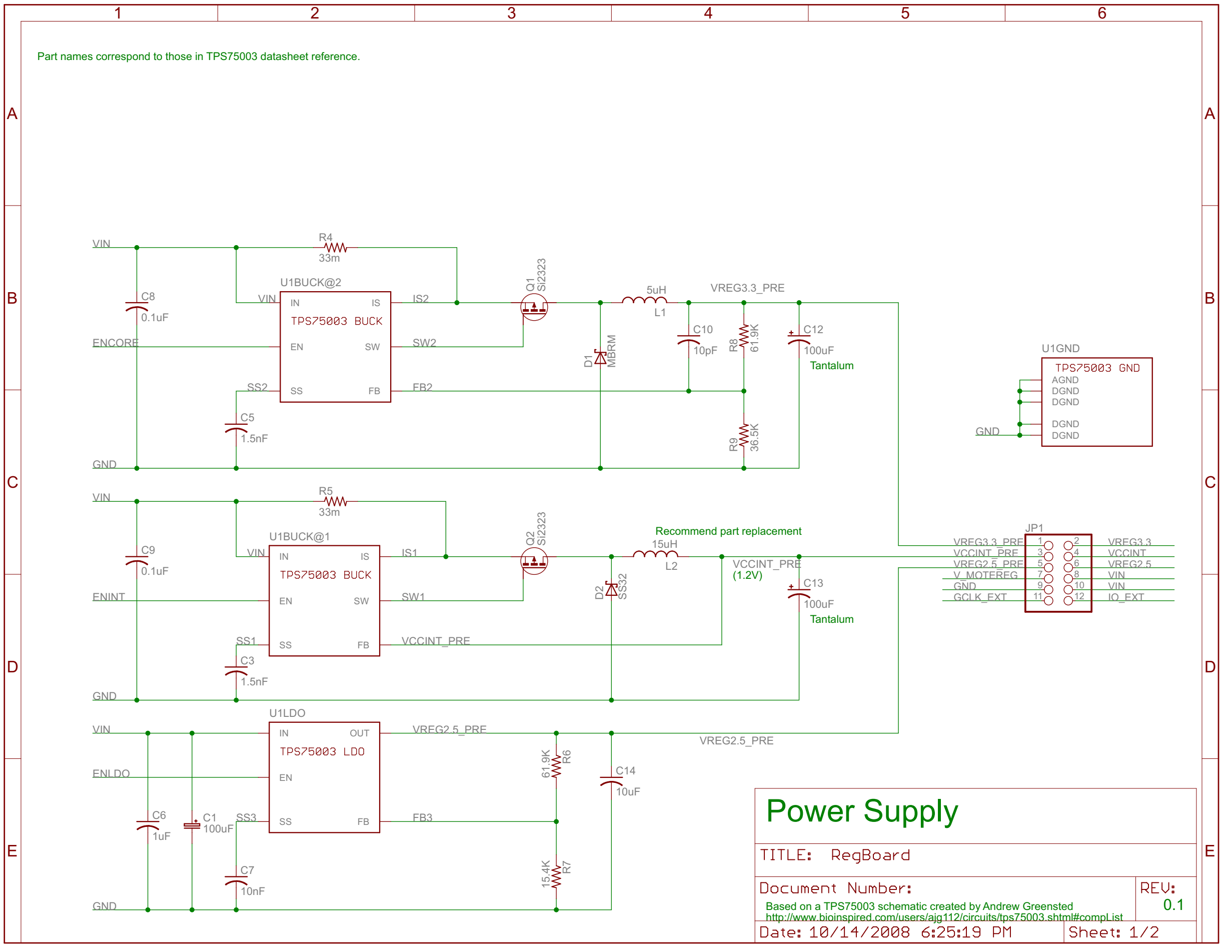


Figure 12: Schematic of the triple voltage regulator circuit.

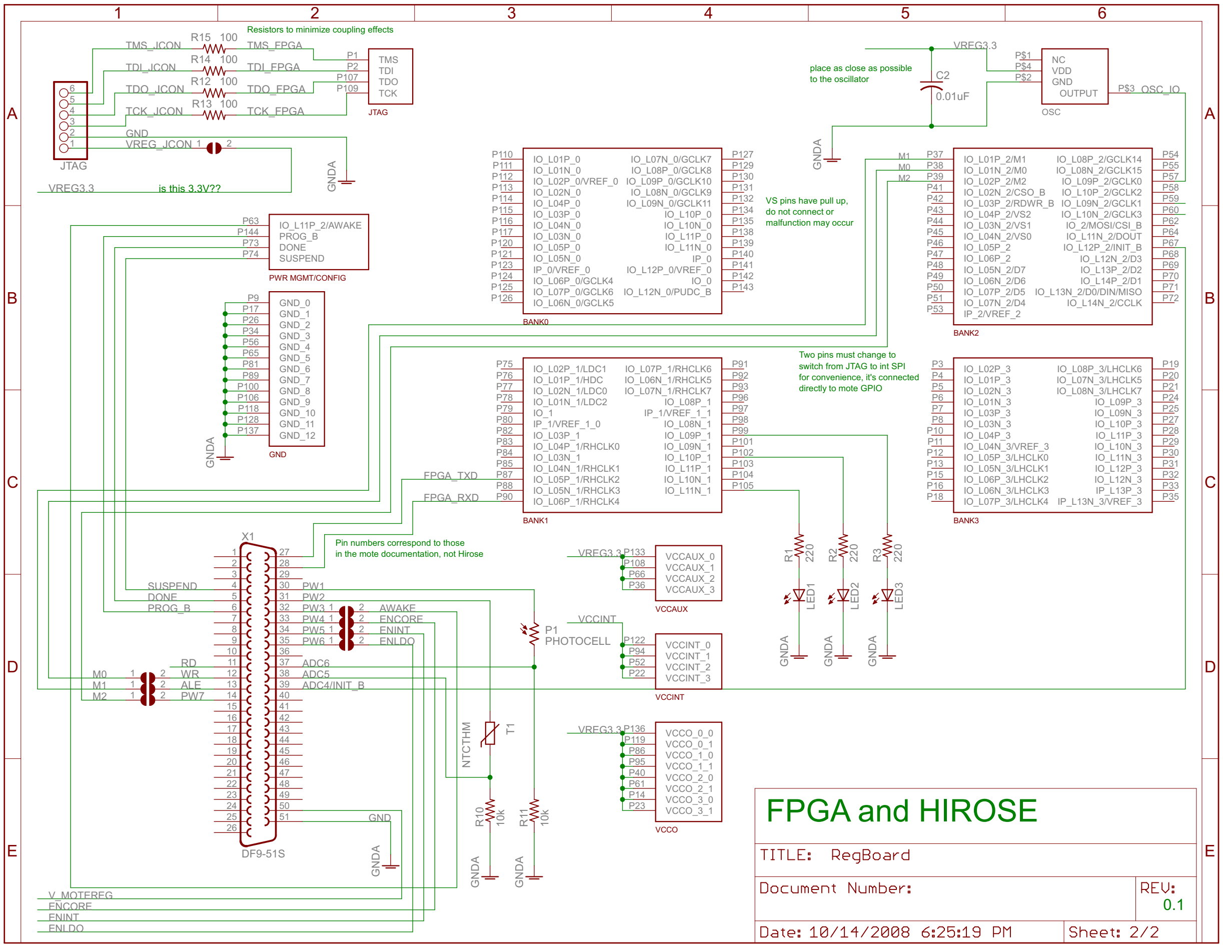


Figure 13: FPGA Schematic

Item	Quantity per board	Unit Cost	Total
50 Mhz Oscillator	1	$2.97	$2.97
6-pin, single-row header	1	$0.30	$0.30
12-pin, dual-row header	1	$0.74	$0.74
Green LED	3	$0.14	$0.42
100uF Tantalum capacitor (large, low ESR)	2	$0.67	$1.33
100uF Tantalum capacitor	1	$1.02	$1.02
10uF ceramic capacitor	1	$0.39	$0.39
1uF ceramic capacitor	1	$0.03	$0.03
1.5nF ceramic capacitor	2	$0.02	$0.04
10pF ceramic capacitor	1	$0.03	$0.03
0.01uF ceramic capacitor	2	$0.16	$0.32
0.1uF ceramic capacitor	2	$0.01	$0.02
0.033ohm current sense resistor	2	$0.37	$0.74
61.9kohm resistor	2	$0.04	$0.08
15.4kohm resistor	1	$0.04	$0.04
36.5kohm resistor	1	$0.04	$0.04
220ohm resistor	3	$0.03	$0.09
100kohm resistor	2	$0.02	$0.05
5uH Power inductor	1	$2.06	$2.06
15uH Power inductor	1	$0.66	$0.66
PMOS SI2323	2	$0.56	$1.12
Schottky Rectifier SS32	1	$0.45	$0.45
Jumpers	2	$0.11	$0.22
Hirose Connector	1	$3.13	$3.13
TPS75003	1	$2.95	$2.95
Spartan 3 AN FPGA	1	$12.06	$12.06
Photocell	1	$1.79	$1.79
Thermistor	1	$0.45	$0.45
Schottky Rectifier	1	$0.26	$0.26
Shipping	1	$5	$5.00
PCB manufacturing	1	$17	$17.00
Total	44		$55.79

Table 1: The costs associated with fabricating and populating one board.

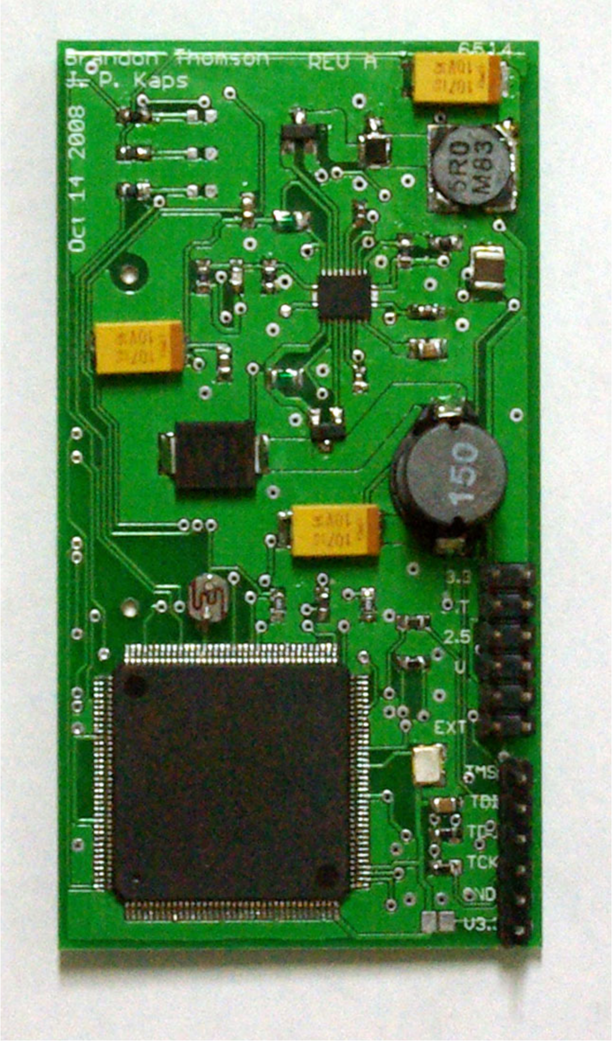


Figure 14: The front of the finished PCB device.

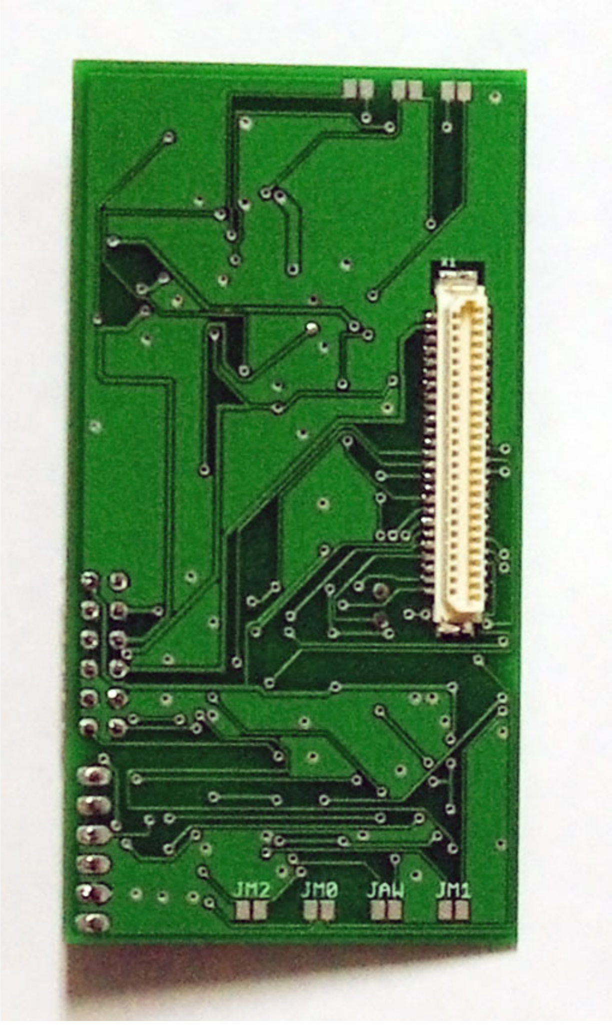


Figure 15: The back of the finished PCB device.

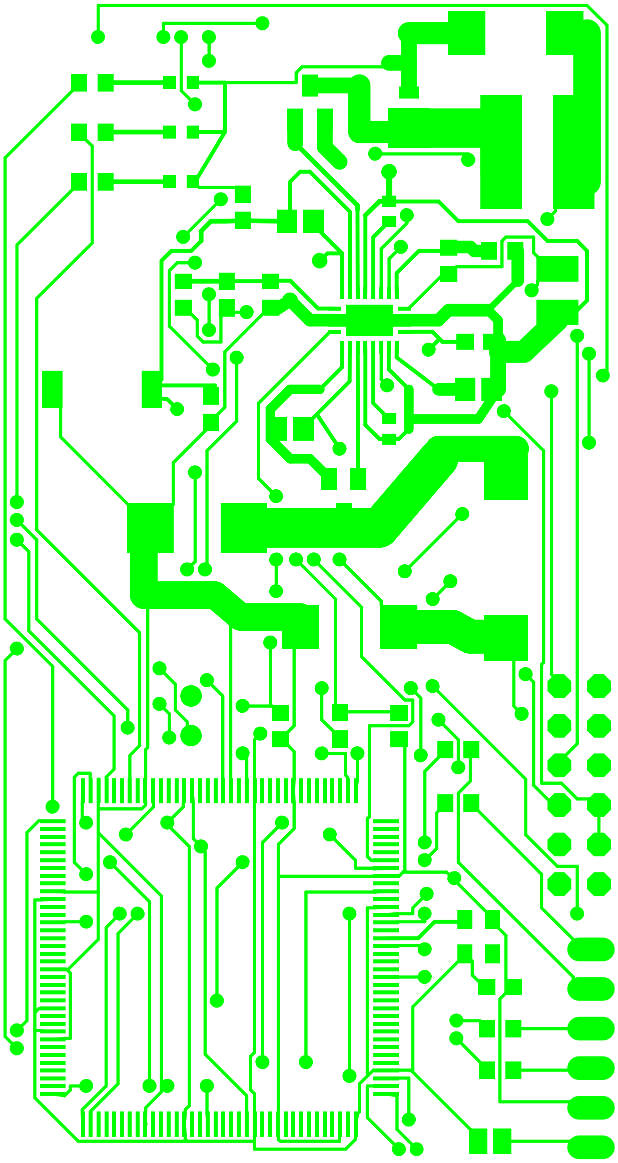


Figure 16: The top copper layer as a gerber file.

have decided that any FPGA package that uses ball grid array pins will not be acceptable for this project.

2.4 Voltages

Spartan-3 devices are designed and characterized to support various I/O standards for V_{CCO} values of +1.2 V, +1.5 V, +1.8 V, +2.5 V, and +3.3 V. The ATMega128L (V_{cc} = 3.0 V) supports an input low voltage from -0.5 V to 0.2V_{cc} (0.6 V) and an input high voltage from 0.6V_{cc} (1.8 V) to V_{cc} + 0.5 V (3.5 V). The output low voltage is 0.5 V and the output high voltage is 2.2 V. We anticipate that if the FPGA IO pins that are used to connect to the mote are configured in TTL logic mode, there will be no voltage-related issues that hamper communication between the devices [7],[8].

2.5 Device Communication Interface

After much comparison between possible communication methods, we concluded that a USART connection operating in asynchronous mode would be the ideal communication method, mostly because it allows us to operate the mote and the FPGA from unsynchronized clock sources.

To implement the UART, we initially planned to use (and modify if necessary) code written by someone else and made available as open-source on `www.opencores.org`. Eventually we decided to write our own UART as it proved to be difficult to understand the opencores code (which didn't really have sufficient documentation).

Being able to use unsynchronized clock sources is a huge advantage because we are then able to use the 8MHz clock provided by the ATMega128's internal oscillator rather than the $\tilde{7}$MHz external oscillator provided by the mote (which is the only clock available on the data bus). The 8MHz oscillator consumes less power and is recommended by Crossbow for battery-powered operation.

2.5.1 About USARTs

A USART device's maximum possible data rate is based on its input clock rate: the input clock rate divided by 16 gives the maximum data rate (or divided by 8 for DDR). On the FPGA we are limited by the maximum speed of the cores (slightly less than 80MHz) and the available oscillator (can be changed), while on the mote we are limited by the maximum speed of the microcontroller oscillator (8 MHz). A

64 MHz oscillator for the Spartan 3e starter board would probably be ideal because it divides evenly to 8MHz for higher rate UART communication, allows a higher FPGA calculation rate, and costs only $3 from DigiKey (SG531). As an alternative to using an external oscillator, we could consider simply using a DCM on one of the FPGAs to create a modified clock signal at the appropriate frequency.

It's also worth noting that the maximum transmit rate for the mote is only 38.4K baud and thus even if the microcontroller can talk to the mote at the higher rate it can't actually send the data anywhere. Clearly then the FPGA can be run at a somewhat higher speed than is necessary for continuous transmission of data with this particular hardware.

It was also necessary to synchronize the UART settings between the ATmega and the VHDL code. We eventually chose a configuration of 500kbaud with an unorthodox 8-1-1 data/start/stop configuration (8-N-1 is more typical).

On the other hand, since demonstrating the high throughput of the FPGA is an important part of the project, it makes sense to try and get as high of a bandwidth link between the FPGA and the microcontroller as we can even if we can't transmit the resulting data in realtime from the microcontroller.

2.6 Pin Functions

This section lists the functions of and our purposes for the specific FPGA pins we'll be using.

PROG_B When pulsed low, this pin causes the FPGA to reprogram from one of multiple sources. With the Spartan 3AN, it can be used to configure the device from an internal flash memory source. With the Spartan 3E starter board, it can be used to configure the device from an on-board Xilinx platform PROM which can be preconfigured via USB. We can connect to to the pin on the Digilent board using the JP8 header.

DONE This signal is asserted when the FPGA has finished configuration.

AWAKE This signal is deasserted when the FPGA enters suspend mode.

SUSPEND This pin can be driven high to put the FPGA into a low power state.

Two general (non-differential I/O) pins will also be used for the Tx and Rx connections of the USART. Differential I/O should be unnecessary at the speed we plan to use the USART.

3 Experimentation Plan

Our experimentation plan involves testing all the components as a system to verify that it performs the required tasks. We begin with a description of the dataflow and then move on to the details of how we'll verify correct operation.

The mote connected with the FPGA will be a remote, battery-powered device. This mote will also be connected with the sensor board and will be programmed to sense some physical stimulation like temperature, pressure, humidity, or other physical values at specific time intervals. The sensor board will gather data and pass it to the microcontroller of the mote. To acquire encrypted data for transmission, the microcontroller on the FPGA mote will transfer data to the FPGA, which will encrypt it and then send the encrypted data back to the mote. This mote will next transmit the encrypted data to the mote attached to the programmer and a personal computer. XServe and XSniffer will be used to monitor and document the communication flow between the two motes. The sensor nodes will be programmed with TinyOS applications, using MoteWorks, to detect and acknowledge the networking communication between the microcontroller and the FPGA.

