# 6.111 Group 1 Final Project
# A Wireless Headphone and Speaker Set

Nivedita Chandrasekaran, Jessica Nesvold, and Aditi Shrikumar

May 17, 2007

## Abstract

The goal of this project was the design and Field Programmable Logic Array implementation of a wireless headphone set capable of transmitting an audio signal wirelessly across a distance of 20 feet. Ideally, the system would convert an analog signal from a 3.5mm headphone jack to a digital signal, introduce compression and error correction codes, transmit and recieve it wirelessly, decompress and error check it, and then convert it back to an analog signal which could be output to headphones. Currently, the wireless transmission and reception are not integrated into the system, which means that the analog to digital and compression/error checking modules communicate with the corresponding decompression/error checking and digital to analog modules using wires. The uses one channel of audio, sampled at 90-kHz with a resolution of 16 bits. The compression algorithm reduces the size of the data by 50% by encoding differences between five consecutive samples. The wireless transcievers are interfaced with using Verilog modules and operate at 2.4-GHz.

## Contents

**1 Introduction** **3**

**2 The Digital-Analog Interface** **3**
2.1 Analog-to-Digital conversion . . . . . . 4
2.2 The Digital-to-Analog Modules . . . . 6

**3 Compression and Decompression** **6**
3.1 Compression . . . . . . . . . . . . . . 6
3.2 Decompression . . . . . . . . . . . . . 8
3.3 Error Safety Encoding . . . . . . . . . 9
3.4 Error Detection and Correction . . . . 10

**4 The Buffers** **11**
4.1 The Transmit Buffer . . . . . . . . . . 11
4.2 The Receive Buffer . . . . . . . . . . . 12

**5 Wireless Communication** **13**
5.1 Hardware and SPI Interface . . . . . . 13
5.1.1 Hardware . . . . . . . . . . . . 13
5.1.2 SPI Interface . . . . . . . . . . 13
5.1.3 Configuration . . . . . . . . . . 14
5.1.4 Transmission . . . . . . . . . . 16
5.1.5 Reception . . . . . . . . . . . . 17
5.1.6 Integration With The FPGA - Creation of a 27/4-MHz clock . 18

**6 Conclusion** **18**

## List of Figures

1 Block diagram of the wireless heaphone and speaker set.) . . . . . . . . 3
2 The timing diagram for the serial interface to the AD7656.(Diagram taken from existing documentation which is attached.) . . . . . . . . . . . . . . . . 4
3 Block diagram of the analog to digital conversion interface. . . . . . . . . . . 4
4 The sampling module's modes of operation. . . . . . . . . . . . . . . . . . 5

5 The module that takes in successive samples and outputs chunks of five samples each. . . . . . . . . . . . . . . 5
6 Timing diagram for the interface to the AD5063. . . . . . . . . . . . . . . . . 6
7 The digital-to-analog interface module. 6
8 The serializer. . . . . . . . . . . . . . . 7
9 The structure of the compressed data 7
11 Matlab simulation of codec, the large graph contains the info from both of the smaller ones. top left = original audio, bottom left = output of decompression . . . . . . . . . . . . . . . . . 9
12 The structure of added error correction 9
13 ModelSim of compression/ error_correction_in/error_correction_out/decompression 11
14 ModelSim ouput for test of buffer_aditi 12
15 ModelSim ouput for test of buffer_nivedita . . . . . . . . . . . . . 12
16 The inputs and outputs of the CC2420 13
17 Timing diagram for the CC2420 . . . 14
19 The configuration and command registers accessed by the configuration FSM. 15
20 The transmission finite state machine. 16
22 The Receive FSM. . . . . . . . . . . 18
10 ModelSim of compression/decompression modules with working error correction recursion. . . 20
18 The configuration finite state machine. 20
21 ModelSim of the Transmit FSM in action. . . . . . . . . . . . . . . . . . . . 21
23 The Receive FSM. . . . . . . . . . . 21

## 1 Introduction

The wireless headphone and speaker set is a system designed to break confines. In a world where people like to be unrestricted and unhindered, our project seeks to allow people to both efficiently and accurately transmit and receive their music without the use of wires. As is shown in Figure 1, the wireless headphone and speaker set consists of two separate sections, a transmitting end and a receiving end. Connected by nothing but air, this frees both the user doing the transmitting and the user doing the receiving to move about as the wish (assuming, of course, that they are willing to take their labkit with them).

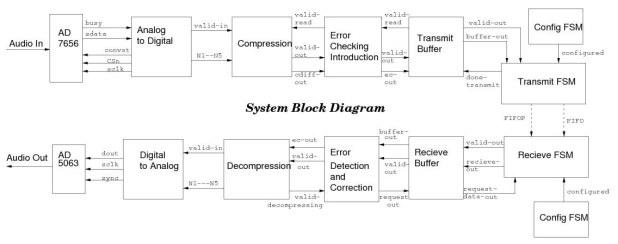


Figure 1: Block diagram of the wireless heaphone and speaker set.)

Our system contains four main components, the first of which is the A/D and D/A converter pair which allow us to take analog output from the user and convert it into digital signals that can be processed and subsequently transmitted over the air. The second component consists of the compression and decompression modules which utilize an algorithm designed to compress inputted data by fifty percent. These modules allow more data to be transmitted per wireless packet than would normally occur, helping to increase the overall efficiency of the system.

The third component consists of an error correction algorithm. Before the data is transmitted it is passed through an error correction module which adds redundant data that is used by the error correction module on the receiving end to fix as many errors as possible that are introduced by the wireless transmission itself.

The fourth part is made up of the wireless transmitter and receiver itself. This component uses the CC2420 chip to establish a wireless link, and we communicate with the chip solely through the use of Verilog. This is a particularly unique and interesting approach because it completely bypasses the onboard microcontroller normally used to interface to the chip, allowing us to achieve a potentially higher data rate.

As is also shown in Figure 1 the receiving end is essentially the inverse of the transmitting end. This is because ideally we should be getting out the exact same data that we are putting into the system. Potential errors that can be introduced along the way, however, include error from the compression and decompression modules (which implement a lossy algorithm) and error from the wireless transmission itself (in which bit flips or loss of data may occur). In order to minimize this error, the codec used was tested extensively in Matlab before being integrated into the system. Additionally, the CC2420 itself implements a CRC checking system. Thirdly, we have implemented our own error correction modules, as discussed above. A lot of attention to detail was paid with regards to the minimization of errors to ensure that our project would be both easy to use and practical.

In this paper, each component and the corresponding modules and logic implemented is discussed in detail. The results of various ModelSim tests are provided to show functionality of the code used. Finally, a discussion of the overall outcome of our project is presented.

## 2 The Digital-Analog Interface

The Digital-Analog interface is how the system communicates with the outside world. On the transmitting end, it consists of an Analog-to-Digital module which samples the audio into 16-bit words and sends chunks of five words each to the compression module. On the recieving end, it consists of a Digital-to-Analog module which recives decompressed chunks from the decompression module, serializes the separate words, and sends them to a digital to analog converter.

## 2.1 Analog-to-Digital conversion

To convert the incoming analog signal to a digital signal this project uses an AD7656 chip by Analog Devices. The chip is a 16-bit successive-approximation analog to digital converter with a maximum conversion rate of 250,000 samples per second.

The interface has two parts: configuring and controlling the analog to digital converter, and converting the serial output data into chunks that are usable by the compression module. Figure 2 shows the timing diagram for the serial interface to the AD7656. The first part, configuring and communicating with the chip, involves generating the correct signals at the specified times and clocking the serial data in after the chip has finished converting the data.

[htb]

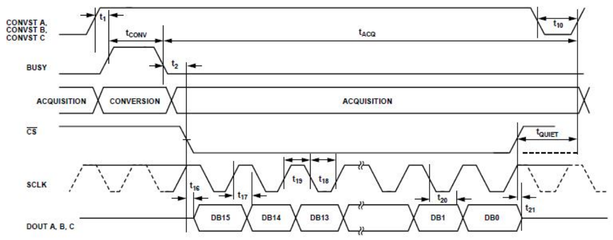


Figure 2: The timing diagram for the serial interface to the AD7656.(Diagram taken from existing documentation which is attached.)

The chip operates as follows: on the positive edge of the `convst` signal, it begins converting the acquired analog signal into a digital signal. While it is converting, it outputs a high `busy` signal. Approximately 3$\mu$s later, conversion finishes and busy goes low indicating that the data is ready to be read out. When the user wishes to read it, he or she must set the $\bar{C}S$ chip select (active-low) signal to zero, and the chip immediately begins clocking out the sixteen bits of data (MSB first) on the `sdata` line, with each bit ready on the positive edge of `sclk`.

