Wireless Audio Effects Processor

6.111 Introduction to Digital Systems

Spring 2007

TA: Javier

Lohith Kini, Spyros Zoumpoulis, Rahul Shroff *

Abstract

Our final project will be designing and implementing an audio system in which the audio signal is input, wirelessly transmitted, audio filtered and equalized. The audio is inputted through a microphone and is digitized using an Analog-To-Digital converter with the LM4550 AC'97 Codec. The audio signal is then compressed using the Discrete Cosine Transform (DCT) and is wirelessly transmitted and received using a low power RF transceiver. The signal, after being decompressed, is then processed using a Fast Fourier Transform module (FFT) and its Inverse (IFFT), as well as an equalizer module.

* {*lkini,szoumpou,rshroff*}@mit.edu

Introduction

The aim of this project was to input an audio signal, wirelessly transmit it after compression, decompress it at the receiving end and then add the audio equalisation as desired by the user. Our idea was inspired by the various sound manipulation schemes prevalent in today's world, and our aim was to create a device similar in nature to Apples AirTunes in its outwardly functionality; the only difference being that our wireless audio effect processor would be programmed on an FPGA, rather than the more traditional means. What follows is a detailed report describing our efforts to achieve these goals.

Specifically, the report is divided as per the contributions of the various members of Team 15. First, we will present the compression and decompression algorithms implemented by Lohith Kini using the MDCT. The following section deals with wireless communication and error correction algorithms as implemented by Spyros Zoumpoulis, and finally, the report culminates with an explanation on Audio equalisation and visualisation worked on by Rahul Shroff.

A detailed Appendix enlisting the Verilog code for the various modules is added to the end of this report. This Appendix also contains Timing Specification diagrams, and screenshots from ModelSim as appropriate.

Audio Input from Microphone [Lohith]

AC97 Codec

The audio is input from a microphone and fed through the LM4550 AC97 Codec. The audio is preprocessed, including changes to the volume level, and then digitized by the internal analog-to-digital converter in the codec. The user can manually change the audio volume using the up and down buttons on the labkit interface. The resulting data, which are interfaced with the codec serially, are divided into frames. The bit clock is generated by the codec, and runs at 12.288MHz. There are 256 bits per frame at a frame speed of 48000 fps. The ADC operates optimally at 48 kHz sample rate (like most audio we hear). The frame-by-frame data are stored in an internal RAM dedicated to store audio input.

FIFO

When working under two clock domains, one which is the 27 MHz labkit clock and the other is the AC97 bit clock, a FIFO (First In First Out) module is needed to store AC97 data and output after every 8 audio points. The reason we need to output only 8 audio points for a 16 point MDCT is because we are implementing overlapping between consecutive audio frames. If we only have 8 audio points, we can combine that with the 8 audio points that were stored from the last frame and obtain a 16 point audio input for the MDCT.

FIFO Controller

The FIFO controller was implemented to provide lapping of consecutive audio frames. It uses a COREGEN BRAM module to store all eight points that come into the MDCT module. Its action mechanism is simple: It fist outputs all eight audio points stored from the previous frame and then writes the new set of eight points into memory. The new set of audio points is manually delayed such that the old eight points of the audio are input into the MDCT and then the new set is simultaneously input into the MDCT as well as the BRAM module.

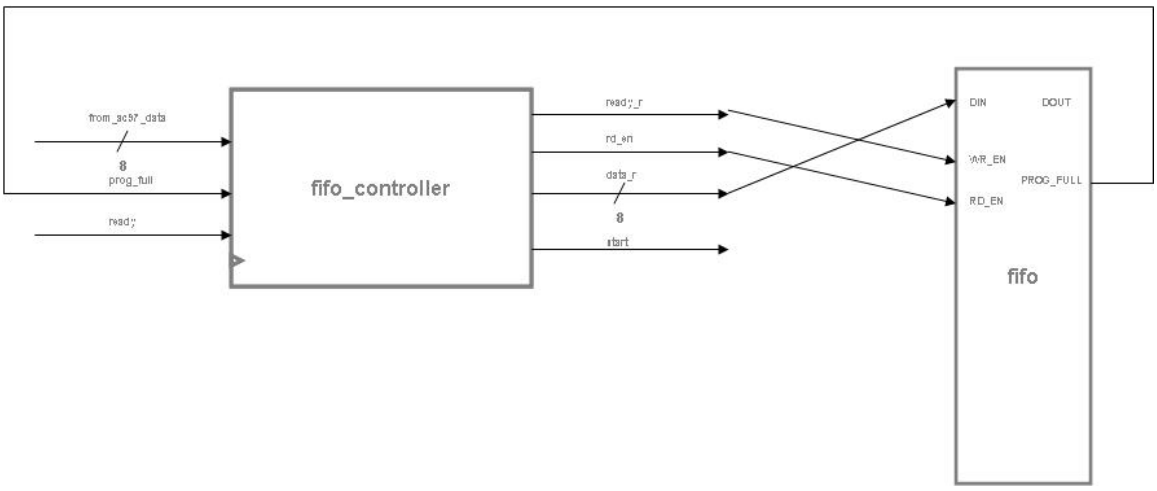


Figure 1: The above picture shows the signals between FIFO and the FIFO Controller

Debugging

We had a lot of trouble implementing CORGEN modules, specifically FFT and FIFO. Rahul will talk about the FFT module. The team spent a significant amount of time creating our own version of the FIFO before Gim recommended using FIFOs with Distributed RAMs, which cleared up the problem with the COREGEN FIFO module. The FIFO Controller had a few timing issues which was cleared up by carefully timing variable ADDR with data that's input into the FIFO.

MDCT [Lohith]

Theory

The MDCT (Modified Discrete Cosine Transform) is very similar to the well known DCT (Discrete Cosine Transform) but the modification is used to allow overlapping of consecutive frames of audio. To this effect, a 16 point audio signal creates a 8 point frequency spectrum and also allows for quick compression techniques. The sound quality is not too different since the overlapping effect takes care of the loss of information when few precision bits of the compression are thrown away.

Modern audio technology such as MP3 and AAC use MDCT for audio compression. Since it is a lapped transform, the MDCT maps a $\mathbf{R}^{2N}$ point signal into the $\mathbf{R}^{N}$ frequency domain. The $2N$ real numbers x_i for $0 \leq i \leq 2N-1$ are MDC Transformed into N real frequency components, X_j for $0 \leq j \leq N-1$.

$$X_j = \sum_{i=0}^{2N-1} x_i \cos \frac{\pi}{N}\left(i + \frac{N+1}{2}\right)\left(j + \frac{1}{2}\right)$$

Computation

Computing the MDCT is simple, especially for a fixed $N = 8$. I created a cosine lookup table which outputs the different cosine scalars in the MDCT equation above. As we get the 16 point audio input (remember 8 first comes out of the BRAM in the FIFO Controller and then the next 8 is from the current audio frame), each coefficient X_j is being computed simultaneously, with multipliers outputting x_n with the cosine scalars and adders accumulating the sum for $i = 0 \ldots 2N-1$. Most of these mathematical modules were readily available on COREGEN and allowed for seamless integration with the other pieces of the module.

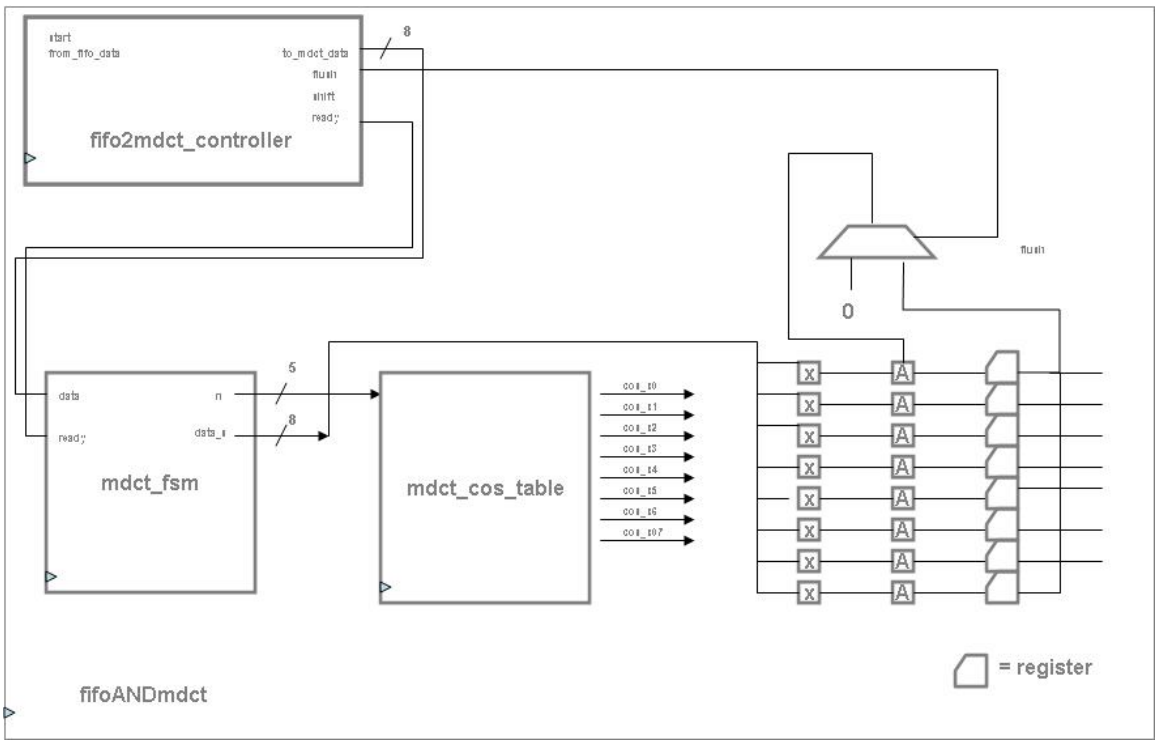


Figure 2: The above picture is an overall level schematic of the FIFO and MDCT

The precision of the computations in the MDCT module vary thoughout. The audio input is 8 bits wide but the multipliers allow for a 16 bit bus width for all coefficients. After all computation is complete, I simply clip the bottom 8 bits of each coefficient to allow for compact presentation to Spyros' wireless module. All lossy compression, such as precision loss was tested in MATLAB with sample audio inputs and outputs, which we verified qualitatively.