3.1 Hardware Components

The hardware components will be tested to ensure proper functioning of the devices both individually and as a system. The devices themselves will be tested as follows:

1. Programming board (MIB520): The MIB520 (see figure 19) will be used to install application programs onto the sensor nodes. We will use NesC to code the programs and the tool Programmers Notepad 2 or a similar text editor to compile and download the programs to the sensor nodes.

2. Sensor Nodes (MICA2): Each MICA2 mote (see figure 20) will be first tested individually and then operating in tandem as a small point-to-point network. For proper program installation and accurate operation in accordance with the application specifications, each node will be connected with the programming board. The code will be compiled and downloaded onto each node via the MIB520. The correct loading of the code can be verified by using the

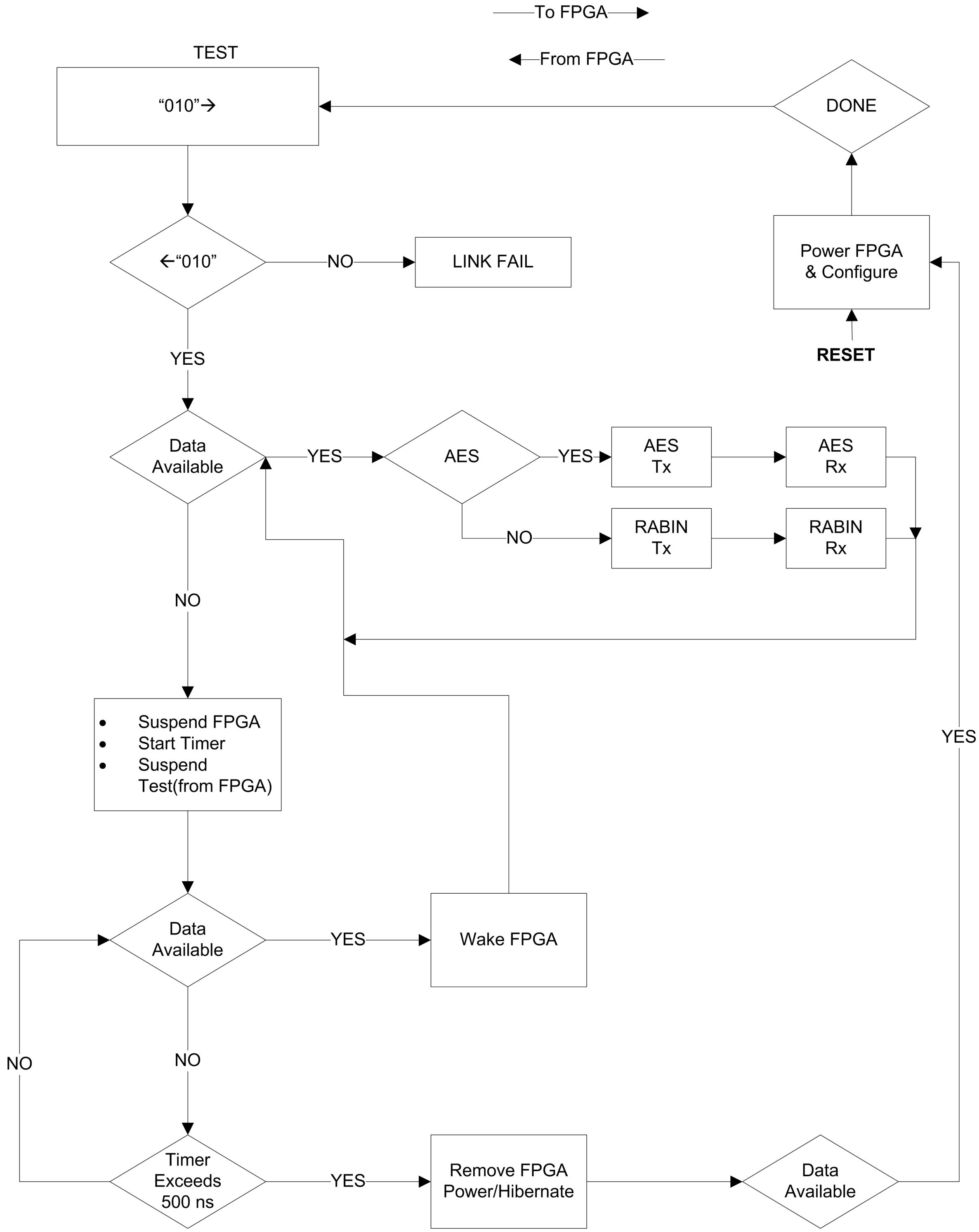


Figure 17: State diagram for the control on the mote.

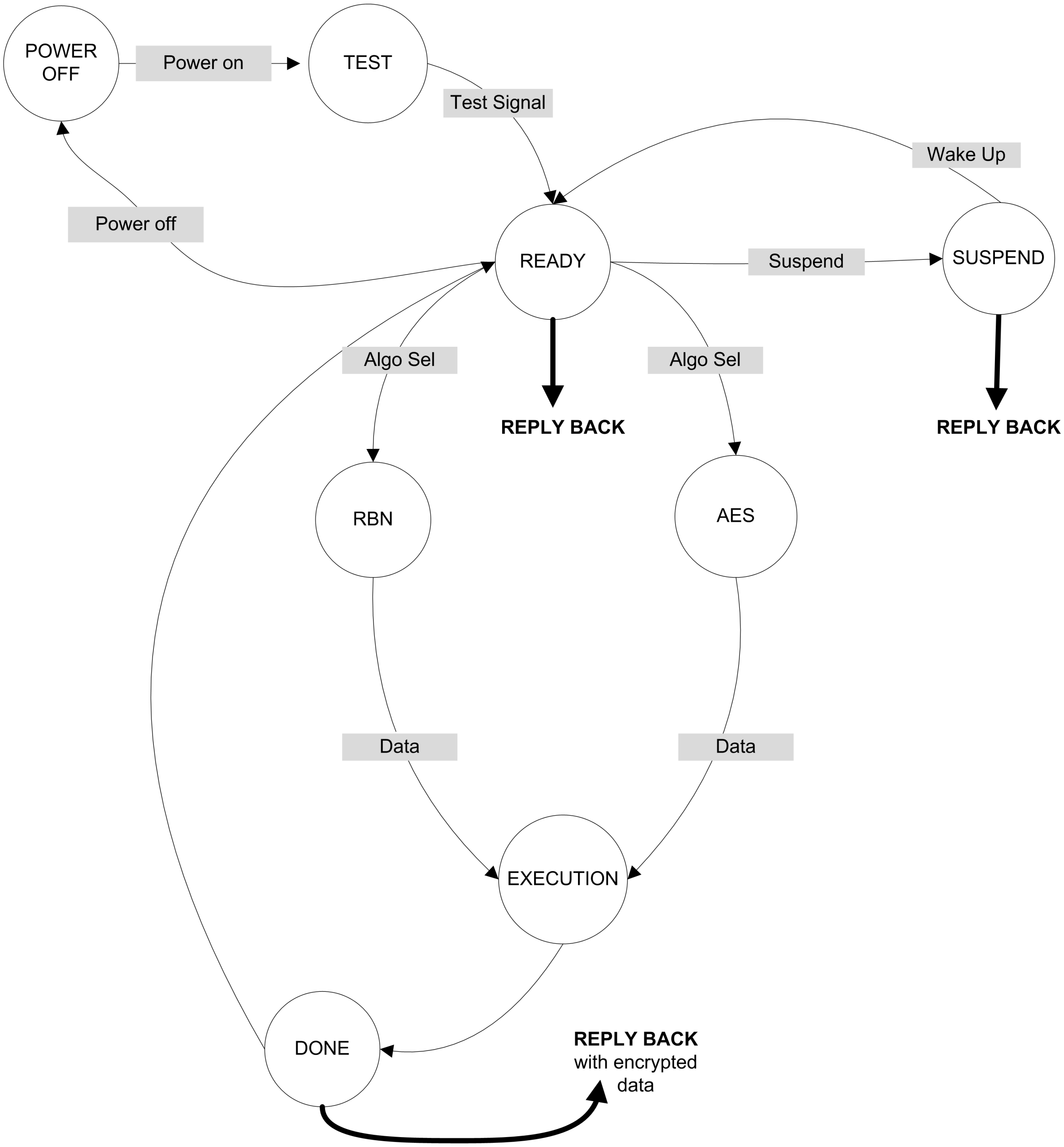


Figure 18: FPGA State diagram.

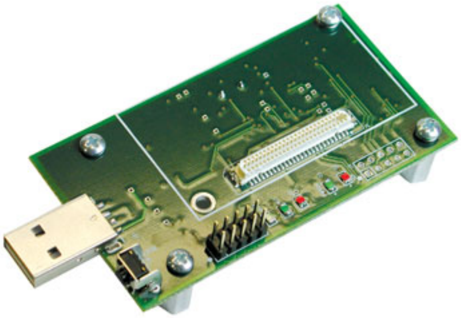


Figure 19: The MIB520 User Interface Board (©2006 Crossbow Technologies)

three LEDs available on each sensor node. Once the nodes are programmed, they will then be configured to operate as a point-to-point network.

XSniffer and XServe will be used to test the communication activity in the network. XServe provides the platform to monitor the communication between the base station and the wireless node in the mesh network, while XSniffer allows monitoring of the multi-hop communication between the nodes in the mesh network.

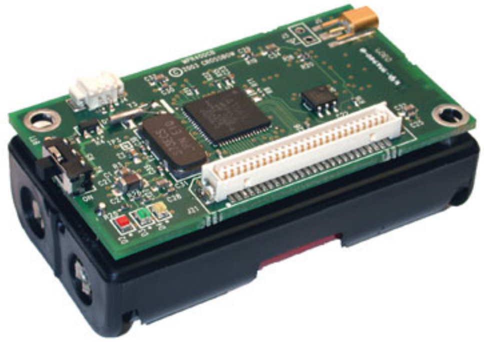


Figure 20: The MICA2 916 Wireless Mote (©2006 Crossbow Technologies)

3. Sensor board (MDA100): The sensor board (see figure 21) will provide the gateway between the mote and the FPGA if we choose one of the applications that doesn't involve building a custom PCB. In this case the FPGA pins will be connected to the fifty-one pins on the sensor board and one of the nodes will be programmed to read the sensor board activity. The node can transmit samples of information from the sensor board to another sensor node at the base station or in the mesh network, although we will not be testing this application.

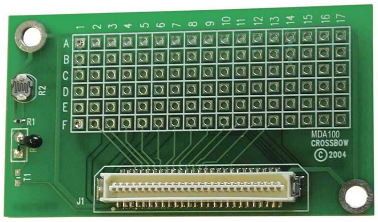


Figure 21: The MICA2 MDA100 Sensor Board (©2006 Crossbow Technologies)

3.2 Evaluation Criteria

The final system will be primarily evaluated on the power consumption of the device. Considering the future extension of this project to an implementation using ASICs instead of FPGAs, the simulation results of the FPGA system will be compared to the estimated results obtained from the ASIC simulation. The results will be divided into three sections—speed, number of resources on chip being used, and the power consumption of the chip—and each will be analyzed for compliance with the requirement specifications provided earlier in this document.

The system will also be evaluated on the following secondary criteria: reliability, accuracy, compatibility, and compactness.

Reliability: The code used for programming the motes and encrypting data on the FPGA should be reliable under all circumstances, including varying speeds of transmission, varying levels of network traffic, and different levels of load on the encryption unit. For practical reasons we won't be testing the equipment to an extreme degree or specifying all possible program states like NASA would, but these issues are still of practical concern.

Accuracy: The FPGA will be used to run encryption algorithms for a secure data transmission between the sensor nodes. The VHDL code running on the FPGA and the hardware itself must provide accurate results every time a transmission is made.

Compatibility: Ideally, the code for programming the FPGA should be simple in complexity and easily transferred to another FPGA family. We don't have much control over Dr. Kaps' code or anything we download from `www.opencores.org`, but we will try to ensure that our code meets these standards. Also, since the future concept for this project is an implementation using an ASIC instead of FPGA, highly-compatible code will provide the preliminary groundwork for such an implementation.

Compactness: Since a fundamental characteristic of sensor nodes is their small size, the add-on module must have a similar form factor. A compact, application-specific FPGA implementation is highly desirable but considering the monetary and time constraints, it may not be achieved.

4 Experimentatal Evaluation

An experimental evaluation is necessary to ensure we acheived what we set out to accomplish. We had a few major evaluation critiera when beginning the project. Here is a quick summary:

4.1 Original Evaluation Criteria

Reliability: The code used for programming the motes and encrypting data on the FPGA should be reliable under all circumstances, including varying speeds of transmission, varying levels of network traffic, and different levels of load on the encryption unit. For practical reasons we won't be testing the equipment to an extreme degree or specifying all possible program states like NASA would, but these issues are still of practical concern.

Accuracy: The FPGA will be used to run encryption algorithms for a secure data transmission between the sensor nodes. The VHDL code running on the FPGA and the hardware itself must provide accurate results every time a transmission is made.

Compatibility: Ideally, the code for programming the FPGA should be simple in complexity and easily transferred to another FPGA family. We don't have much control over Dr. Kaps' code or anything we download from `www.opencores.org`, but we will try to ensure that our code meets these standards. Also, since the future concept for this project is an implementation using an ASIC instead of FPGA, highly-compatible code will provide the preliminary groundwork for such an implementation.

Compactness: Since a fundamental characteristic of sensor nodes is their small size, the add-on module must have a similar form factor. A compact, application-specific FPGA implementation is highly desirable but considering the monetary and time constraints, it may not be achieved.

And of course, the design requirements:

- Develop communication protocol between MICA mote and FPGA
- Enable AES encryption on MICA2 platform with FPGA
- Develop software platform to demonstrate encryption
- Test with S3E Starter Board
- Increase speed compared to calculation on CPU
- Decrease CPU memory utilization
- Design portable, production-quality FPGA support platform
- Cost under $100 in mass-production
- Appropriate size for deployment with sensor node
- Power consumption profile:
- Suitable for battery operation
- Superior to CPU for encryption task

4.2 Results

The final conclusion is that most of the requirements were successful, with the exception of testing the FPGA board and demonstrating that the FPGA (either the demo board or the finished production version) were superior to the CPU for encryption tasks. These things are some primary tasks that will be left for the next group to take a look at.

5 Administrative Details

Ah, progress... There is much to discuss. Unfortunately, we did not complete all the tasks we set out to complete successfully. To finish the core part of the project we had to sacrifice some ancillary functionality, namely some things related to the mote communication and the testing and implementation of the printed circuit board. I'm still quite happy with how

Description	Cost
First parts order	$42.92
Second parts order	$54.27
Third parts order	$31.45
PCB Manufacture	$52.30
Hot Plate	$17.74
Poster	$50.00

Table 2: The parts we bought and their associated costs.

far along we got, but it's unfortunate that we ran out of time.

There were plenty of extra, not-originally-planned tasks that we had to carry out. Initially we did not plan to implement a UART from scratch, for instance; we planned to use open source code to acheive this functionality. When that didn't go according to plan one of our members had to spend more than 2 months coding the required functionality from scratch. Naturally this affected our timeline significantly.

5.1 Funds spent

The expenses for the project included costs for the PCB fabrication, costs for the parts for the PCB, a one-time cost for the poster production, and some other miscalenous expenses. See table 2 for the full description.

6 Source Code

All together about 2000 lines of custom source code were written for this project. To save space we are not including all this source in the document here, but it can be provided upon request. Also be sure to see Figure 3 for the flow of the nesC test component. Some source code is available in the proposal and design documents which follow.

7 Lessons Learned

7.1 VHDL Design

As can be seen, it takes approximately 700 s for the data to be received from the sensor node. Once the input signal is done, the system then goes to the compute stage. It can be a bit difficult to see from the diagram, but the total computation time of the encryption takes 514 clock cycles or approximately 10.7 s with a system clock of 50 MHz. Once the compute stage is done, the output stage sends the data back to the mote which takes about 350 s. When adding up the total transmission time, it is easy to see that the serial transmission is over 100 times slower than it takes to encrypt the data. However, this next point could easily be overlooked. The baud rate was mostly constrained by the mote and not the clock on the FPGA. The AES speed is only constrained by the clock speed of the FPGA. If the FPGA had a slower oscillator, the AES time would have taken longer, and the difference between transmission and encryption would not have been so grossly accentuated. In any case, the next group to work on this project should definitely implement their system using a parallel interface.

The FPGA we decided to use was mostly constrained by us designing our own PCB and making sure our FPGA had a suspend mode to save power. The Spartan 3AN series does have this functionality, so we needed to use this type. However, the only size FPGA that came in the TQ packaging was the smallest possible FPGA, the 50 series. All other sizes were built using ball grid array arrangements. Therefore, we had no choice but to choose the Spartan 3AN 50. This limited the applications we could actually perform on our board, but the AES and UART design did fit on the board using approximately 70% of the available slices. However, recall that this is a custom FPGA board; therefore it doesnt necessarily have to perform cryptography tasks. There are many applications for which this board will be fine for. However, if it is going to be used for cryptography, the next group should definitely think about using a larger FPGA so public-key applications can be performed.

7.2 Other Experiences

My experience in work and in this project has been that plans are great, but that they are ultimately not very useful. It's hard to predict what's going to happen and it is not really possible to anticipate in advance all the issues that you will run into, especially when the project you are working on is something that you have never worked on before. Experience helps here.

The teaming experience was good and useful, though I wouldn't say it was representative of the kind of teaming experiences I've had the in the workplace. When no one is getting paid, there is no single loca-

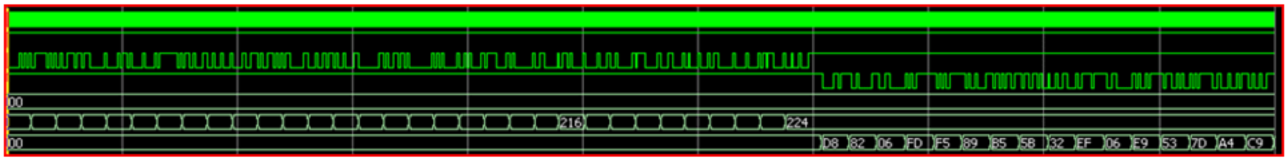


Figure 22: A simulation run in Modelsim of the total transmission and encryption from beginning to end.

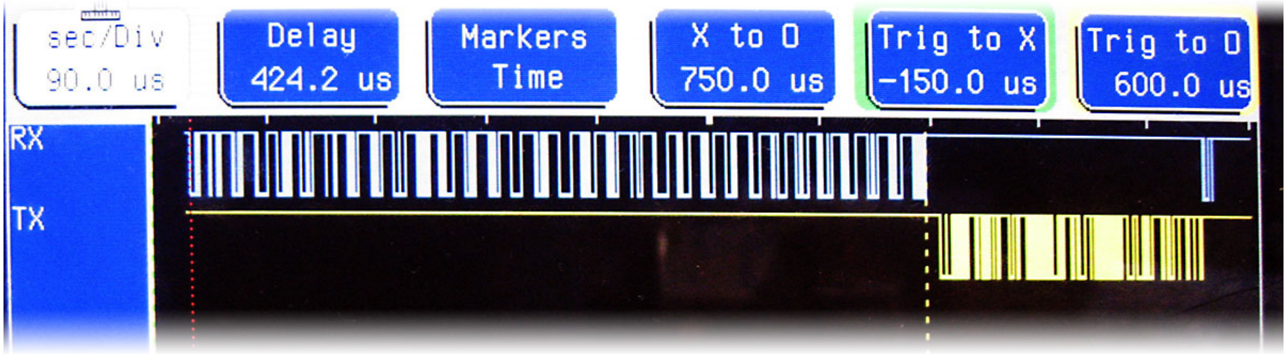


Figure 23: Same as the modelsim, but on a logic analyzer.

tion where everyone can work and store their equipment, and no one is motiviated to drive their part of the project to completion, it's a completely different team situation than you have when those conditions are different. A lot of effort is wasting doing things that are automatically and much more easily handled in a workplace environment. It wasn't worthless, but it wasn't great either.

A major change I would recommend to GMU's senior design project are a reduction in the paperwork requirements. Sure, projects in the real world require documentation and they require a lot of clerical work, but in the real world you get paid to do boring stuff that is required. Here we are paying to attend the university and it would be nice if we could focus on the fun stuff, i.e., the actual design, without having to slog though miles and miles of boring paperwork and documentation.

I know it probably wouldn't make the industry folks happy, but being a student who wasted a lot of valuable time that could have been spent on the design doing meaningless paperwork, I am naturally somewhat annoyed with the situation.