The number of positive edges of `convst` per second determines the sampling rate. In order to avoid aliasing in the audible range, this rate must be at least 44.4kHz (because humans can hear upto 22.2kHz). Beyond that, increases in sampling rate have no audible effect. Thus, if this project transmitted the sampled audio raw, with no compression or error checking, a sampling rate of 44.4kHz would suffice. Nevertheless, limitations on the speed of wireless transmission meant that some compression would be necessary. Jessica Nesvold, one of the team members, came up with a scheme to compress the data by 50% (described in section 3) by encoding the differences between five successive 16-bit samples. The closer the samples are to each-other, the less lossy the compression. This means that an increase in sampling rate beyond 44.4kHz could have audible effects because it gives samples that are closer together in time, and potentially closer together in value. After some experimentation, the team members found that the improvement from a 45-kHz sampling rate to 90-kHz sampling rate was audible. The resultant increase in data rate could still be accomodated, so a sampling rate of 90-kHz was decided upon.

Figure 3 is a block diagram of the interface designed to signal conversions and receive data. The

[htb]

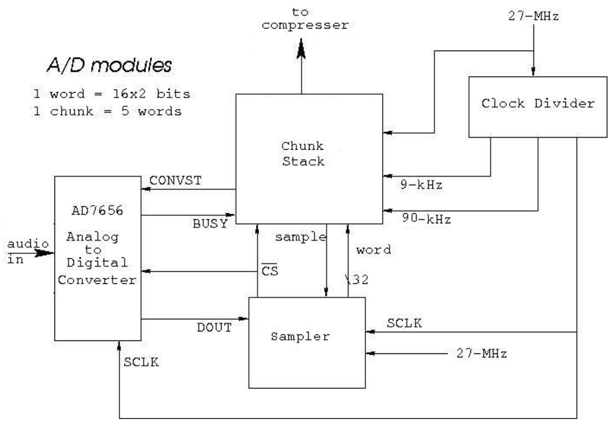


Figure 3: Block diagram of the analog to digital conversion interface.

90-kHz sampling signal `convst` is generated using a divider module (code in the appendix) that divides the 27-MHz FPGA clock by 300. The serial clock `sclk` was chosen to be 5.4-Mhz, 60 times faster than the sampling clock, to allow enough time for conversions to finish before reading out the serial data. In this setup, the third signal $\bar{CS}$, which controls when the chip starts clocking out data, is not triggered by the `busy` signal going low. Instead, it stays high for 20 `sclk` cycles (3.7$\mu$s) after the `convst` positive edge before going low. The specifications for the AD7656 guarantee that conversion will have finished by this time, so data can safely be clocked out. The main reason for not triggering this signal off the falling edge of `busy` is to avoid timing errors: there is no relationship between the falling edge of `busy` and the next `sclk` positive edge, so there is a risk of missing the MSB if `busy` falls at the wrong time and there is not enough time between the falling edge of `convst` and the next rising edge of `sclk`.

The sampling module begins the sampling cycle with the AD7656 when the `sample` signal from the chunk module is high. It outputs a `word` of 32 bits to the chunk module along with a `word-ready` signal to indicate that the word is valid. Figure `samplefsm` shows the modes of operation of the sampling module.

[htb]

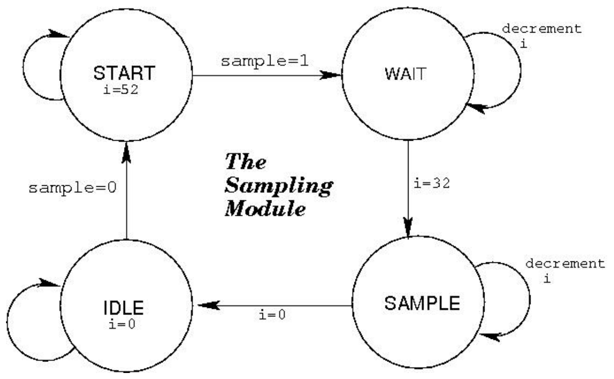


Figure 4: The sampling module's modes of operation.

The system was originally designed for stereo audio, in which case the AD7656 would output two samples totaling 32 bits (16 each for the left and right channels) every sampling cycle. This is why the counter 'i' clocks in data for 32 clock cycles, and why `word` is marked as 32 bits wide. Nevertheless, only one digital-to-analog chip arrived in time, so in the actual system, the first 16 bits of `word` are always zero because only one of the audio channels is sampled.

The second task of the analog-to-digital interface is to convert the recieved samples into chunks of five that can be sent to the compression module. This is accomplished using the module shown in Figure 5. This module contins five FIFO stacks onto

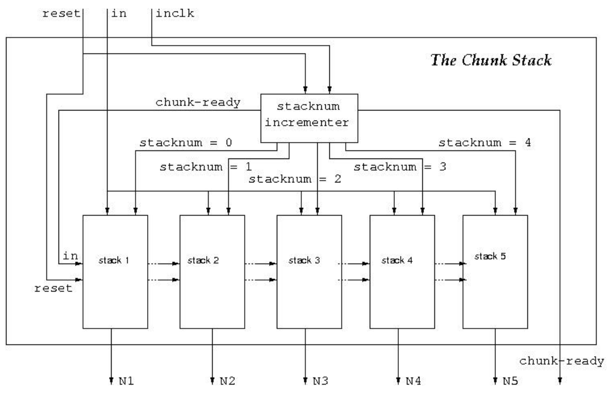


Figure 5: The module that takes in successive samples and outputs chunks of five samples each.

which incoming samples from the sampling module are pushed. Internal logic keeps track of the sample count, and every time five samples have been recieved, the module pushes one sample off each of the five stacks, supplying the compression module with the required five samples. It also outputs a `chunk-ready` signal that tells the compression module when the five samples become valid.

The chunk module is also in charge of supplying some control signals, both to the analog to digital converter and the sampling module. It houses a divider that supplies the `convst` and `sclk` signals, and provides the `sample` signal, which controls whether or not sampling takes place, to the sampling module.

## 2.2 The Digital-to-Analog Modules

The digital to analog interface can be thought of as the reverse of the analog to digital interface: it recieves chunks of five samples each from the decompression module, and sends them to a digital to analog converter in serial form, while supplying all of the appropriate control signals.

The chip used for this part of the project is the AD5063, a 16-bit voltage-out digital to analog converter by Analog Devices. The interface to it is extremely straightforward, everything is accomplished using two control signals: a `sync` signal to start a conversion, and a `sclk` serial clock using which the chip clocks in the user-supplied `d-out` data MSB first. Figure 6 shows the timing diagram for these three signals.

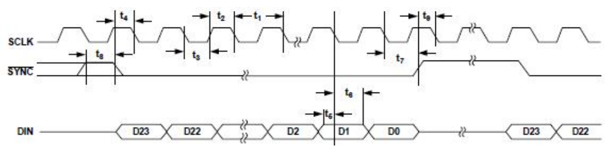


Figure 6: Timing diagram for the interface to the AD5063.

An overall block diagram is shown in figure 7. This module contains five numbered FIFO stacks. When decompression is finished, the decompresse sends in five decoded samples to the digital-to-analog interface module along with a `data-valid` signal that indicates whether the wires hold valid data. When the module receives the numbers, it pushes each of the samples onto a separate stack: N1 is pushed onto `stack1`, N2 onto `stack2` and so on. Meanwhile, combinatorial logic keeps track of which stack to push data out of, making sure that that the samples retain their time-ordering. The module cycles around the stacks from `stack1` through `stack5` in a round-robin fashion.

Once a sample is pushed out of a stack, it is sent to a serializer module. Figure 8 shows a schematic diagram of the serializer's modes of operation. The operation of the serializer is triggered off the same control signals as the digital to analog converter. When

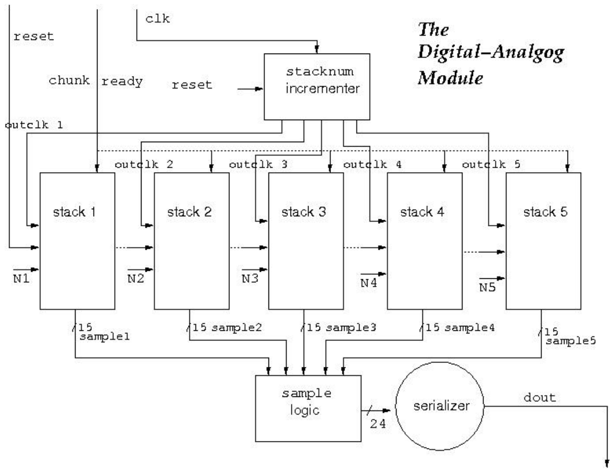


Figure 7: The digital-to-analog interface module.

the `sync` signal goes high, the sample to be output is stored in `sample` and serialization begins. The index of the data bit being output decrements every `sclk` cycle till it hits zero. The module is then idle until the next `sync` pulse, when the cycle repeats. The digital to analog converter requires eight configuration bits before each sample, so the counter decrements from 24 (instead of just 16 for the sample). In the mode of operation needed for this project, the configuration bits are all zeroes, so they are just appended to the actual sample before being sent to the serializer.