The communication with the wireless is simple, a 64-bit coefficients signal (with all eight 8-bit MDCT coefficients bundled together) and a c-avail signal that is asserted for one clock cycle when the next set of MDCT coefficients are available.

Debugging

The only problem I encountered was glitchy inputs to the feedback adders because of incorrect register loading at the output of the adder. The problem was quickly solved with a $flush$ signal from the FIFO Controller, which tells the MDCT to keep accumulating 0 until the next set of audio inputs are input to the MDCT for coefficient computation.

IMDCT [Lohith]

The IMDCT module is similar to the MDCT except the computation is a mirror image of what occurs in the MDCT. All X_j coefficients for $0 \leq j \leq N-1$ are computed using overlapping of audio inputs. For that same reason, the coefficients should also be lapped during IDMCT computations to recalculate the original input signal. The IMDCT transforms N real numbers X_j into $2N$ real numbers as follows

$$y_i = \frac{1}{N} \sum_{j=0}^{N-1} X_j \cos \frac{\pi}{N} \left(i + \frac{N+1}{2} \right) \left(j + \frac{1}{2} \right)$$

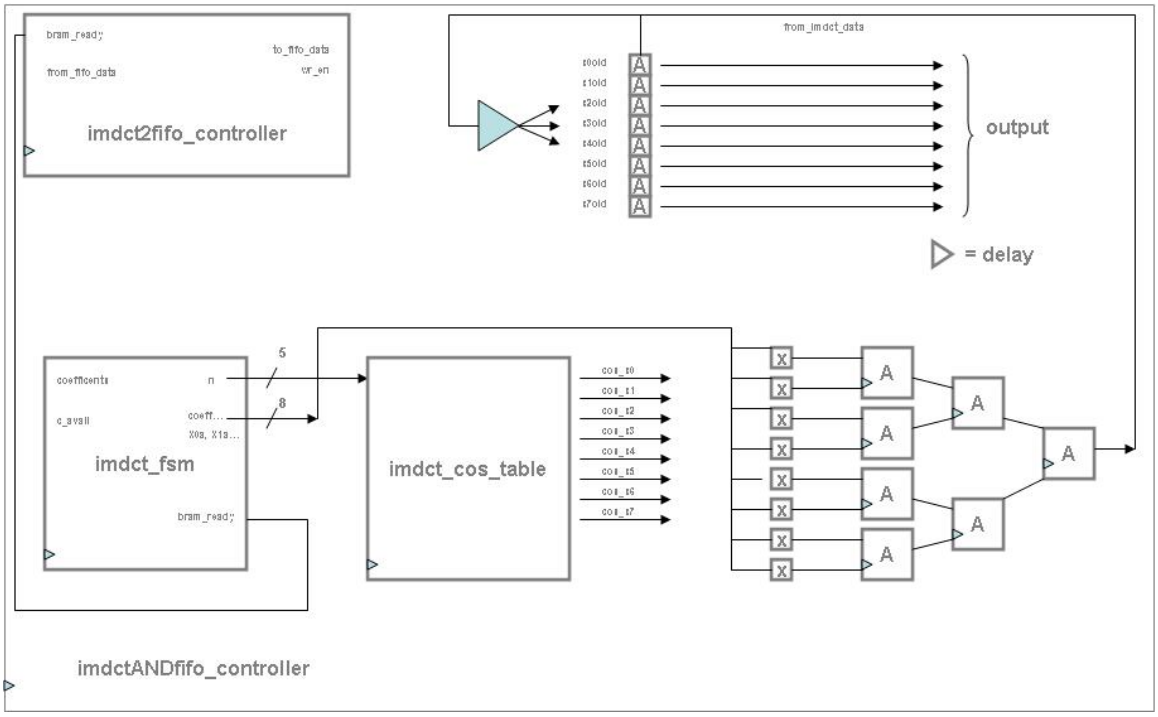


Figure 3: The above picture is an overall level schematic of the IMDCT and lap Controller

Computation

The computation of the IMDCT is very similar to the MDCT. The schematic for the module is shown below.

Debugging

Since the MDCT was fairly easy to debug, all the problems with the transform was resolved in the MDCT module. Thus, there was no subsequent errors in the IMDCT module. All data had to be checked to make sure the output of the IMDCT was actually taking the inverse of a sample input coefficients that were being provided in the testbench.

In the example below, the small red box represents a new 8-point set audio input and the 16-point in the longer red box is all 16 points of the audio signal (audio inputs 1-8 came from the BRAM in the FIFO Controller) with the corresponding MDCT Coefficients highlighted in blue. The IMDCT returns a signal that approximately sounds like the input one (i.e. $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]$) but the values are always a little off due to lossy compression and other factors.

Audio Equalization and Visualization [Rahul]

Overview

All the analogue signal processing occurs on the inputted audio signal through the programmes generated on the LabKit. An analogue to digital conversion is implemented via the AC97 Codec on the 6.111 LabKit. The inputted audio signal is then sent through an asynchronous FIFO to ensure the integrity of the signal which is inputted through the Codec at 48 kHz, but is then eventually processed at the system clock speed, 27 MHz. The audio signal coming out of the FIFO is then sent to an FFT

Figure 4: ModelSim Simulations of Lohith's MDCT/IMDCT Modules

module where it undergoes a Fast Fourier Transform, thereby generating a frequency spectrum for the audio signal. The coefficients thus generated are then sent to two modules - where they are used to produce a simple 8 bin visualisation implemented using the VGA, and audio equalisation, based on user input through the 8 switches located on the LabKit. Ideally, even though we did not get a chance to implement it, a loop is created sending these frequency coefficients through an IFFT module which recreates the modified audio signal and finally outputs it to the AC97 codec.

In the sections that follow we will explore these various modules in some detail. A block diagram and FSM is shown in the figure below.

Audio Module [Rahul, Lohith]

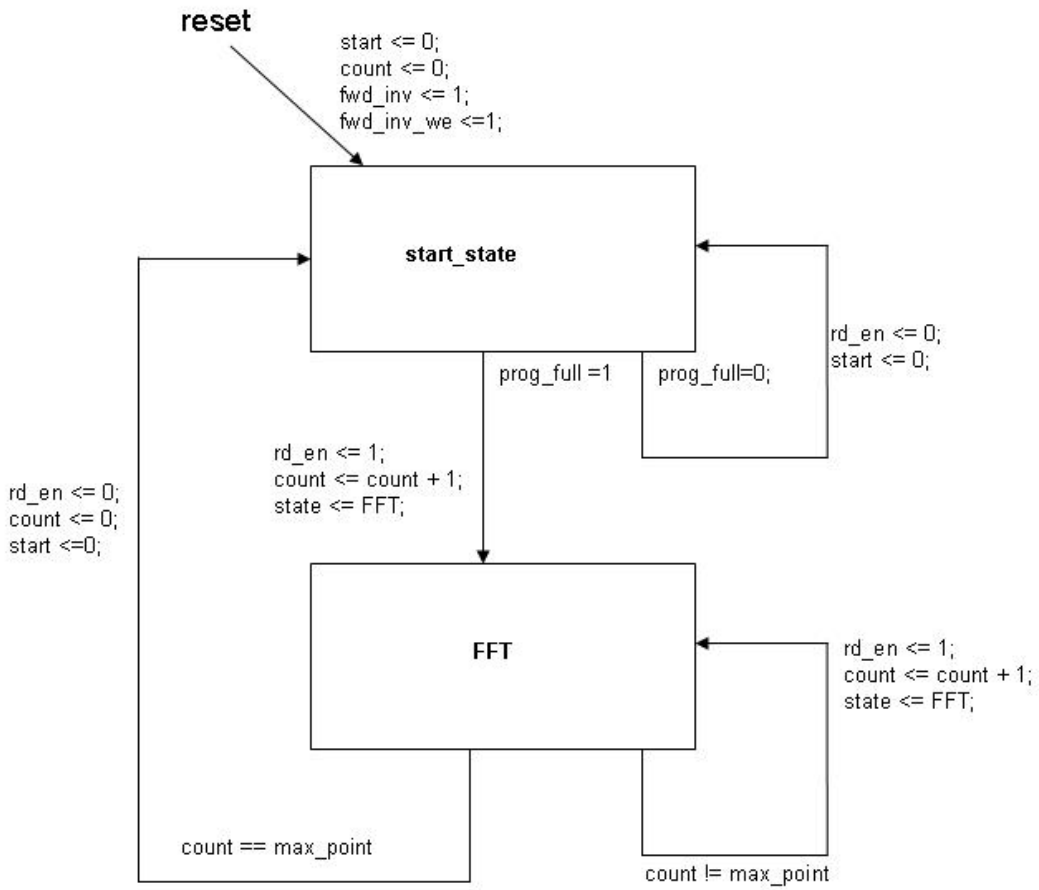
This module is used as a framework for the handling of audio signals, and essentially provides us with a black box interface to the AC97 Codec. It takes in data from the microphone jack on the LabKit and provides us with an 8-bit sampled output (It may be noted that the chip could give us output of up to 18 bits). A handshaking signal ready is also outputted by this module, indicating the presence of valid output data from the chip. The Codec runs on an internal clock, namely the ac97-bit-clock, which has a sampling rate of 48 kHz. As a result, ready goes high every 48 kHz. It is important to note that this sampling frequency is much slower than the system clock, and the ratio between the two clock rates is pivotal for calculations involving timing specifications between the various modules. In addition to a sampled output, this module also provides the ability to loop the sound back to the headphone jack on the LabKit. It also provides a 20-bit sine wave to test the functionality of the Codec.