8 Appendix A: Proposal (ECE 492)

Project Proposal

Cryptographic Coprocessor for Wireless Sensors

Faculty Supervisor:
Dr. Kaps

Team (in alphabetical order):
Faizul Islam
Supreet Kaur
James McCall
Brandon Thomson

March, 2008

Contains materials from `www.digilentinc.com` and `www.xbow.com`

Contents

List of Figures

1 Executive Summary

This project has two primary goals:

1. Create an add-on board with an FPGA for the MICA II wireless mote platform,
2. Demonstrate the superiority of the FPGA add-on for encrypting data compared to the built-in microcontroller.

Since an add-on board for a small wireless mote powered by two AA batteries should be small and use as little energy as possible, we want to create a custom PCB that includes only the essential components and a connector that plugs directly into the mote.

We'll demonstrate the superiority of the FPGA by showing that it can encrypt more blocks of data per unit of energy consumed than the microcontroller. We'll show this first by XPower simulation, and then with physical measurements once the device is completed and working. To the extent we can, we'll also simulate the encryption core implemented as an ASIC and try to produce reasonable projections about power consumption and throughput for that configuration.

We think the project as we've described it is fairly ambitious based on the skills of our group members, but we're ready for the challenge and looking forward to all the things we're going to learn.

1.1 Additional Notes

This is a preliminary document based around our current understanding of the project and our goals. The planning process is not complete and everything is subject to change. Thanks for the comments on the Activity Report, Dr. Kaps: we've integrated your suggested changes and they were very helpful. Some of your comments applied to sections that aren't included in this document, but they are still helpful for future documents and our plans.

2 Problem Statement

Wireless sensor nodes are small devices equipped with an RF transceiver, a battery, multiple sensors, and typically a small microcontroller. The transceiver enables communication with other (often identical) nodes and a base station, while the microcontroller controls the communication and gives the nodes local data-processing ability.

Hundreds or thousands of these sensor nodes can collectively form mesh networks to facilitate monitoring, tracking, and surveillance applications in commercial, industrial, and military environments. A few specific applications include detection of gas leaks and pollution levels at chemical plants, early detection of forest fires or volcanic activity, and position tracking for the armed forces [1].

It's desirable that each node use as little energy as possible because this saves labor and enables the nodes to be deployed in locations where its difficult to change a battery. Remote wilderness locations, toxic and extreme environments, and even human or animal bodies are examples of places where sensor nodes could be useful but it would also be very challenging to replace a battery.

To minimize power consumption, it's necessary to carefully budget microcontroller clock cycle use and transceiver activity. These components can be powered nearly all the way off on most node platforms when they're not in use which means the primary determinant of battery life is the percentage of time the various components are active, or their "duty cycle." Naturally, effectively balancing the use of these components in a wireless sensor application to achieve minimum energy consumption is challenging.

For example, it's obvious that in the vast majority of cases it would be wasteful of both power and bandwidth for a mote to transmit every bit of the data recorded to every other mote and to a mains-powered base station with no processing or local logic at all. What's less clear is exactly how much local processing should be done on each mote to keep the transmitter power consumption low without inadvertently wasting power by running the microcontroller for too long. Like most engineering problems, there are obvious tradeoffs involved: do you want to focus on having a high frequency of updates, maximizing battery life, or getting the software working as quickly as possible?

When you add an FPGA (a device that can calculate extremely quickly but also uses a large amount of power) into the mix, things only get more complex. It should be clear, though, that any device which could potentially reduce the energy consumption of a wireless mote for a particular application would be a useful addition. We'll examine the use of an FPGA for an encryption applicaton and provide a detailed analysis of the problem in the following section.

3 Approach

This section provides a detailed analysis of the problem and our requirements and desirable features for the final design.

3.1 Problem Analysis

For some applications, an FPGA may be a suitable addition to the onboard microcontroller for local data processing. The high power consumption of FPGAs can often be mitigated by the fact that they run many algorithms orders of magnitude more quickly than microcontrollers, and thus may use less total energy than a microcontroller to perform similar tasks. Some FPGAs even have a "suspend" mode that allows them to retain their configuration while operating in a low-power state so they don't need to be reconfigured every time operation is resumed.

We propose an FPGA add-on module for the Crossbow MICA2 wireless sensor node platform that is configured to handle encryption tasks in response to requests from the microcontroller. Our primary goals are to create the FPGA-to-mote interface and to demonstrate the ability of the FPGA to encrypt data at a higher speed and with higher energy efficiency. Although for this project we will be focusing on a particular encryption algorithm (or perhaps several), there are many types of algorithms that are particularly suited to operation on FPGAs. Our implementation will support reconfiguration by RS-232 or USB and so other uses will simply require a new FPGA bitfile and new code for the microcontroller.

We expect it will be easy to show that any FPGA device is faster, but creating one that is also more energy-efficient for heavy encryption loads could be more challenging. The efficiency of FPGAs with respect to total power consumption for encryption is noted in [2], and we hope to reproduce their results, but it's worth noting that they used a customized PCB with only related components and not a prefabbed starter board. We haven't yet measured how much standby power the Digilent Spartan 3E Starter Board (see figure 4) consumes, and so we're unsure whether it will be appropriate to use in a sitution where power consumption must be minimized.

If we end up using a Spartan 3E like the one that's on the Digilent board, we also note the necessity of completely powering down the FPGA every time it is idle. On restart, it would have to be reconfigured with the bitfile. Due to the lack of a suspend mode, there may only be a net energy savings for constant, heavy loads. On the other hand with a 3L or 3A there is a good possibility that we'll be able to suspend the chip frequently and come out ahead on energy consumption even for light or infrequent use of the device. This is another reason developing a custom board would be helpful: we could use the Spartan 3 size and model most ideally suited for the job.

3.2 Requirements Definition

The finished device must be functional. When data is presented from the microcontroller to the FPGA, it must encrypt it correctly according to a pre-shared key and return it promptly to the microcontroller.

The finished FPGA with all its support components must use less energy per unit of data encrypted than the ATMega128L microcontroller would use for the same task. Ideally it would be able to handle small duty cycle loads (that is, perhaps 10 blocks of data to encrypt, and then a 30 second pause, and then 10 more blocks...) while still consuming less energy than the microcontroller, but we are willing to relax this requirement if it turns out to be too difficult to acheive in practice.

The total cost of the project should be kept under $400. So far, we've budgeted about $200 worth of parts and equipment. Hopefully none of our expensive stuff will break.

The finished device must be attractive and in a similar form factor to the original wireless mote. It should be roughly as rugged and resistant to shock as the original sensor node. Constant vibration over long periods must not cause the device to malfunction.

The finished device should be programmable using either a USB or serial interface. Although a USB interface would certainly be more convenient, we are already strapped for time and will prefer to implement a simpler RS-232 interface. Fast programming speed is unimportant.

Plugging in the add-on FPGA board to the sensor mote should not cause any of the features of the mote to become disabled or to malfunction. Extreme care must be taken to ensure that neither device's absolute maximum voltages are exceeded.

4 Preliminary Design

This section provides an overview of our current design proposals. We begin with our primary plan and then specify 2 backup plans to be used in the case of critical failure or time constraint.

4.1 Primary Plan

Taking into consideration the nature of sensor nodes and the specified constraints, we believe the best way to implement the FPGA expansion board is to purchase a low-power, discrete FPGA and design a PCB for direct connection to the sensor nodes through the Hirose 51-pin expansion connector on the motes. Shown in figure 1 is a preliminary look at the functional blocks we propose the PCB layout will consist of. We are not showing more detail in this document because we still need more time to understand the many interconnections involved.

Texas Instruments has developed a triple voltage regulator specifically for the Spartan 3 series which supplies the correct voltages to the FPGA and takes care of many complex regulation issues. Our proposed PCB will also need a flash memory device, the size depending on the FPGA model we use.

The benefits to designing a PCB for a discrete FPGA are two-fold: first, the power consumption will be much lower than using the Diligent Spartan 3E demo board because there will no superfluous components to power. It would also enable us to choose a low-power FPGA model. This means, however, that we would need to order an FPGA from the Spartan 3A or 3L series, which would increase our budget somewhat. Considering that we did not have to pay for the motes as they were loaned to us by Dr. Kaps, there is plenty of room in our budget for the components we will need for this design. The second benefit is size. The Diligent board is a large piece of equipment, and given the nature of sensor nodes and the constraints we have specified, size is a major factor. Naturally, an expansion board designed with a stand-alone FPGA would be much smaller.

Lastly, with a PCB we could order the 51-pin expansion connector from Digikey and affix it to the PCB such that the finished product can be plugged directly into the mote. This adds to the convenience, appearance, and sturdiness of the device as there will not be a lot of unsightly wires and volatile connections to constantly worry about. These features are secondary concerns but they would make for a better design.

There will be added difficulties in designing the FPGA expansion in this way, as tasks which would have already been done for us had we decided to use the Diligent board will need to be addressed. Most notably we will have to deal with power compatibility, whether to use serial or USB ports, memory and initial configuration, and other interconnection issues. A starter board or even a basic breakout board would have taken care of much of this for us.

One last concern is whether we can fabricate the PCB ourselves or whether we will need to send it to a manufacturer, and then whether we would attach the parts or pay for the fab to do it. Doing everything on our own using board stock, etching solution, a laser printer, an iron, and a drill would be interesting and fun, but there are many potential pitfalls. Two major issues are the size of the traces for the FPGA, and the very small holes that would need to be drilled for the expansion connector.

Despite the potential pitfalls, we believe that it's feasible for us to complete a custom PCB (certainly at least the schematics) through careful planning and prototyping. That's why we have chosen this to be our primary preliminary design.

4.2 Backup Plan B

If the primary design proves to be too difficult, we have two "back-up" ideas that will still result in acceptable finished products. Figure 2 shows a bare-minimum Spartan 3E breakout board from Sparkfun Electronics. There are two ways we could use this board. One way would be to break the partition, removing the support module from the actual FPGA and using it as a stand-alone FPGA. This does not seem beneficial because we would be essentially removing a triple voltage regulator that supplies the correct voltage, a 16 Mbit PROM to store the bitfiles, and a serial port for programming. If we go this route we will probably use the whole breakout board, although we may still break it in half to make the form factor more appealing.

To use this secondary design we would need to develop an interface board that maps the pin connections of the Hirose 51-pin expansion connector such that the components can be interfaced with the sensor nodes. We want to meet the requirements that the board be portable and easy to attach to the mote, so we could still affix a purchased Hirose connector to another board of similar size which would be attached to the (broken apart) Sparkfun boards in perhaps a triple-stacked arrangement. This board could have traces which line up with those on the Sparkfun board and be bridged to it by small pins. Of course

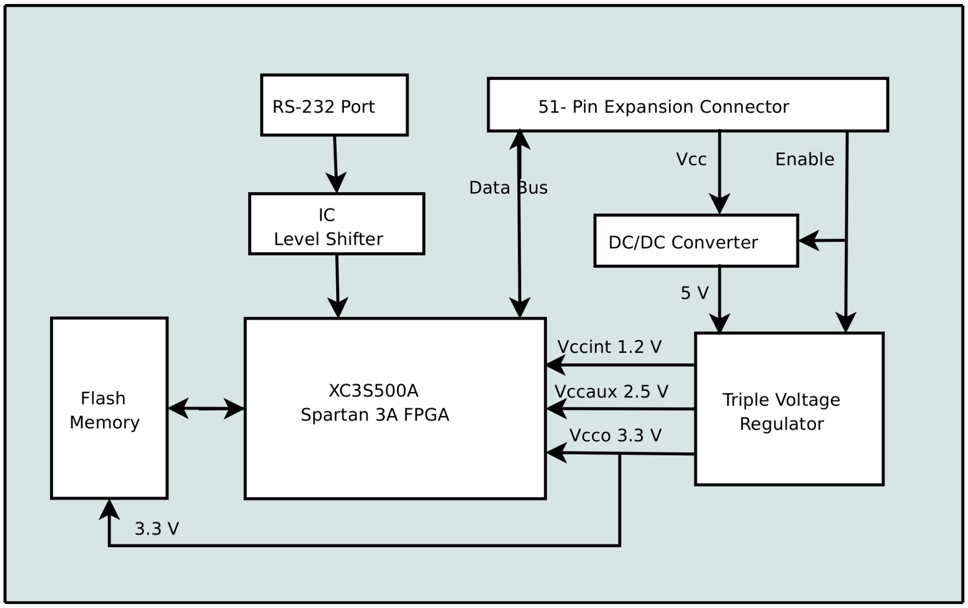


Figure 1: Proposed PCB Components Block Diagram

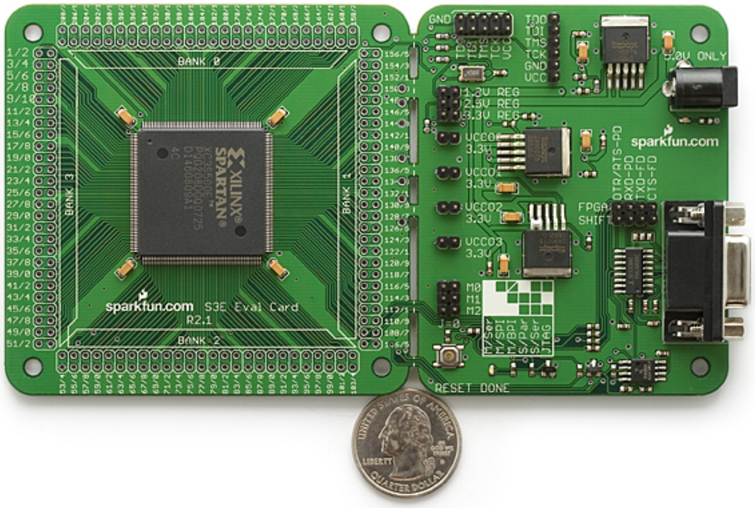


Figure 2: The Sparkfun DEV-08458 Spartan 3E Breakout and Development Board (©2007 Sparkfun Electronics)

the breakout board would still store the bitfile in the PROM on the support module.

Alternatively we could use the MDA100CB interface board (Figure 3) to connect to the sensor node via the 51-pin expansion connector and standard wires to connect to the traces on the breakout board, but we would need some other type of structural enforcement to keep the device small and held together as one unit.

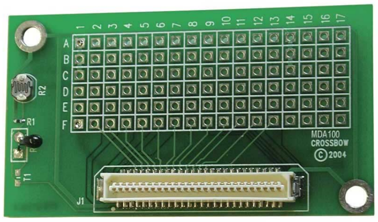


Figure 3: The Crossbow MDA100CB Expansion Connector (©2006 Crossbow Technologies)

4.3 Backup Plan C and Initial Test

There is still one more back-up plan that could come into effect if we do not make nearly as much progress as we hoped to and time becomes a major issue. For preliminary testing purposes, we are using the Digilent Spartan 3E Starter board (provided by Dr. Kaps), shown in figure 4.

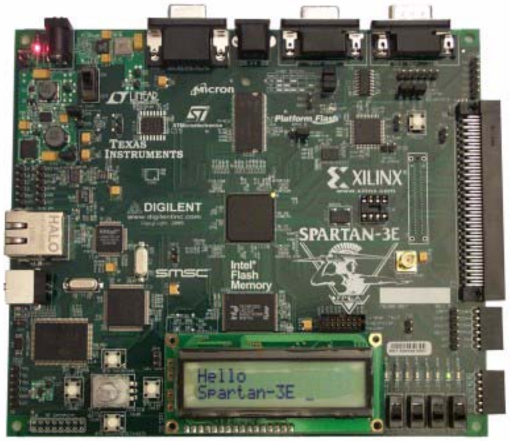


Figure 4: The Digilent Spartan 3E Starter Board (©2007 Digilent)

We'll be using this board to ensure our ideas about interconnections work, that the software developed for the motes is effective, and that the implemented FPGA encrypts correctly.

Since we are going to be using this board anyway to being with, we will always have it as a backup in case our more ambitious plans for a final design don't quite work out. The board has a Hirose 100-pin expansion connector that we would need to interface with the MDA100CB board shown in figure 3. Since this demo board has all the bells and whistles, is no need for external memory or voltage regulation and the main design boils down to simply interfacing the expansion connector on the demo board to the expansion connector on the sensor nodes. This design will be more large and consume more energy than the other designs which is why it is a back-up.

Admittedly as we delve deeper into the many complexities of this project we are learning that there are many difficult aspects which may force us to be satisfied with this last design.

4.4 Device Communication and Bandwidth

This section provides some details about our research on the bus connections between the microcontroller and the FPGA device. We don't anticipate the 57.6K baud USART implementation on the microcontroller will be sufficient to saturate the FPGA with data for encryption if it is running at full speed. Presumably the FPGA will be expecting some input every clock cycle or every few clock cycles, and at 48MHz that adds up to a lot of bits per second.

It's also worth noting that the maximum transmit rate for the mote is only 38.4K baud and thus even if the microcontroller can talk to the mote at the higher rate it can't actually send the data anywhere. This means the FPGA is probably a lot faster than is necessary for simple transmission of data with this particular hardware.

Since demonstrating the high throughput of the FPGA is an important part of the project, I think it makes sense to try and get as high of a bandwidth link between the FPGA and the microcontroller as we can even if we can't transmit the resulting data from the microcontroller. To that end we plan to implement a parallel data bus using 8 GPIO pins on the microcontroller. We could use more, but we can only write to a maximum of 8 in one clock cycle on the ATMega128L and so 8 seems like the ideal number.

By keeping the "data generation" instructions on the microcontroller simple, we to be able to "feed" the FPGA with fairly large amounts of data. Unfortunately we don't yet know how much overhead the MoteWorks software will take up, or how often we'll be able to issue an interrupt to write to the pins. It may turn out that this method doesn't give us much more usable bandwidth than a USART connection would have, but at least it'll be an interesting learning experience.

We'd like to have a second parallel data connection for returning the data to the microcontroller but unfortunately there is not a second set of 8 GPIO pins grouped together that we'll be able to read from in only one clock cycle. The only way to solve this problem may be to accept the use of GPIO pins on different ports, but this will slow down the device since we will need to use extra cycles on the microcontroller to read multiple ports and then concatenate the results. This is an outstanding issue we'll need to think about before we start making detailed schematics.

If we fail to get our parallel connection working during the demo board testing stage, we'll switch over to the USART implementation and find a clever way to work around the fact that we can't actually saturate the FPGA with data from the microcontroller. We may run into the same problem with the parallel connection anyway.

4.5 Voltages

The Digilent board cannot really be powered from our two AA batteries due to the high current requirement. For our other 2 plans (PCB and Sparkfun) we hope that using a DC-to-DC converter and the TI voltage controller device will supply stable enough voltages from 2 lithium AA batteries to power the FPGA and other components.

Spartan-3 devices are designed and characterized to support various I/O standards for V_{CCO} values of +1.2 V, +1.5 V, +1.8 V, +2.5 V, and +3.3 V. The ATMega128L ($V_{cc} = 3.0\,\text{V}$) supports an input low voltage from -0.5 V to $0.2V_{cc}$ (0.6 V) and an input high voltage from $0.6V_{cc}$ (1.8 V) to $V_{cc} + 0.5\,\text{V}$ (3.5 V). The output low voltage is 0.5 V and the output high voltage is 2.2 V. Presumably if the FPGA is configured for $V_{CCO} = 2.5\,\text{V}$ on the connected pins the devices will not have trouble communicating due to out-of-range voltages (and at least will not damage each other due to over-voltage, which is what we're really trying to avoid during early testing).

5 Preliminary Experimentation Plan

Our experimentation plan will include the testing of all the components as a system to verify that system performs the required tasks. To acquire encrypted data, the microcontroller on the mote attached to the programmer and a personal computer will send data to another mote connected with the FPGA (this mote will simulate a sensor or other device and give us the opportunity to become familar with computer-to-mote communications). The microcontroller on the mote with the FPGA will then transfer this data to the FPGA, which will encrypt it and send the encrypted data back to the requesting microcontroller. XServe and XSniffer will be used to monitor and document the communication flow between the requesting mote and the FPGA mote during the encryption phase. The sensor nodes will be programmed with TinyOS applications, using MoteWorks, to detect and acknowledge the networking communication between the microcontroller and the FPGA.