## 3 Compression and Decompression

### 3.1 Compression

The compression module works to take in five signed sixteen bit numbers. It then compresses these five numbers into one single forty-bit number. This constitutes fifty percent compression. Compression is important to our system, as the compression of data allows more data to be contained per packet sent by the wireless link. The more data a single packet contains, the faster the song is transmitted and the faster the user is able to enjoy music on the receiving end

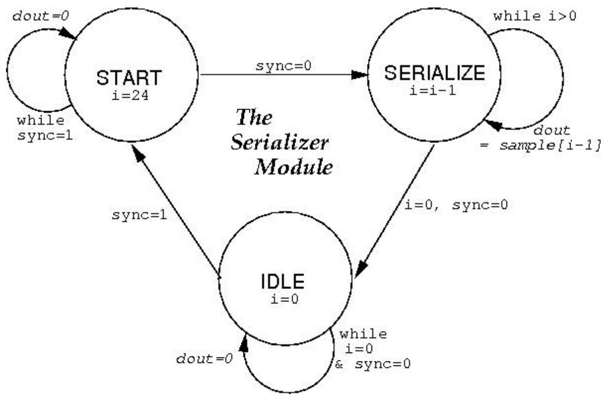
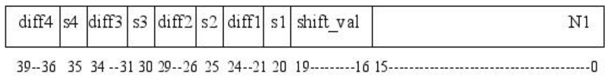


Figure 8: The serializer.

of our system.

The compression implemented in our system used a homemade compression algorithm. The algorithm is lossy as it makes use of only fixed length difference values. The structure of the forty bit number created is shown in Figure 9



| diff4 | s4 | diff3 | s3 | diff2 | s2 | diff1 | s1 | shift_val | N1 |
|---|---|---|---|---|---|---|---|---|---|
| 39--36 | 35 | 34 --31 | 30 | 29--26 | 25 | 24--21 | 20 | 19---------16 | 15-------------------------------------0 |

Figure 9: The structure of the compressed data

In Figure 9, N1 represents the first sixteen bit number inputted to the system. First refers to the first number of the five given to the compression module that was originally outputted by the D/A converter. As is shown, the number N1 is copied exactly to the bottom 16 bits of the outputted data. This will serve as a reference number for the decompression module.

`shift_val` is short for shift value. It is the amount by which all of the following difference magnitudes ($\text{diff}_1$, $\text{diff}_2$, $\text{diff}_3$ and $\text{diff}_4$) are to be shifted by. Shifts are measured as the location of the first digit in the actual difference value. For example, if $\text{diff}_1$ is equal to 1100 and `shift_val` is equal to 1001, or 5, we know that we would like the first bit of the number 1100 to begin at the binary location representing $2^5$. In other words, $\text{diff}_1$ is encoded as the number 110000. The same is true for $\text{diff}_2$, $\text{diff}_3$ and $\text{diff}_4$. Also, all 4 difference values share the same `shift_val`.

The one-bit numbers $s_1$ through $s_4$ represent the sign of their respective difference magnitudes. A value of zero represents a positive difference while a value of 1 represents a negative difference.

This string of bits is decoded quite simply. The first number, N1, is given. The second number is found by first shifting the value of $\text{diff}_1$ and either adding or subtracting (as given by $s_1$) this from N1. The third number is found by shifting $\text{diff}_2$ and either adding or subtracting this from the previously found second number. The third, fourth and fifth numbers are found in a similar manner.

A finite state machine was used to implement this compression algorithm, the state transition diagram for which can be found in the appendix. This FSM is very linear, following a step-by-step method to compress the data.

To ensure proper functionality, the compression fsm must first be reset. This places the FSM in `state_wait`. In this state the fsm is literally 'waiting for the inputted data to become valid. This occurs when the `valid_in` signal goes high. When `valid_in` becomes high the five inputted numbers are registered as N1 through N5. By registering these numbers only when `valid_in` is high, it allows the inputs to change once `valid_in` goes low again without adverse affects to the system.

After registering the numbers N1 through N5, the FSM enters `state_diff`. This is the state in which the actual difference magnitudes are calculated. 'Actual means that they are exact, found simply as the difference between two adjacent numbers. For example, $\text{diff}_1$ is set to be abs(N2 N1), $\text{diff}_2$ is set to be abs(N3-N2) and so on for all four difference magnitudes. The values of $s_1$ through $s_4$ are set in this state as well. This is done by looking at the two numbers being subtracted. If the larger is being subtracted from the smaller, we know that the output will be negative and the corresponding s value is set to one, otherwise it is set to zero.

After the initial difference values have been found (note that these can be as large as sixteen bits as they have not yet been limited to four bits in length), `shift_val` must be calculated. This is calculated as being the location of the first one in the largest difference. This means that when decompression occurs, at the very least the largest difference will be recovered to its order of magnitude. This helps the algorithm follow the sound as closely as possible, thereby producing sound that is as near to the original as possible. `shift_val` is therefore calculated in the progression of states `state_max_diff` to `state_max1`. Note that for the purposes of this FSM `shift_val` actually corresponds to the value max1 (the location of the first one in the maximum difference magnitude).

Once `shift_val` has been calculated, the FSM determines the sixteen possible difference values that can actually be encoded by the compression system itself. There are only sixteen values that can be encoded because the compression format only allows for four-bit long values of $\text{diff}_1$ through $\text{diff}_2$. Hence, if `shift_val` is five, the sixteen possible values that $\text{diff}_1$ can take on are 000000, 000100, 0001000, 0001100, etc. To take a practical example, the number four would actually be encoded as $\text{diff}_1 = 0001$. This is because shifting up the most significant bit of 0001 to the `shift_val` position of 5 gives 000100, or four. The finding of possible encoded values is done in state_vals.

Now all that is left for the FSM to do is to determine which of the possible values best corresponds to the actual difference value and store the four bit representation of these back as $\text{diff}_1$ through $\text{diff}_4$.

Because this represents a lossy type of compression, there is the possibility for compounding errors on the decompression side. This is because N3 is found using N2, hence if there was any error in the difference magnitude used to fine N2, this will also be found in N3. To prevent this from happening, every time a four-bit difference value is calculated, the compression FSM cycles back and recalculates the new difference magnitudes.

For example, consider that $N2N1 = 10$ and that $N3N2 = 14$. If $N2N1$ is estimated as 8, when N2 is decompressed it will be estimated as two less than the actual value. This means that to recover N3 on the decompression side, a difference of 16 is actually needed to get back N3 exactly. Thus, once $N2 - N1$ has been calculated the compression FSM enters a state called state_recal_diff2 where the actual value of $\text{diff}_2$ is calculated using the value of N2 that will be found by the decompressor.

An example of this working is shown below in Figure 10 which shows compression and decompression of various signals.

The interesting case occurs where the number $-314$ in test case 1 is recovered exactly by the decompression module despite the fact that there was error introduced in the second number decompressed. Had the additional recursive difference calculations not been introduced, the error would have compounded and $-314$ would not have been recovered.

## 3.2 Decompression

The decompression module takes in a forty-bit number that has been created by the compression module and outputs five signed sixteen bit numbers. To achieve proper functionality, the decompression module must first be reset before it is used. This is because the module makes use of an FSM to control the progression of the decompressing which must be initialized into the right state. Reseting the module causes the FSM to be placed in `state_wait`, a state in which it can wait for valid data to be provided to it.

When the data inputted to the system becomes valid, the data is sectionalized appropriately and registered. The sections the data is divided into can be classified as difference values, sign values, `shift_val` and N1. N1 is the reference value, representing a complete uncompressed version of the first sixteen bit number. `shift_val` is the value of the bit placement of the most significant bit of the difference values (discussed in the section on compression). The sign values are stored as N2_sign through N5_sign and represent the sign of their corresponding difference values. To note, the difference values here are assigned as N2_diff through N5_diff whereas the compression module referred to them as $\text{diff}_1$ through $\text{diff}_4$. The numerical inconsistency is unfortunate, but they correspond as follows: N2_diff = $\text{diff}_1$, N3_diff = $\text{diff}_2$, etc.