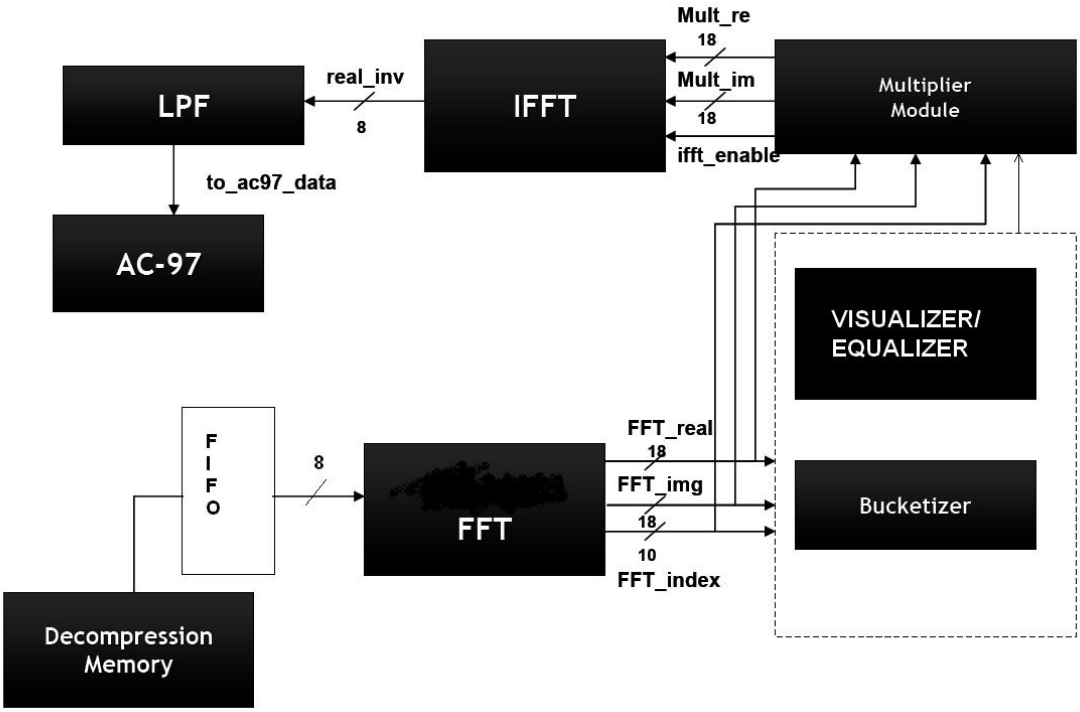
Primary FFT [Rahul, Lohith]

This module forms the overarching connecting module, which is finally instantiated in the Audio Module. This module is responsible for connecting the block-mem-fft-controller, the FIFO, the FFT module, the block-mem-ifft-controller and the IFFT module. It takes as inputs: ac97-bit-clock, clock-27mhz, from-ac97-data, reset and ready, and produces as outputs: xk-re, xk-rm, xk-re2, xk-im2. An explanation of these various modules follows in the sections below.

Asynchronous First In First Out (FIFO) [Rahul, Lohith]

This is a CoreGen module that implements an asynchronous FIFO, which is needed to buffer the data as it comes from the AC97 Codec sampled at 48 kHz, and is passed to the FFT module clocked at the system clock. The FIFO takes as input: DIN, rd-en and clock, and generated as output: DOUT, prog-full and wr-en. The prog-full signal indicates when the FIFO is filled with data of the specified count. When this signal goes high, we release the data for processing to the FFT module.


reset
start <= 0;
count <= 0;
fwd_inv <= 1;
fwd_inv_we <=1;
start_state
rd_en <= 0;
start <= 0;
prog_full =1
prog_full=0;
rd_en <= 1;
count <= count + 1;
state <= FFT;
rd_en <= 0;
count <= 0;
start <=0;
FFT
rd_en <= 1;
count <= count + 1;
state <= FFT;
count == max_point
count != max_point



FFT (IFFT) Module

This output signal DOUT is passed into the FFT module, which is responsible for describing the audio input as a frequency spectrum which can then be analyzed as needed. The FFT core computes an N-point forward DFT where N can be $2^3 - 2^16$. The input data is a vector of N complex values represented as two's complement numbers. All memory needed to make the computation is on-chip using either Block RAM or Distributed RAM. The FFT core uses the Cooley-Tukey algorithm for computing the FFT. The from-ac97-data serves as the time domain audio data that will eventually be described in the frequency domain via a Fast Fourier Transform. The FFT module generated by CoreGen takes in as input a start signal which acts as a signal that triggers the computation of the FFT, as and when each frame of data is available. The module stores the FFT in three important output signals: xk-re, xk-im and xk-index. One of the design choices for the FFT module was the point size of the transform being produced. A 512 pt FFT was successfully tested on sample audio input. However, during the implementation of the Visualiser and Equaliser, an 8 pt FFT was used thereby creating 8 buckets for audio processing. This was done, as the IFFT was not successfully implemented, and an 8 pt IMDCT transform created by Lohith Kini was used to reconstruct the audio signal. Another design choice was to select the Pipelined, Streaming Input/Output architecture of the FFT core, purely because it allows for continuous data processing which was necessary for our system. This solution pipelines several radix-2 butterfly processing engines to offer continuous data processing. Each engine has its own memory banks to store the input and intermediate data. The core has the ability to simultaneously perform transform calculations on the current frame of data, load input data for the next frame and unload the results of the previous frame. The unscaled option was selected to prevent errors involving a faulty scaling schedule, which needed to be tackled during the development of the system. Finally, the FFT module has a fwd-inv signal, which is high when the forward transform in being computed and is pulsed low to compute an IFFT.