The hardware components will be tested to ensure proper functioning of the devices both individually and as a system. The devices themselves will be tested as follows:

1. Programming board (MIB520): The MIB520 (see figure 5) will be used to install application programs onto the sensor nodes. We will use NesC to code the programs and the tool Programmers Notepad 2 or a similar text editor to compile and download the programs to the sensor nodes.

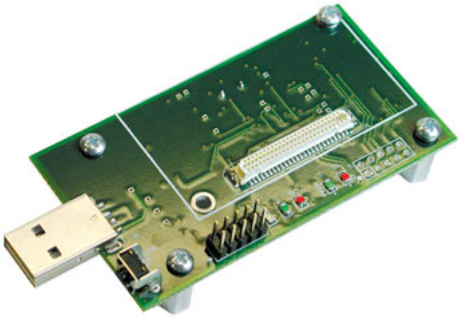


Figure 5: The MIB520 User Interface Board (©2006 Crossbow Technologies)

2. Sensor Nodes (MICA2): Each MICA2 mote (see figure 6) will be first tested individually and

then operating in tandem as a small point-to-point network. For proper program installation and accurate operation in accordance with the application specifications, each node will be connected with the programming board. The code will be compiled and downloaded onto each node via the MIB520. The correct loading of the code can be verified by using the three LEDs available on each sensor node. Once the nodes are programmed, they will then be configured to operate as a point-to-point network.

XSniffer and XServe will be used to test the communication activity in the network. XServe provides the platform to monitor the communication between the base station and the wireless node in the mesh network, while XSniffer allows monitoring of the multi-hop communication between the nodes in the mesh network.

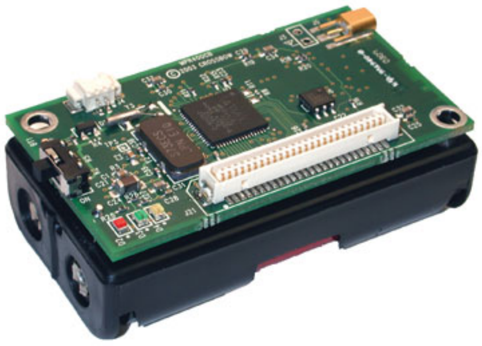


Figure 6: The MICA2 916 Wireless Mote (©2006 Crossbow Technologies)

3. Sensor board (MDA100): The sensor board (see figure 3) will provide the gateway between the mote and the FPGA if we choose one of the applications that doesn't involve building a custom PCB. In this case the FPGA pins will be connected to the fifty-one pins on the sensor board and one of the nodes will be programmed to read the sensor board activity. The node can transmit samples of information from the sensor board to another sensor node at the base station or in the mesh network, although we will not be testing this application.

5.1 Evaluation Criteria

The final system will be primarily evaluated on the power consumption of the device. Considering the future extension of this project to an implementation using ASICs instead of FPGAs, the simulation results of the FPGA system will be compared to the estimated results obtained from the ASIC simulation. The results will be divided into three sections—speed, number of resources on chip being used, and the power consumption of the chip—and each will be analyzed for compliance with the requirement specifications provided earlier in this document.

The system will also be evaluated on the following secondary criteria: reliability, accuracy, compatibility, and compactness.

Reliablilty: The code used for programming the motes and encrypting data on the FPGA should be reliable under all circumstances, including varying speeds of transmission, varying levels of network traffic, and different levels of load on the encryption unit. For practical reasons we won't be testing the equipment to an extreme degree or specifying all possible program states like NASA would, but these issues are still of practical concern.

Accuracy: The FPGA will be used to run encryption algorithms for a secure data transmission between the sensor nodes. The VHDL code running on the FPGA and the hardware itself must provide accurate results every time a transmission is made.

Compatibility: Ideally, the code for programming the FPGA should be simple in complexity and easily transferred to another FPGA family. We don't have much control over Dr. Kaps' code or anything we download from `www.opencores.org`, but we will try to ensure that our code meets these standards. Also, since the future concept for this project is an implementation using an ASIC instead of FPGA, highly-compatible code will provide the preliminary groundwork for such an implementation.

Compactness: Since a fundamental characteristic of sensor nodes is their small size, the add-on module must have a similar form factor. A compact, application-specific FPGA implementation is highly desirable but considering the monetary and time constraints, it may not be achieved.

6 Preliminary List of Tasks

This is our preliminary list of tasks and goals. Many tasks listed for this semester will be complete before the final document is submitted and will contribute to the research and specifications in the other sections of the final design document. We will use some type of a progress-tracking Gantt chart to demonstrate our progress now all of these tasks have been entered into MS Project.

Preliminaries I thought we should get these tasks done right away to help the team get familiar with the project and the tools we'll be working with.

Toolchain setup and installation A lot of support software is required for this project. We need the MoteWorks software distribution, drivers for the mote programmers, Xilinx ISE for the FPGA, EAGLE for the PCB layout, and whatever software we'll be using for ASIC simulation. For the most part this step is complete although we all have different versions of the toolchain.

Hardware testing and verification Nothing is worse than spending hours trying to figure out why your code doesn't work and then realizing that it's because your hardware platform is bad. As such, we'll be testing all of the hardware (motes, FPGA starter board, any purchased hardware) before we try to use it. So far we've run sample programs on the motes, but we haven't had a chance to test the Digilent board yet.

Demo Board Setup The first setup we'll develop will use the demo board as a target platform. In the case that the power-efficient setup runs into problems, we'll also have this platform to fall back to for demonstrations and testing.

Locate/Obtain FPGA encryption core For the final part of the project we expect to use a core supplied by Dr. Kaps, but in the interim we may use an open-source core from `www.opencores.org`.

Core synthesis/implementation We need to make sure early that the core will synthesize on a Spartan 3 target. We should also figure out the smallest chip that the core will synthesize on.

Mote connections We need to connect the demo board to the mote. These will be temporary connections with probably an IDE cable attached to one of the pin headers on the demo board and as such shouldn't take very long.

Testing The fun part—making sure it works. If everything goes fine this will take 5 minutes, but if it doesn't we'll have a lot of work to do troubleshooting. That's why I allocated so much time for this.

Power consumption measurements Once everything is verified working, we can measure the power consumption for our reports. This will be a combination of simulation results and physical measurements. It'll be interesting to see how much power the demoboard consumes and how much we can potentially save with a custom PCB. If the amount is small, perhaps we'll rethink our plan at this point. Or perhaps we should find a way to measure it earlier.

Throughput analysis We'll need to find out how much data the core is passing when it's running at full speed. The throughput could be limited by the speed of the communication link between the FPGA and the microcontroller, and we'll need to figure out what to do about this issue if it is (preferably before we get there).

Mote Software The software running on the mote is a significant part of the project. We've already made some good progress understanding the

Locate encryption implementation We want to run an implementation of the same algorithim that's going to be used on the FPGA on the microcontroller so that we can compare power consumption and throughput for both of the devices. The first step is finding an implementation.

Throughput analysis After the implementation is compiled and running, we'll see how much data we can get through. From there it's a trivial step to determine how many blocks of data per unit energy is being used.

Communication software (for testing interface) Although we could go directly to a device driver, we thought it might be simpler for development to get some code running on the microcontroller that does something stupid like flip the output bits of the communication interface every clock cycle. This will probably make it easier to troubleshoot the connections and we'll also be able to get it done earlier so work on these two main tasks can proceed in parallel.

Device driver The device driver itself could be a bit of a challenge. As we noted eariler, a significant amount of information is available on creating device drivers for TinyOS 1.1 (which MoteWorks is derived from) and 2.0 but documentation for the MoteWorks platform is scarcer. We've allocated a lot of time for this task.

ASIC Simulation The purpose of doing an ASIC simulation is for our learning experience and to demonstrate the advantages of ASICs. Obviously we wouldn't be developing one.

Synthesis We gather the first step is running the VHDL through a synthesis tool, similar to the VHDL workflow we're already familiar with. The impression I got from Dr. Gaj's slides was the GMU has one licensed, but if that's not the case we'll need to reevaluate our options.

Projected power consumption This may include a serious amount of estimation. There are tools available that automate this process, but probably not student versions and GMU probably doesn't have them licensed.

Projected throughput analysis If the synthesis tool works, hopefully it will provide a maximum clock cycle and we'll be able to use that number to determine throughput.

Power efficient setup For the final deliverable, we'd really like to create an FPGA package that does the job while using the minimum amount of power possible. If it turns out to be impossible to get this done before the deadline, we'll get as far through the process as we can.

Build schematic We've tentatively decided to use the EAGLE software for this part of the project. The first step is creating a schematic. Depending on whether we decide to implement the power management components on the chip or buy a premade solution, we may simply copy significant amounts of this from the TI datasheet (they have a nice power controller IC for the Spartan 3s, but it needs a lot of external components)

Circuit layout Once the schematic is complete we have to do layout. I'm not very familiar with this process, but we'll learn as we go. It would be nice to use SMD components (apparently it is possible to affix them yourself if you have a steady hand), but we'll see as we go along.

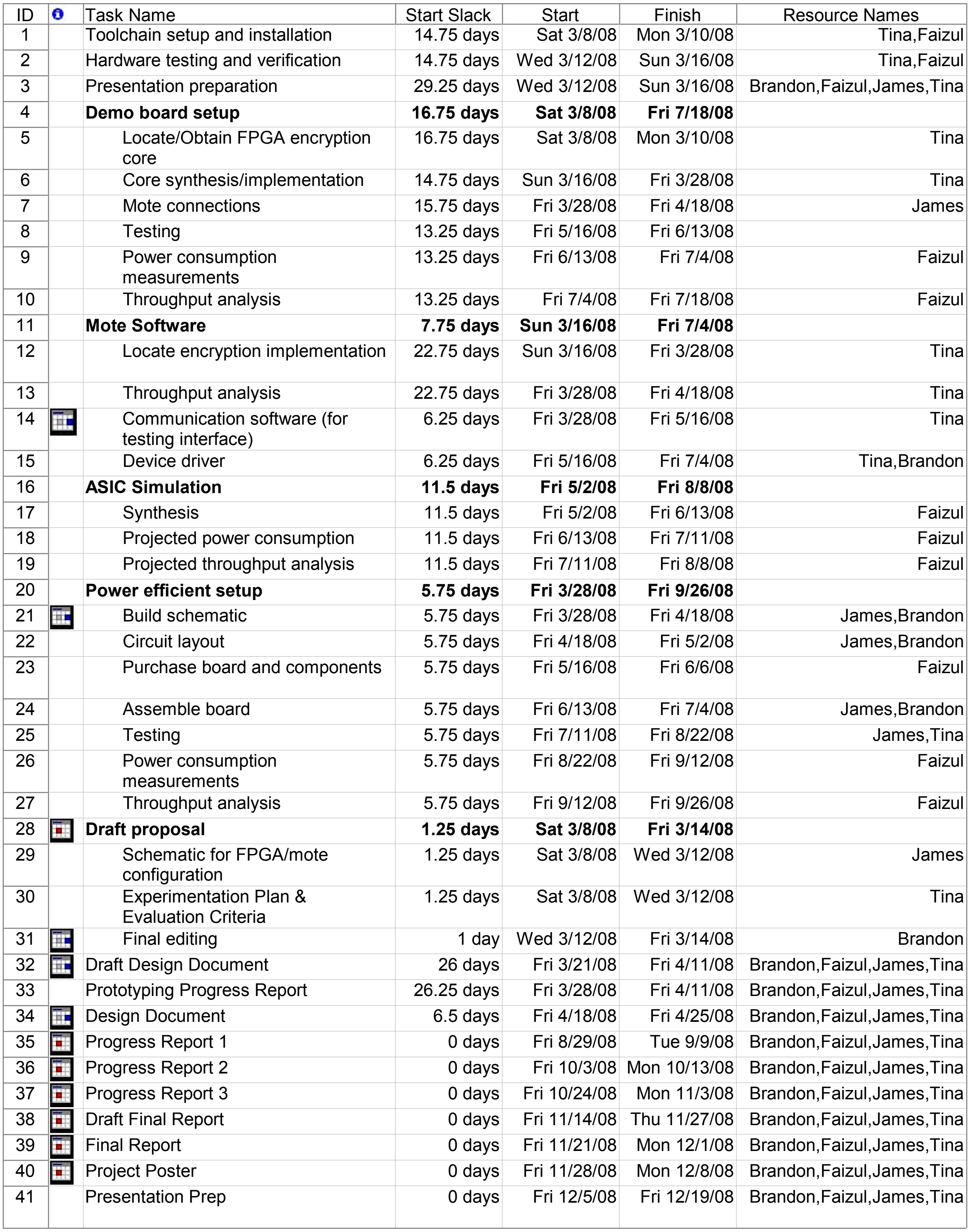
Purchase board and components We're still a little shaky on whether we'll have time to implement the voltage controller on the board and whether we'll be able to solder an FPGA with all those tiny pins ourselves (others have done it, but we're not pros!) Thus, we don't know exactly which parts we need to buy yet. For the PCB, we'll send the schematic off to one of to the cheapest Chinese supplier we can buy. Will it work? Who knows, but it'll be interesting.

Assemble board Some PCB suppliers seem willing to affix components they stock for a low cost, but I suspect we'll still end up doing it ourselves. I'm more nervous about doing SMD components than through-hole ones, but we'll see how it goes. The FPGA will be the only really challenging one.

Testing Naturally there's going to be a lot of testing and this may well be the stage where we get stuck. Ideally this stage will not only check whether "it works," but also the many other constraints we specified in the design requirements section.

Power consumption measurements After we've ensured the board is working correctly we'll measure the power consumption of the FPGA by putting an ammeter inline with the power supply while the chip is operating. The result should be close to the simulation, plus the inefficiencies in the voltage regulators.

Throughput analysis If everything's working, our final task is to ensure our throughput results from simulation match what we're actually getting in the hardware.



ID	ℹ	Task Name	Start Slack	Start	Finish	Resource Names
1		Toolchain setup and installation	14.75 days	Sat 3/8/08	Mon 3/10/08	Tina,Faizul
2		Hardware testing and verification	14.75 days	Wed 3/12/08	Sun 3/16/08	Tina,Faizul
3		Presentation preparation	29.25 days	Wed 3/12/08	Sun 3/16/08	Brandon,Faizul,James,Tina
4		**Demo board setup**	**16.75 days**	**Sat 3/8/08**	**Fri 7/18/08**	
5		Locate/Obtain FPGA encryption core	16.75 days	Sat 3/8/08	Mon 3/10/08	Tina
6		Core synthesis/implementation	14.75 days	Sun 3/16/08	Fri 3/28/08	Tina
7		Mote connections	15.75 days	Fri 3/28/08	Fri 4/18/08	James
8		Testing	13.25 days	Fri 5/16/08	Fri 6/13/08	
9		Power consumption measurements	13.25 days	Fri 6/13/08	Fri 7/4/08	Faizul
10		Throughput analysis	13.25 days	Fri 7/4/08	Fri 7/18/08	Faizul
11		**Mote Software**	**7.75 days**	**Sun 3/16/08**	**Fri 7/4/08**	
12		Locate encryption implementation	22.75 days	Sun 3/16/08	Fri 3/28/08	Tina
13		Throughput analysis	22.75 days	Fri 3/28/08	Fri 4/18/08	Tina
14	📅	Communication software (for testing interface)	6.25 days	Fri 3/28/08	Fri 5/16/08	Tina
15		Device driver	6.25 days	Fri 5/16/08	Fri 7/4/08	Tina,Brandon
16		**ASIC Simulation**	**11.5 days**	**Fri 5/2/08**	**Fri 8/8/08**	
17		Synthesis	11.5 days	Fri 5/2/08	Fri 6/13/08	Faizul
18		Projected power consumption	11.5 days	Fri 6/13/08	Fri 7/11/08	Faizul
19		Projected throughput analysis	11.5 days	Fri 7/11/08	Fri 8/8/08	Faizul
20		**Power efficient setup**	**5.75 days**	**Fri 3/28/08**	**Fri 9/26/08**	
21	📅	Build schematic	5.75 days	Fri 3/28/08	Fri 4/18/08	James,Brandon
22		Circuit layout	5.75 days	Fri 4/18/08	Fri 5/2/08	James,Brandon
23		Purchase board and components	5.75 days	Fri 5/16/08	Fri 6/6/08	Faizul
24		Assemble board	5.75 days	Fri 6/13/08	Fri 7/4/08	James,Brandon
25		Testing	5.75 days	Fri 7/11/08	Fri 8/22/08	James,Tina
26		Power consumption measurements	5.75 days	Fri 8/22/08	Fri 9/12/08	Faizul
27		Throughput analysis	5.75 days	Fri 9/12/08	Fri 9/26/08	Faizul
28	📅	**Draft proposal**	**1.25 days**	**Sat 3/8/08**	**Fri 3/14/08**	
29		Schematic for FPGA/mote configuration	1.25 days	Sat 3/8/08	Wed 3/12/08	James
30		Experimentation Plan & Evaluation Criteria	1.25 days	Sat 3/8/08	Wed 3/12/08	Tina
31	📅	Final editing	1 day	Wed 3/12/08	Fri 3/14/08	Brandon
32	📅	Draft Design Document	26 days	Fri 3/21/08	Fri 4/11/08	Brandon,Faizul,James,Tina
33		Prototyping Progress Report	26.25 days	Fri 3/28/08	Fri 4/11/08	Brandon,Faizul,James,Tina
34	📅	Design Document	6.5 days	Fri 4/18/08	Fri 4/25/08	Brandon,Faizul,James,Tina
35	📅	Progress Report 1	0 days	Fri 8/29/08	Tue 9/9/08	Brandon,Faizul,James,Tina
36	📅	Progress Report 2	0 days	Fri 10/3/08	Mon 10/13/08	Brandon,Faizul,James,Tina
37	📅	Progress Report 3	0 days	Fri 10/24/08	Mon 11/3/08	Brandon,Faizul,James,Tina
38	📅	Draft Final Report	0 days	Fri 11/14/08	Thu 11/27/08	Brandon,Faizul,James,Tina
39	📅	Final Report	0 days	Fri 11/21/08	Mon 12/1/08	Brandon,Faizul,James,Tina
40	📅	Project Poster	0 days	Fri 11/28/08	Mon 12/8/08	Brandon,Faizul,James,Tina
41		Presentation Prep	0 days	Fri 12/5/08	Fri 12/19/08	Brandon,Faizul,James,Tina

Figure 7: Project responsibilities and other relevant data exported from our project management software

6.1 Allocation of responsibilities

Although we've done our best to allocate responsibilities according to the expertise, work habits, and desires of the group members, at least some changes will probably be necessary at some point. The responsibilities are shown in figure 7.

7 Preliminary Schedule and Milestones

Our schedule includes everything from the date we started tracking tasks, including project milestones and deliverables for this semester and next semester. The Gantt chart is shown in figure 8.

8 Outstanding Issues and Problems

This section provides a rough list of issues that we've been discussing at the meetings but haven't worked out yet as of the printing of this document (15 MAR 2007). We are still trying to find solutions and any comments are appreciated!

- Does GMU have any tools that can be used for ASIC synthesis licensed? Are there any student/research versions? How about ASIC power analysis software? What do we do if we can't get access to this stuff?
- How do we clock the parallel bus(ses) between the FPGA and the microcontroller? If we have 8 wires coming from the microcontroller, and we use one pin as a "clock," and we wait a couple of FPGA clock cycles after the "clock" pin has changed before latching the data pins into the I/O blocks, is that going to work?

References

[1] J. Yick. (2006, March) Wireless sensor networks. UC Davis Network Research Lab. [Online]. Available: http://networks.cs.ucdavis.edu/~yick/Research.html

[2] P. Hamalainen, T. Alho, M. Hannikainen, and T. Hamalainen, "Design and implementation of low-area and low-power aes encryption hardware core," *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pp. 577–583, 2006.