The process of decompression is fairly simple. First the stored difference values are shifted to represent the actual encoded difference values. This happens in `state_shiftl`. After the difference values have been shifted, N1 is first calculated by adding or subtracting diff_2 from N1, as determined by N2_sign. Namely, an N2_sign value of 0 represents an addition and an N2_sign value of 1 represents a subtraction.

N3 through N5 are then found by adding or subtracting the shifted versions of N3_diff through N5_diff from N2 through N4. Testing of the decompression module was done in ModelSim, the result of which is shown in Figure 10. Such a test, however, is not indicative of what kind of sound the decompression module will output. For this reason, more extensive testing was done using Matlab, the results of which are shown in figure 11

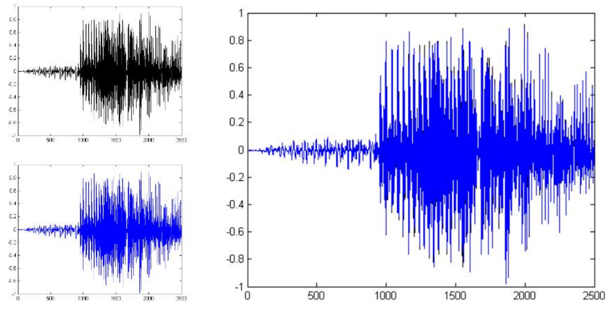


Figure 11: Matlab simulation of codec, the large graph contains the info from both of the smaller ones. top left = original audio, bottom left = output of decompression

As can be seen in Figure 11, decompressed audio is very similar to that of compressed audio, with most of the differences occurring on large transitions of data. In testing the Verilog implementation of the codec, errors introduced by compression and decompression in low frequencies were negligible while really high frequencies were repressed. A demo of this can be seen in the video tape posted on the project website. All in all, the codec used was really successful and produced good results when applied to audio signals.

## 3.3 Error Safety Encoding

The error_correction_input module takes in a forty bit packet from the compression module and adds ten bits of error correction to it. The parts of the packet that are error corrected for directly correspond to the importance of the bits. In our case, we are mostly concerned with the bottom twenty bits of any given packet. This is because the first sixteen bits correspond to the N1 number which is used as the basis for reference for the rest of the difference values. The next four bits correspond to the `shift_val` which is also incredibly important as it dictates the amount by which every difference value is to be shifted before it is applied in the decompression algorithm.

The error correction scheme used here locates the position of ones in the data. For example if the sixteen bit number, N1 is equal to 0001000100000001 we know that the first one, the most significant one, is located at bit number twelve. Similarly the second one is located at bit number eight. Because N1 is sixteen bits long, it takes four bits to encode the position of the first one. If N1 is equal to zero, the position of the first one is simply set to zero as well. To note, `shift_val` is only a four bit number so the position of the first one in `shift_val` need only be encoded with two bits.

The ten bits added to the forty bit data packet are as shown in figure 12

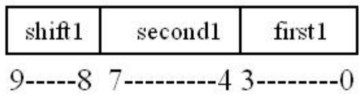


Figure 12: The structure of added error correction

The four bit value first1 corresponds to the location of the first one in the value N1. The four bit value second1 corresponds to the location of the second one in the value N1. Also, the two bit value shift1 corresponds to the location of the first one in the value `shift_val`. The error_correction_output module uses these values to zero everything that is higher than the location of the first one, as well as everything located between the first and second ones in the case of N1. It will also make sure that there are ones located at the specified values. As such, this guarantees that the sign and most significant bits of N1 are correct and that the most significant bits of the `shift_val` are correct.

This error correction is implemented via a finite state machine that first calculates the value of the location of the first one in both `shift_val` and N1 once valid_in has gone high. After the position of the first ones have been located, the position of the second one in N1 can be found and the error correction can be added. The data is thus ready for transmission and can be passed to the FIFO buffer on the transmit side.

Unfortunately the effectiveness of this system could not be tested with the wireless module, but testing was carried out in ModelSim. This testing will be discussed in the section on Error Correction Output as the modules were tested together to be sure that they had no adverse effects on any of the data.

## 3.4 Error Detection and Correction

The error_correction_output module takes in a fifty bit packet formatted for error correction and outputs a forty bit error corrected packet. The error correction module implements the same algorithm regardless of whether or not an error has occurred. It can do this because it knows what the first few bits of both `shift_val` and N1 are meant to be and then just sets them as such.

To implement this functionality, an FSM is used. When reset, the FSM is placed in the wait state where it waits for the signal valid_in to become high. When this happens it registers the data and parses it into the values, first1, second1, shift1, N1, `shift_val` and rest_data where rest_data is simply the rest of the inputted data which is not affected by the error correction algorithm.

The error correction process involves left shifting the data to the position of the first one, setting the MSB of the shifted number to one and then right shifting the data back by the same amount. In this way, the first one is set to one regardless and the higher order bits are all zeroed appropriately.

In the case of the number N1 which has associated with it both a first one and a second one, the number N1 is first treated for the value of second1 and then for the value of first1. When second1 is accounted for, the first one will be zeroed because everything above the value of the second1 will gets zeroed. This is alright, however, because N1 is adjusted for first1 in the following states.

When error compression has been completed, the valid_out signal is set to one. The FSM will not transition back to `state_wait` until it has received confirmation that its outputted data has been read by the following module. This is indicated when the valid_read signal goes high. (valid_read here corresponds to the valid_decompressing signal outputted by the decompression module when it begins decompression).

Testing of the error_correction_ouput module was done together with both the error_correction_input module and the compression and decompression modules. Namely, the modules were hooked up as: compression → error_correction_input → error_correction_output → decompression.

The same tests that were used to test the compression and decompression modules (shown in the section on compression) were run again with the error correction modules in the middle. Because error correction is implemented regardless of whether or not there was any error, this was a good way to test the functionality of the modules. As is shown in figure 13, the same results were obtained with error correction inserted as were obtained with no error correction, thereby showing that the error correction modules work at least in principle. Unfortunately full practical testing was not able to be carried out do to the lack of the wireless portion of our project in its final form.

Another interesting thing that is portrayed in Figure error_correction_out.1 is the formation of the busy signals in the bottom four rows of the output. Here it is shown how the system as a whole is able to pipeline the data through. For example, once the busy signal for valid_compressing goes low, it is able to take in more data on the next valid_in signal even though the data_out has not yet been calculated by the rest of the system. This further shows the correct functionality of interaction between different modules in the overall system.

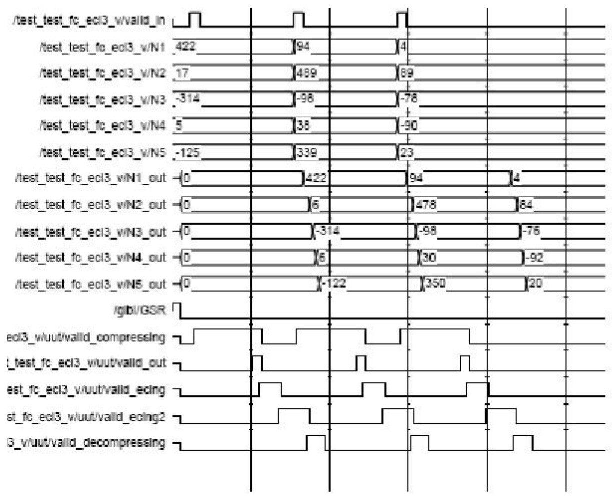


Figure 13: ModelSim of compression/ error_correction_in/error_correction_out/decompression

## 4 The Buffers

### 4.1 The Transmit Buffer

The buffer_aditi module refers to the FIFO buffer located on the transmit side. It is responsible for taking in the fifty bit numbers outputted by the error_correction_in module and creating eight-hundred bit packets compatible for transmission by the wireless module. As such there are two important signals concerning the input and output of this system. The first signal is valid_in, which signals that data is ready to be taken into the system. The second signal is done_transmit which signals that the transmit module is ready to send a new packet.

To create the eight-hundred bit packets, the buffer_aditi module makes use of an internal FSM. On reset, this FSM is placed in state_wait where it waits for inputted data to become valid. Once this occurs it registers the data as the bottom fifty bits of data_in. Data_in represents a sequence of eight-hundred registers where data packets can be built up.

After the data has been registered, the FSM transitions to a sequence of ackin states which acknowledge that the module has taken in the data. These states are needed to interface properly with the error_correction_in module which waits for confirmation that its data has been read before moving on to error correct more data. The ackin states transition to state_store. This is a registered value set by the FSM before the ackin states begin. In the case of the transition from state_wait to state_ackin, state_store is set as state_100. This state is similar to the state_wait, except for the fact that when valid_in becomes high the inputted data is stored as bit numbers 50 to 99 of data_in, creating a 100 bit long packet. The FSM then acknowledges the intake of data and subsequently transitions to state_150. A similar transition of states continues until an eight hundred bit packet is built up. Once this occurs the data is stored in the FIFO as dictated by the in_pointer.