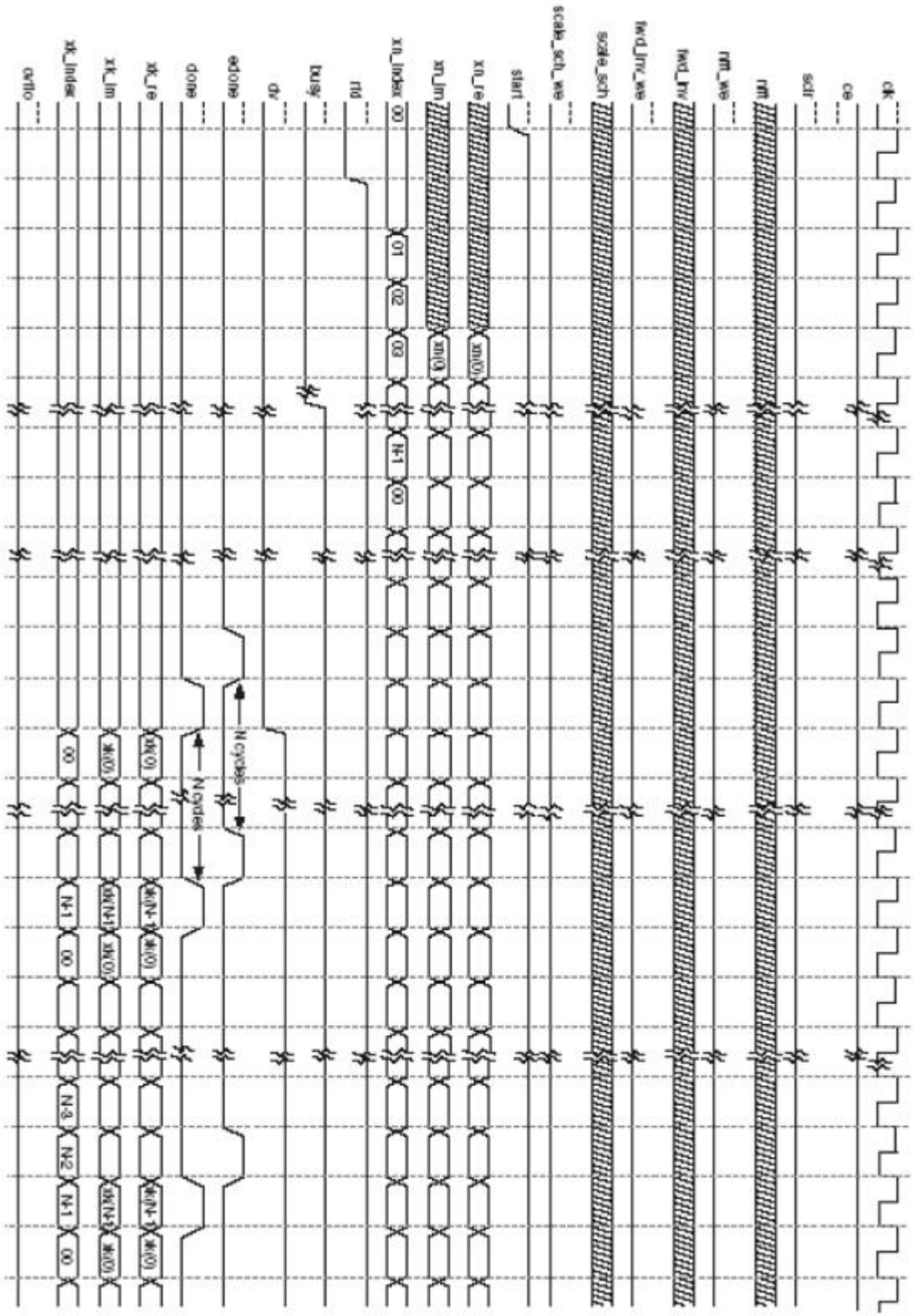


Figure 5: Timing Specifications for Pipelined Streaming I/O

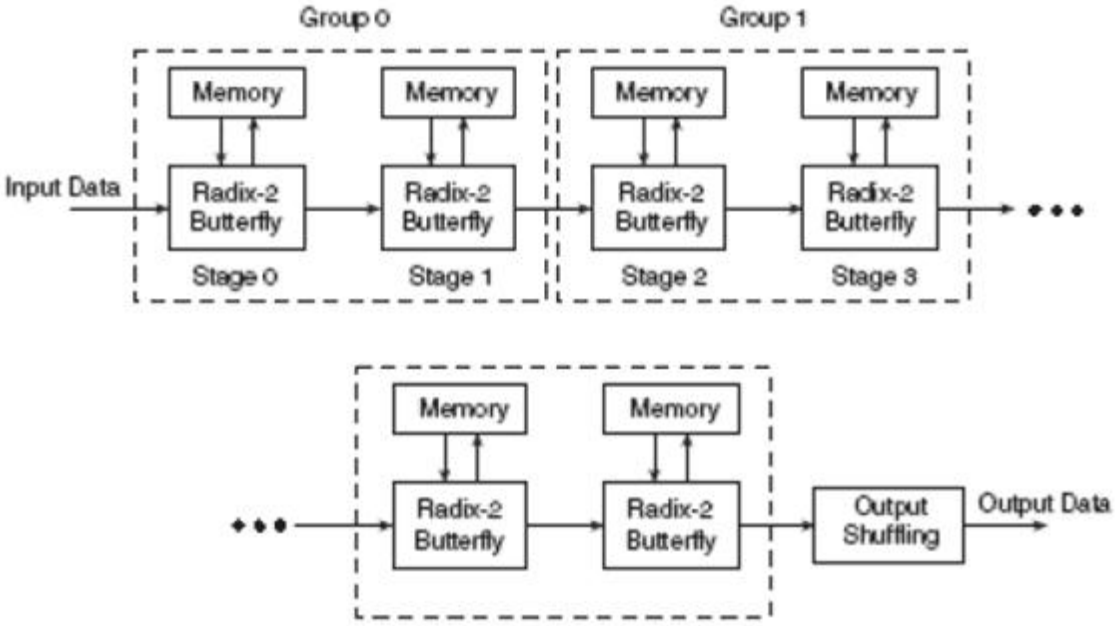


Figure 6: Pipelined Streaming I/O

Block-Mem-fft-Controller

This module provides the FSM that is instrumental in interfacing between the FIFO and the FFT modules, sending the correct triggering signals and inputs to each of these modules. It takes as inputs: reset, clock-28mhz, from-ac97-data, prog-full and ready, and produces the following outputs: fwd-inv, fwd-inv-we, data-r and start sent to the FFT module, and rd-en to the FIFO. This module also has a parameter max-point, which determines the maximum number of points per frame in the input. This module initially registers the input data in from-ac97-data and ready, to ensure that there will be no metastability problems. This data is stored in data-r and ready-r. As mentioned earlier, this module implements an FSM to generate the required control signals. IT may be noted that all the FSMs used in this project are Mealy Machines, in the sense that the output at each state is a function not only of the current state but also of its inputs. In case of reset being asserted high, we transition to the start-state, and set start and count low. We also set fwd-inv high, as we are computing the FFT in this case. The following are the transitions that take place in the FSM. It may be noted that all the transitions occur at the rising edge of the system clock.

start-state: We enter this state if reset is asserted. We check to see if prog-full is high, i.e., if the current frame of data has been loaded into the FIFO. If it is indeed high, we set start and rd-en high, increment count (which is later compared to max-point) and transition to FFT. If prog-full is not high we assert start and rd-en low, and loop back to start-state.

FFT: We enter this state if the FIFO is full, and has the data of the current frame to be read into the FFT CoreGen module for computation. Here we check to see if the counter has reached its maximum value, and if it has we set start, rd-en and count low, and transition back to start-state. If not, we merely increment the counter, set rd-en high, and return to FFT.

Equalisation & Visualisation

The coefficients generated by the FFT module are then sent to the VGA to have a real-time visualisation of the Frequency spectrum. The visualiser is created using the Rectangle, Display Field and VGA modules from Laboratory 4. It basically consists of 8 rectangles of different colours corresponding to

the 8 points of the FFT of the audio signal. This is implemented in the Audio Module. Using CoreGen multipliers and adders we find the magnitude of each point of the Fourier Transform, and send this to the Display Field module, to ensure that the rectangle heights are scaled appropriately. The 8 coefficients of the FFT are also displayed on the alphanumeric display of the LabKit.

The equalisation was a little more tricky to implement due to the inability to successfully take the IFFT of the frequency components of the signal. We used the equivalence of the DCT and FFT to achieve this goal. Having already implemented a successful loopback using the MDCT and IMDCT, we used this to implement the reconstruct the equalised sound.

First let us understand why the DCT and FFT are equivalent transforms for some real audio signal $x[n]$. The FFT is just a set of samples of the DTFT at the points $2\pi k/N$. There are many versions of the DCT, but the MDCT implemented in this project, samples this same DTFT with a half sample delay. Thus the samples are at the points $2\pi k/N + \pi/N$. For real signals, the DTFT exhibits conjugate symmetry, and taking advantage of this we can establish the following reduction property. Having a IMDCT module, we can easily compute the IFFT by shifting the input in frequency by a half sample. Thus we can use the IMDCT to implement the loopback instead of the IFFT. The equalisation was implemented in the Audio Module, and involved a simple scaling of coefficients as imputed by the user with the help of switches. The processed audio was then replayed through the headphone jack on the LabKit.

The visualiser and equaliser could be selected using the buttons on the LabKit. The enter button on the Labkit also allows the user to record his own voice, and either watch the changing frequency spectrum, or change its frequency components and listen to the regenerated audio signal.

Block-Mem-ifft-Controller

Although this module did not successfully take the IFFT of an inputted frequency spectrum, it was implemented with the aim of sending the correct input signals to the CoreGen IFFT module. Like the block-mem-fft-controller, this module was also implemented using an (Mealy) FSM, with similar states and transitions. The main difference in this module was the fact that fwd-inv was set low; indicating an inverse FFT was being performed on the input signal.

Testing & Debugging

Most of the testing and debugging involved meeting the timing specifications. Careful simulation was needed to ensure that each of the control signals was given the appropriate value after a predetermined number of delays. One of the main problems faced was to ensure that the data was inputted into the FFT module only after the start signal had been high for 5 clock cycles. Meeting the specifications perfectly relied heavily on testing through ModelSim. A Verilog Test Fixture was independently created for every module to enable modular testing, thus enabling the system to work effectively once strung together as a whole. The testing of the CoreGen FFT Module was especially difficult. In the end, it was noted that the Specification sheet as published by Xilinx was incorrect, with respect to the timing of the done and edone signals. Once this module was working as desired, it was coupled with input from the FIFO, controlled by the block-mem-fft-controller module. A test fixture was created for the module as well. The main challenge in getting this working was once again the timing. Eventually, the DOUT signal from the FIFO was delayed by 5 cycles to ensure that it was fed to the fFT Module the correct number of cycles after start was asserted. Once this was working, a lot of time was spent trying to successfully take the IFFT of an input signal. The same procedure was followed; however, unfortunately we were unable to create a successful audio loopback. As mentioned earlier this was finally achieved by using the equivalence of the IMDCT and IFFT, which enabled us to successfully implement the audio equaliser.