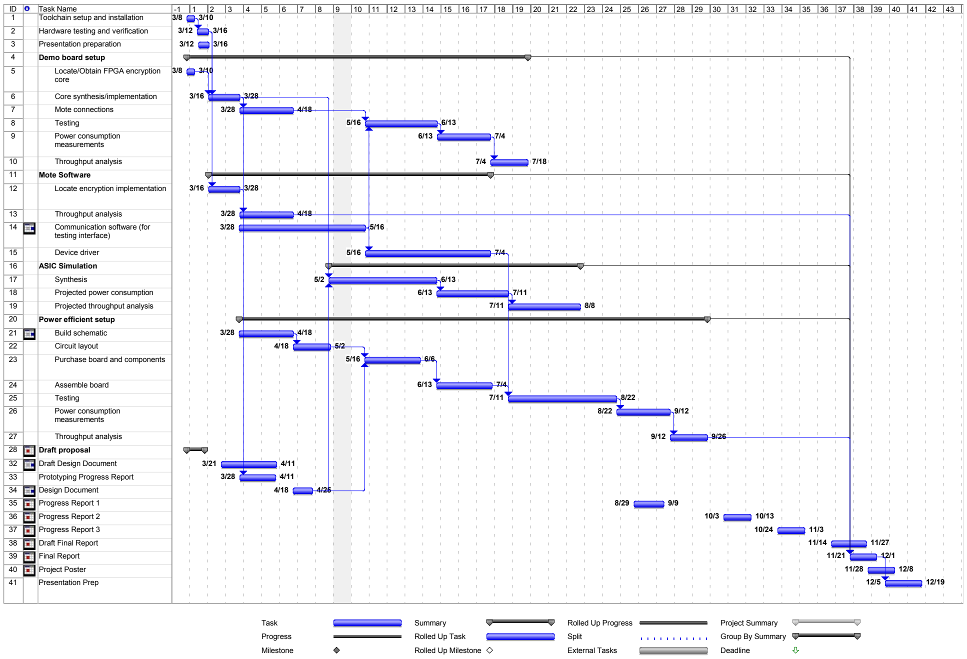


Figure 8: Project Gantt chart, exported from our project management software

9 Appendix B: Design Document (ECE 492)

Cryptographic Coprocessor for Wireless Sensors

ECE 492 Design Document

Faculty Supervisor:
Dr. Kaps

Team (in alphabetical order):
Faizul Islam
Supreet Kaur
James McCall
Brandon Thomson

April, 2008

Department of Electrical and Computer Engineering
George Mason University
Fairfax, VA 22030

Contains materials from `www.digilentinc.com` and `www.xbow.com`

Contents

List of Figures

Acknowledgments

We would like to thank our faculty supervisor, Dr. Kaps, for introducing us to such an interesting project, for spending a lot of time with us and helping with the necessary research, and for tolerating us and working around our sometimes intolerable schedules.

We'd also like to thank Dr. Gaj for sitting through our (admittedly long and somewhat unorganized) proposal presentation and giving us some very useful comments that have changed and clairified some critical details of this project.

1 Introduction

This design document specifies the details of our plan to integrate a Field Programmable Gate Array (FPGA) device with the Crossbow MICA2 wireless mote platform. The FPGA will be used to host two encryption cores that have been provided to us by Dr. Kaps in VHDL. The project is scheduled for completion at the end of 2008, but may become a continuation for a group to work on next year depending on whether we are able to carry out the full vision for the project.

The FPGA device is mainly intended for prototyping and experimentation. Ideally, this project will lead to the development of an ASIC that could be integrated with future mote devices to provide fast encryption with maximal power efficiency. Since our small, unfunded group will be unable to see any type of ASIC project through to completion, though, we aim to use various techniques to make the FPGA implementation as power-efficient as possible.

We'll begin with a gentle introduction to the various aspects of this project and then ease into the details of our specific design.

1.1 Wireless Sensor Nodes

Wireless sensor nodes (WSNs) are small devices equipped with an RF transceiver, a battery, multiple sensors, and typically a small microcontroller. The transceiver enables communication with other (often identical) nodes and a base station, while the microcontroller controls the communication and gives the nodes local data-processing ability.

Hundreds or thousands of these sensor nodes can collectively form mesh networks to facilitate monitoring, tracking, and surveillance applications in commercial, industrial, and military environments. A few specific applications include detection of gas leaks and pollution levels at chemical plants, early detection of forest fires or volcanic activity, and position tracking for the armed forces [1].

TinyOS is a free and open source operating system and support platform developed specifically for wireless sensor networks. It's written in the specially-developed nesC programming language and provides a common set of commands and libraries that can be used on multiple hardware platforms. The wireless motes we'll be using for this project are supported by TinyOS 2.x, and so we'll be using it extensively. Details about nesC are discussed in §3.5.

1.2 Rationale for Encryption on WSNs

Many potential applications of wireless sensor nodes can benefit from message authentication and confidentiality. For instance, a sensor node deployed in a critical military network with a remote command interface should only accept commands from authorized users. Likewise, it may be undesirable for unauthorized users to be able to view data the sensor nodes are collecting. Encryption can assist with both of these goals.

There are two major classes of encryption: symmetric and asymmetric. In general, symmetric-key encryption is only effective if the symmetric key is kept secret by both parties involved in a communication. For this reason, it is not ideal for deployment in wireless sensor network scenarios where individual motes may be compromised and a private key recovered by an attacker.

Asymmetric (or public-key) encryption involves a pair of keys, public and private. Each public key is published while its corresponding private key is kept secret. Asymmetric encryption allows parties to disguise information they send to each other, to ensure data has not been modified in transit, to confirm a sender's identity, and to prevents a sender of information from claiming at a later date that the information was never sent. An additional benefit for WSNs is the fact that each mote need only contain the public keys of other motes and any base stations, which would limit security risks if some motes were collected and their contents analyzed by an attacker [2][3].

This has been only a brief discussion of the reasons for encryption and security in WSNs. A full analysis of the benefits and drawbacks of symmetric vs asymmetric cryptography and their applications for WSNs are outside the scope of this document. For more information about the implementations and their implications, see [2] and [4].

2 Functional Design/Architecture

There are many functional design aspects to consider for an encryption platform for WSNs. For instance, it's desirable that a sensor node use as little energy as possible because this saves labor, enables the nodes to be deployed in locations where its difficult to change a battery, and provides additional runtime. Remote wilderness locations, toxic and extreme environments, and even human or animal bodies are examples of places where sensor nodes could be useful but it would also be very challenging to replace a battery.

To minimize power consumption, it's necessary to carefully budget microcontroller clock cycle use and transceiver activity. These components can be powered nearly all the way off on most node platforms when they're not in use which means the primary determinant of battery life is the percentage of time the various components are active, or their "duty cycle." Naturally, effectively balancing the use of these components in a wireless sensor application to achieve minimum energy consumption is challenging [4].

For example, it's obvious that in the vast majority of cases it would be wasteful of both power and bandwidth for a mote to transmit every bit of the data recorded to every other mote and to a mains-powered base station with no processing or local logic at all. What's less clear is exactly how much local processing should be done on each mote to keep the transmitter power consumption low without inadvertently wasting power by running the microcontroller for too long. Like most engineering problems, there are obvious tradeoffs involved: do you want to focus on having a high frequency of updates, maximizing battery life, or getting the software working as quickly as possible?

When you add an FPGA (a device that can calculate extremely quickly but also uses a large amount of power) into the mix, things only get more complex. It should be clear, though, that any device which could potentially reduce the energy consumption of a wireless mote for a particular application would be a useful addition. We'll examine the use of an FPGA for an encryption application and provide a detailed analysis of the problem in the following section.

2.1 Adding an FPGA

For some applications, an FPGA may be a suitable addition to the onboard microcontroller for local data processing. The high power consumption of FPGAs can often be mitigated by the fact that they run many algorithms orders of magnitude more quickly than microcontrollers, and thus may use less total energy than a microcontroller to perform similar tasks. Some FPGAs even have a "suspend" mode that allows them to retain their configuration while operating in a low-power state so they don't need to be reconfigured every time operation is resumed.

We propose an FPGA add-on module for the Crossbow MICA2 wireless sensor node platform that is configured to handle encryption tasks in response to requests from the microcontroller. On the FPGA will be one public key algorithm and one private key algorithm: AES/Rijndael as a private-key algorithm and RSA/Rabin as a private key algorithim. Our primary goals are to create the FPGA-to-mote interface and to demonstrate the ability of the FPGA to encrypt data at a higher speed and with higher energy efficiency than the Atmega128L microcontroller on the mote. Although for this project we will be focusing on two particular *encryption* algorithms, there are many other types of algorithms that are particularly suited to operation on FPGAs. Our implementation will support reconfiguration by either RS-232 or USB and so other uses will simply require a new FPGA bitfile and new code for the microcontroller.

We expect it will be easy to show that any FPGA device is faster, but creating one that is also more energy-efficient for heavy encryption loads could be more challenging. The efficiency of FPGAs with respect to total power consumption for encryption is noted in [5], and we hope to reproduce their results, but it's worth noting that they used a customized PCB with only related components and not a prefabbed starter board. We haven't yet measured how much standby power the Digilent Spartan 3E Starter Board (see figure 3) consumes, and so we're unsure whether it will be appropriate to use in a situation where power consumption must be minimized.

If we end up using a Spartan 3E like the one that's on the Digilent board, we also note the necessity of completely powering down the FPGA every time it is idle. On restart, it would have to be reconfigured with the bitfile. Due to the lack of a suspend mode, there may only be a net energy savings for constant, heavy loads. On the other hand with a 3L or 3A there is a good possibility that we'll be able to suspend the chip frequently and come out ahead on energy consumption even for light or infrequent use of the device. This is another reason developing a custom board would be helpful: we could use the Spartan 3 size and model most ideally suited for the job.

3 System Design/Architecture

Considering the nature of sensor nodes and the specified constraints, we believe the best way to implement the FPGA expansion board is to purchase a low-power, discrete FPGA and design a PCB for direct connection to the sensor nodes through the Hirose 51-pin expansion connector on the motes. This is quite an ambitious project, and admittedly the plan may not come to fruition in the time we have available. Still, we've provided many details herein that describe our vision for the platform. Even if we don't get to this part of the project, this documentation may prove useful for any future groups who work on it.

3.1 Device Communication Process

Initially we considered using Block RAMs on the FPGA to buffer incoming data so it was available locally when requested by the encryption code. After some discussion we concluded that it would make sense to send the data "just in time," especially since for AES the bytes for the key and the data are requested by the core in the same order each time. In this case discrete chunks of the data and the key will go across the bus alternately until all of the data has been transferred. We acknowledge that it is wasteful of both power and transmission bandwidth to repetitively send data over the serial bus in this way, but since bandwidth will not be a constraint and this will simply the overall design immensely it seems like a worthwhile tradeoff. A total of 256 bits will be transferred for AES; 128 bits for the key, and 128 bits for the data.

In figure 1, we show our anticipated path for a chunk of data through the system. Initially data will be sampled from a sensor to memory in the microcontroller. Next the data would pass over a USART link from the microcontroller to the FPGA for encryption, and the encrypted data would pass back over the USART to the microcontroller. The microcontroller would then send the data to the CC1000 for transmission, and the data would be received by an identical CC1000 on a base station mote. The data would go through the mote's microcontroller and out over USB to a host PC where MoteView could be used to access and decrypt the data.

As mentioned in [2], there is some concern to have key data travelling over an open bus if the nodes are to be deployed in an untrusted environment. Our design does not make any effort to protect the key data, and so using a public-key core like Rabin would be essential if good security were desired in this situation.

3.2 Power Consumption

To determine whether it was really feasible to power an FPGA from two AA batteries, we examined the

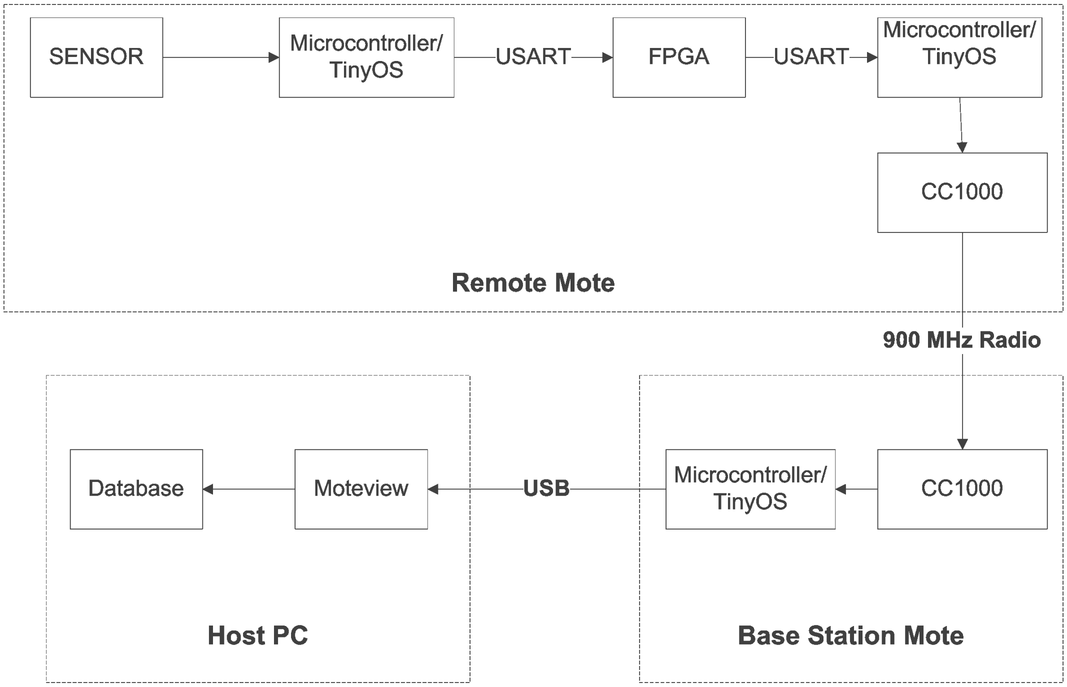


Figure 1: A representation of the dataflow during device operation.

power requirements of a Spartan 3AN and it's necessary support circuitry (a voltage regulator and perhaps a DC–to–DC converter, depending on the number of battery cells used). Note that this is not meant to be a highly accurate analysis, simply enough to determine whether it is feasible to power an FPGA from alkaline batteries.

Based on the smallest Spartan 3AN available, the XC3S50AN, our order-of magnitude estimate is that the device and its support circuitry will dissipate about 3W of power when operating in full active mode. To make the math simple, let's assume 3 battery cells in series, which means 1W of power would need to be provided by each cell.

Since Duracell alkaline AA batteries have characteristics similar to other alkaline batteries, we've used the Duracell AA alkaline datasheet to determine some general performance characteristics for AA batteries. Alkaline batteries operate over a fairly wide voltage range from 0.8V to 1.5V. If we average it out to assume that most of the time they are operating at 1.1V, then roughly 909mA of current will be needed from each battery on average. A constant 1 A drain would cause the batteries to be depleted extremely quickly, in less than one hour (total) of operation.

It looks like 3 AA batteries are not an appropriate power source for the FPGA, and that means the 2 that come with the mote are definitely not appropriate. We'll need to use a separate, higher capacity power supply for the FPGA. 3 D cell batteries would last longer; roughly 8 hours for 1 A of constant drain. We would not be honoring the requirements of keeping the size of the mote device small if we used multiple D cell batteries to power it, but this tradeoff may be unavoidable. 3 D cells are not likely to save much space vs 4 D cells, and including an extra cell would increase runtime significantly, so 4 D cells is probably the magic number.

Of course, the FPGA will not be operated at the full power consumption level for 3 straight hours: the idea is to only turn it on only very briefly when an encryption operation is desired and then turn it off again as quickly as possible. What follows then is a short analysis of the throughput of the device for AES and Rabin.

3.3 Throughput Analysis

Based on the numbers in [2], it seems likely that the FPGA will offer significant throughput and power consumption benefits for the more complex Rabin scheme but fewer if any benefits for the simpler AES scheme. A major part of the project will be to test these assumptions and report the results.

3.4 MICA2 Mote Pseudocode

What follows is a pseudocode-ish description of the behavior of the motes in the final system. Alas, we can't find a good way to integrate the MS Word 2007 which Tina wrote into this LaTeXdocument.

The mote to be connected to the FPGA add-on board will be programmed before it is connected to the FPGA module. This initial programming will allow the mote to communicate with another mote at the base station and also with the FPGA module. The mote to be connected to the base station will, then, be programmed to enable communication with other mote.

1. Pseudo-code for programming FPGA mote:

   1.1 Communication link with base station mote:

```
➢ Initialize components
    o Leds: to signal stages of communication
    o PhotoControl: to sense Led toggles
    o Message Packet: data to be sent
        ▪ Set Header information -
                • Sensor Board ID = MDA100
                • Packet ID = *any number
                • Node ID = TOS_LOCAL_ADDRESS
                • Rsvd = 0
    o Timer: repeating timer to signal start data sampling
➢ Start Time
➢ Timer Fired = TRUE
    o Data Receive Ready
        • Toggle Red Led
        • Start Photo Control
    o Sample Data from sensor board at PhotoADC
        • Data sample done = TRUE
            ❖ Pass data to GenericComm.SendMsg
            ❖ Toggle Yellow Led
    o Deliver Sampled Data
        • Stop PhotoControl
        • Prepare Message Packet
            ❖ Pack message in packet
            ❖ Pack Header information for
```

1.2 Communication link with FPGA module:

- INITIALIZE COMPONENTS
  - Leds: to signal stages of communication
  - PhotoControl: to sense Led toggles
  - Message Packet: data to be sent
    - Set Header information -
      - Sensor Board ID = MDA100
      - Packet ID = *any number
      - Node ID = TOS_LOCAL_ADDRESS
      - Rsvd = 0

- START FPGA:
  - Data to be sent ready = TRUE
    - Send PowerOn signal to FPGA
    - Start timer for t = 10ns
    - Start PhotoControl
      - Timer expires
        - Check if micro-controller received READY signal
          - FALSE
            - ✘ No LED toggle
          - TRUE
            - ✓ Toggle Yellow LED
        - Check if Yellow LED toggled
          - FALSE
            - ✘ Powering FPGA – FAILURE, system not ON
          - TRUE

- TEST COMMUNICATION BUS:
  ‘TEST SEND’
  - Send test data sequence “0100..” on TX bus
  - Check if complete data sequence sent
    - FALSE
      - ✘ No LED toggle
    - TRUE
      - ✓ Toggle Green LED
    - Check if Green LED toggled
      - FALSE
        - ✘ ‘Send Stage’ of micro-controller failed
      - TRUE
        - ✓ Move to ‘Test Receive’

‘TEST RECEIVE’

- Start timer for t = 10ns
  - Timer expires
    - Check if micro-controller received same test sequence at RX bus
      - FALSE
        - ✗ No LED toggle
      - TRUE
        - ✓ Toggle Red LED
    - Check if Red LED toggled
      - FALSE
        - ✗ Communication Link setup – FAILURE
      - TRUE
        - ✓ Move to ‘ENCRYPTION’ stage

- ENCRYPTION:
  - Send signal for algorithm selection on TX bus (AES or RSA Algorithm)
  - Send data and key on TX bus
  - Check if complete data sequence sent
    - FALSE
      - ✗ No LED toggle
    - TRUE
      - ✓ Toggle Green LED
  - Check if Green LED toggled
    - FALSE
      - ✗ Required data not sent yet
    - TRUE
      - ✓ Complete data sent, wait for encrypted data
  - Check if micro-controller received DONE signal from FPGA
    - FALSE
      - ✗ Encrypted data not ready yet
    - TRUE
      - ✓ Encryption done – data ready to send back
  - Collect data at RX bus after receiving DONE signal
  - Check if more data available to be sent
    - TRUE
      - ✓ Repeat from ‘ENCRYPTION’ stage
    - FALSE
      - ✗ Move to ‘SUSPEND’ stage
- SUSPEND:
  - More data available with mote = FALSE
    - Send ‘Suspend’ signal to FPGA
    - Start timer for t = 10ns

- Start PhotoControl
  - Timer expires
    - Check if micro-controller received OK signal
      - FALSE
        - ✗ No LED toggle
      - TRUE
        - ✓ Toggle Yellow LED
    - Check if Yellow LED toggled
      - FALSE
        - ✗ Suspending FPGA – FAILURE, FPGA still ON
      - TRUE
        - ✓ Start timer for time t for which mote waits for more data to be sent
          - Timer expires
            - Check if micro-controller has more data available to be sent for encryption
              - FALSE
                - ✗ No data available – Move to ‘POWER OFF’ stage
              - TRUE
                - ✓ Send WAKE UP signal to FPGA
                  - Check if micro-controller received OK signal
                    - FALSE
                      - ✗ No LED toggle
                    - TRUE
                      - ✓ Toggle Yellow LED
                  - Check if Yellow LED toggled
                    - FALSE
                      - ✗ Disconnecting FPGA – FAILURE, system not OFF
                    - TRUE
                      - ✓ Move to ‘ENCRYPTION’ Stage

- POWER-OFF FPGA:
    - (Data available = FALSE) & (Wait timer expired = TRUE)
        - Send PowerOff signal to FPGA
        - Start timer for t = 10ns
        - Start PhotoControl
            - Timer expires
                - Check if micro-controller received OK signal
                    - FALSE
                        - ✗ No LED toggle
                    - TRUE
                        - ✓ Toggle Yellow LED
                - Check if Yellow LED toggled
                    - FALSE
                        - ✗ Disconnecting FPGA – FAILURE, system not OFF
                    - TRUE
                        - ✓ Move to 'START FPGA' stage

2. Pseudo-code for programming FPGA to communicate with the mote: Corresponding to the Finite State Machine diagram provided
    - POWER ON
        - Received voltage signal from voltage controller
            - Turn on FPGA
            - Read Flash memory
            - Return DONE signal to micro-controller on mote
            - Transfer to TEST state
    - TEST
        - Received signal "010…" at TX bus
            - Repeat signal in response
            - Return same signal on RX bus to micro-controller on mote
            - Transfer to READY state
    - READY
        - Detect Algorithm selection signal at TX bus
            - Transfer to AES or RBN state according to selection signal
            - Start gathering data from TX bus
    - EXECUTION
        - Received required data at TX bus from micro-controller
            - Execute encryption algorithm
            - Transfer to DONE state
    - DONE
        - Completed execution and transferred from EXECUTION state
            - Return output on RX bus to micro-controller on mote
            - Return DONE signal to micro-controller

- SUSPEND
    - Received SUSPEND signal from micro-controller
        - Return OK signal to micro-controller
        - Turn on suspend mode of FPGA
        - Wait until receive wake up signal
    - Received WAKE UP signal from micro-controller
        - Return OK signal to micro-controller
        - Turn on FPGA power
        - Transfer to READY state
- POWER OFF
    - Received POWER OFF signal from micro-controller
        - Return OK signal to micro-controller
        - Turn off FPGA power
        - Wait until receive wake up signal

3.5 nesC Program Structure

The nesC language is essentially C with some added constructs for creating “components” and enabling concurrency. All nesC applications consist of one or more components assembled (or wired) together statically to form an executable image. The components provide and use interfaces. “Provided” interfaces represent the functionality the component provides to its user, while “used” interfaces represent the functionality the component needs to perform its job [6].