The FIFO implemented by the buffer_aditi module can store up to ten eight hundred bit packets at a time. It makes use of two pointers, which are stored as registered values internally. These are the in_pointer and the out_pointer. When an eight-hundred bit packet is written to the FIFO, it is written to the slot in_pointer points to. Afterwards, in_pointer is incremented by one. When an eight-hundred bit packet is read from the FIFO, it is read from the slot out_pointer points to. After a read has taken place, out_pointer is also incremented by 1. If out_pointer or in_pointer ever reach a value of ten, they will be set back to 1 instead of incremented to 11 producing a cyclic effect whereby data can be continuously read and written to the ten slot FIFO. There are two ways by which the module knows that a read has been requested. The first is a detection of a positive edge of the done_transmit signal. The second is if the valid_out signal is low. The done_transmit signal represents a direct request from the transmit module for more data. If there is data in the FIFO when this occurs, the data will be outputted and valid_out will be set high. If there is no data in the FIFO, valid_out is set low. This lets the transmit module know that there is no data to transmit. This also means that the buffer_aditi module will continue to try to output new data every clock cycle. Thus, as soon as new data is inputted into the FIFO, it can be outputted to the transmitter.

Testing of this module was done both in ModelSim and via compilation onto the labkit. The results of ModelSim testing can be seen in Figure 14. In this figure, note that the data inputted to the system is referred to as data_right.

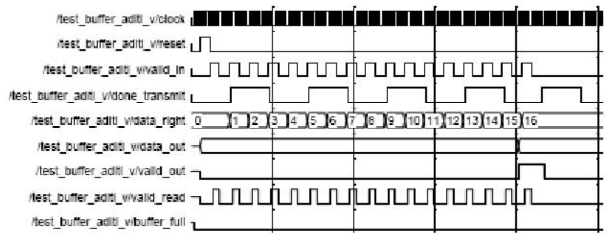


Figure 14: ModelSim ouput for test of buffer_aditi

As is shown in Figure 14, the outputted data does not become valid until sixteen numbers have been received (the number necessary to create one eight-hundred bit packet). Also, after valid_out goes high it becomes low again once done_transmit becomes high. This is because after the first eight hundred bit packet was outputted there was nothing left in the FIFO. The behavior shown here is all in accordance with the expected behavior of the system.

## 4.2 The Receive Buffer

The buffer_nivedita module refers to the FIFO buffer located on the receive side of the system. In addition to implementing a FIFO it also takes care of parsing the eight-hundred bit packets produced by the wireless system back into the fifty bit packets compatible with the error_correction_out module. The implementation of the FIFO is analogous to that found in the module, buffer_aditi. The only difference is that there are 160 slots as opposed to only 10. This corresponds to the same amount of storage space, however, as the slots here are only fifty bits wide as opposed to eight-hundred. To parse the received data packets, an internal FSM is used. This FSM makes use of only three states; a wait state, an acknowledge state and a store state. On reset, the FSM is placed in the wait state where it waits for the valid_in signal to become high. Once this happens the inputted data can be registered and the FSM can transition to state_ackin. This acknowledge state is for ensuring that the same data does not get read twice and is essential for interfacing to the transmitter. The transmitter runs on a much slower clock than does the buffer module. Therefore, even if the valid_in signal is only high for one transmitter clock cycle, this may constitute may buffer clock cycles. The addition of the ackin state prevents the buffer FSM from cycling back into the wait state and seeing a valid_in signal that is still high from the previously registered and processed data. After the valid_in signal has gone low, the FSM can transition to the store state. This is where the data is parsed and stored in the FIFO. The lower fifty bits represent the first data value that is to be stored, the next fifty bits represent the second and so on. There are sixteen fifty bit data values stored in every eight hundred bit packet, thus every packet received represents an increase of sixteen values in the FIFO. Another purpose served by this buffer is to prevent periods of silence in the sound outputted at the receiving end. In other words, the data is buffered up before it is sent on to compression and decompression so that even if there is an occasional delay in the received wireless information the user will not be affected. It was decided that the packet length themselves would be enough to buffer the incoming data. In other words, every packet brings with it a buffer of fifteen. This is because once the first fifty bit data value is outputted there are still fifteen more behind it in the FIFO. It was thought that this would be enough to buffer the system on. Unfortunately, full testing of the functionality of this buffering mechanism was never carried out as the wireless was never integrated in the system. Testing of the buffer was therefore done in Modelsim, the results of which are shown in Figure 15. Again,

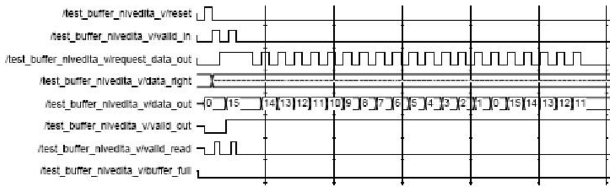


Figure 15: ModelSim ouput for test of buffer_nivedita

the inputted data is found on the port, data_right. Although it is unclear on this picture, the data inputted is an eight-hundred bit packet representing the sequence of numbers 15 to 0. Two of these packets are inputted into the system and are then clocked out according to the request_data_out signal, which has the same functionality as the done_transmit signal discussed in the buffer_aditi section. As is shown, valid_out only becomes high after the data_out signal has changed to the first number to be outputted, 15. Additionally, new data is outputted only on the rising edge of request_data_out. This combines to confirm correct functionality of the system.

# 5 Wireless Communication

## 5.1 Hardware and SPI Interface

### 5.1.1 Hardware

The wireless portion of the system was implemented using a Serial Peripheral Interface (SPI) link that communicated with the CC2420 2.4Ghz wireless chip from Chipcon, which was mounted on the CC2420 Demonstration Board Kit (DBK). The design and implementation process for the Verilog code for the SPI interface was implemented by using the "CC2420 Examples Release" and "CC2420DBK Libraries" C code from the Chipcon as a starting point and guideline for the functionality and operations the SPI interface would have to implement. Two CC2420DBKs were used to form the wireless link. The SPI interface was implemented in a series of Verilog modules that implemented complex operations like the configuration of the CC2420, the transmission of data, and the reception of data in relatively simple finite state machines. This CC2420 was chosen because it could support the high data rates needed to stream audio wirelessly since, according to the specification sheets, the CC2420 is capable of supporting data rates of up to 250kbps.

The SPI interface is a novel feature because it talks directly to the CC2420. The overhead involved in passing commands to the microprocessor on the C2420DBK to command the CC2420 to transmit and receive data is completely eliminated. In this section of the report, the timing constraints and functions of the inputs and outputs between the Verilog-implemented SPI and CC2420 will be discussed. Then, the operation and function of the three main modules, the ConfigureFSM, TransmitFSM, and ReceiveFSM will be discussed. Then, the design process and method of integration with the FPGA will be detailed. Finally, the degree of the success of the implementation of the wireless system and its integration with the system will be evaluated.

### 5.1.2 SPI Interface

Figure 16 shows the four one-bit input and output lines the Verilog interface uses to communicate with the CC2420.

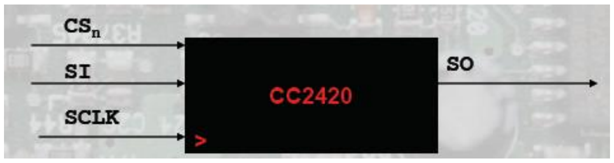


Figure 16: The inputs and outputs of the CC2420

The three input lines to the CC2420 are the SI, CSn, and SCLK lines. The single clocked output line from the CC2420 is the SO line. Other outputs from the pin itself include the FIFO, FIFOP, and SFD pins. These go low and high as packets of data are received and transmitted. More details of their operation are included in the ReceiveFSM and TransmitFSM sections.

The Verilog interface assumes master role normally taken on by the microprocessor in the CC2420DBK board and dictates all actions of the CC2420. All communication with the chip is performed most significant bit (MSB) first. The chip is enabled when the CSn signal is low. The chip can take in data from the SI line and CSn lineon each rising edge of the SCLK signal. In order for the chip to properly read bits in from the SI line, the SI and CSn inputs to the chip must be set up at least 25ns before the rising edge of the SCLK signal. This places a maximum constraint of 10Mhz on the maximum frequency at which the SPI interface can be clocked.