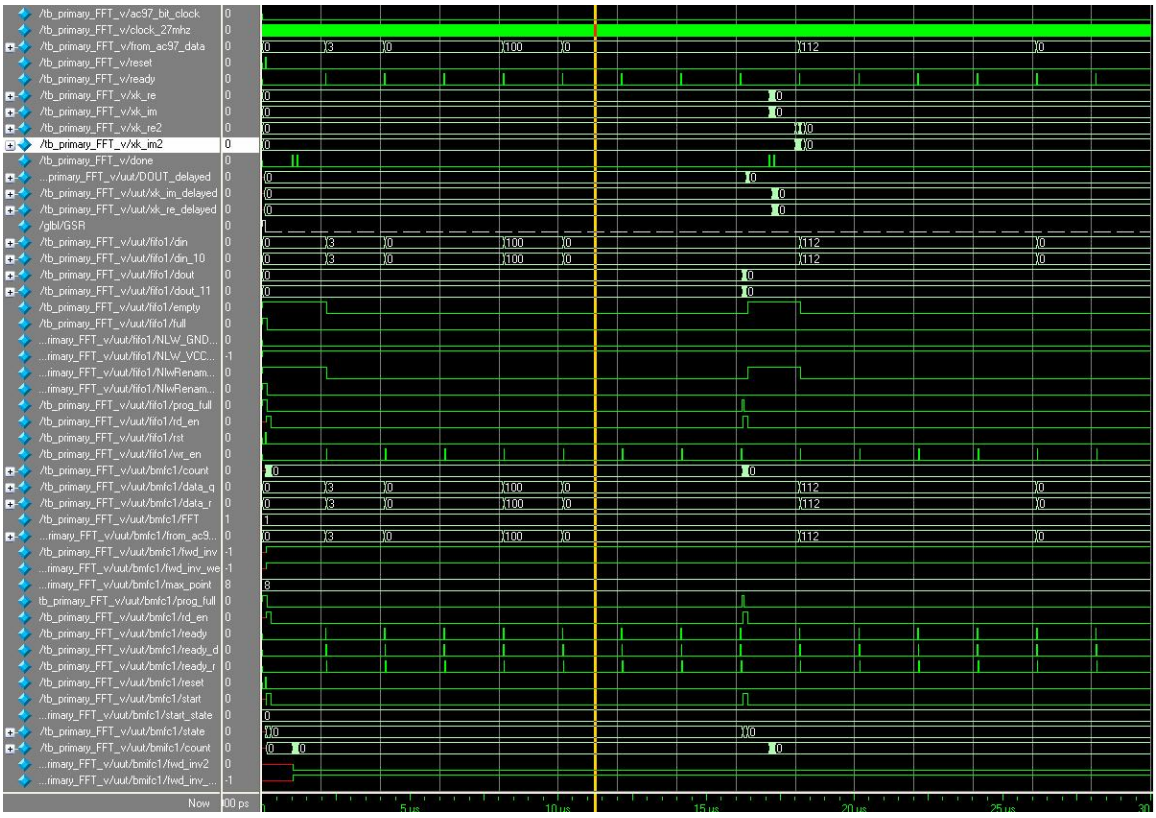


Figure 7: Primary FFT Test Bench

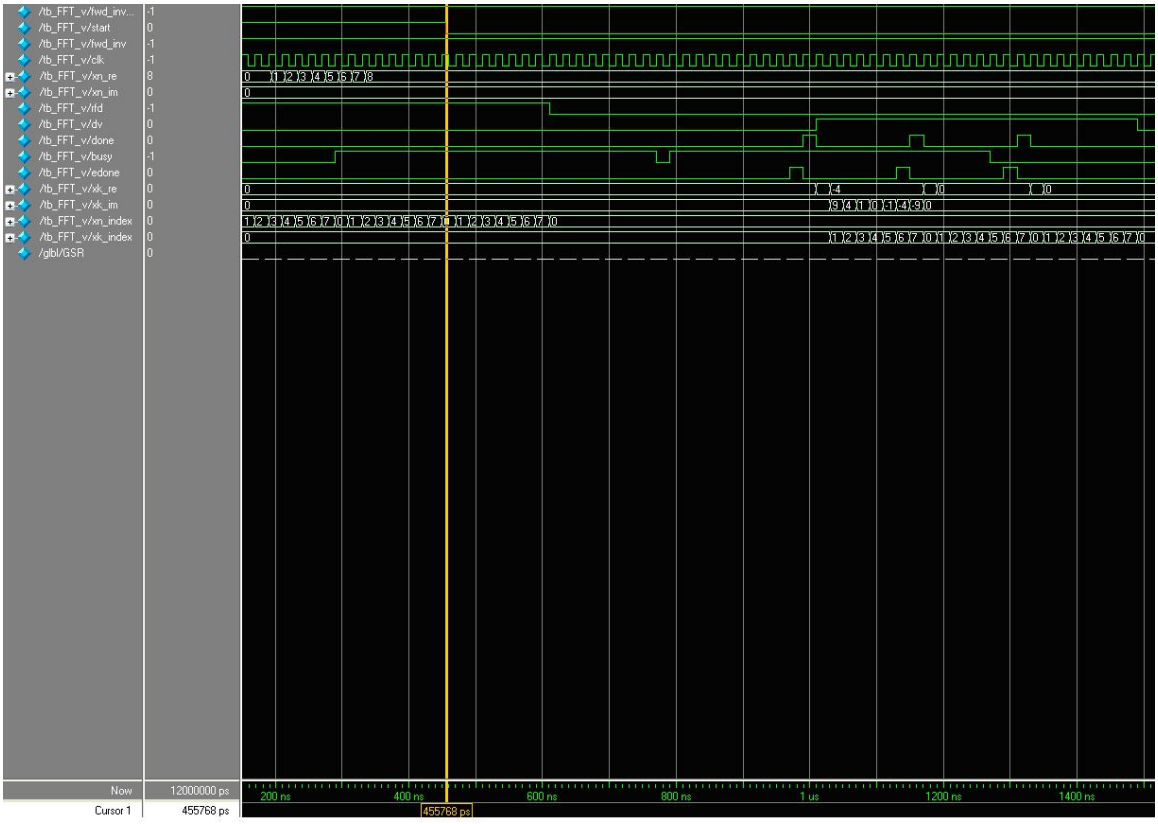


Figure 8: FFT (CoreGen) Test Bench

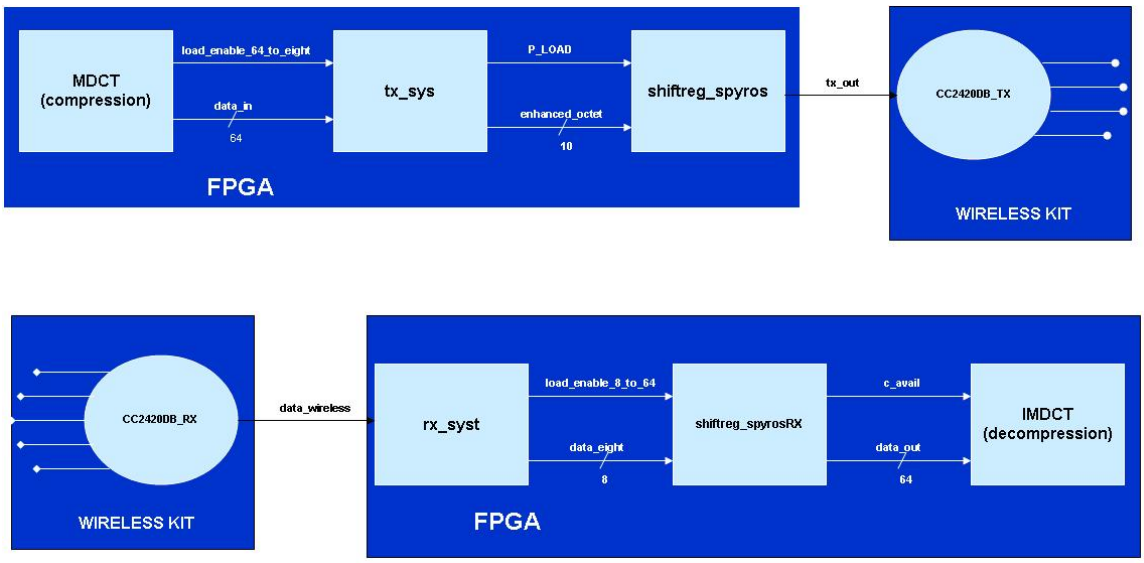


Figure 9: Block diagram for transmission and reception

Module Description/Implementation - Wireless

The modules that support the wireless processing of the data can be divided into modules on the transmitting end and modules on the receiving end (depending on whether they are part of the transmission or reception of the signal). As shown in Figure 9, tx-sys (written in Verilog), shiftreg-spyros (written in Verilog), and CC2420DB-TX (written in C) are the modules on the transmitting end; CC2420DB-RX (written in C), rx-syst (written in Verilog), and shiftreg-spyrosRX (written in Verillog) are the modules on the receiving end of the data flow.