In figure 2 we show an automatically generated flow diagram for our USART test code created using the free GraphViz tool. Since large, well-designed nesC applications tend to have a little bit of code spread out over many small text files, it can sometimes be hard to grasp the purpose of the various linkages all at once. The use of the GraphViz tool to create these component maps makes the code layout easier to understand.

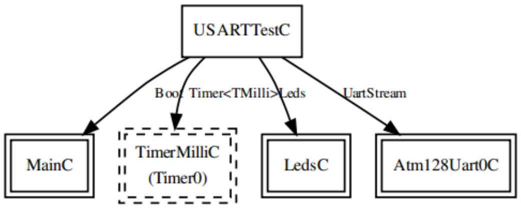


Figure 2: Automatically-generated flow diagram for our USARTTest application.

3.6 Voltages

Spartan-3 devices are designed and characterized to support various I/O standards for V_{CCO} values of +1.2 V, +1.5 V, +1.8 V, +2.5 V, and +3.3 V. The ATMega128L ($V_{cc} = 3.0\,\text{V}$) supports an input low voltage from -0.5 V to $0.2V_{cc}$ (0.6 V) and an input high voltage from $0.6V_{cc}$ (1.8 V) to $V_{cc} + 0.5\,\text{V}$ (3.5 V). The output low voltage is 0.5 V and the output high voltage is 2.2 V. We anticipate that if the FPGA IO pins that are used to connect to the mote are configured in TTL logic mode, there will be no voltage-related issues that hamper communication between the devices [7],[8].

4 Detail Design

This section provides implementation details about the designs.

4.1 Initial Testbed and Preliminary Design

For preliminary testing purposes, we are using the Digilent Spartan 3E Starter board (provided by Dr. Kaps), shown in figure 3.

We’ll be using this board to ensure that our USART interconnections work, that the software developed for the motes is effective, and that the cores

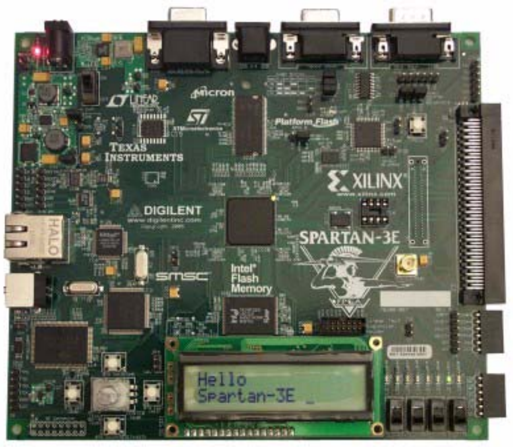


Figure 3: The Digilent Spartan 3E Starter Board (©2007 Digilent)

implemented on the FPGA return the expected results for the keys and data that are input.

Since we are going to be using this board anyway to begin with, we will always have it as a backup in case our more ambitious plans for a final design don't quite work out. The board has a Hirose 100-pin expansion connector that we would need to interface with the MDA100CB board shown in figure 12. Since this demo board has all the bells and whistles, there is no need for us to add external memory or voltage regulation and the main design boils down to simply interfacing the expansion connector on the demo board to the expansion connector on the sensor nodes. This design will be larger and consume more energy than a full-custom PCB design, and as such it will not really be suitable for operation in a real mesh network.

4.2 Custom PCB

Shown in figure 4 are the functional blocks the PCB layout will consist of.

Texas Instruments has developed a triple voltage regulator specifically for the Spartan 3 series which supplies the correct voltages to the FPGA and takes care of many complex regulation issues. Our proposed PCB will not need a flash memory device because it is included on the 3AN FPGA.

The benefits to designing a PCB for a discrete FPGA are two-fold: first, the power consumption will be much lower than using the Diligent Spartan 3E demo board because there will no superfluous components to power, and second it enables us to choose a small Spartan 3AN FPGA that supports suspend mode and lower power operation. It would also enable us to choose a low-power FPGA model. This means, however, that we would need to order an FPGA from the Spartan 3A or 3L series, which would increase our budget somewhat. Considering that we did not have to pay for the motes as they were loaned to us by Dr. Kaps, there is plenty of room in our budget for the components we will need for this design. The second benefit is size. The Diligent board is a large piece of equipment, and given the nature of sensor nodes and the constraints we have specified, size is a major factor. Naturally, an expansion board designed with a stand-alone FPGA would be much smaller.

Lastly, we will order the 51-pin expansion connector from Digikey and affix it to the PCB such that the finished product can be plugged directly into the mote. This adds to the convenience, appearance, and sturdiness of the device as there will not be a lot of unsightly wires and volatile connections to constantly worry about. These features are secondary concerns but they would make for a better design.

There will be added difficulties in designing the FPGA expansion in this way, as tasks which would have already been done for us had we decided to use the Diligent board will need to be addressed. Most notably we will have to deal with power compatibility, whether to use serial or USB ports, memory and initial configuration, and other interconnection issues. A starter board or even a basic breakout board would have taken care of much of this for us.

One additional concern is whether we can fabricate the PCB ourselves or whether we will need to send it to a manufacturer, and then whether we would attach the parts or pay for the fab to do it. Doing everything on our own using board stock, etching solution, a laser printer, an iron, and a drill would be interesting and fun, but there are many potential pitfalls. Two major issues are the size of the traces for the FPGA, and the very small holes that would need to be drilled for the expansion connector.

Choosing the best size for the FPGA is also challenging as well. Larger sizes offer more flexibility for simulating larger cores (like the public-key Rabin, and any others that Dr. Kaps might like to try) but also consume significantly more power. Another big issue is that most large FPGAs are only available in a ball-grid array pattern which is impossible to solder by hand. Some “breakout”–type boards for ball grid array packages do exist, but they are large, unwieldy, and expensive. After much consideration we have decided that any FPGA package that uses ball grid array pins will not be acceptable for this project.

4.3 Device Communication Interface

After much investigation between possible communication methods, we concluded that a USART connection operating in asynchronous mode would be the ideal communication method, mostly because it allows us to operate the mote and the FPGA from unsynchronized clock sources. As long as the clocks have sufficient accuracy and their speeds are integer multiples of one another, the USART connection will be highly accurate.

Being able to use unsynchronized clock sources is a huge advantage because we are then able to use the 8MHz clock provided by the ATMega128's inter-

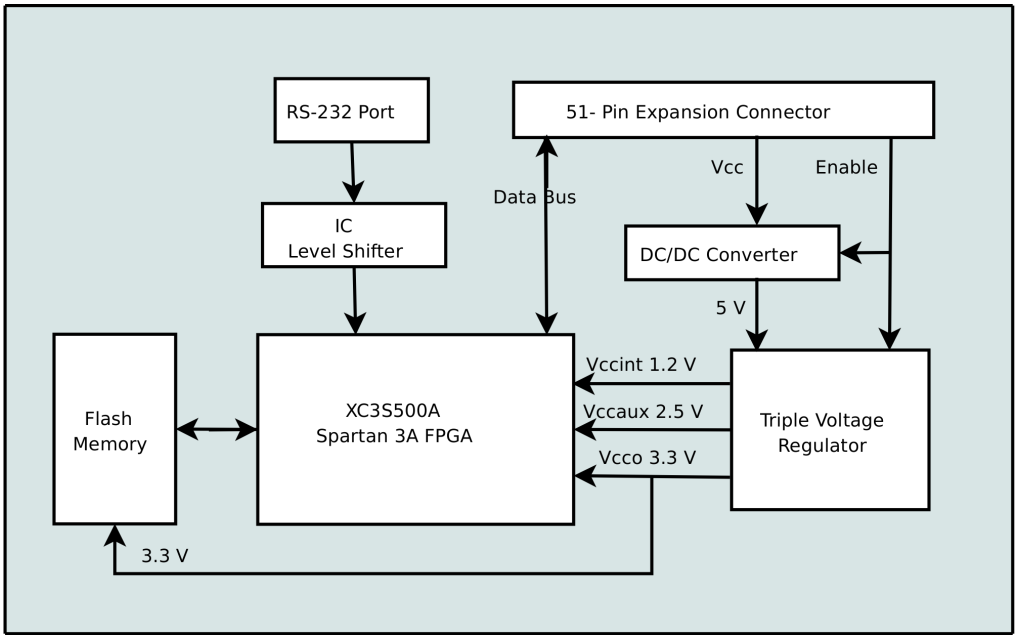


Figure 4: Proposed PCB Components Block Diagram

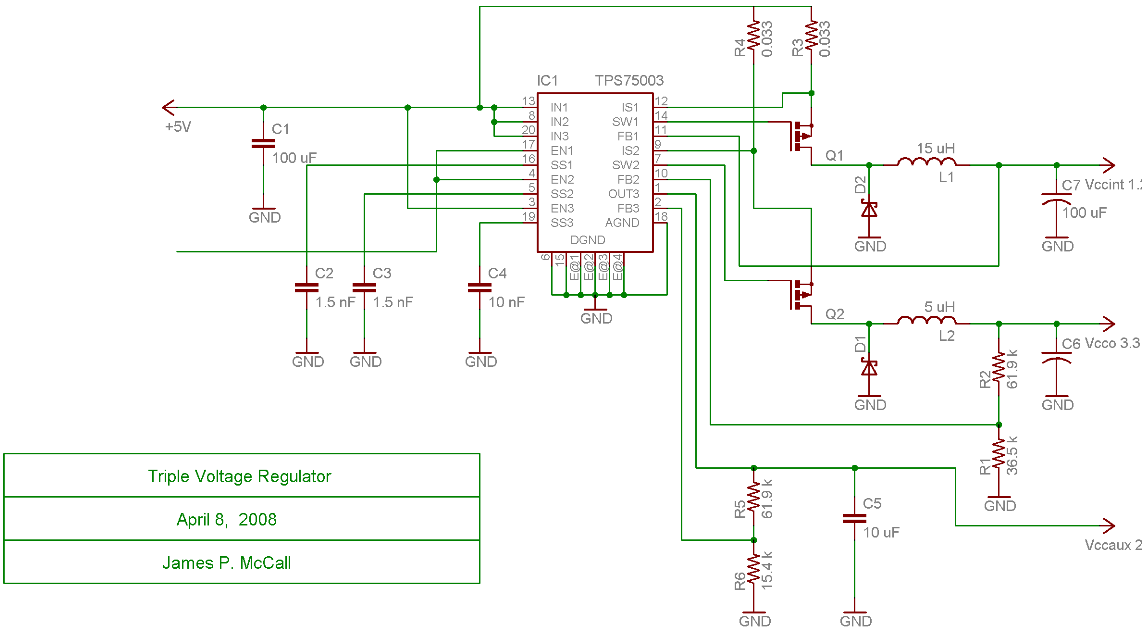


Figure 5: Schematic of the triple voltage regulator circuit.

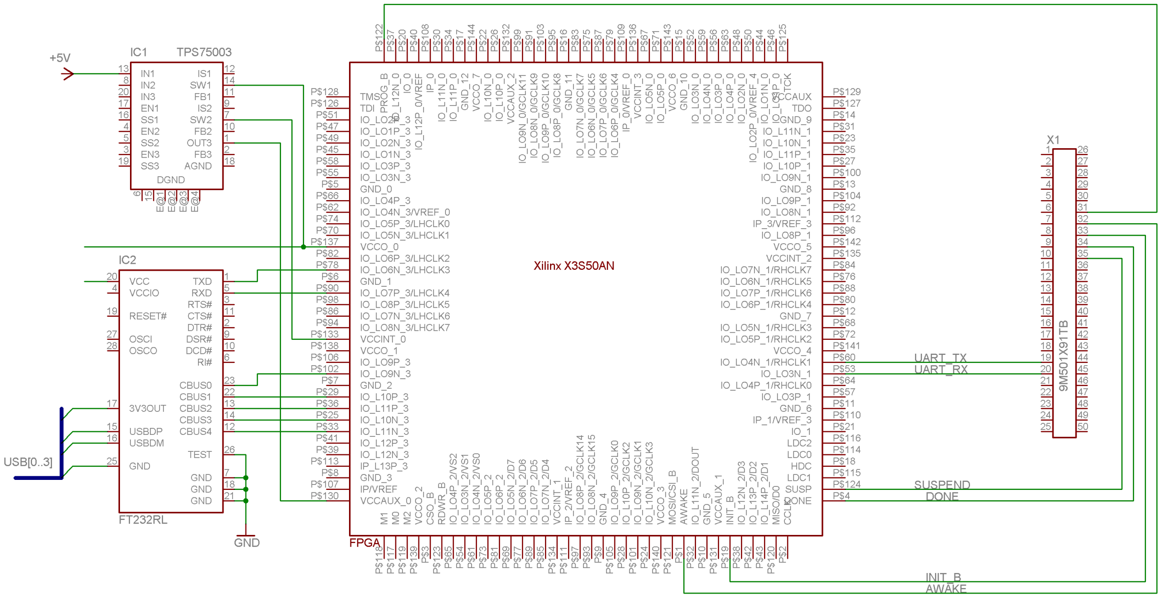


Figure 6: FPGA Schematic

nal oscillator rather than the $\tilde{7}$MHz external oscillator provided by the mote (which is the only clock available on the data bus). The 8MHz oscillator consumes less power and is recommended by Crossbow for battery-powered operation.

A USART device's maximum possible data rate is based on its input clock rate: the input clock rate divided by 16 gives the maximum data rate (or divided by 8 for DDR). On the FPGA we are limited by the maximum speed of the cores (slightly less than 80MHz) and the available oscillator (can be changed), while on the mote we are limited by the maximum speed of the microcontroller oscillator (8 MHz). A 64 MHz oscillator for the Spartan 3e starter board would probably be ideal because it divides evenly to 8MHz for higher rate UART communication, allows a higher FPGA calculation rate, and costs only $3 from DigiKey (SG531). As an alternative to using an external oscillator, we could consider simply using a DCM on one of the FPGAs to create a modified clock signal at the appropriate frequency.

A very high data rate is probably unnecessary; based on our calculations only about 100kbit/s are required for the AES core, and less would be required for the Rabin core. It's also worth noting that the maximum transmit rate for the mote is only 38.4K baud and thus even if the microcontroller can talk to the mote at the higher rate it can't actually send the data anywhere. Clearly then the FPGA can be run at a somewhat higher speed than is necessary for continuous transmission of data with this particular hardware.

On the other hand, since demonstrating the high throughput of the FPGA is an important part of the project, it makes sense to try and get as high of a bandwidth link between the FPGA and the microcontroller as we can even if we can't transmit the resulting data in realtime from the microcontroller.

To implement the UART, we will use (and modify if necessary) code written by someone else and made available as open-source on `www.opencores.org`. The license for the code is compatible with our project. It will be necessary to synchronize the UART settings between the ATmega and the VHDL code. For instance, the VHDL code is currently configured to use 1 start bit, eight data bits, and one stop bit. The ATmega supports this configuration, but it will need to be told that this format is going to be used during the initialization process.

4.4 Pin Functions

This section lists the functions of and our purposes for the specific FPGA pins we'll be using.

PROG_B When pulsed low, this pin causes the FPGA to reprogram from one of multiple sources. With the Spartan 3AN, it can be used to configure the device from an internal flash memory source.

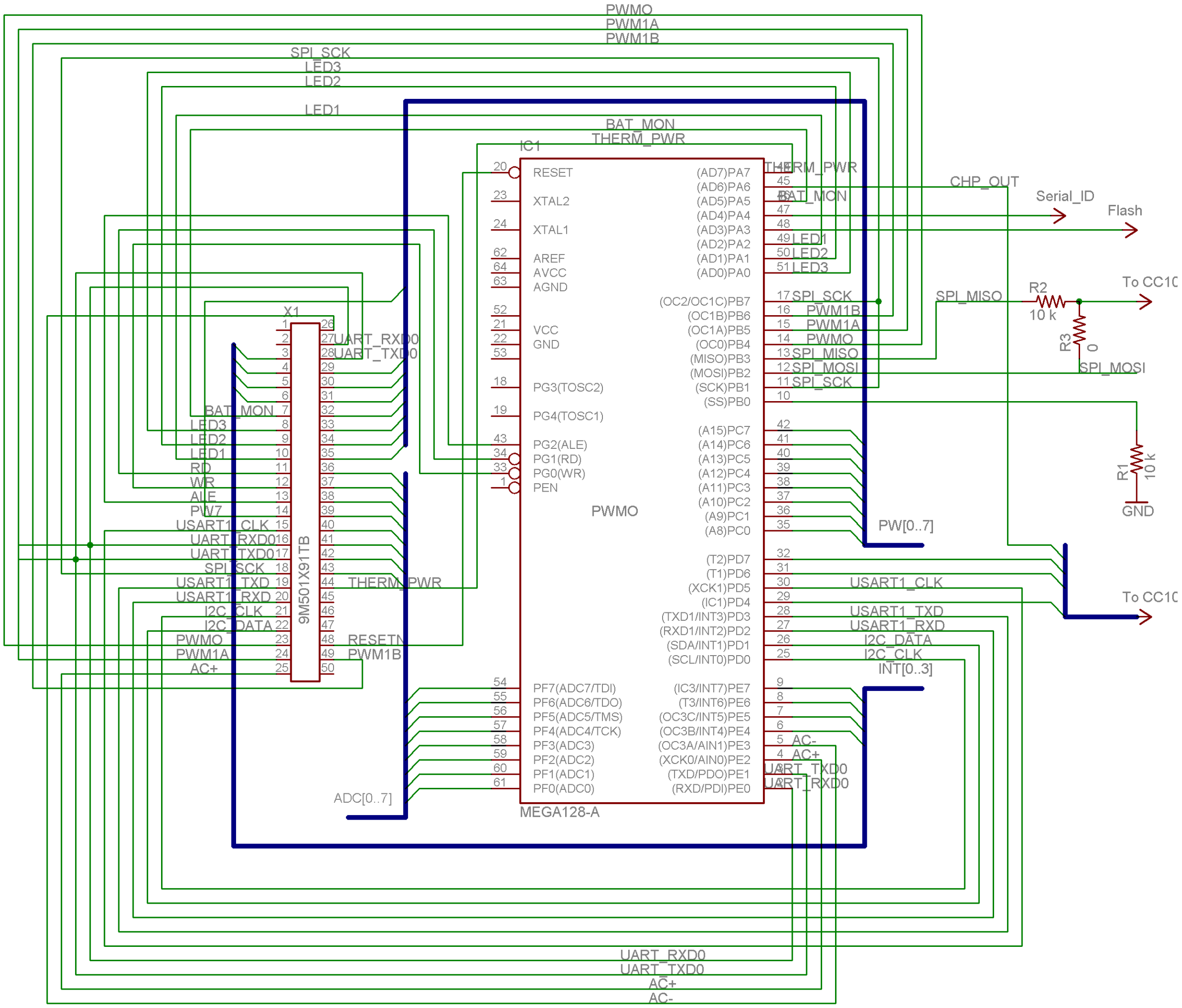


Figure 7: Schematic of the Microcontroller.