All operations performed by the CC2420 are dictated by 33 16-bit configuration registers, 15 command strobe registers, 2 8-bit registers to access the separate receive and transmit FIFOs, and the data that can be written the local RAM on the chip which holds configuration information about the chip as well as data to be transmitted or data that has been received. Register and RAM access can only be performed when CSn is low.

The command strobe registers are relatively easy to message. They contain no data and only serve to set certain signals in the chip high and low. As a result, a typical command strobe access involves pulling CSn low, making sure that SCLK is running, and then shifting in 8 bits of data on 8 rising edges of SCLK, 6bits of which are the command strobe register address, MSB first to the CC2420. The first bit of the register access command tells the chip whether a RAM or register access command will follow while the next bit selects a read or write operation. The next 6 bits select one of the 50 configuration and command registers. A command strobe register address will never cause the chip to expect a 16-bit data sequence, while a 6-bit configuration address will cause the CC2420 to treat the next 16 bits that come down the SI line as data to be written into the selected configuration register.

```
[RAM/REGISTER][dont care][6-bit address]
Command strobe register access
[RAM/REGISTER][READ/WRITE][6-bit address][16-bit data]
Configuration register access
```

RAM access, however, is slightly more complicated. Figure 17 is a timing diagram taken from the timing diagrams in the CC2420 documentation specifying RAM access. Table 6 in the CC2420 chip documentation[1] lists the various addresses and contents expected in the CC2420 RAM.

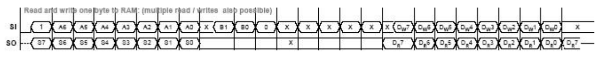

Read and write one byte to RAM: (multiple read / writes also possible)
SI
SO
1
A6
A5
A4
A3
A2
A1
A0
X
B1
B0
0
$D_W7$ $D_W6$ $D_W5$ $D_W4$ $D_W3$ $D_W2$ $D_W1$ $D_W0$
S7 S6 S5 S4 S3 S2 S1 S0
$D_R7$ $D_R6$ $D_R5$ $D_R4$ $D_R3$ $D_R2$ $D_R1$ $D_R0$ $D_R7$

Figure 17: Timing diagram for the CC2420

[1]`http://www.chipcon.com/files/CC2420_Data_Sheet_1_4.pdf`

The first bit of the expected command is a 1 for RAM access. The next 7 bits in the command specify the address in the RAM to which the data must be written. The next 2 bits select which of the three memory banks the address is located in the TXFIFO bank (00), the RXFIFO bank (01), or the security bank (10). Then, the bit specifying a read/write (0) or read (1) operation is transferred. After this, 5 dont care bits are transmitted. After this sequence of data, the data written into the RAM can be transferred, one byte at a time, MSB first. Successive writes and reads to memory can take place, which the caveat that memory locations automatically increment with each successive read or write. After a RAM strobe, memory access can only be terminated by bringing CSn high for the duration of an SCLK cycle.

Successive register access and RAM operations (provided that the RAM operations access successive addresses) can be undertaken with CSn low for each operation. As a result, no wait time is needed between one operation and next unlike a microprocessor, which needs to exit out of each command and redo any microprocessor-specific setup operations before executing the next command to the CC2420.

All CC2420 related modules are clocked on CC2420 clock, which must be 10Mhz or slower. Data to the chip on the SI and CSn lines is registered out in order to prevent any glitching on the line while a read or write to CC2420 registers or RAM is taking place. In order to properly set up SI and CSn well before the rising edge of SCLK, which is when the CC2420 will register in the data, SCLK is the inverse of the clock any module accessing the CC2420 runs off of. This is especially key in making certain that the setup and hold time constraints on CSn and SI are met in order to ensure the expected operation of the CC2420.

For the rest of this document, the registers accessed with be referred to by the names assigned to them in the CC2420 chip documentation.

### 5.1.3 Configuration

Certain registers must be strobed or written into before the CC2420 is ready to transmit data. This module implements a sequential FSM structure that performs the various operations needed to ready the

CC2420 for the transmission or reception of data.

In the first stage RESET, the entire CC2420 is reset with a hard chip reset by pulling the reset_chipn pin of the CC2420 low for a single clock cycle. After waiting for an additional 16 clock cycles, the RESET stage is exited. This wait is absolutely crucial, as after a reset to the chip certain capacitors and other elements of the DBK must be allowed to discharge before command strobes will function properly.

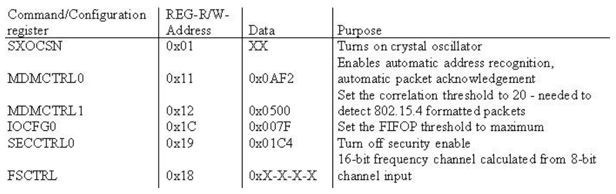
In the second stage, STROBE, the command register SXOCSN, is strobed in order to turn on the crystal oscillator inside the chip. Without the crystal oscillator turned on, RAM and register access of the CC2420 are not possible.

In the third stage, WAITLONG, we must wait for at least 1ms (or 10,000 100ns clock cycles) for the crystal oscillator to turn completely on. This number can be found in the CC2420 specifications for the CC2420 chip. In the end, it was decided to force the chip to wait for at least 2ms in order to essentially guarantee that register and RAM access would be possible by the end of the WAITLONG stage.

In the REGWRITE stage, 5 configuration registers (MDMCTRL0, MDMCTRL1, SECCTRL0, and FSCTRL) are consecutively written to in order to configure the chip to receive IEEE 802.15.4 formatted packets, enable automatic address recognition by the hardware, set the FIFOP threshold to maximum (the consequence of this will be discussed in the Reception section), and to program the frequency channel on which the chip will be transmitting. The consecutive writes are performed by simply constructing a 120-bit register that contains each of the 5 8bit register access commands followed by immediately by the corresponding 16-bits of data to be written into this register. Then, the data in the register is simply shifted out on the SI line on each clock cycle.

Table 1 lists all of the configuration and command registers accessed in the STROBE and REGWRITE stages.

After the REGWRITE stage is completed, we enter a single clock-cycle long stage called WAIT2 in which CSn is pulled high before we enter the next stage for safetys sake and in order to prevent incorrect values of SI from being entered into the RAM on the next stage. This stage is an artifact of the debugging process, and can be taken out in order to take full advantage of the feature of consecutive register and RAM accesses allowed by the SPI interface.



| Command/Configuration register | REG-R/W-Address | Data | Purpose |
|---|---|---|---|
| SXOCSN | 0x01 | XX | Turns on crystal oscillator |
| MDMCTRL0 | 0x11 | 0x0AF2 | Enables automatic address recognition, automatic packet acknowledgement |
| MDMCTRL1 | 0x12 | 0x0500 | Set the correlation threshold to 20 - needed to detect 802.15.4 formatted packets |
| IOCFG0 | 0x1C | 0x007F | Set the FIFOP threshold to maximum |
| SECCTRL0 | 0x19 | 0x01C4 | Turn off security enable |
| FSCTRL | 0x18 | 0xX-X-X-X | 16-bit frequency channel calculated from 8-bit channel input |

Figure 19: The configuration and command registers accessed by the configuration FSM.

Finally, after the REGWRITE stage, the MEMWRITE1 stage is entered. In this stage, the MSB(most significant byte) and LSB (least significant byte) of the PANID are written to the CC2420 memory locations 0x169 and 0x168, respectively. First, the memory address is strobed and set up, and then two consecutive bytes of data from the 16-bit PANID are passed in. Note that the LSB must be written first, followed by the MSB , as the memory location pointer automatically increments with each memory access. In the WAIT3 stage, we then bring CSn high for a single clock cycle. This wait stage is absolutely crucial since non-consecutive RAM accesses must be terminated by setting CSn high.

In the MEM2WRITE stage, we make the second write to memory in order to write the device address, or the SHORTADDR to the CC2420 memory. The MSB and LSB are written to memory locations 0x16B and 0x16A, respectively. Again, the LSB must be written to memory before the MSB since the address pointer of CC2420 RAM automatically increments.

The following stages (except for the IDLE stage) were introduced for debugging purposes and were then kept because they performed the useful function of allowing the user to check whether or not the PANID and SHORTADDR had been written into memory correctly. The sequence of stages WAIT3, ASKREAD1, WAIT4, and ASKREAD2 allows the user to read from the ram locations that were written to in the MEM1WRITE and MEM2WRITE stages in order to make sure that the PANID and SHORT-

ADDR were written to properly in those memory write stages.

Finally, the IDLE state is entered and signals that the chip has been configured by setting the "configured" flag high for all time. The FSM will stay in this state until the next reset signal. SI is low and CSn is high as well.