As expected, the functionality of the Verilog modules (tx-sys, shiftreg-spyros, rx-syst, and shiftreg-spyrosRX) is implemented on the FPGAs: more precisely, the FPGA of the transmitting end for tx-sys, shiftreg-spyros, and the FPGA of the receiving end for rx-syst, shiftreg-spyrosRX. On the other hand, the modules written in C are used to program the two CC2420DB kits: the wireless kit on the transmitting end is programmed by CC2420DB-TX and the wireless kit on the receiving end is programmed by CC2420DB-RX. In Figure 10 the top view of the CC2420 Demonsration Board Kit is shown.

The UART (part of the USART - Universal Synchronous and Asynchronous serial Receiver and Transmitter) was used for the interface between the FPGA and the CC2420DB kit on both the transmitting and the receiving end. The main features of the USART, as described in ATMEL's documentation of its Atmega128L Microcontroller (the microcontroller on the CC2420DBK's, as shown in Figure 10) are Full Duplex Operation (independent serial receive and transmit registers), Asynchronous or Synchronous Operation (asynchronous was used for this project), High Resolution Baud Rate Generator, Serial Frames with 5, 6, 7, 8, or 9 Data Bits and 1 or 2 Stop Bits (8 data bits and 1 stop bit was used), Odd or Even Parity Generation (no parity was used), etc.

The various modules of the design are hereby described.

Transmitting End

Divider

The baud rate for the UART interface of wireless transmission and reception was chosen to be 250Kbps (the baud rate of the UART interface for the wireless kits is programmable, as can be observed in the C code provided in the Appendix. The desired rate is programmed in the lines ENABLE-UART1(); INIT-UART1(UART-BAUDRATE-250K, UART-OPT-8-BITS-PER-CHAR); .) Thus, for the 27MHz clock

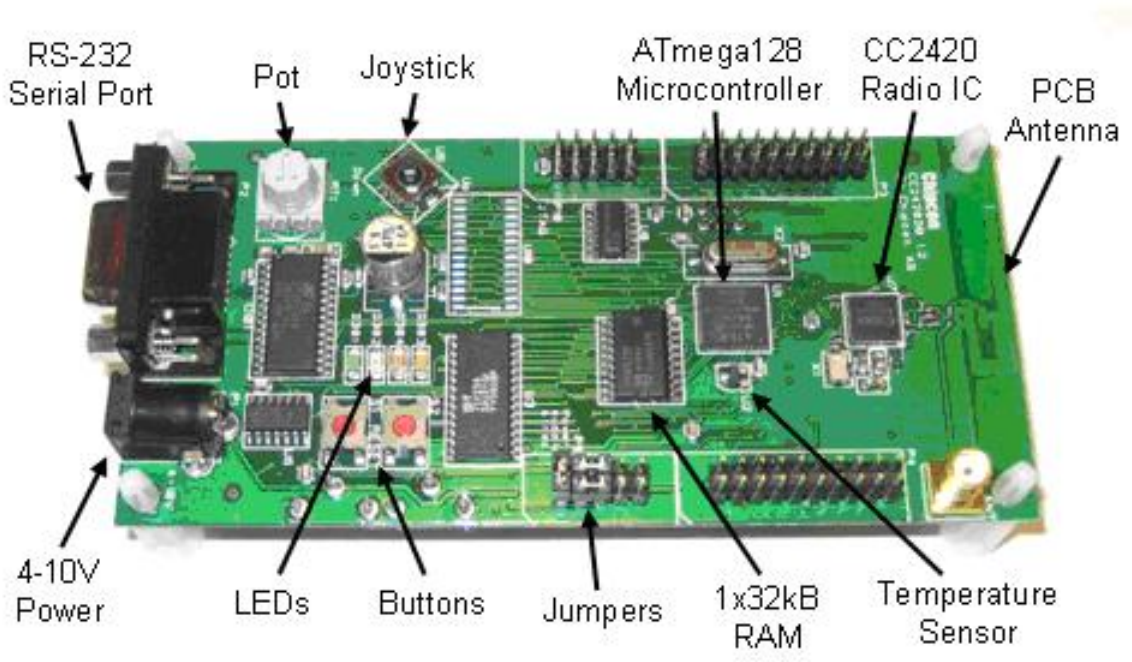


Figure 10: CC2420DBK evaluation board - top view

to support the system, every bit should be held for 108 cycles, so that (108cycles/bit) * (250Kb/second) = 27MHz. In the divider.v module, an enable signal is set high once every 108 cycles; this enable signal is what clocks the tx-sys and shiftreg-spyros modules.

Synchronizer

The reset signal pass through the synchronizer before going to other blocks as reset-sync. The purpose of the synchronizer is to ensure that the inputs are synchronized to the system clock (27MHz).

tx-sys

The module tx-sys takes as input the 64-bit data-in compressed signal (which carries the 8 bits for each of the 8 MDCT coefficients that Lohith sends in), as well as the load-enable-64-to-eight signal (and of course reset-sync). On a load-data (which is an internal control signal) request, the temporary sixty-four-bus register gets updated with the data-in. On a shift-enable request, the sixty-four-bus is shifted to the right (and the eight most significant bits are now 0's.) On a load-enable-64-to-eight request, which comes from the compression of the data (from Lohith), the load-data is set high; with load-data high, the word-count and bit-count counters are reset to 0, and the P-LOAD output to the shiftreg-spyros module becomes high. word-count counts the bytes (octets) of the 64-bit data-in, and bit-count counts the bits within each octet. These counters get incremented until they reach the values 7 and 9, accordingly. The reason why bit-count loops until 9 and not 7 is because octet is assigned to be the least significant bits of the sixty-four-bus, and then enhanced-octet is assigned to be the concatenation of a 1 (MSB), octet, and a 0 (LSB). The 0 serves as the start bit and the 1 as the stop bit of every byte for the UART interface. Within the loop of bit-count and word-count the shift-enable control signal and the P-LOAD output are updated. There is also the lohith-busy output to Lohith's compression modules, which is high while the counters are looping (while the 64 bit input is shift-registered into octets) to notify the inputting module of tx-sys that tx-sys is busy. (The lohith-busy signal can be omitted from Lohith's modules'and

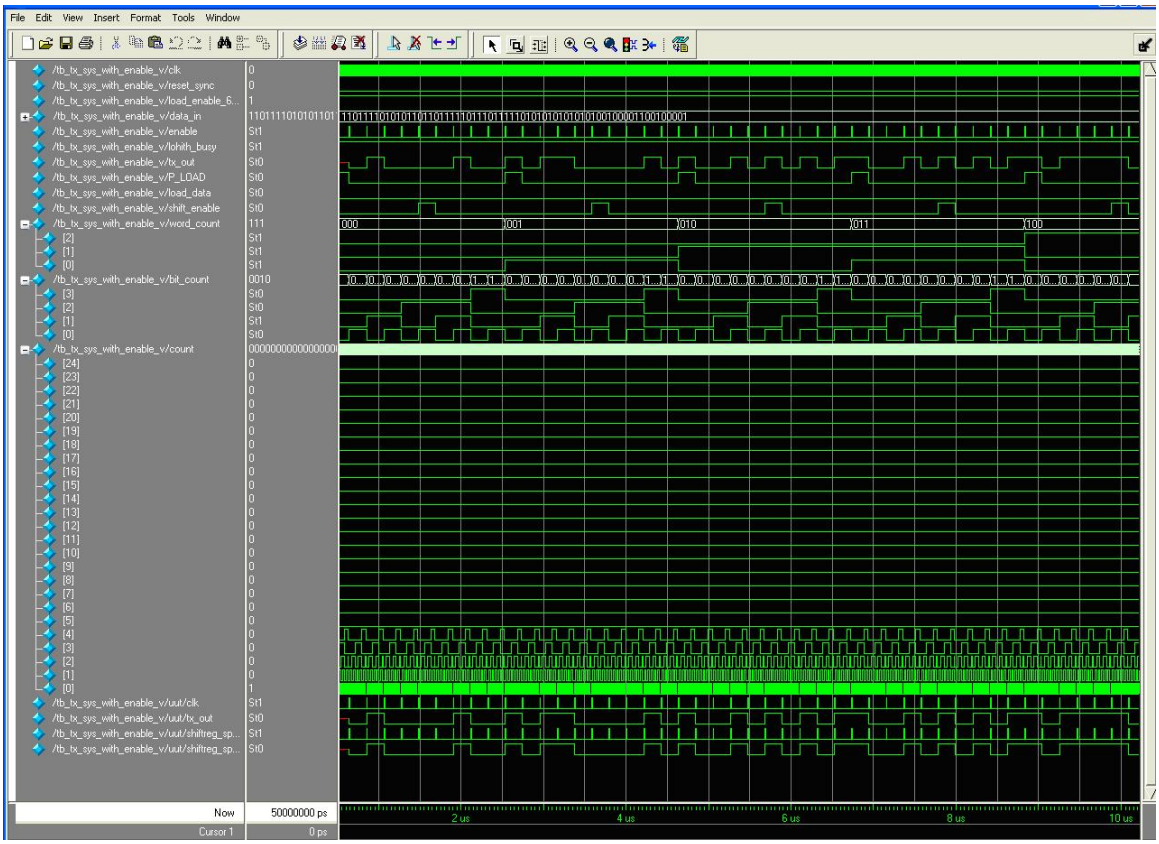


Figure 11: Screen capture of ModelSim simuation of tx-sys and shiftreg-spyros

Spyros's modules' interaction with slight modifications.) The enhanced-octet 10-bit bus (data bits with start and stop bits) drives the shiftreg-spyros module.