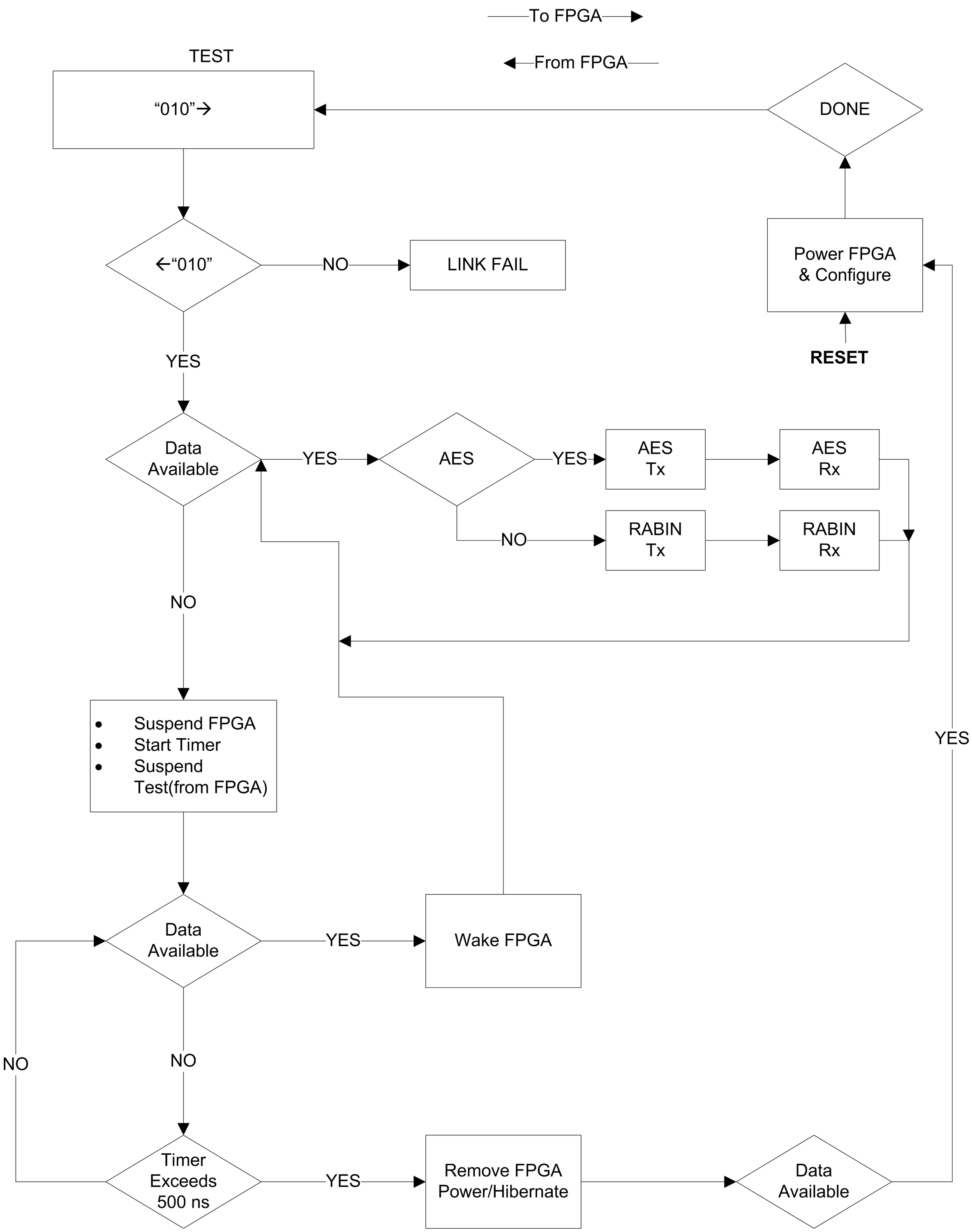


Figure 8: State diagram for the control on the mote.

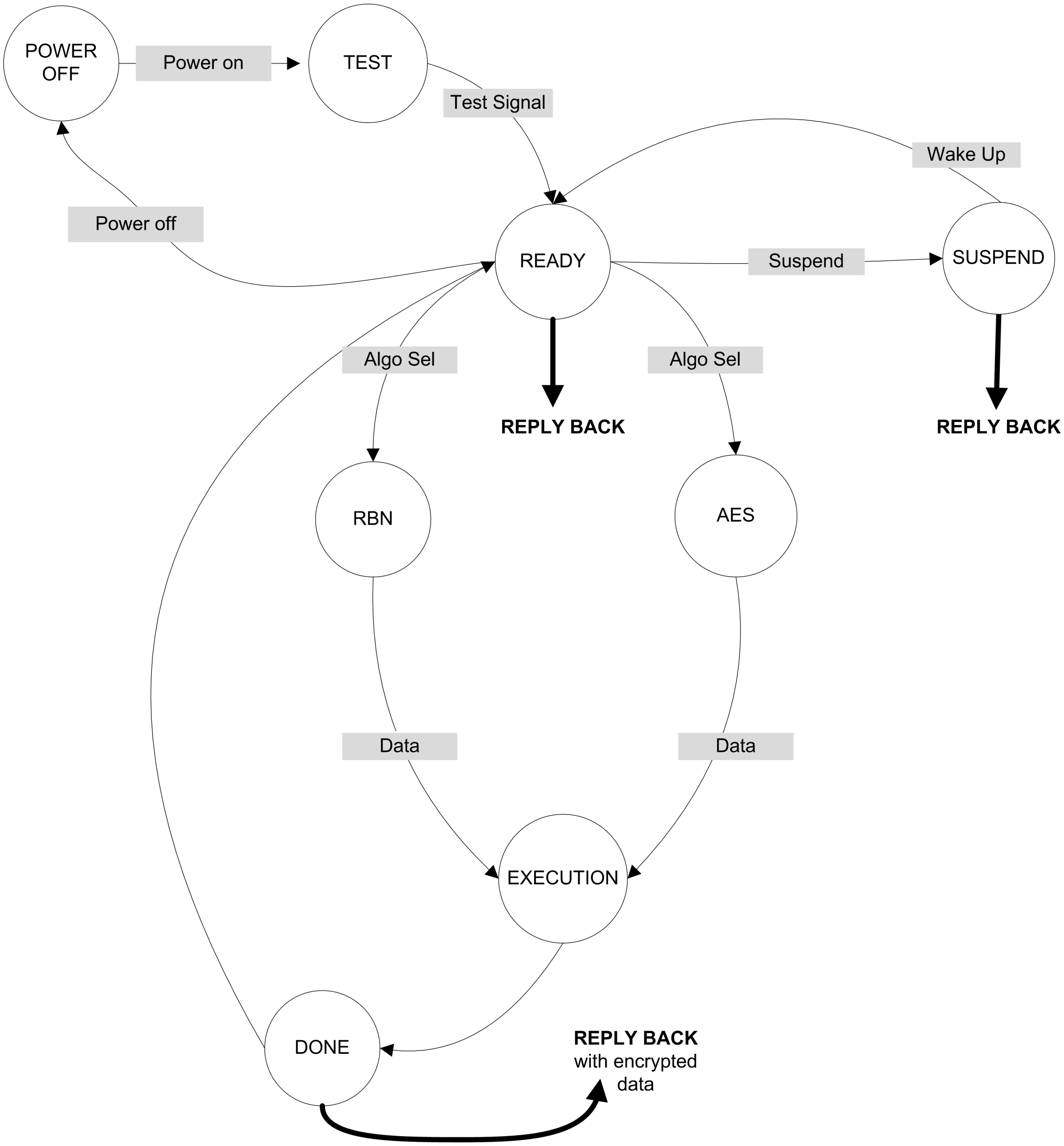


Figure 9: FPGA State diagram.

With the Spartan 3E starter board, it can be used to configure the device from an on-board Xilinx platform PROM which can be preconfigured via USB. We can connect to to the pin on the Digilent board using the JP8 header.

DONE This signal is asserted when the FPGA has finished configuration.

AWAKE This signal is deasserted when the FPGA enters suspend mode.

SUSPEND This pin can be driven high to put the FPGA into a low power state.

Two general (non-differential I/O) pins will also be used for the Tx and Rx connections of the USART. Differential I/O should be unnecessary at the speed we plan to use the USART.

5 Experimentation Plan

Our experimentation plan involves testing all the components as a system to verify that it performs the required tasks. We begin with a description of the dataflow and then move on to the details of how we'll verify correct operation.

The mote connected with the FPGA will be a remote, battery-powered device. This mote will also be connected with the sensor board and will be programmed to sense some physical stimulation like temperature, pressure, humidity, or other physical values at specific time intervals. The sensor board will gather data and pass it to the microcontroller of the mote. To acquire encrypted data for transmission, the microcontroller on the FPGA mote will transfer data to the FPGA, which will encrypt it and then send the encrypted data back to the mote. This mote will next transmit the encrypted data to the mote attached to the programmer and a personal computer. XServe and XSniffer will be used to monitor and document the communication flow between the two motes. The sensor nodes will be programmed with TinyOS applications, using MoteWorks, to detect and acknowledge the networking communication between the microcontroller and the FPGA.

5.1 Hardware Components

The hardware components will be tested to ensure proper functioning of the devices both individually and as a system. The devices themselves will be tested as follows:

1. Programming board (MIB520): The MIB520 (see figure 10) will be used to install application programs onto the sensor nodes. We will use NesC to code the programs and the tool Programmers Notepad 2 or a similar text editor to compile and download the programs to the sensor nodes.

Figure 10: The MIB520 User Interface Board (©2006 Crossbow Technologies)

2. Sensor Nodes (MICA2): Each MICA2 mote (see figure 11) will be first tested individually and then operating in tandem as a small point-to-point network. For proper program installation and accurate operation in accordance with the application specifications, each node will be connected with the programming board. The code will be compiled and downloaded onto each node via the MIB520. The correct loading of the code can be verified by using the three LEDs available on each sensor node. Once the nodes are programmed, they will then be configured to operate as a point-to-point network.

XSniffer and XServe will be used to test the communication activity in the network. XServe provides the platform to monitor the communication between the base station and the wireless node in the mesh network, while XSniffer allows monitoring of the multi-hop communication between the nodes in the mesh network.

3. Sensor board (MDA100): The sensor board (see figure 12) will provide the gateway between the mote and the FPGA if we choose one of the applications that doesn't involve building a custom PCB. In this case the FPGA pins will be connected to the fifty-one pins on the sensor board and one of the nodes will be programmed to read the sensor board activity. The node can transmit samples of information from the sensor board to another sensor node at the

Figure 11: The MICA2 916 Wireless Mote (©2006 Crossbow Technologies)

base station or in the mesh network, although we will not be testing this application.

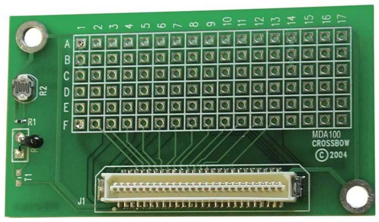


Figure 12: The MICA2 MDA100 Sensor Board (©2006 Crossbow Technologies)

5.2 Evaluation Criteria

The final system will be primarily evaluated on the power consumption of the device. Considering the future extension of this project to an implementation using ASICs instead of FPGAs, the simulation results of the FPGA system will be compared to the estimated results obtained from the ASIC simulation. The results will be divided into three sections—speed, number of resources on chip being used, and the power consumption of the chip—and each will be analyzed for compliance with the requirement specifications provided earlier in this document.

The system will also be evaluated on the following secondary criteria: reliability, accuracy, compatibility, and compactness.

Reliability: The code used for programming the motes and encrypting data on the FPGA should be reliable under all circumstances, including varying speeds of transmission, varying levels of network traffic, and different levels of load on the encryption unit. For practical reasons we won't be testing the equipment to an extreme degree or specifying all possible program states like NASA would, but these issues are still of practical concern.

Accuracy: The FPGA will be used to run encryption algorithms for a secure data transmission between the sensor nodes. The VHDL code running on the FPGA and the hardware itself must provide accurate results every time a transmission is made.

Compatibility: Ideally, the code for programming the FPGA should be simple in complexity and easily transferred to another FPGA family. We don't have much control over Dr. Kaps' code or anything we download from `www.opencores.org`, but we will try to ensure that our code meets these standards. Also, since the future concept for this project is an implementation using an ASIC instead of FPGA, highly-compatible code will provide the preliminary groundwork for such an implementation.

Compactness: Since a fundamental characteristic of sensor nodes is their small size, the add-on module must have a similar form factor. A compact, application-specific FPGA implementation is highly desirable but considering the monetary and time constraints, it may not be achieved.

6 List of Tasks

This is the list of tasks for the project. Many prototyping tasks listed have already been completed and contribute to the research and specifications in the other sections of this design document. All the tasks are listed for completeness' sake.

Preliminaries I thought we should get these tasks done right away to help the team get familiar with the project and the tools we'll be working with.

Toolchain setup and installation A lot of support software is required for this project. We need the MoteWorks software distribution, drivers for the mote programmers, Xilinx ISE for the FPGA, EAGLE for the PCB layout, and whatever software we'll be using for ASIC simulation. For the most part this step is complete although we all have different versions of the toolchain.

Hardware testing and verification Nothing is worse than spending hours trying to figure out why your code doesn't work and then realizing that it's because your hardware platform is bad. As such, we'll be testing all of the hardware (motes, FPGA starter board, any purchased hardware) before we try to use it. So far we've run sample programs on the motes, but we haven't had a chance to test the Digilent board yet.

Demo Board Setup The first setup we'll develop will use the demo board as a target platform. In the case that the power-efficient setup runs into problems, we'll also have this platform to fall back to for demonstrations and testing.

Locate/Obtain FPGA encryption core For the final part of the project we expect to use a core supplied by Dr. Kaps, but in the interim we may use an open-source core from `www.opencores.org`.

Core synthesis/implementation We need to make sure early that the core will synthesize on a Spartan 3 target. We should also figure out the smallest chip that the core will synthesize on.

Mote connections We need to connect the demo board to the mote. These will be temporary connections with probably an IDE cable attached to one of the pin headers on the demo board and as such shouldn't take very long.

Testing The fun part—making sure it works. If everything goes fine this will take 5 minutes, but if it doesn't we'll have a lot of work to do troubleshooting. That's why I allocated so much time for this.

Power consumption measurements Once everything is verified working, we can measure the power consumption for our reports. This will be a combination of simulation results and physical measurements. It'll be interesting to see how much power the demoboard consumes and how much we can potentially save with a custom PCB. If the amount is small, perhaps we'll rethink our plan at this point. Or perhaps we should find a way to measure it earlier.

Throughput analysis We'll need to find out how much data the core is passing when it's running at full speed. The throughput could be limited by the speed of the communication link between the FPGA and the microcontroller, and we'll need to figure out what to do about this issue if it is (preferably before we get there).

Mote Software The software running on the mote is a significant part of the project. We've already made some good progress understanding the

Locate encryption implementation We want to run an implementation of the same algorithm that's going to be used on the FPGA on the microcontroller so that we can compare power consumption and throughput for both of the devices. The first step is finding an implementation.

Throughput analysis After the implementation is compiled and running, we'll see how much data we can get through. From there it's a trivial step to determine how many blocks of data per unit energy is being used.

Communication software (for testing interface) Although we could go directly to a device driver, we thought it might be simpler for development to get some code running on the microcontroller that does something stupid like flip the output bits of the communication interface every clock cycle. This will probably make it easier to troubleshoot the connections and we'll also be able to get it done earlier so work on these two main tasks can proceed in parallel.

Device driver The device driver itself could be a bit of a challenge. As we noted earlier, a significant amount of information is available on creating device drivers for TinyOS 1.1 (which MoteWorks is derived from) and 2.0 but documentation for the MoteWorks platform is scarcer. We've allocated a lot of time for this task.

ASIC Simulation The purpose of doing an ASIC simulation is for our learning experience and to demonstrate the advantages of ASICs. Obviously we wouldn't be developing one.

Synthesis We gather the first step is running the VHDL through a synthesis tool, similar to the VHDL workflow we're already familiar with. The impression I got from Dr. Gaj's slides was the GMU has one licensed, but if that's not the case we'll need to reevaluate our options.

Projected power consumption This may include a serious amount of estimation. There are tools available that automate this process, but probably not student versions and GMU probably doesn't have them licensed.

Projected throughput analysis If the synthesis tool works, hopefully it will provide a maximum clock cycle and we'll be able to use that number to determine throughput.

Power efficient setup For the final deliverable, we'd really like to create an FPGA package that does the job while using the minimum amount of power possible. If it turns out to be impossible to get this done before the deadline, we'll get as far through the process as we can.

Build schematic We've tentatively decided to use the EAGLE software for this part of the project. The first step is creating a schematic. Depending on whether we decide to implement the power management components on the chip or buy a premade solution, we may simply copy significant amounts of this from the TI datasheet (they have a nice power controller IC for the Spartan 3s, but it needs a lot of external components)

Circuit layout Once the schematic is complete we have to do layout. I'm not very familiar with this process, but we'll learn as we go. It would be nice to use SMD components (apparently it is possible to affix them yourself if you have a steady hand), but we'll see as we go along.

Purchase board and components We're still a little shaky on whether we'll have time to implement the voltage controller on the board and whether we'll be able to solder an FPGA with all those tiny pins ourselves (others have done it, but we're not pros!) Thus, we don't know exactly which parts we need to buy yet. For the PCB, we'll send the schematic off to one of to the cheapest Chinese supplier we can buy. Will it work? Who knows, but it'll be interesting.

Assemble board Some PCB suppliers seem willing to affix components they stock for a low cost, but I suspect we'll still end up doing it ourselves. I'm more nervous about doing SMD components than through-hole ones, but we'll see how it goes. The FPGA will be the only really challenging one.

Testing Naturally there's going to be a lot of testing and this may well be the stage where we get stuck. Ideally this stage will not only check whether "it works," but also the many other constraints we specified in the design requirements section.

Power consumption measurements After we've ensured the board is working correctly we'll measure the power consumption of the FPGA by putting an ammeter inline with the power supply while the chip is operating. The result should be close to the simulation, plus the inefficiencies in the voltage regulators.

Throughput analysis If everything's working, our final task is to ensure our throughput results from simulation match what we're actually getting in the hardware.

6.1 Allocation of responsibilities

We are no longer using project management software to allocate responsibilities to our team members. Instead, this simple list outlines the tasks each member will ultimately be held responsible for:

- Faizul
  - PCB Layout
- Tina
  - Mote software design and implementation
- James
  - VHDL code and core glue logic
- Brandon
  - LaTeXdocumentation
  - Technical assistance with all other areas

7 Schedule and Milestones

Our schedule includes everything from the date we started tracking tasks, including project milestones and deliverables for this semester and next semester. The Gantt chart is shown in figure 13.

8 Appendix A: Parts List and Estimated Project Cost

In table 1 we provide a preliminary estimate of the costs for both the basic design and the PCB layout. Many of the costs for components used in the basic design are listed as zero because Dr. Kaps has graciously provided the equipment. This does not imply they have no cash value!

All estimated costs do not include shipping and handling, which is included as a separate line item and is somewhat unpredictable. Although we have done research to ensure that we can obtain all the parts listed in quantities of one at these prices, the most unpredictable factor in the total cost of the project will be unanticipated purchases.