### 5.1.4 Transmission

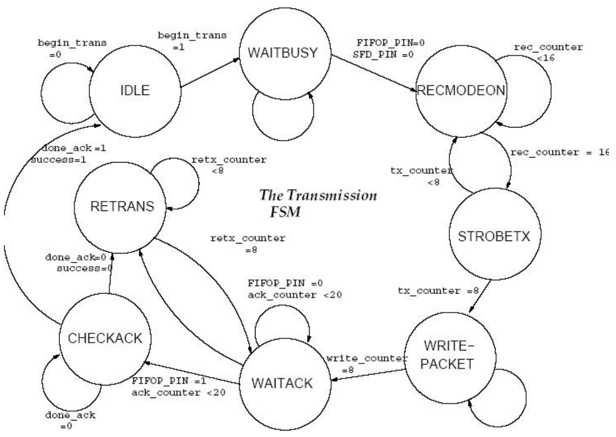


Figure 20: The transmission finite state machine.

After the configuration stage, the chip is ready to transmit packets of data. This module is responsible for handling the transmission of 40-bit (or 5-byte) packets of data. While this module is currently hardwired to handle the 5-byte packets of data, it can be easily modified to handle larger amounts of data by simply (1) changing the allowed size of the PAYLOAD input and (2) changing the counters in the WRITEPACKET stage. This module also waits for an acknowledgement(ACK) frame from the receiver it targets. If the TransmitAggressiveFSM does not receive an ACK frame, the FSM ensures that it will retransmit the packet until an ACK frame is received. If the ACK frame is received, TransmitAggressiveFSM will call a minorFSM called CheckAck responsible for parsing ACK frames in order to make sure that the ACK frame

WAITBUSY is the initial state. This part of the transmit operation checks to see whether a transmit operation is in progress by making sure that neither the SFD nor the FIFO pin are high. - if a transmit operation is in progress, the TransmitAggressiveFSM will continue to WAITBUSY. Otherwise, the operation will move to the RECMODEON state.

In the next stage, RECMODEON, the FSM strobes two command registers in succession. The first strobe to the FLUSHTX register (0x09) flushes the TXFIFO, the bank of the RAM that stores the packet to be transmitted, and the second strobe to the RXON command register (0x03)turns on the receive mode so that the transmitter will be able to receive acknowledgement signals from the receiver. Once this has been completed, the FSM will move on to the STROBETX state.

In the STROBETX, the FSM strobes the STXON (0x04) register, which turns on transmit mode. This strobe of the STXON register is what makes this an aggressive transmit  there is another register, STXCCAON that only turns on the transmit mode when the channel the transmitter sends packets out on is clear. While this is cleaner and more polite to other transmitters using this range, it was decided that it was simpler to blast the packet from the transmitter regardless of other traffic on the line since the transmitter retransmits if an acknowledgement frame is not received from the receiver. This stage also strobes the WRITE_TXFIFO (0x3E) register, which will allow us to write the packet to the TXFIFO in the WRITEPACKET stage.

The WRITEPACKET stage is especially crucial as this is where the packet constructed by the FSM is shifted into the chips TXFIFO. At this point, it is considered that this is probably the state in which packets were not written to the TXFIFO correctly. Instead of strobing the TXFIFO byte access register (0x3E), it is suggested that this strobe be removed and that a memory write to location 0x000 replace this attempt at writing to the TXFIFO. At any rate, the packet constructed takes the following form:

1. 1-byte LENGTH  The most significant bit is reserved as a 0. The following 7 bits encode up to 128 bytes of data that can be stored in the TXFIFO. However, this length also includes the overhead of storing various preamble information

that must be included in the packet and that will always take up 11 bytes of space. As a result, the maximum PAYLOAD size can only be 117 bytes long. So, when the user enters their LENGTH size, the LENGTH must equal the length of the PAYLOAD in bytes + 11 for overhead. For example, the LENGTH of a 5-byte payload packet is 16. This must match the rest of the packet.

2. The 2-byte FCS (frame control sequence) is next. To request an acknowledgement, this must be set to 0x8861.

3. Next is the 2-byte DATASEQNO, which essentially numbers the packet so that the TransmitAggressiveFSM can match which packets it has received acknowledgements for.

4. 2-byte PANID The PANID of the transmitter and receiver.

5. 2-bytes DESTADDR The SHORTADDR of the receiver.

6. 2-bytes MYADDR The SHORTADDR of the transmitter.

7. 5-bytes PAYLOAD The 5-byte payload currently hardcoded into TransmitAggressiveFSM.

This data is shifted into the TXFIFO. As can be seen from the above packet structure, there is an overhead of 9 bytes and a payload of 5 bytes. The chip itself tracks the length, and when n-2 bytes are written to the TXFIFO, the chip writes in another 2 bytes for a frame check sequence which contain a checksum on the rest of the packet, and then automatically transmits the packet after adding an additional SFD (start of frame delimiter) preamable. While the packet is being transmitted, the SFD pin goes high when the SFD field has been completely transmitted.

The FSM then enters the WAITACK state, where it waits for 6760 clock cycles for an acknowledgement frame. The number of wait cycles was calculated from the documentation and includes the time it takes for the packet to be sent out as well as fact that it takes a certain amount of time for the ACK frame to be transmitted back. The CC2420 chip knows that an ACK frame has been received when the FIFOP_PIN goes high.

If an ACK frame has not been received, the FSM simply strobes the TXON register without rewriting to the TXFIFO. This causes the CC2420 to retransmit the packet still in the TXFIFO.

If an acknowledgement is received, the FSM enters the CHECKACK stage, where it calls the CheckAck FSM to read the RXFIFO of the transmitter and chunk through any data in the RXFIFO to check for the proper format of an acknowledgement packet, which simply consists of: reading from the RXFIFO by strobing 0x7F (the second bit of the read command must be a 1 for a read) and then chunking in the packet and outputting a high success value for a clock cycle if the length of the ACK packet is 5, if the DATA_SEQ_NO matches the numbered packet the transmitter sent, if the frame control sequence is 0x0002, and if the MSB of the CRC byte from the ACK packet is high, which means that the checksum passed on the receiver end. The mechanics of this can be seen in the code for the Check_ACK module.

Figure 21 shows the TransmitAggressiveFSM in action with a ModelSim of inputs and outputs over time as it exits the STROBETX state and enters the WRITEPACKET state.

### 5.1.5 Reception

The ReceiveFSM was relatively straightforward to code since it was simply a larger version of the CheckAckFSM. Admittedly, this ReceiveFSM is hardcoded to parse a packet with a 5-byte test PAYLOAD, but it should be relatively simple to adjust the counters in the state that parses the packet in order to parse a 100-byte packet. An intelligent parser capable of reading the length byte and then parsing the packet for that length of time would have been ideal. However, time constraints and debugging the system did not allow for this to happen.

The Receive FSM in Figure 22 is far less complicated that the TransmitAggressiveFSM. In the initial stage RECON, the RXON command register is strobed to enable the reception of packets. The ReceiveFSM then waits for either a packet to come in in this case both the FIFOP_PIN and FIFO_PIN in-

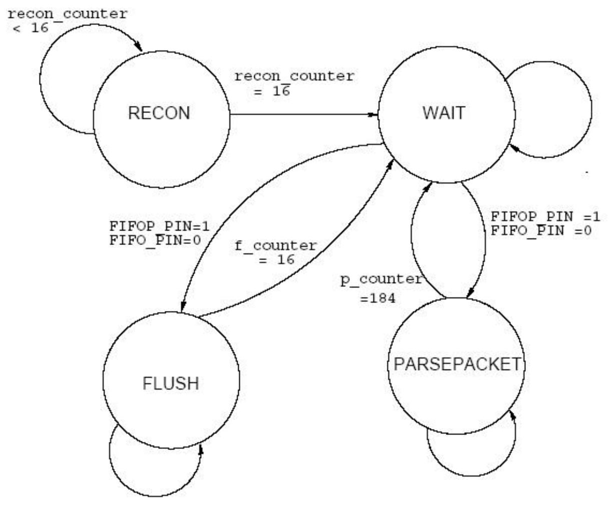


Figure 22: The Receive FSM.

puts from the CC2420 will go high. If a packet is received, the PARSEPACKET stage is enteredl, where the RXFIFO is strobed with the 0x3E command and the assumed 5-byte packet is parsed. The mechanics of this packet-parsing can be seen in the code for ReceiveFSM. The parsing of the packet does depend on the format of the packet being sent from the transmitter. If the ReceiveFSM is in the WAIT stage and FIFOP_PIN goes high while FIFO_PIN is low, this signals an overflow. The FIFO_PIN on the CC2420 goes high when more than a byte of data is in the RXFIFO, while the FIFOP_PIN only goes high when a certain nonzero threshold of bytes is in the RXFIFO (This threshold is set in the ConfigurationFSM with the programming of a register). If FIFOP is high and FIFO is low, an underflow has occurred and the RXFIFO needs to be flushed. This is accomplished by strobing the FLUSHRX command register (0x08) twice for good measure.

Figure 23 shows the ReceiveFSM in the PARSEPACKET stage, where various contents of the packet are read out and outputted from the ReceiveFSM.

#### 5.1.6 Integration With The FPGA - Creation of a 27/4-MHz clock

In order to integrate with the FPGA, a 6.75Mhz clock was created by passing the 27Mhz clock through a module containing a 4-bit registered counter. Then, the MSB of the counter was passed out as the 6.75 Mhz clock. The counters reset was different from the reset of the other modules to ensure that the clock of the wireless modules would be completely decoupled from the wireless modules themselves. The general configuration of the hardware interface between the CC2420 and the 6.111 labkit can be seen below. The wires are connected from a cable between the user2 ports on the labkit and the pins on the P4 header strip on the CC2420DBK.

Key things learned while wiring up the CC2420DBK:

1. The voltage regulator pin on the CC2420DBK P4 pin number 10 must be wired to high (to 3.3 volts) in order for the CC2420DBK to function correctly.

2. An external power source between 4-7 volts must be supplied to the CC2420DBK in order to allow the chip to properly drive outputs like SO, FIFO, FIFOP, and SFD.

3. The CC2420DBK microprocessor reset (P4, pin 3) must be wired to ground to bypass the microcontroller in order to use to SPI interface. This ground must connected to both labkit ground and CC2420DBK ground (P4 pin 20).

## 6 Conclusion

In the end, only the transmit side of the wireless portion of our project was functional. Even after intense debugging, we were unable to make the receiver side of the wireless link function properly with automatic address recognition. Instead, we could only receive what appeared to be portions of the test 5-byte packet being sent from the transmitter side of the network. Problems with the writing of the packet to the transmitter may be the cause of this problem.

As a result, we were unable to integrate the wireless portion of our system into the project. We were, however, able to hardwire the A/D, compression, decompression, and D/A systems directly together to take in music, compress it, decompress it, and play it on a set of speakers or headphones. While our project was not fully integrated, we do feel that we learned a lot both from the construction of our individual projects as well as group discussion and debugging of the D/A, compression and decompression, and wireless transmitter and receiver sides of our project. We would like to thank the awesome 6.111 staff for providing us with the tools to construct and implement such an ambitious project and for the help they provided in debugging various parts of system and offering design tips and solutions. Thanks especially to our long-suffering and patient TA, Dave Wentzloff!

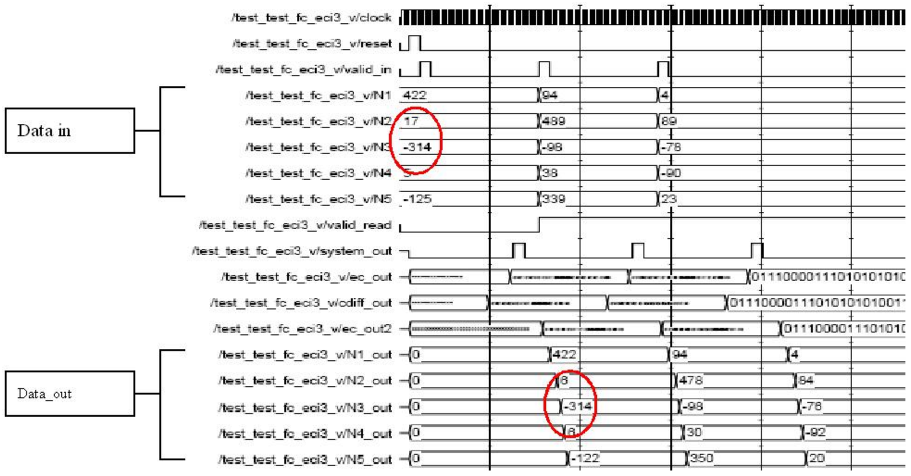


Figure 10: ModelSim of compression/decompression modules with working error correction recursion.

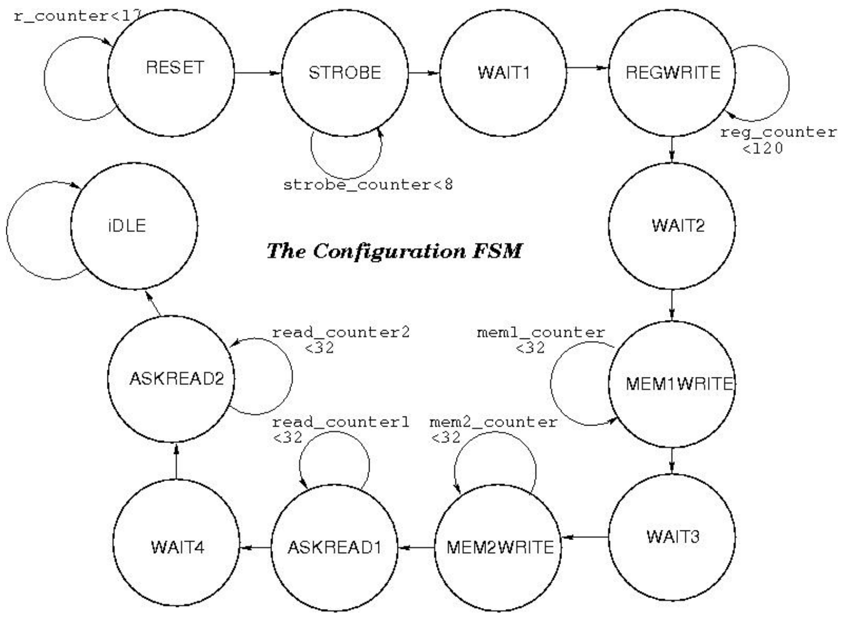


Figure 18: The configuration finite state machine.

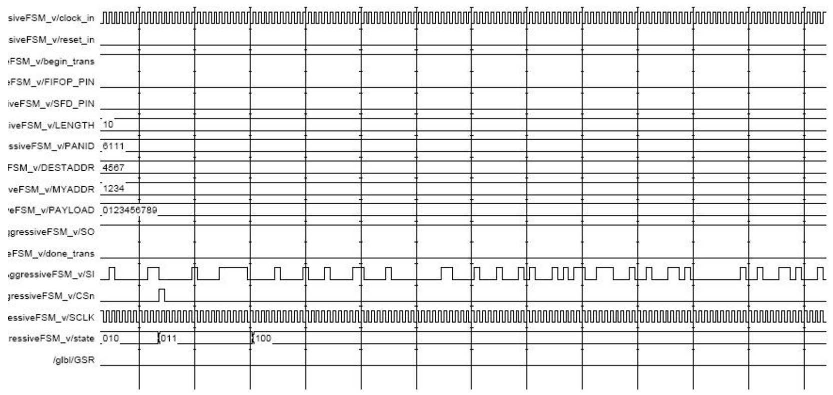


Figure 21: ModelSim of the Transmit FSM in action.

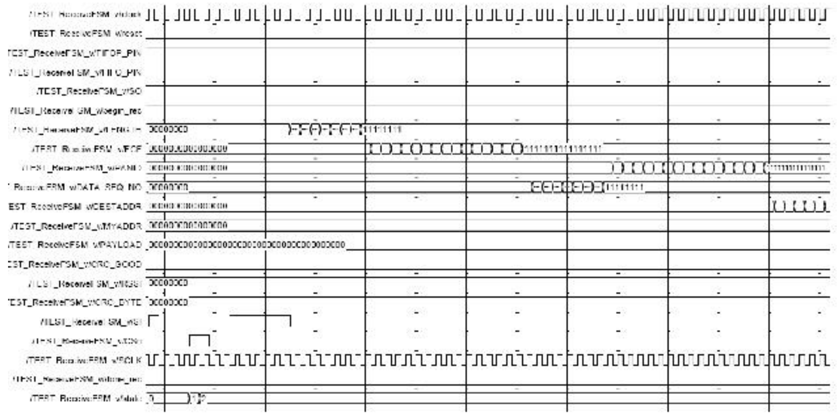

TEST_ReceiveFSM_v/FIFOP_PIN
TEST_ReceiveFSM_v/SO
TEST_ReceiveFSM_v/PAYLOAD
TEST_ReceiveFSM_v/CRC_GOOD
TEST_ReceiveFSM_v/CRC_BYTE
TEST_ReceiveFSM_v/SCLK

Figure 23: The Receive FSM.