shiftreg-spyros

Instantiated within tx-sys, shiftreg-spyros takes in the driven enhanced-octet and P-LOAD and is essentially a shift register. On a P-LOAD request, the ten-bus internal register is updated with the new value of the input enhanced-octet. The ten-bus register is shifted each time by one to the right, and the output tx-out (an 1 bit signal) is assigned to be the LSB of ten-bus. The names D, P-LOAD and SDOUT for the inputs and outputs of shiftreg-spyros (see Appendix) were inspired by the Coregen implementation of a shift register. The shiftreg-spyros module is clocked by the clock of the tx-sys module, which is the enable output of divider.v. The sytem of tx-sys and shiftreg-spyros can alternatively be presented as an FSM, but since the functionality is analogous to the modules on the receiving end (rx-syst and shiftreg-spyrosRX), of which rx-syst is analytically described later as an FSM, a state transition diagram is not included here. Nevertheless, an alternative Verilog implementation of tx-sys, which is clearly written as an FSM, is included in the Appendix (tx-shifter.v).

The system of tx-sys and shiftreg-spyros takes in the 64-bit data-in and serially outputs one bit at a time (overall 80 bits: 64 data bits, 8 start bits, 8 stop bits) on tx-out. In Figure 11 a ModelSim simulation of tx-sys and shiftreg-spyros is shown, for which the data-in was hardcoded to be 64'hDE-AD-BE-EF-AA-AA-43-21 (see tb-tx-sys-with-enable.v in the Appendix).

The 7th signal fom the top is tx-out. The first value tx-out gets is 0 (start bit). The right next is 1, followed by four 0's, a 1, two 0's, and then a stop bit (1). So the data bits on tx-out are 10000100. The last byte in the hardcoded data-in was "21", which in binary is 0010-0001. The shift register starts

reading from the LSB, which is why tx-out is 0010-0001 inversed. Here is the sequence of the last 16 bits (LSB) in the hardcoded data-in = 64'hDE-AD-BE-EF-AA-AA-43-21.

$$0100001100100001$$

Adding start (LSB) and stop (MSB) bits:

$$1 - 01000011 - 01 - 00100001 - 0$$

which is exactly what the screen capture reads. In the waveform, the functionality of the enable signal (5th from top) is shown: every bit is held for as many cycles as wanted (108 for the project application, 20 for this ModelSim simulation).

CC2420DB-TX

This packet sender module was written in C. In it, the UART interface is initialized at a 250Kbps rate. In it the first slot in the buffer on the transmitting end, pTxBuffer[0], is set to have the value in the UDR1 register for the first transmitted byte. For the rest of the bytes in the packet, pTxBuffer[n], where n is the index of the respective byte, is set to UDR1. The packet is then sent and the transmitter waits for an acknowledgement from the receiver. If an ACK is not received, the sender keeps resending the packet. The yellow LED is programmed to stay on while ACK for a sent packet has not yet been received. The C code for the packet sender (called here 2420DB-TX) is included in the Appendix.

Receiving End

The divider and synchrnonizer modules have exactly the same functionality as described for the transmitting end.

CC2420DB-RX

This module receives the packets that are wirelessly transmitted into the pRxBuffer (the buffer on the receiving kit) and fills its UDR1 register with the contents of the buffer. This is repeated for all 8 bytes of a packet. Notice that pTxBuffer and pRxBuffer are the pPayload fileds of rfTxInfo (an instance of the BASIC-RF-TX-INFO structure) and rfRxInfo (an instance of the BASIC-RF-RX-INFO structure) respectively.

rx-syst

rx-syst takes in the data-wireless ouput of the receiving wireless kit (the CC2420DB-RX module) and ouputs a load-enable-8-to-64 control signal and a data-eight information packet to shiftreg-spyrosRX. rx-syst is essentially an FSM that takes in the serial output of the receiving wireless kit, registers it in "enhanced" octets of 10 bits (an enhanced octet is an octet with start and stop bits added) which are the data-eight-enhanced registers, and finally clears out the low start bit and the high stop bit off data-eight-enhanced to ouput a data-eight of width 8. The transition from data-eight-enhanced to data-eight can only be done on a high load-enable-8-to-64 (see sequential always block of rx-syst.v in Appendix). This modules samples the data-wireless input signal, by first checking for a start bit at each positive edge of the clock. rx-syst repeatedly counts to a value of 107 (every bit need to be held for 108 cycles) until 8 data bits are detected.

Thus, rx-syst is an FSM with states STOP-BIT, START-BIT, BIT-0, BIT-1, BIT-2, BIT-3, BIT-4, BIT-5, BIT-6, BIT-7, and IDLE (11 states). On a reset-sync, the current state becomes the STOP-BIT state, in which if data-wireless is low, the MSB of data-eight-enhanced is set to high and the FSM transitions into the START-BIT state.

From the START-BIT state though BIT-7 state, if check-enable is high, the data-eight-enhanced bit at the next index gets the value of the data-wireless input. The next state for START-BIT is BIT-O, for

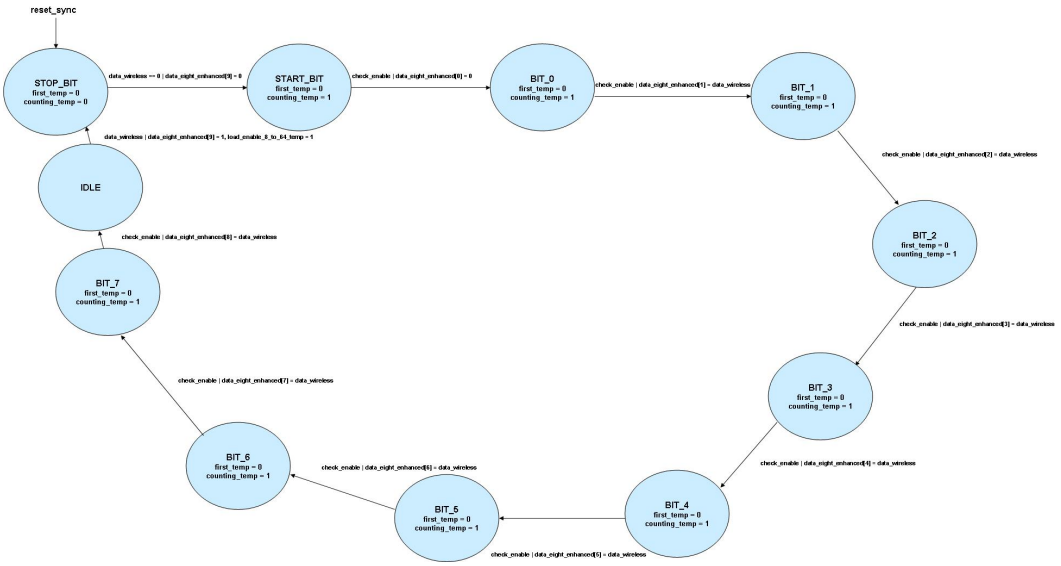


Figure 12: State transitioning diagram for rx-syst FSM

BIT-i it is BIT-i+1 (i ¡ 7) . At the BIT-7 state, if check-enable is high, data-eight-enhanced[9] becomes high and the FSM transitions into the STOP-BIT state. The functionality is best visualized in the state transition diagram of Figure 12. Transitions that are self-cycles are not shown on the diagram. The detailed Verilog code for the rx-syst FSM is included the Appendix (rx-syst.v).

shiftreg-spyrosRX

This modules takes as inputs the load-enable-8-to-64 signal, as well as the data-eight packet from rx-syst. It is a shift register that takes in an octet of eight bits and outputs data-out, the recovered 64 bits that were originally sent by Lohith's compression module (on data-in) on data-out. Along with data-out, which drives the decompression module, shiftreg-spyrosRX also outputs a c-avail signal to the decomression module, which is high when the new set of 64 bits (encoding 8 coefficients of MDCT) is sent for decompression.

The system of rx-syst and shiftreg-spyrosRX was tested with tb-shiftreg-syrosRX-v, which is included in the Appendix. In the testbench, the data-wireless input is hardcoded to start with a low (start bit), then alternate between highs and lows and finally have a high(stop bit). In Figure 13, the waveform for data-out, the 64-bit output, is noteworthy.

The data-out signal (the 7th simulated signal from bottom) gets shifted to the right (with a new octet coming in in the most significant bits positions as data-eight) on a high load-enable-8-to-64.

Testing/Debugging - Wireless

The systems of tx-sys, shiftreg-spyros (transmission) and rx-syst, shiftreg-spyrosRX (reception) were tested with tb-tx-sys-with-enable-v and tb-shiftreg-spyrosRX-v respectively, as discussed previously.

Specifically for the generation of the tb-tx-sys-with-enable-v testbench, an alternative, different form the original, version of tx-sys was created, in which internal control signals such as load-data, shift-enable, word-count and bit-count were also outputted and processed by the Logic Analyzer. Originally a discrepancy was observed between the tx-out (tested with hardcoded data-in as explained above) in the ModelSim and the tx-out in the Logic Analyzer signal analysis, which was resolved after the control signals were also outputted and processed by the Logic Analyzer.

The most interesting part of testing and debugging was done for CC2420DB-TX and CC2420DB-RX (C code written to program the microcontroller of the wireless kits). Although the code for the

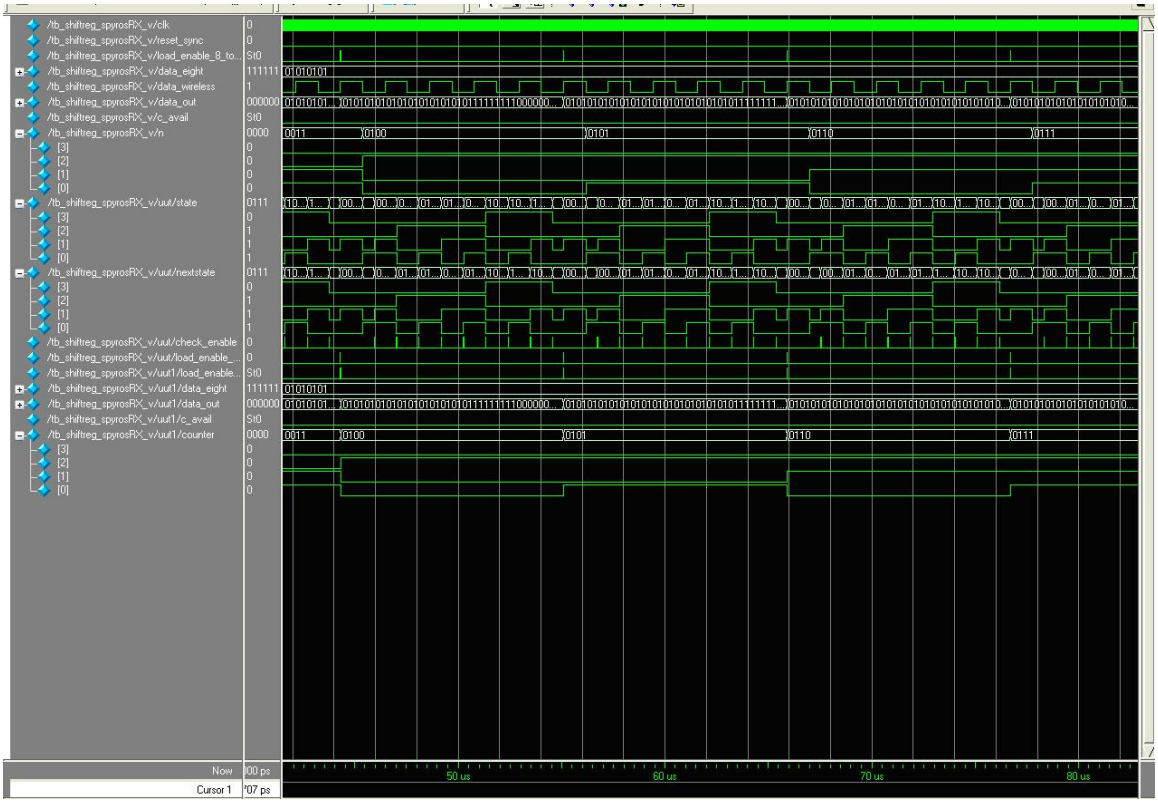


Figure 13: Screen capture of ModelSim simuation of rx-syst and shiftreg-spyrosRX

packet sender and the packet receiver is written on the same file (rf-blink-led.c), the difference between the transmitter and the receiver is established by pressing the joystick on the kit down to make the kit a sender, or in any other direction to make the kit a receiver. Debugging was done in levels; for example, if wireless-debug is 1, the main while loop in the main method executes the test for the wireless link proposed in the CC2420DBK tutorial . This demo is for verifying that wireless transmission and reception actually works.

For the packet sender (CC2420DB-TX), the code was written in the main method of rf-blink-led.c (included in the Appendix). The next step taken for debugging was checking whether the RXC1 flag of the UCSR1A register is set (which means that the receive buffer - of the transmitting kit - has unread data). While the receive buffer does not have any unread data, the green LED stays on. When the receive buffer gets data, or the joystick is pressed down, the while loop (see code for rf-blink-led.c) is exited and the buffer is actually read. If the hex value for an "a" ASCII character (0x61) is read, then the green LED blinks, else the red LED blinks. The packet is then sent over the air and if an ACK is not received, the sender retransmits the data, while the yellow LED is on. Else, if an ACK is indeed received, the yellow LED goes off.

A problem that came up was that when sending the packet, the data first needs to be received in the UART interface. (Note that paradoxically UART-RECEIVE() is used for the transmitter, UART-SEND() is used for the receiver.) Receiving in UART means reading the UDR1 register and setting the argument of UART-RECEIVE() equal to it. Nevertheless, the UDR1 register was read already while testing whether the incoming byte was an "a" or not, so the second time it was accessed it would yield the previous value loaded in it. To resolve this bug, a temporary BYTE variable, temp, was used to hold the value of UDR1 and access it, and then temp was used both for testing whether the incoming byte was an "a" or not and for sending the packet off. This way, the receiver can actually read the data that was sent in the current frame, rather than the data sent in the previous frame, as was the case before the modification.

For the packet receiver (CC2420DB-RX), every bit of the receive buffer (pRxBuffer[0]) was checked to see whether it was a 1 or a 0. The green LED lit on in case of a 1, the red lit on in case of a 0, and the yellow LED blinked between reading two successive bits (to give the eye time to see the LEDs blinking and decode the received bit sequence). This way, by seeing the LEDs on the receiving kit, the observer can decide what message is being received. The basicRfReceivePacket() is part of the basic RF library, yet it must be declared by the application.

The next step was to connect both the transmitting and the receiving kit (through the RS232 serial port) to the COM port of two terminals, load data to the transmitter from the HyperTerminal, and then verify that the same data is received on the receiving end (again using HyperTerminal). The connection used for this simulation was a 9600bps connection (so the baud rate of the wireless kits needed to be reprogrammed at 9,6Kbps). Again, testing and debugging was done in stages: first, integrity in loading the data onto the transmitter kit was verified (if an "a" is written in the HyperTerminal, the green LED should blink as explained above, if some other key is pressed, then the red LED should blink). Second, correct transmission and reception was verified using the LEDs on the receiving wireless kit (e.g., if an "a" is sent, then knowing that "a" in ASCII is 0x61, or 0b0110-0001, the LEDs on the receiving kit should blink as follows: red-green-red-green). Third, the sent data should be displayed right on the HyperTerminal of the receiving end. As mentioned earlier, before the addition of the temp internal BYTE variable in the main method, the receiving kit could receive the right data (the LEDs on the receiving end blinked as they were supposed to), but the data in the previous frame was displayed on the HyperTerminal screen on the receiving end. This was because the UDR1 register was read twice. After assigning temp to be UDR1, UDR1 was actually accessed only once, thus the bug was fixed and the HyperTerminal on the receiving end showed the ASCII character that was inputted in the HyperTerminal on the transmitting end. Lastly, the code was tested and debugged for wirelessly sending and receiving multiple-byte packets (rather than single-byte ASCII characters), specifically 8-byte packets. The for loops were used for this in the code for CC-2420DB-RX (basicRfReceivePacket()) and CC2420DB-TX (main).

Conclusion

The aim of our project was to implement a wireless audio effects processor. By the projects conclusion, we were indeed able to successfully implement many of the goals that we set out to achieve.

There were many important things that we learned from this lab. First, careful forethought and planning is extremely important. This goes back to the concept of modular design. Determining all requirements as per specifications, and then breaking down the system into several interconnected but self contained blocks allows for rapid and straightforward implementation, and also greatly help in debugging and testing. Secondly, thorough simulation and testing of each component independently of others is extremely important as this greatly hastens the debugging process and increases the overall reliability of the system.

If we were to do this project again, we would try to first analyse the timing characteristics between each module, and thoroughly understand the relation and interaction between each module (especially the FSMs). Also, we would try to be more meticulous while writing the code, and comment it more effectively in order to minimise errors caused simply due to overlooking a particular aspect of design (such as unconnected wires etc.).

Finally, we would like to have spent more time integrating the system together. Although each of the individual parts worked effectively, due to scarcity of time we were unable to integrate these aspects of the project to achieve the data flow that we had proposed.

6.111 in general has been an extremely enriching experience for each one of us on the team. Concepts that once seemed so complex are second nature now, and we hope that we can take what we have learned through this class outside the academic environment in which it was taught. Time management, attention to detail, and effective teamwork are just some the things that will most definitely stand us in good stead in the future.

Acknowledgements

We would like to extend our thanks to Professors Chandrakasan and Akhinwande, and Gim Hom for their unceasing support through every aspect of our project, and throughout the term in general. Also, we would like to profusely thank our extremely helpful TAs, Javier Castro, David Wentzloff and Amir Hirsch for all their help throughout this semester. Finally, we would also like to thank our LA, Alex Valys for his valuable help debugging the system.