Part Description	Specification	Quantity	Cost
MICA 2 916 Wireless Motes	-	2	$0.00
Programming board (MIB520)	-	1	$0.00
Sensor board (MDA100)	-	1	$0.00
Digilent Spartan 3 Starter Board	-	1	$0.00
Oscillators	56 MHz	1	$3.00
	64 MHz	1	$3.00
	72 MHz	1	$3.00
Hirose 100-pin female and cable	FX2B-100SA-1.27R	1	$9.50
Hirose 51-pin connector	DF9B-51S-1V	1	$3.13
Spartan X3S50AN FPGA	4TQG144C	2	$25.00
Triple-voltage Regulator	TPS75003-EP	1	$4.00
PCB printing fees	Site not determined	1	$75.00
USB Header	URB-1001B	1	$3.00
Dual USB UART/FIFO IC	FD2232D	1	$6.99
Diodes	Vishay SS32	1	$0.32
	On-semi MBRM120	1	$0.49
MOSFET	Siliconix Si232BDS	2	$2.66
Inductors	Sumida CDRH6D38-5R0	1	$1.10
	Sumida CDRH8D43-150	1	$1.10
Capacitors	10 pF	1	$0.30
	1.5 nF	2	$0.60
	0.01 uF	1	$0.30
	0.1 uF	2	$0.60
	1 uF	1	$0.30
	10 uF	1	$0.30
	100 uF	3	$0.90
Resistors	0.033	2	$0.60
	15.4 k	1	$0.30
	36.5 k	1	$0.30
	61.9 k	2	$0.60
Shipping and Handling	-	-	$40.00
Grand Total			$146.39

Table 1: List of parts, details, and their anticipated costs.

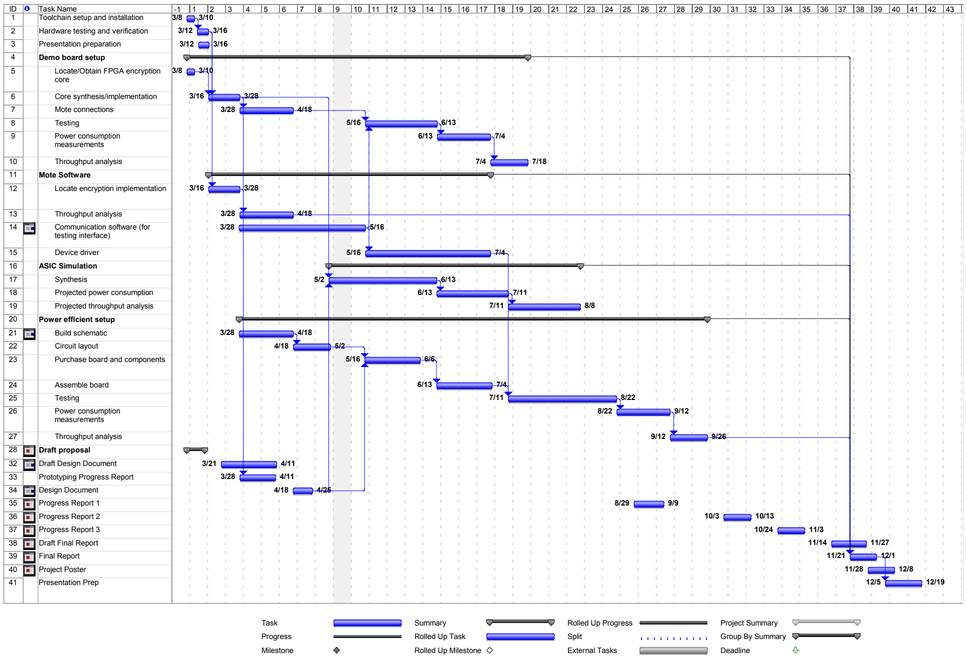


Figure 13: Project Gantt Chart.

9 Appendix B: Source Code

Although we are just getting started with this project, we've already written some source code to test out the various hardware components (especially the USARTS) that will be essential during the hardcore prototyping period. We don't want to run into any unanticipated problems, and the best way to avoid them is to test early and test often. Also be sure to see Figure 2 for the flow of the nesC test component.

Regrettably syntax highlighting is unavailable due to relative obscurity of nesC. For the sake of brevity, we're only including the code that we as a team have actually written; the UART implementation that we are using is not included, for instance, because the code was obtained from `www.opencores.org`.

9.1 USART Test NesC Code (for the Motes)

This section contains the nesC source code for the TinyOS 2.x toolchain that is compiled onto the motes.

9.1.1 USARTTestAppC.nc

```
configuration USARTTestAppC
{
}
implementation
{
    components MainC, USARTTestC, LedsC, HplAtm128UartC, HplAtm128GeneralIOC;
    components new TimerMilliC() as Timer0;
    components new TimerMilliC() as Timer1;

    USARTTestC -> MainC.Boot;
    USARTTestC.Timer0 -> Timer0;
    USARTTestC.Timer1 -> Timer1;
    USARTTestC.Leds -> LedsC;
    USARTTestC.HplAtm128Uart -> HplAtm128UartC.HplUart0;
    USARTTestC.Uart0TxControl -> HplAtm128UartC.Uart0TxControl;
    USARTTestC.Uart0RxControl -> HplAtm128UartC.Uart0RxControl;

    USARTTestC.PWPin0 -> HplAtm128GeneralIOC.PortC0;
    USARTTestC.PWPin1 -> HplAtm128GeneralIOC.PortC1;
    USARTTestC.PWPin2 -> HplAtm128GeneralIOC.PortC2;
    USARTTestC.PWPin3 -> HplAtm128GeneralIOC.PortC3;
    USARTTestC.PWPin4 -> HplAtm128GeneralIOC.PortC4; // in use
    USARTTestC.PWPin5 -> HplAtm128GeneralIOC.PortC5; // in use
    USARTTestC.PWPin6 -> HplAtm128GeneralIOC.PortC6; // in use
    USARTTestC.PWPin7 -> HplAtm128GeneralIOC.PortC7;

    USARTTestC.INTPin0 -> HplAtm128GeneralIOC.PortE4;
    USARTTestC.INTPin1 -> HplAtm128GeneralIOC.PortE5;
    USARTTestC.INTPin2 -> HplAtm128GeneralIOC.PortE6;
    USARTTestC.INTPin3 -> HplAtm128GeneralIOC.PortE7;
}

```

9.1.2 USARTTestC.nc

```
/**
 * Test module for the USART
 *
 * More text goes here, to give more details to the description.
 *
 * @author Brandon
 * @created  4/11/2008 First version! Huzzah!
 * @modified 5/31/2008 Added meaningful documentation.
 * @modified 6/7/2008  Debugging outputs for USART.
 *
 **/


#include "Timer.h"

module USARTTestC
{
    uses interface Timer<TMilli> as Timer0;
    uses interface Timer<TMilli> as Timer1;
    uses interface Leds;
    uses interface HplAtm128Uart;
    uses interface StdControl as Uart0RxControl;
    uses interface StdControl as Uart0TxControl;
    uses interface Boot;
    uses interface GeneralIO as PWPin0;
    uses interface GeneralIO as PWPin1;
    uses interface GeneralIO as PWPin2;
    uses interface GeneralIO as PWPin3;
    uses interface GeneralIO as PWPin4;
    uses interface GeneralIO as PWPin5;
    uses interface GeneralIO as PWPin6;
    uses interface GeneralIO as PWPin7;

    uses interface GeneralIO as INTPin0;
    uses interface GeneralIO as INTPin1;
    uses interface GeneralIO as INTPin2;
    uses interface GeneralIO as INTPin3;
}

implementation
{
    uint8_t blinkcounter = 0;
    uint8_t receivedData = 0;

    task void USARTbyteReceivedTask();
    task void bootTask();

    task void timer0task() {
        call HplAtm128Uart.tx(170);
    }

    // This task is used to blink the green LED twice at startup to verify
    // successful power on or programming
    task void timer1task() {
        call Leds.led1Toggle(); // green
        blinkcounter = blinkcounter + 1;
        if (blinkcounter != 4)
            call Timer1.startOneShot(500); //500 ms between toggles
    }

    // This event is called at boot-time
    event void Boot.booted() {
        post bootTask();
    }

    task void bootTask() {
        error_t result;

        call Timer0.startPeriodic(5000); // USART send timer
        call Timer1.startOneShot(500); // initial blink LED timer
        result = call HplAtm128Uart.enableTxIntr();

        if (result == FAIL)
            call Leds.led0On(); // red

        // Appears these pins are outputs by default but we’ll leave
        // this in there.
        call PWPin0.makeOutput();
        call PWPin1.makeOutput();
        call PWPin2.makeOutput();
        call PWPin3.makeOutput();
        call PWPin4.makeOutput();
        call PWPin5.makeOutput();
        call PWPin6.makeOutput();
        call PWPin7.makeOutput();

        call INTPin0.makeOutput();
        call INTPin1.makeOutput();
        call INTPin2.makeOutput();
        call INTPin3.makeOutput();

        // fire up the UARTS (consumes additional power)
        result = call Uart0RxControl.start();
        result = call Uart0TxControl.start();
    }

    event void Timer0.fired() {
        post timer0task();
    }

    event void Timer1.fired() {
        post timer1task();
    }

    /**
    * USART Byte received handler. Call a task to handle the data.
    *
    * @param data The received byte.
    *
    * @return none.
    **/
    async event void HplAtm128Uart.rxDone( uint8_t data ) {
        receivedData = data;
        post USARTbyteReceivedTask();
    }

    /**
    * if correct data received, toggle green LED. If incorrect data
    * received, toggle amber LED. Also output the value of the received
    * byte on some GPIO pins (PW0..7).
    *
    * @return none.
    **/
    task void USARTbyteReceivedTask() {
        uint8_t capturedData;

        atomic capturedData = receivedData;

        if (capturedData == 170)
            call Leds.led1Toggle(); // green
        else
            call Leds.led2Toggle(); // orange

        // clear all debugging pins
        call PWPin0.clr();
        call PWPin1.clr();
        call PWPin2.clr();
        call PWPin3.clr();
        call INTPin0.clr();
        call INTPin1.clr();
        call INTPin2.clr();
        call INTPin3.clr();

        // set debugging pins based on the data value
        if (capturedData & 1)
```

```
            call PWPin0.set(); // F2
        if (capturedData & 2)
            call PWPin1.set(); // F3
        if (capturedData & 4)
            call PWPin2.set(); // F4
        if (capturedData & 8)
            call PWPin3.set(); // F5
        if (capturedData & 16)
            call INTPin0.set(); // D5
        if (capturedData & 32)
            call INTPin1.set(); // D4
        if (capturedData & 64)
            call INTPin2.set(); // D3
        if (capturedData & 128)
            call INTPin3.set(); // D2
    }

    // do nothing when tx is done, but event needs to be specified anyway!
    async event void HplAtm128Uart.txDone() {

    }
}
```

9.2 USART Test VHDL Code (for the FPGA)

This section contains the VHDL code for the USART test that we synthesize onto the FPGA. The critical path delay after synthesis was 6.538 ns, which would enable a clock speed near 150 MHz. This bodes well for the maximum speed of the USART, indicating that at least this component of the FPGA will not be the speed bottleneck. We also include a testbench and our custom UCF file.

The code for the UART itself is available online from http://opencores.org/cvsweb.shtml/miniuart/. We have not modified this code for this application and do not anticipate any changes will be necessary.

9.2.1 topLevel.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.UART_Def.all;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity top_level is
    Port ( clk_50mhz   : in  STD_LOGIC;
           reset       : in  STD_LOGIC;
           rxd         : in  STD_LOGIC;
              txd         : out  STD_LOGIC;
              LED         : out  STD_LOGIC_VECTOR (7 downto 0));

end top_level;

architecture Behavioral of top_level is
```

```

component miniUART is
  port (
     sysclk   : in  Std_Logic;  -- System Clock
     Reset    : in  Std_Logic;  -- Reset input
     CS_N     : in  Std_Logic;
     RD_N     : in  Std_Logic;
     WR_N     : in  Std_Logic;
     RxD      : in  Std_Logic;
     TxD      : out Std_Logic;
     IntRx_N  : out Std_Logic;  -- Receive interrupt
     IntTx_N  : out Std_Logic;  -- Transmit interrupt
     Addr     : in  Std_Logic_Vector(1 downto 0); --
     DataIn   : in  Std_Logic_Vector(7 downto 0); --
     DataOut  : out Std_Logic_Vector(7 downto 0)); --
end component;

signal reset_l : std_logic;
signal cs_n, rd_n, wr_n : std_logic;
signal intrx_n, inttx_n : std_logic;
signal addr : std_logic_vector(1 downto 0);
signal datain : std_logic_vector(7 downto 0);
signal dataout: std_logic_vector(7 downto 0);
signal counter: std_logic_vector(3 downto 0);
signal temp: std_logic;

begin

reset_l <= not reset;
addr <= "00";
awesome_process: process(clk_50mhz)
begin
    if rising_edge(clk_50mhz) then
        if reset = ’1’ then
            temp <= ’0’;
            wr_n <= ’1’;
            rd_n <= ’1’;
            cs_n <= ’1’;
        elsif intTx_n = ’0’ then
            if temp = ’0’ then
                temp <= ’1’;
                datain <= X"AA";
                wr_n <= ’0’;
            cs_n <= ’0’;
            elsif temp = ’1’ then
                temp <= ’0’;
            end if;
        elsif intRx_n = ’0’ then
            RD_N <= ’0’;
            CS_N <= ’0’;
      else
         RD_N <= ’1’;
         CS_N <= ’1’;
         WR_N <= ’1’;
         DataIn <= "ZZZZZZZZ";
      end if;
    end if;
end process;

process(clk_50mhz)
begin
    if rising_edge(clk_50mhz) then
        if reset = '1' then
            led <= X"00";
        else
            if rd_n = '0' then
                if dataout = X"AA" then
                    led(0) <= '1';
                    led(1) <= '0';
                else
                    led(0) <= '0';
                    led(1) <= '1';
                end if;
            end if;
        end if;
    end if;
end process;

comp:  miniuart port map ( sysclk => clk_50mhz,
                           reset => reset_l,
                                     cs_n => cs_n,
                            rd_n => rd_n,
                            wr_n => wr_n,
                             rxd => rxd,
                             txd => txd,
                         intrx_n => intrx_n,
                         inttx_n => inttx_n,
                            addr => addr,
                          datain => datain,
                         dataout =>    dataout);

end Behavioral;
```

9.2.2 topLevelTB.vhd

```

--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   16:54:15 04/18/2008
-- Design Name:   top_level
-- Module Name:   C:/Xilinx91i/uart_code2/top_level_tb.vhd
-- Project Name:  uart_code2
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: top_level
--
-- Dependencies:
```

```
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY top_level_tb_vhd IS
END top_level_tb_vhd;

ARCHITECTURE behavior OF top_level_tb_vhd IS

    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT top_level
    PORT(
        clk_50mhz : IN std_logic;
        reset : IN std_logic;
        rxd : IN std_logic;
        txd : OUT std_logic;
        LED : OUT std_logic_vector(7 downto 0)
        );
    END COMPONENT;

    --Inputs
    SIGNAL clk_50mhz :  std_logic := ’0’;
    SIGNAL reset :  std_logic := ’1’;
    SIGNAL rxd :  std_logic := ’0’;

    --Outputs
    SIGNAL txd :  std_logic;
    SIGNAL LED :  std_logic_vector(7 downto 0);

    signal rxd_sel : std_logic := ’0’;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: top_level PORT MAP(
        clk_50mhz => clk_50mhz,
        reset => reset,
        rxd => rxd,
        txd => txd,
        LED => LED
    );
    clk_50mhz <= not clk_50mhz after 10 ns;

    tb : PROCESS
    BEGIN
        wait for 100 ns;
      reset <= ’0’;

        wait;
    end process;

    rxd_sel_proc : process(rxd_sel)
    begin
        if rxd_sel = ’1’ then
            rxd <= txd;
        else
            rxd <= ’0’;
        end if;
    end process;

    rxdproc : process
    begin
        rxd_sel <= ’0’;
        wait for 5 ms;
        rxd_sel <= ’1’;
        wait for 5 ms;
        rxd_sel <= ’0’;
        wait for 5 ms;
        rxd_sel <= ’1’;
        wait;
    end process;
END;
```

9.2.3 uartTest.ucf

```
NET "LED<7>" LOC = "F9"  | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<6>" LOC = "E9"  | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<5>" LOC = "D11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<4>" LOC = "C11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<3>" LOC = "F11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<2>" LOC = "E11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<1>" LOC = "E12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED<0>" LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;

NET "RESET" LOC = "H13" | IOSTANDARD = LVTTL | PULLDOWN;

# ==== Clock inputs (CLK) ====
NET "CLK_50MHZ" LOC = "C9" | IOSTANDARD = LVCMOS33 ;
# Define clock period for 50 MHz oscillator (40%/60% duty-cycle)
NET "CLK_50MHZ" PERIOD = 20.0ns HIGH 40%;
#NET "CLK_AUX" LOC = "B8" | IOSTANDARD = LVCMOS33 ;
#NET "CLK_SMA" LOC = "A10" | IOSTANDARD = LVCMOS33 ;

NET "Txd" LOC = "D7" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 6 ;
NET "Rxd" LOC = "C7" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 6 ;
```

References

[1] J. Yick. (2006, March) Wireless sensor networks. UC Davis Network Research Lab. [Online]. Available: http://networks.cs.ucdavis.edu/~yick/Research.html

[2] G. Gaubatz, J.-P. Kaps, and B. Sunar, "Public key cryptography in sensor networks—revisited," in *1st European Workshop on Security in Ad-Hoc and Sensor Networks (ESAS 2004)*, ser. Lecture Notes in Computer Science (LNCS), H. Hartenstein, C. Castellucia, C. Paar, and D. Westhoff, Eds., vol. 3313. Heidelberg: Springer, August 2004, pp. 2–18.

[3] (1998, October) Introduction to public-key cryptography. Sun Microsystems. [Online]. Available: http://docs.sun.com/source/816-6154-10/contents.htm

[4] G. Gaubatz, J.-P. Kaps, E. Öztürk, and B. Sunar, "State of the art in ultra-low power public key cryptography for wireless sensor networks," in *Third IEEE International Conference on Pervasive Computing and Communications Workshops, Workshop on Pervasive Computing and Communications Security–PerSec'05.* IEEE Computer Society, Mar 2005, pp. 146–150.

[5] P. Hamalainen, T. Alho, M. Hannikainen, and T. Hamalainen, "Design and implementation of low-area and low-power aes encryption hardware core," *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pp. 577–583, 2006.

[6] (2008, April) Tinyos documentation wiki (english). [Online]. Available: http://docs.tinyos.net/index.php/Getting_Started_with_TinyOS#Compiling_and_Installing

[7] (2007, December) Spartan-3an fpga family data sheet. Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds557.pdf

[8] (2007, August) Atmega128l 8-bit microcontroller with 128k bytes in-system programmable flash. Atmel Corporation. [Online]. Available: http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

References

[1] J. Yick. (2006, March) Wireless sensor networks. UC Davis Network Research Lab. [Online]. Available: http://networks.cs.ucdavis.edu/~yick/Research.html

[2] G. Gaubatz, J.-P. Kaps, and B. Sunar, "Public key cryptography in sensor networks—revisited," in *1st European Workshop on Security in Ad-Hoc and Sensor Networks (ESAS 2004)*, ser. Lecture Notes in Computer Science (LNCS), H. Hartenstein, C. Castellucia, C. Paar, and D. Westhoff, Eds., vol. 3313. Heidelberg: Springer, August 2004, pp. 2–18.

[3] (1998, October) Introduction to public-key cryptography. Sun Microsystems. [Online]. Available: http://docs.sun.com/source/816-6154-10/contents.htm

[4] G. Gaubatz, J.-P. Kaps, E. Öztürk, and B. Sunar, "State of the art in ultra-low power public key cryptography for wireless sensor networks," in *Third IEEE International Conference on Pervasive Computing and Communications Workshops, Workshop on Pervasive Computing and Communications Security–PerSec'05.* IEEE Computer Society, Mar 2005, pp. 146–150.

[5] (2008, April) Tinyos documentation wiki (english). [Online]. Available: http://docs.tinyos.net/index.php/Getting_Started_with_TinyOS#Compiling_and_Installing

[6] A. Greensted. (2008, June) Tps75003 spartan-3e voltage regulator. [Online]. Available: http://www.bioinspired.com/users/ajg112/circuits/tps75003.shtml

[7] (2007, December) Spartan-3an fpga family data sheet. Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds557.pdf

[8] (2007, August) Atmega128l 8-bit microcontroller with 128k bytes in-system programmable flash. Atmel Corporation. [Online]. Available: http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf