A Time Interleaved Analog to Digital Converter and Digital Filter

Final Project Report
December 9, 1999
E157

Emily Hill and John Ward

Abstract:

Some signal processing systems such as high-speed digital oscilloscopes or sensor systems require broadband analog to digital conversion to allow digital processing. Depending on the bandwidth and required resolution, there may not be an ADC that could perform the conversion by itself. Multiple ADCs can be time-interleaved to achieve the desired resolution. Our project was to design a two-channel time-interleaved ADC with a 16-tap FIR filter on the FPGA to perform digital filtering. The output would be transferred to a PC running LabView via a digital data acquisition card. Several attempts to create a prototype were made. New techniques and considerations in creating the system were discovered, however, due to time constraints and board manufacturing delays, we were unable to produce a working system.

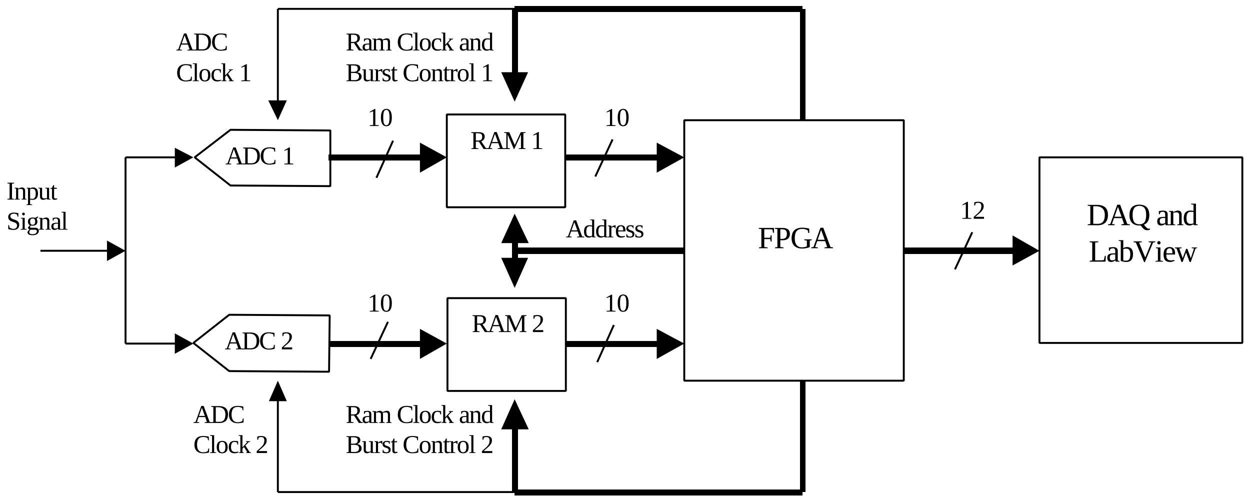
Introduction

For engineering clinic, the Aerojet clinic team is building a system with a 400 MHz bandwidth input signal that is to be digitized in its entirety and then filtered into 9 specific bands grouped into 5 channels with varying bandwidths. The calculated resolution needed to meet Aerojet's requirements is 10 bits. With our bandwidth, the Nyquist rate is 800 MSPS. Current ADC technology can only produce a 200 MSPS 10 bit ADC. Given that there will be some error introduced in the process, the Aerojet team will go with 12 bit ADCs, where the maximum sampling rate found for a 12 bit ADC has only been 100 MSPS.

A solution to the digitizing problem is to time interleave several ADCs. For example, given a set of ADCs that ran at 100 MSPS and a system to sample at 400 MSPS, we would have one sample at time t, then the next at 2.5 ns after the first, followed by a third at 5 ns after the first, the fourth at 7.5 ns, and at 10 ns the first would be ready again. If the system does not need to or is to slow to perform the signal processing in real-time, the samples can be stored in a RAM dedicated to each ADC. When the sampling stage is finished, the samples can then be read into a DSP or FPGA to be digitally filtered.

As an preliminary step to building a full scale system for Aerojet, our goal was to build a two channel system. This would aid us in understanding how the various parts of the system interface with each other, learn software packages that we will need to use such as Accel and LabView, and to understand some of the problems involved with creating the physical system.

Our system was initially to be two 10 bit ADCs running at 50 MHz, producing an equivalent sampling rate of 100 MSPS. Each ADC would send data to a Burst SRAM module which would send the samples to the FPGA when the sampling phase was finished. The FPGA would filter the samples at a rate slow enough to be transferred to the PC and would provide the control signals and desired clock signals to the ADCs and RAMs.



Initial Design

PC Board Design

To create a system sampling at 50 MHz with two time interleaved converters, a clock frequency in excess of 25 MHz would be necessary to run the analog to digital converters at this speed. Systems with such frequencies tend to be fairly noisy if not designed carefully. One item of care is the creation of a printed circuit board with power and ground planes. By having power and ground planes rather than simply power and ground nets, cross talk is drastically reduced.

The board we had hoped to use is a 4-layer design. Layers one and four are signal layers, and the middle two are power and ground layers. This way all signals are next to a small signal ground, minimizing capacitive noise. Since part of the intention of this board was to understand the effect of noise at this speed, many precautions such as splitting the ground plane into analog and digital were not attempted. However, the layout was designed such that the analog circuitry was all close to the power supply, and the digital was farther. The quick switches of digital logic create a great deal of noise. Such noise is propagated on the power and ground planes. Since our system is only analog on the front end, creating a layout that traces the signal across the board in one direction was fairly simple. Through our layout, we hope to have created enough of a separation of analog and power from digital circuitry to still have a functioning system.

Accel was used to create both the schematics and layout of our board, generating the Gerber files following the procedures outlined by the Sun Microsystems Clinic 1998-99 final report. Several issues were later discovered resulting in holds by the manufacturer. The first of these was the subject of the drill hole pattern. This pattern is created by a separate function within Accel and is saved by default to a slightly different location after generation. Secondly, a board outline, simply a line and a stay-out generated around the outer edge on the board in the layout needed to be placed on the mask layers of the board, so that the edges of the board can be defined.

Time really seemed to be our biggest problem. Due to the time constraints of the project, and the delays from holds, the boards did not arrive until December 8, too late for use. Additionally, while the schedule was moved from a 5 to a 4 day return, movement beyond that to a shorter cycle would have increased the price over the clinic credit card limit, requiring a purchase order and increasing the delay.

FPGA

The FPGA was to perform three functions: generate the clock and control signals to the ADCs and RAM buffers, digitally filter the signal using a 16 tap FIR filter, and send the output data to the DAQ. The initial design used one Spartan FPGA, the same as used in E157, utilizing all but one of its available I/O pins.

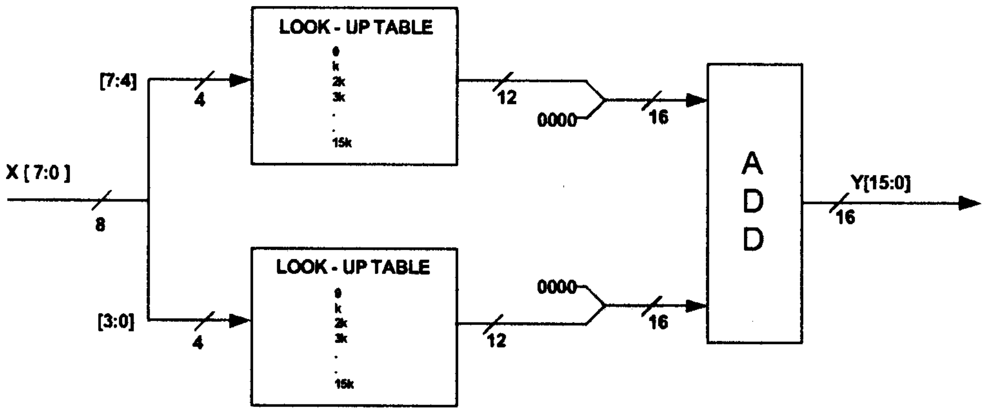
Generating accurate clocks using the FPGA proved to be the most difficult part of the FPGA design. Since we were designing for a two-channel system, the ADCs needed to be 180 degrees out of phase with each other. This can be done easily by inverting the input clock and then sending the clock and inverted clock to the ADCs. The output from the ADC is valid during a certain time period and the input to the RAM needs to meet a setup and hold time for the RAM to store the correct value. From the timing diagrams and data sheets, we determined that if the ADC and RAM in one channel were run by a single clock, the ADC data may not be valid when the SRAM would be reading it in. To meet the specified desired times, the ADC clock needed to lag the RAM clock so that the ADC output data was valid during the RAM's setup and hold times. A final consideration that we had to take into account was that we were using single port RAM and both the ADC and FPGA were connected to that port. If the ADC were producing sampled data during the period that the FPGA was transferring data to the DAQ, we would not receive the desired output.

To satisfy these requirements, we assigned four clock signals with specific delays relative to each other and that disabled the ADCs while sending data to the DAQ. To do this, the FPGA would first generate the RAM clocks from the input clock. The RAM clocks would then be anded with an internal control signal that indicated whether the system was sampling or filtering to produce the ADC clocks. Our intention was that the gates would delay the ADC clocks long enough to meet the timing requirements. This approach produced mixed results. With the Constraints Editor in the Foundation Tools, offsets and path delays can be specified to be less than some time. When implemented, the result of the constraint may vary. For example, using a constraint of 20ns from the clock input to data on some output pin, Xilinx may create an one implementation with a time delay of 10 ns, well within the constraint, while the next implementation may have a delay of 19 ns, still meeting the constraint but not by as much. With our design, we needed to have the delays at specific times and the variability in the results using timing constraints was unacceptable. Sometimes the lag between RAM and ADC clocks would be about a nanosecond when we wanted 4 nanoseconds or the ADC clocks would not be perfectly out of phase. Although some difference is to be expected, the hundreds of picoseconds or even a couple nanoseconds we received most of the time were outside our desired bounds of less than the realizable goal of 200 picoseconds.

We had more success with variations of different output pins for the clocks and buffering the ADC clocks. The idea behind the pins was to induce about a 4 nanosecond path delay difference between the two. This proved mildly successful although the results of the combination were not always predictable. For one implementation, the ADC clocks at two pins might lag the RAMs by one nanosecond while for another implementation, using the pins right next to the ones used in the first implementation would produce a 4 nanosecond delay. Buffering the clocks internally also provided useful results, although the router might place things differently between implementations that would negate or sometimes reduce the lag between the RAM and ADC clocks despite the addition of another buffer.

The control signals to the RAM consisted of an address bus and burst control. Instead of trying to increment the 16-bit address counter every cycle while sampling, RAM has a burst feature that takes in a new address every four cycles, easing the speed requirement on the adder. The burst control signals would tell the RAM to take in an external address instead of the internally generated address to avoid data being overwritten since the RAM loops through four addresses while on burst. During the filtering stage, burst was disabled and the RAM would always take in the external address from the FPGA. It takes several clock cycles for the output value to appear after taking in a sample from the RAM and is ready for the next sample. Since we only needed a new address after many clock cycles instead of every cycle, there was no need for burst to be implemented. After the filter received data from the same address from both RAMs, the address would be incremented. By the time the filter was ready for a new sample, the data would already appear at the output of the RAM.

Since the focus of our design was on a time interleaved system and not on digital filter design, we used the Core Generator program in Xilinx to create an FIR filter. Under certain conditions, an FIR filter can be efficiently created in FPGAs. With constant coefficients, instead of creating a full multiplier, the FPGA can use a look up table with subsets of the input sample providing the address within the LUT for the result, which is much faster than a multiplier. All the results are summed to produce the result for each coefficient.



From Xilinx Application Note: "Building High Performance Filters Using KCM's"

The above figure shows an example using an 8 bit input sample with one coefficient. With a symmetric filter, the number of unique coefficients is half the total number of coefficients, so two taps can share a set of LUTs, decreasing the area at the cost of some speed.

Xilinx's Core Generator allows us to provide some of the parameters of the filter such as number of taps, coefficients, and number of input and output bits. The coefficients were produced in Matlab using the Signal Processing Toolbox and stored in a text file for Xilinx to look up when creating the filter. The filter module introduces three control signals, RDY, RFD, and ND. RDY is an output control signal that indicates when a new result has been calculated. Upon the assertion of this signal, the result is stored in output registers connected to the DAQ interface pins. The output signal RFD indicates that the filter is ready for new data. Upon receiving this, the control module will take the next data sample from one of the RAMs and store it in registers connected to the filter. The ND pin is then asserted by the control module and when the filter sees this, it will take in the new sample.

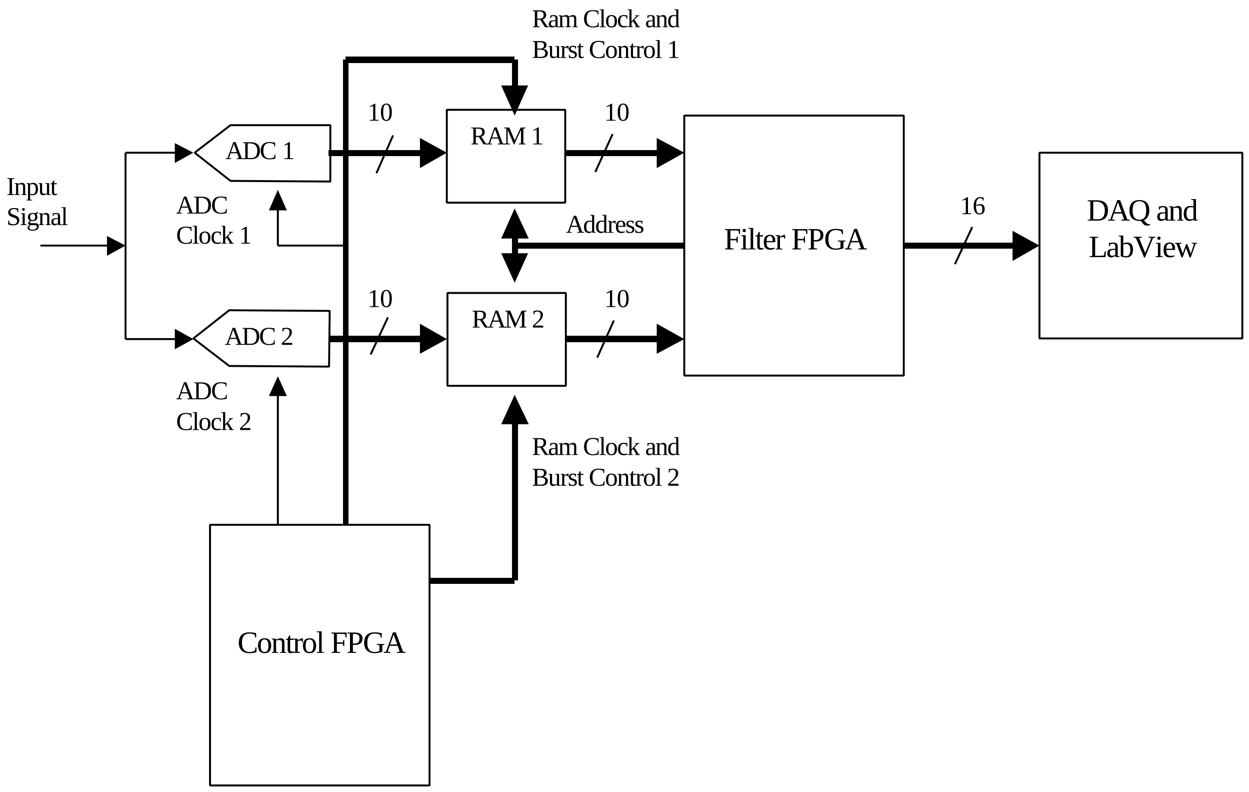
The final function of the FPGA was to oversee the transfer of data to the DAQ card. For ease of application and to reduce the number of control signals, the DAQ card was to use change detection. Whenever one of the output bits changed, the DAQ card would collect the data point. The maximum transfer rate is 2 MHz with this mode. If slower speeds were needed, the input clock to the filter could be slowed down, reducing the speed of the output. In retrospect, one problem with this mode would be if consecutive outputs were the same, then only one output data point would be taken. For future clinic use, one of the handshaking modes should be used to insure all the output points are transferred.

Perf Board

With the change to the perf board, we were given the opportunity to implement changes to the design that we discovered after we had sent the board. The only change to our design was the use of two FPGAs instead of one. Our reasons centered around flexibility and increased output resolution. With extra pins, we could try a greater variety of pin combinations that were not tested due to time constraints in designing the board. In our original design, we used 12 output bits from the filter, although more could be produced. Since the DAQ card has four 8 bit wide ports, we went with 2 ports, increasing the filter resolution to 16 bits, since we had the available pins.

One FPGA would be used for clock generation, burst control, and RAM addressing. The other FPGA was designed as the filter. It would take the samples from each of the RAMs, filter, and output to the DAQ card. A couple control signals ran between the two FPGAs like a mode signal for when the filter should take in data and when it shouldn't and an ADC_choice signal which indicated the ADC to take the next sample from. Other than output resolution and the added control signals between the two FPGAs, no additional changes were made.

The design intent behind using the perforated board was that while it would not be capable of running nearly as cleanly as the PC board would, we could still use our ADCs (in a surface mount package) and solder the components together fairly easily. This would have been true if it was not for the small packages and sheer number of pins to be connected. With the equipment available, it was difficult to solder small wires to the 28 pin- small surface mount ADCs so they could be connected. There were also the 2 SRAMs, each with a 52-pin socket, and a connection to nearly every pin. Given time, this solution would have also been viable.

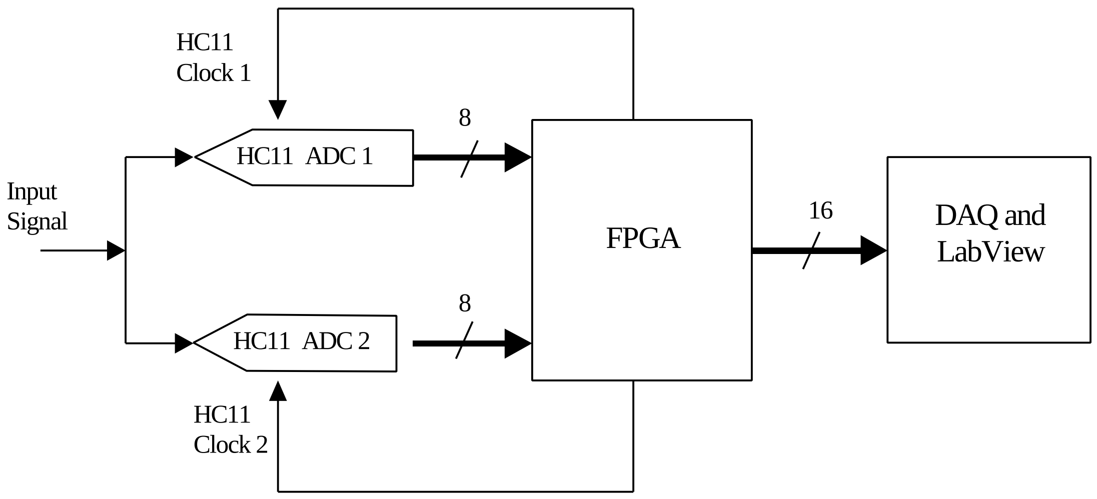


Wire Wrapping

After determining that the time needed to solder did not give a chance for us to finish the project, we chose to wire wrap our system since this alternative seemed to have a better chance of success. Since it was unlikely we would be able to successfully wire wrap the ADCs we purchased, we decided to use the most available ADCs we knew of, the onboard ADCs of the HC11. Our goal was to interleave the HC11s using the EXTAL pin and a clock generated by the FPGA. The program to run the HC11 was simple. All that had to be done was take the value in the ADC output register and write it to the port B register. This output port was then wired to the FPGA. Since we had successfully used both these components in previous E157 labs, this implementation did not seem problematic.

The sampling rate of the HC11 is 1/32nd of the input clock. With an 8 MHz clock, we were only looking at 62.5 kHz from each HC11. The filter could take in samples faster than they were being produced, so we decided to go with a real-time filtering system and not use the RAM. The FPGA could be programmed to take in a new data point at the same rate as they were being output by pulsing the new data input at 1/16th of the input clock rate and alternating the HC11 from which the data was being taken.

Since we no longer needed the control signals and address bus to the RAM, a single FPGA had enough pins to interface with the HC11s and the DAQ board at the desired resolutions.



Ribbon Cable and Breadboard

After receiving poor data results and being unable to track where the problems were due to the tangle of wires making probing with an oscilloscope difficult, we decided to use the ribbon cable attachment of the HC11 and connect it to the FPGA via a breadboard. Although this option had been avoided due to the noise incurred by the rows of wire underneath a breadboard, it was the most viable option in the time crunch created.

Since the only change to the system was the physical data bus, the HC11 and FPGA modules remained the same. Now came the part for the DAQ card and LabView to show their stuff. Although we were able to collect some data, there were several problems. Not all the data lines for each port seemed to work. They would just be stuck at one value while the other lines would be changing. Doing some quick testing, slightly more than half of all 32 lines in the card seemed to be working. A solution was developed that should have worked, but there was little time to develop tests to make sure it was correct.

The LabView interface was also lacking in some areas. Although we were hoping to use change detection with our DAQ card, we were unable to develop a solution that used it. After receiving help from Professor Spjut, an interface was developed that worked, but used a fixed sampling rate. We assumed that the user input sample rate in LabView could be set to at least the 62.5 kHz sample rate of the HC11, but it appeared to be based on an integer number of ticks of a millisecond timer in LabView. With this limitation, we were limited to a maximum sample time of 1 kHz and we did not have enough time to determine an alternate method.

The data received seemed to have dubious value. At best in the time domain, LabView would display a very noisy time input signal. The power spectral density also had limited usefulness. We were unable to set up a meaningful frequency axis and we were unsure of the sampling rate, making it hard to interpret the data. We did seem to make some progress near the end. When sending a square wave, we could see the fundamental usually standing out some from the noise. When increasing the frequency from near 0 Hz to almost 500 Hz, the peaks would move closer together and nearly touch at the halfway point, which is to be expected if performing an FFT with a 1 kHz sampling rate due to the FFT’s periodicity and with the left and right edges of the graph representing the DC component.

Results and Recommendations

Although a working prototype is not one of our deliverables, the project provided many key lessons that will be applicable in the future.

Scheduling on many different levels became a key issue that should be evaluated further. From the obvious standpoint, time was not efficiently allocated, as any of the systems should be possible to build with enough time. Designing the version most plausible within the given time should be balanced with the desire to build a preliminary design that will be complex enough to be useful to understand issues to be faced with the final product. Other time issues uncovered were time without direction, or time without a sanity check. This would include time spent before checking in with a professor or other expert. The time left for board manufacturing and de-bug was also found to be problematic. In such a short design window, we had such a low margin of error that minor mistakes would cause us to lose enough of our time that the probability of producing a working system on time was small. On a similar note, the board manufacture time from the standpoint of cost issues and the Harvey Mudd administrative procedure were brought to light. It would seem that if one plans to build something large, or plans to need something quickly, one must ask for a purchase order before the design is anywhere near completion.

Parts need to be identified and acquired as soon as possible. This will help nail down the final board design sooner instead of having to make numerous changes. The two parts that we did not nail down until the near the end were the ADC and a single-ended to differential op-amp. The problem with the ADC was that we had some samples of a 10 bit ADC and only a few days before the board design was due, it was pointed out that it used ECL logic. The ADC had a sister module using CMOS logic, so we had to find a place to purchase two. Such a search could easily have ended empty-handed. For the differential op-amp, a model was decided on based on some data sheets from Analog Devices using it in conjunction with an ADC that showed the various passive support components. Two were ordered off the Analog Devices website, but the availability was not checked. After discovering that they would not be coming in until January, and not finding any to purchase, we decided to scrap the op-amp at the cost of increased noise. While searching for the new ADC, we tried to obtain two samples from Analog Devices, and, since we were at it, we tried to sample two op-amps. At this point since we did not physically have the hardware and the board was about due, we decided it was riskier to place it on the board and not receive the op-amps than to design it with a part that never arrived. We did get the two free op-amp samples from Analog Devices despite not being able to purchase them from Analog Devices.

Using Accel and creating the artwork for a design are invaluable skills. Accel is not intuitively obvious, nor do all schematic capture programs behave the same. Having both the schematic and layout packages in one led to unique loopholes, such s the need to create a part and a pattern, and link them together prior to placement on the schematic. Sending out a board also presented some challenges. The first of these was the subject of the drill hole pattern. This pattern is created by a separate function within Accel and should accompany the Gerber files. It should be particularly noted that if one is working on the design center copy of Accel because it is saved by default to a slightly different location after generation, and must be found and included in the files sent for production. Secondly, a board outline, simply a line and a stay-out generated around the outer edge on the board in the layout (using the draw functions on the Accel toolbar) needs to be placed on the mask layers of the board, so that the edges of the board can be defined. Lastly, if any files need to be re-sent, be sure to specify that the new file or set of files is to be used as a replacement, and whether to use any of the previous set, or if the new set is a total replacement.

More time should have been spent early on for the data acquisition system to install the hardware, understand the software, and test the whole DAQ system. A DAQ system we had confidence in could have helped in debugging the project, mainly by us being sure our problems were not coming from the DAQ card or a weak LabView interface. With a short tutorial from Professor Spjut, much understanding was gained on how to get the interface set up and how LabView worked in general, but not enough time was left to build on this knowledge and create a good interface. This knowledge will help greatly next semester when we set up the DAQ interface for clinic, although we may need to replace the DAQ card if only half of the pins are in fact working. Once the basics are learned, LabView seems pretty easy to use, it’s just we did not spend enough time learning those basics. For next semester clinic, getting an interface working should be done as early as possible. Tests can be implemented using an FPGA as the signal source to determine how effective it is. While doing timing simulations for the control module of the FPGA, a module that created a sawtooth wave was placed to make sure that the correct input was being taken. This could be transferred to an FPGA and used as a test for the DAQ setup.

Although disappointed that we did not produce a working system, we did learn many valuable lessons in design and testing, which was also an important goal of our project. We will now be able to apply many of these lessons to the Aerojet Clinic project to increase the chances of the final project working by not having to make many of these discoveries next semester. Now we have a better feel for how to design a board and get it manufactured and how to set up a data acquisition system using LabView. Some other lessons learned are to lock down the parts to be used in the design so a good schematic can be developed, make sure the parts are available and are the correct type such as CMOS versus ECL logic, and allow at least an extra week for board manufacture. Timing between the ADCs and RAMs also needs to be considered more than it was by the Aerojet Clinic team before this project and the RAM used will also probably be changed to a dual port module so as not to have to find some way to disable the ADCs on the real board. As long as we apply this properly next semester to produce a good final design, this project will have been worth the effort expended.

Budget and Parts List

Part	Manufacturer Part No.	Source	Vendor Part No.
PC Board Fabrication		Advanced Circuits	
Motorola 64k x 18 Burst SRAM Module	MCM67B618AFN10	Newark Electronics	66F 7193
Analog Device 10 bit, 100 MSPS ADCs	AD9071BR	Newark Electronics	48F 1965
52 Lead PLCC Socket for SRAM		Radio Shack	99005757
Zero Ohm Jumper Wire		Digi-Key	0.0QBK-ND
Waffle Board		Radio Shack	2760147
Total Cost			

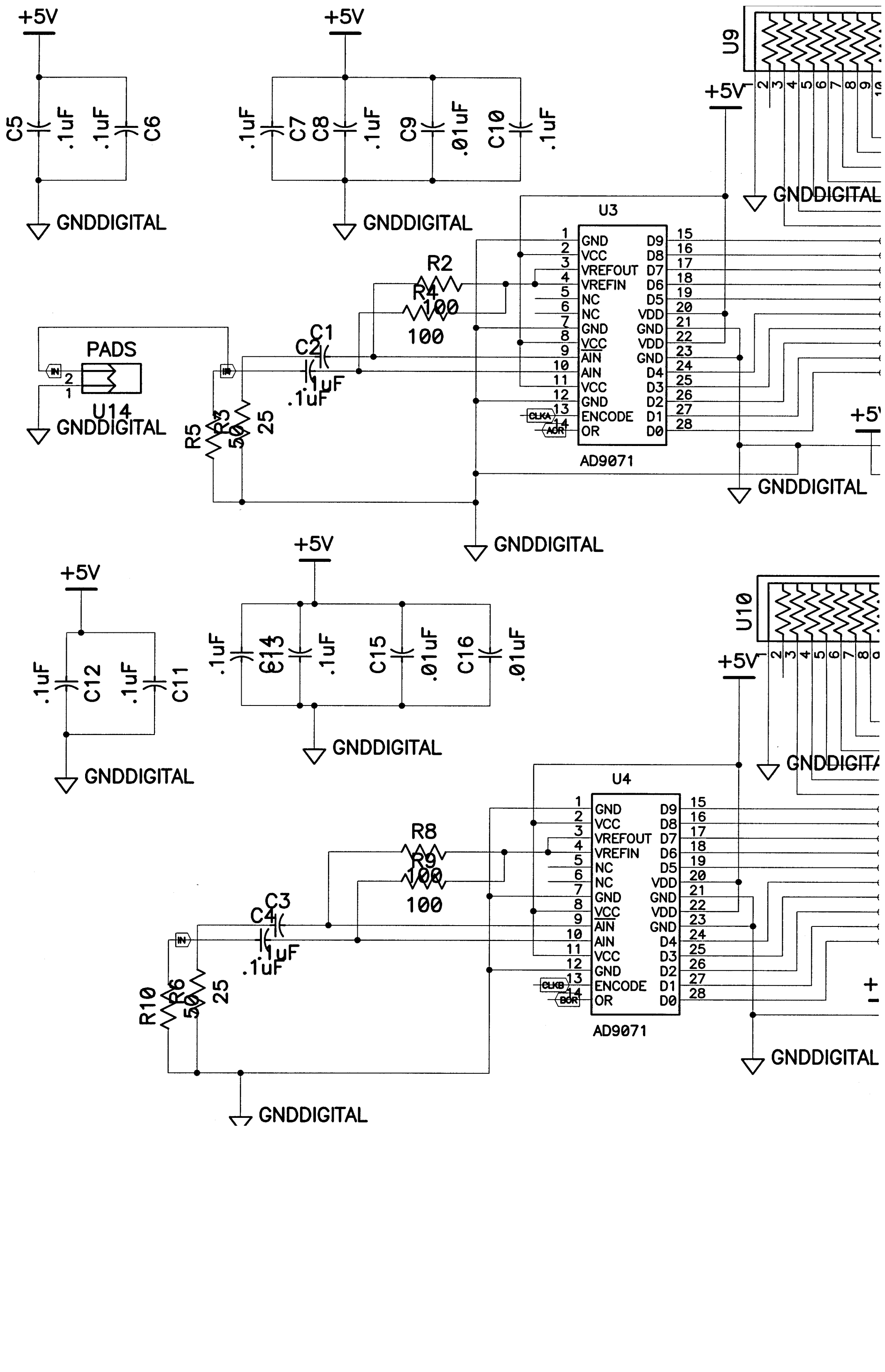
References:

[1] H. Johnson and M. Graham, *High Speed Digital Design: A Handbook of Black Magic*, Upper Saddle River, NJ: Prentice Hall,
[2] K. Chapman, *Xilinx Application Note: Building High Performance FIR Filters Using KCM's*, http://www.xilinx.com/appnotes
[3] G. Goslin and B. Newgard, *Xilinx Application Note: 16-Tap, 8-Bit FIR Filter Applications Guide*, http://www.xilinx.com/app
[4] W. Black and D. Hodges, *Time Interleaved Converter Arrays*, IEEE Journal of Solid-State Circuits, December 1980, pgs 1022
[5] *AD9071 Datasheet*, http://www.analog.com/pdf/AD9071_b.pdf
[6] *MCM67B618AFN10 Datasheet*,http://mot-sps.com/cgi-bin/get?/books/dl156/pdf/mcm67b618ar*.pdf
[7] *Ch. 4: Spartan and SpartanXL Families*, Xilinx Data Book, http://www.xilinx.com/partinfo/spartan.pdf, January 1999.

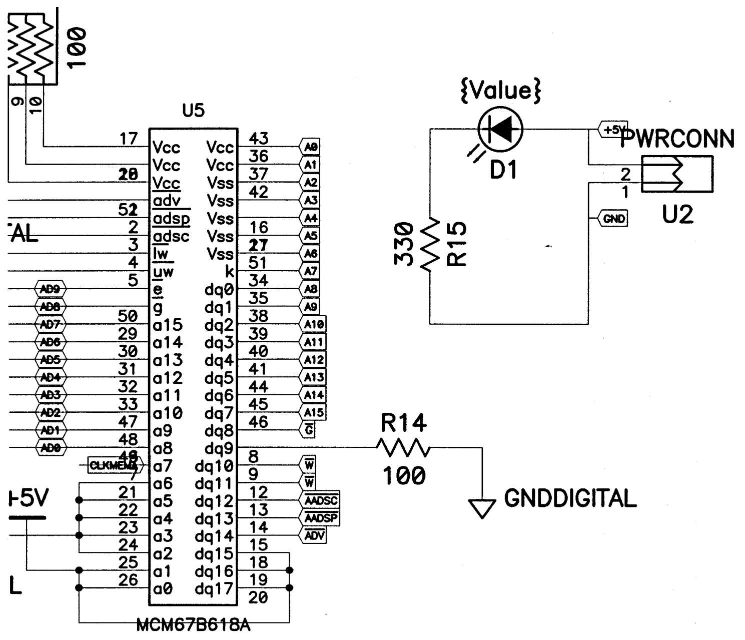
Appendices

Appendix A

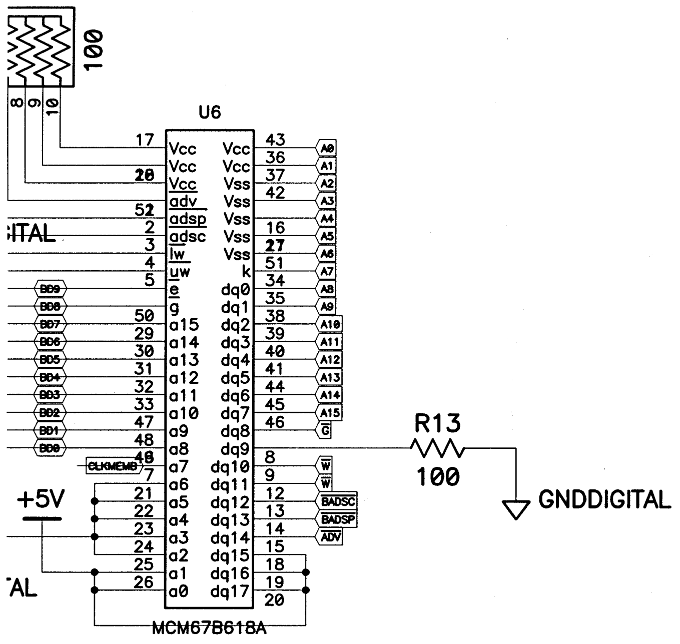
Schematics for Manufactured PCB



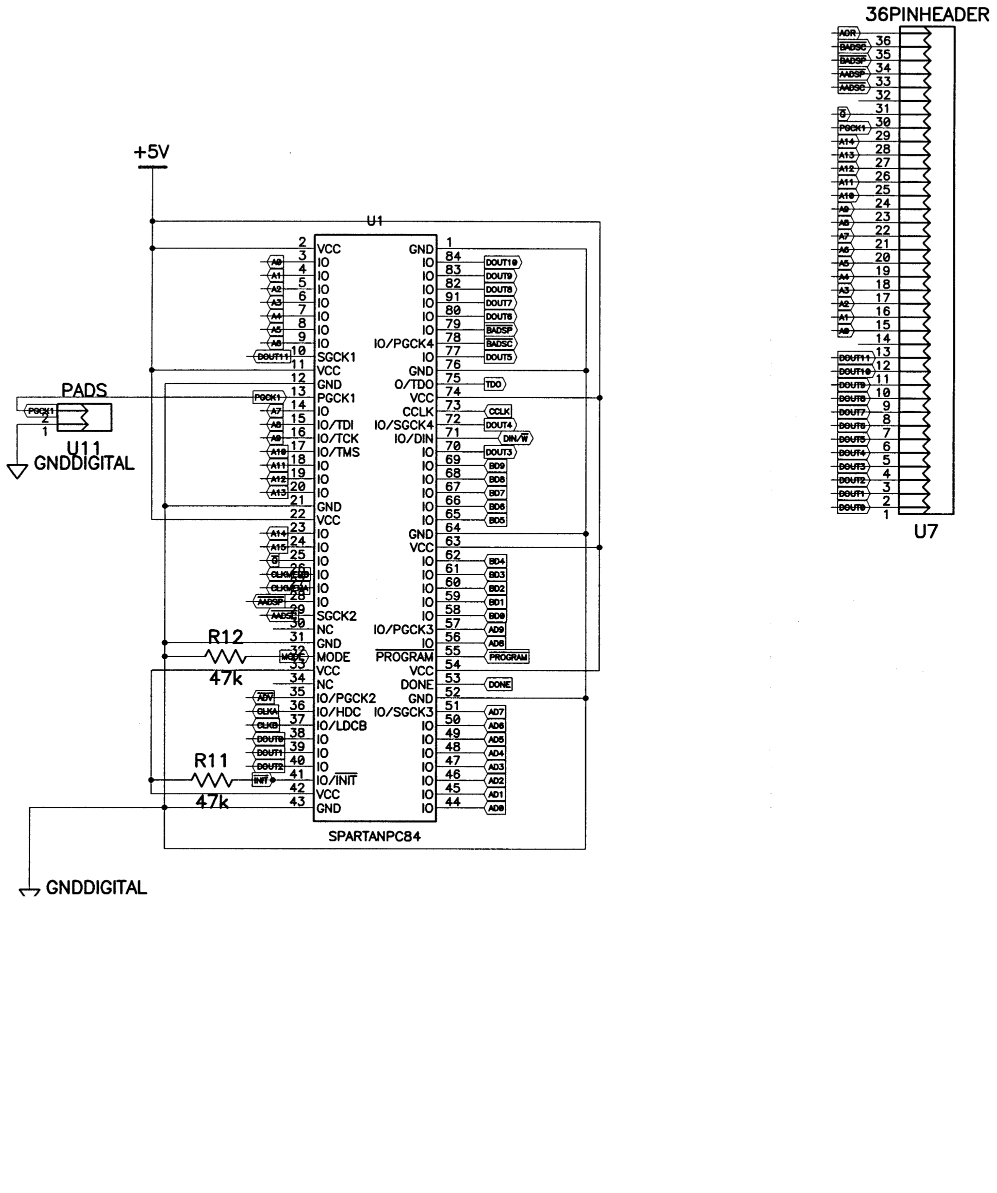
+5V
C5
.1uF
.1uF
C6
C7
C8
C9
C10
.01uF
GNDDIGITAL
U9
U3
U4
U10
AD9071
R2
R4
100
C1
C2
PADS
U14
R5
R3
50
25
C3
C4
R8
R9
R10
R6
C11
C12
C13
C14
C15
C16
GND
VCC
VREFOUT
VREFIN
NC
AIN
ENCODE
OR
VDD
D9
D8
D7
D6
D5
D4
D3
D2
D1
D0
CLKA
CLKB

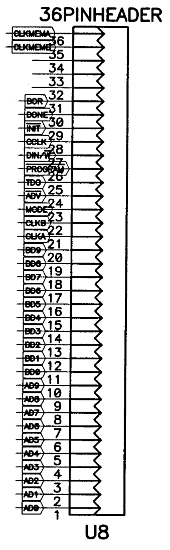


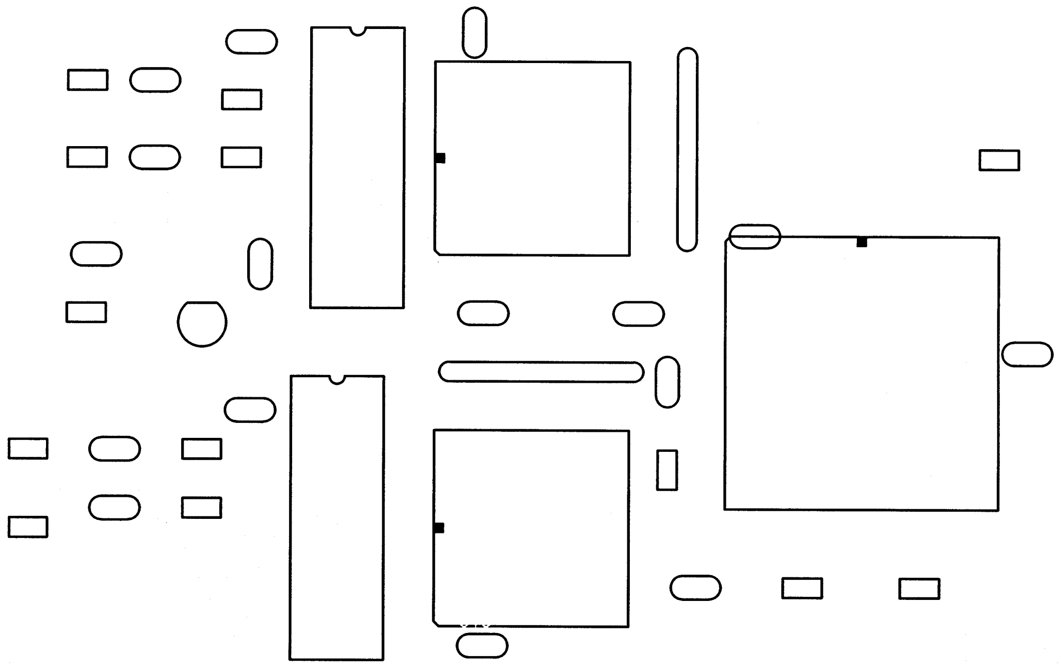
100
U5
{Value}
D1
PWRCONN
U2
330
R15
R14
100
GNDDIGITAL
+5V
MCM67B618A

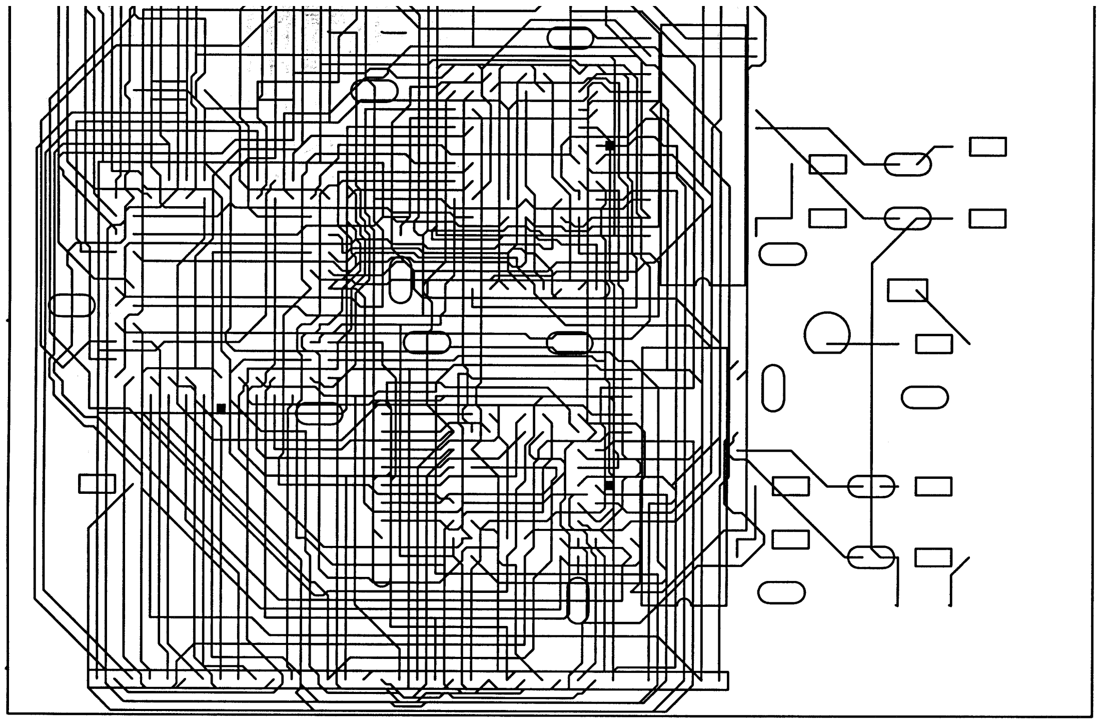


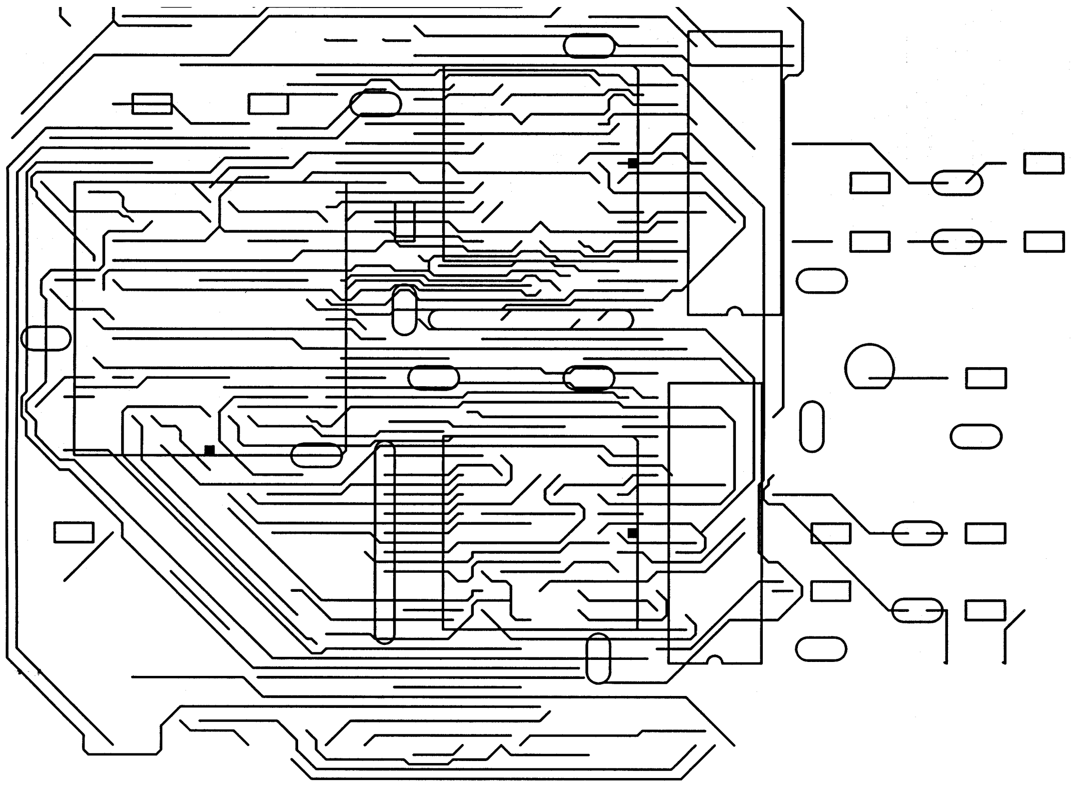
100
U6
R13
100
GNDDIGITAL
+5V
MCM67B618A

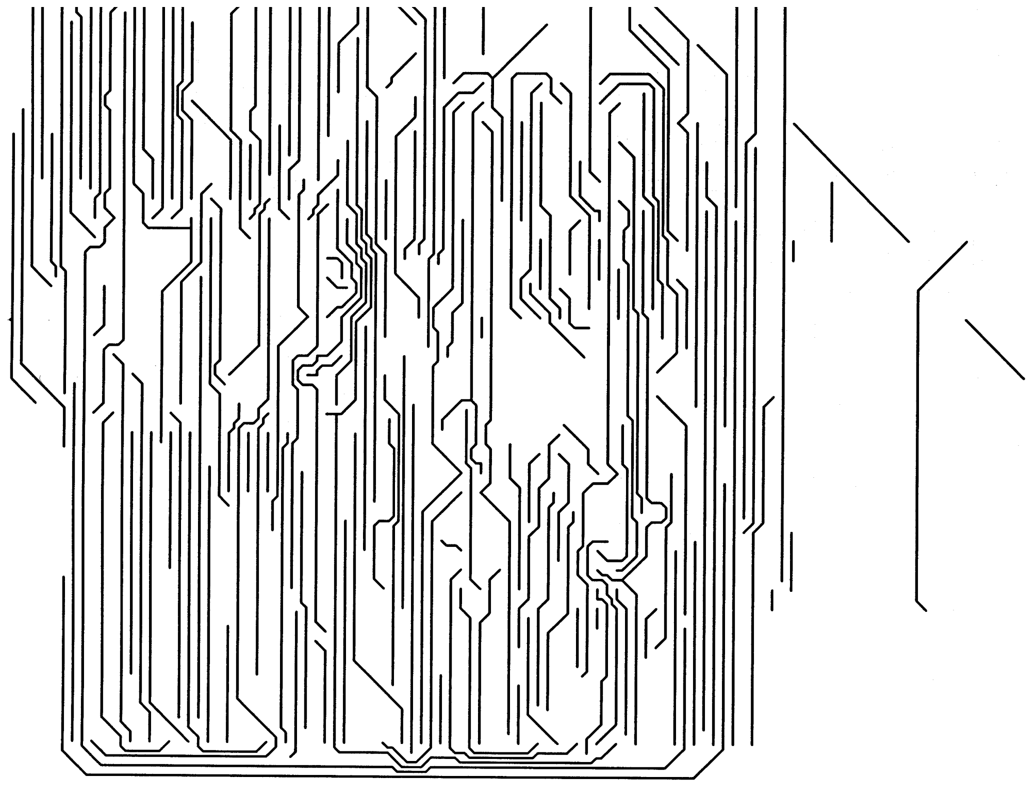

36PINHEADER
U7
+5V
U1
SPARTANPC84
PADS
U11
GNDDIGITAL
R12
47k
R11
47k
GNDDIGITAL


36PINHEADER
CLKMEMA
CLKMEMB
BOR
DONE
INIT
CCLK
DIN/W
PROGRAM
TDO
ADV
MODE
CLKB
CLKA
BD9
BD8
BD7
BD6
BD5
BD4
BD3
BD2
BD1
BD0
AD9
AD8
AD7
AD6
AD5
AD4
AD3
AD2
AD1
AD0
36
35
34
33
32
31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
U8



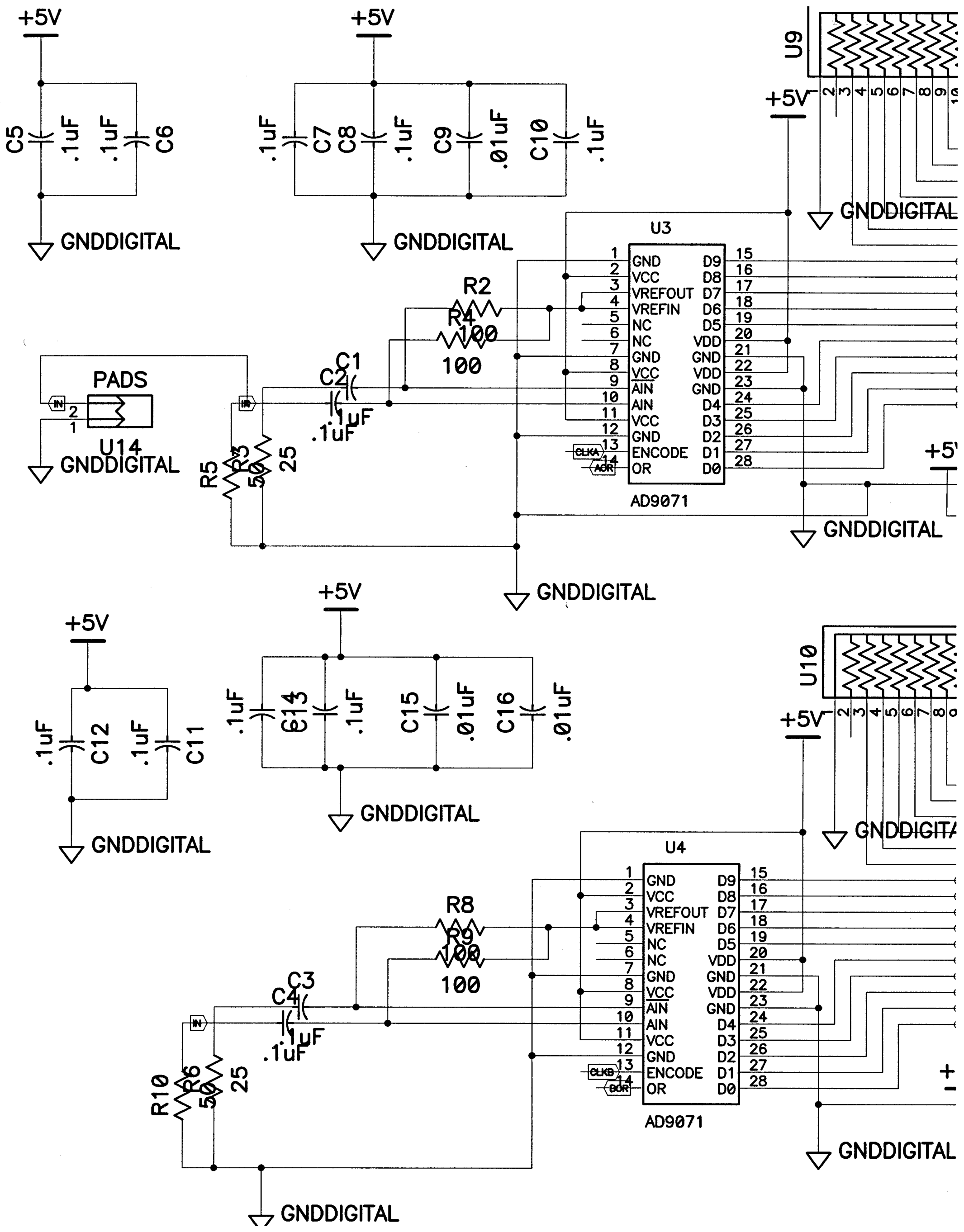


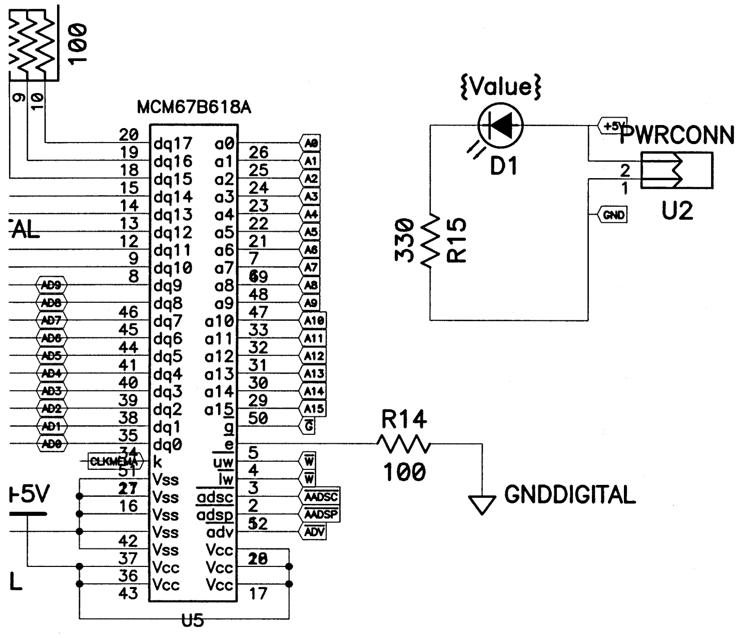




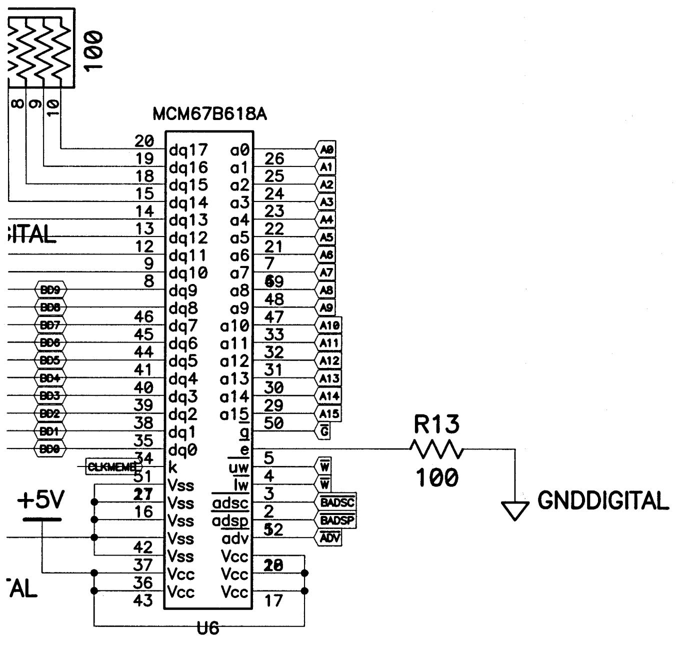
Appendix B

Corrected PCB Schematics

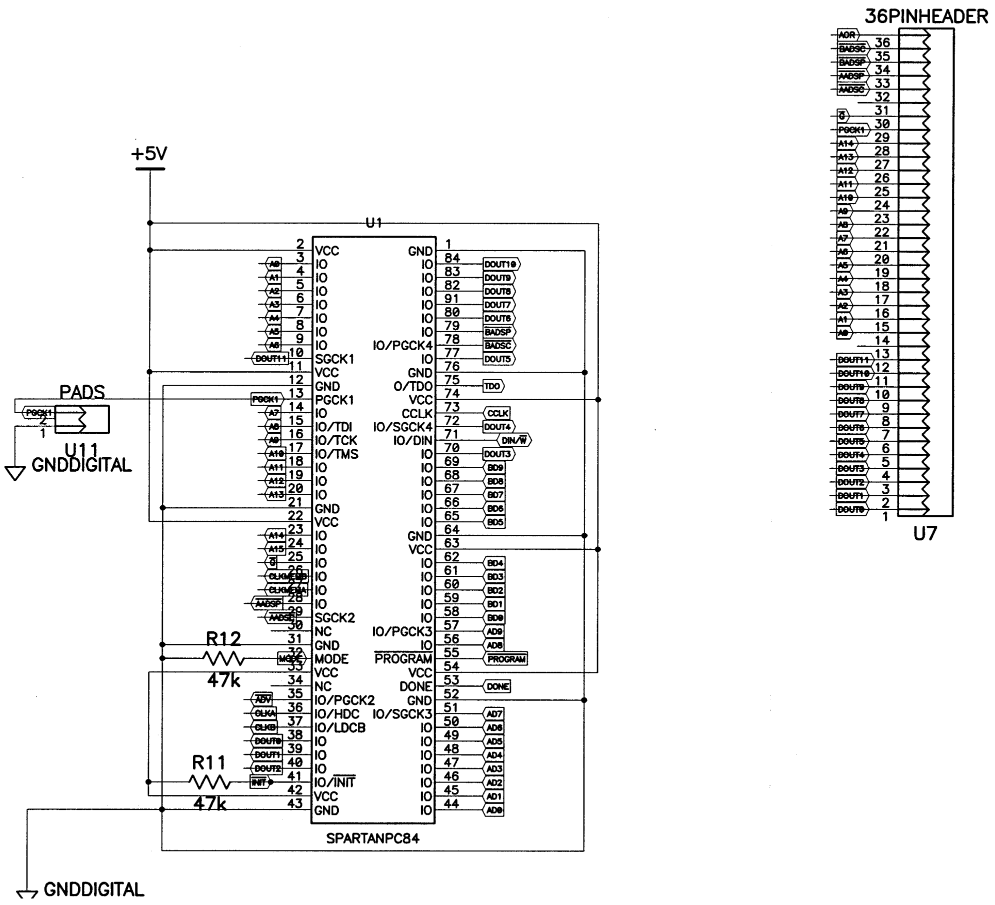

+5V
C5
.1uF
.1uF
C6
GNDDIGITAL
C7
C8
C9
C10
.01uF
U9
U3
GND
VCC
VREFOUT
VREFIN
NC
AIN
ENCODE
OR
D9
D8
D7
D6
D5
VDD
D4
D3
D2
D1
D0
AD9071
R2
R4
100
C1
C2
PADS
U14
R5
R3
50
25
CLKA
AOR
C11
C12
C13
C14
C15
C16
U10
U4
R8
R9
C3
C4
R10
R6
CLKB
BOR



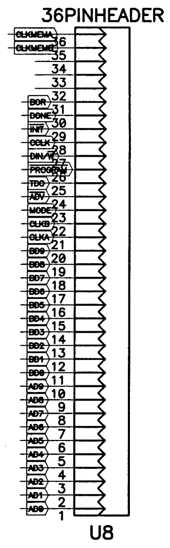
100
MCM67B618A
{Value}
D1
PWRCONN
U2
330
R15
R14
100
GNDDIGITAL
+5V
U5



100
MCM67B618A
R13
100
GNDDIGITAL
+5V
U6



36PINHEADER
U7
+5V
U1
SPARTANPC84
PADS
U11
GNDDIGITAL
R12
47k
R11
47k
GNDDIGITAL


36PINHEADER
CLKMEMA
CLKMEMB
BOR
DONE
INIT
CCLK
DIN/W
PROGRAM
TDO
ADV
MODE
CLKB
CLKA
BD9
BD8
BD7
BD6
BD5
BD4
BD3
BD2
BD1
BD0
AD9
AD8
AD7
AD6
AD5
AD4
AD3
AD2
AD1
AD0
36
35
34
33
32
31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
U8

Appendix C

Assembler Code for HC11

```
                * ADC Data Collector and Parallel Transfer to FPGA

                * Our goal is to use the 8 bit ADC built into the HC11
                * to digitize an analog input signal and transfer the data
                * over port B to the FPGA to be filtered

                * HC11 Registers
1004            portb  equ   $1004 Port B address
1030            adctl  equ   $1030 ADC Control Register
1039            option equ   $1039 ADC Options register
1031            adr1   equ   $1031 ADC Result Register
0000            zero   equ   $0000

                * Masks
0080            bit7   equ   %10000000   Mask for ADC Options register
0027            adc    equ   %00100111   Continuous scan, channel 8 for ADC

d100             org   $d100

                * MAIN
                * Initialize control registers
```

```
0022

0023 d100 86 80               ldaa  #bit7       Store options for ADC

0024 d102 b7 10 39            staa  option

0025

0026 d105 c6 27               ldab  #adc        Store setting for ADC
into ADACTL

0027 d107 f7 10 30            stab  adctl

0028

0029                         * Repeat copying ADC values to output port

0030 d10a fc 10 31           here:  ldad  adr1

0031 d10d fd 10 04            stad  portb

0032 d110 b6 10 31            ldaa  adr1

0033 d113 97 00               staa  zero

0034 d115 b6 10 31            ldaa  adr1

0035 d118 97 00               staa  zero

0036 d11a b6 10 31            ldaa  adr1

0037 d11d 97 00               staa  zero

0038 d11f 20 e9               bra   here

0039

0040 d121 cf                  stop
```

Appendix D

Initial Design Verilog

```
module control (clk, data_out, reset, dataa, datab, clka, clkb, clkmema, clkmemb, adv, adspa, adsca, adspb,
adscb, g, lw, address) ;

input reset;
input clk ;
output [11:0] data_out ;
output clka, clkb, clkmema, clkmemb;
output adv, adspa, adsca, adspb, adscb, g, lw;
output [15:0] address;
output [9:0] dataa, datab;

wire [9:0] dataa, datab, signal ;
wire [11:0]  data ;
wire rfd, clk2 ;
wire adv, adspa, adsca, adspb, adscb, g, lw, rfda, rfdb, nd, indata;
wire address;

reg rdy;
reg [11:0] data_out;
reg sinr, sinf, soutr, soutf;
reg [1:0] counter;
reg mode;

// This block provides some timing control for the various parts and also
// sets some functions high because Xilinx didn't like it when I put the inputs
// to the filter as just 1's and I received some errors while simulating. Instead
// I use registers that are just 1's. The counter will be used to control how long
// the memory is written to and when to write addresses.

assign clka = clk;
assign clkb = ~clk;

buf         adcb (clkmemb, clkb);
buf     adca (clkmema, clka);

always @ (posedge clk)
begin
  sinr = 1'b1;
  sinf = 1'b1;
  soutr = 1'b1;
  soutf = 1'b1;
  if (reset) counter = 2'b00;
  else counter = counter + 1'b1;
end

// Take half the full frequency for other applications that have to
// run slower.
assign clk2 = counter[1];

// Generate a sawtooth wave test signal
testsignal test1 (clk, rfda, rfdb, reset, dataa, datab, mode);

// Memory controller instantiation
memfsm     memcontrol(clk, mode, rfd, reset, dataa, datab, adv, adspa, adsca, adspb, adscb, g, lw, rfda,
rfdb, nd, signal, address);

// 16 tap FIR filter
filt            filter (clk2, nd, sinr, sinf, signal, rdy, rfd, soutr, soutf, data);

always @ (posedge rdy)
  data_out = data;

endmodule
```

```
//----------- Begin Cut here for LIBRARY inclusion --------// LIB_TAG

// synopsys translate_off

`include "XilinxCoreLib/sdafirVHT.v"

// synopsys translate_on

// LIB_TAG_END ------- End LIBRARY inclusion -------------

module filt (ck, nd, sinr, sinf, indata, rdy, rfd, soutr, soutf, outdata) ;

input ck ;
input nd ;
input sinr ;
input sinf ;
input [9:0] indata ;
output rdy ;
output rfd ;
output soutr ;
output soutf ;
output [11:0] outdata ;

// This filter was generated using Xilinx's Core Generator using
// the Serially Distributed Arithmetic FIR Filter

filter1 filter2 (
        .DATA(indata),
        .ND(nd),
        .RFD(rfd),
        .SINF(sinf),
        .SINR(sinr),
        .SOUTF(soutf),
        .SOUTR(soutr),
        .CK(ck),
        .RSLT(outdata),
        .RDY(rdy));

endmodule


//*******************************************************************
//* This file was created by the Xilinx CORE Generator tool, and     *
//* is (c) Xilinx, Inc. 1998, 1999. No part of this file may be      *
//* transmitted to any third party (other than intended by Xilinx)   *
//* or used without a Xilinx programmable or hardwire device without *
//* Xilinx's prior written permission.                               *
//*******************************************************************/

// The following code must appear after the module in which it
// is to be instantiated. Ensure that the translate_off/_on compiler
// directives are correct for your synthesis tool(s).

//----------- Begin Cut here for MODULE Declaration -------// MOD_TAG
module filter1 (
        DATA,
		ND,
		RFD,
		SINF,
		SINR,
		SOUTF,
		SOUTR,
		CK,
		RSLT,
		RDY);

input [9 : 0] DATA;
input ND;
output RFD;
input SINF;
input SINR;
output SOUTF;
output SOUTR;
input CK;
output [11 : 0] RSLT;
output RDY;

// synopsys translate_off

		SDAFIRVHT #(
			0,
			0,
			-2,
			3,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			2,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			5,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			0,
			26,
			-58,
			-92,
			115,
			0,
			0,
			10,
			10,
			16,
			13,
			0,
			1,
			1,
			0)
		inst (
			.DATA(DATA),
			.ND(ND),
			.RFD(RFD),
			.SINF(SINF),
			.SINR(SINR),
			.SOUTF(SOUTF),
			.SOUTR(SOUTR),
			.CK(CK),
			.RSLT(RSLT),
			.RDY(RDY));

// synopsys translate_on

endmodule
// MOD_TAG_END ------- End MODULE Declaration -------------
```

```
module memfsm (clk, mode, rfd, reset, dataa, datab, adv, adspa, adsca, adspb, adscb, g, lw, rfda, rfdb, nd,
indata, address) ;

input clk ;
input rfd ;
input reset ;
input [9:0] dataa, datab;
output mode;
output adv ;
output adspa, adsca ;
output adspb, adscb ;
output g ;
output lw ;
output rfda, rfdb ;
output nd ;
output [9:0] indata ;
output [15:0] address ;

reg [15:0] address;
reg [1:0]  counter;
reg [3:0]  state_reg1;
reg [2:0] constantb;  // In phase ADC Channel Write Control
reg [1:0]  state_reg2;  // Out of phase ADC Channel Write Control
reg state_reg3;          // Read control
reg lw, g, adv, adspa, adsca, adspb, adscb, nd, rfda, rfdb, mode, ce, ci, clr, carry;
reg [9:0] indata;

// This FSM has two main components, the sample stage and the filter stage.
// During the sample stage, it is generating the appropriate control signals
// so that the RAM takes in an address every four cycles and that the address
// is incremented by four every four cycles.  Once 50k samples have been taken
// the FSM switches to filter mode where we have to wait for the filter to be
// ready to take in a new data point, get the data, tell the filter that the data
// is ready, and then wait for the next time the filter is ready for a new sample.

always @ (posedge clk)
  if (reset) begin
    state_reg1 = 3'b000;      // Initialize variables for write mode
    mode = 1'b1;
    adspa = 1'b1;
    adsca = 1'b1;
    g = 1'b1;
    counter = 2'b01;
    ce = 1'b1;
    ci = 1'b0;
    constantb = 3'b100;
    clr = 1'b0;
    end
  else begin
    if (mode) counter = counter + 1'b1;          // While during write mode, want a slower clock to adder
    case (state_reg1)
          4'h0:     begin
                       clr = 1'b0;
                       adspa = 1'b1;         // Switch back to burst mode
                       adsca = 1'b1;
                       state_reg1 = 4'h1; end
```

```
4'h1:        state_reg1 = 4'h2;  // Waste time
4'h2:     if (address >= 16'hC350) // If have taken 50k samples
               begin
               state_reg1 = 4'h4;  // to go into read mode
               lw = 1'b1;
               mode = 1'b0;
               g = 1'b0;
               clr = 1'b1;
               adv = 1'b1;
               constantb = 1'b1;  end
             else                    // If haven't taken 50k samples
               begin
               mode = 1'b1;
               state_reg1 = 4'h3;  end // Increment to next four bit address
4'h3:        begin  adspa = 1'b1;        // Switch to suspend burst for new address
             adsca = 1'b0;
             state_reg1 = 4'h0; end
4'h4:     begin  clr = 1'b0;
             adspa = 1'b0;
         if (rfd && ~nd) begin  indata = dataa;  // If filter ready for data
                                                        // provide new data
          rfda = 1'b0;
               rfdb = 1'b1;
          state_reg1 = 4'h5;     // Provided data from one ADC, now
          end                                  // go to next one for data
         else state_reg1 = 4'h4;  // Wait for rfd from filter
             end
4'h5:        begin
               nd = 1'b1;
               state_reg1 = 4'h6;
             end
4'h6:        if (~rfd)  begin
               nd = 1'b0;                               // Deassert new data pin
               counter = counter + 2'b10;    // Increment to next address
               state_reg1 = 4'h7;  end
             else
          state_reg1 = 4'h6;                 // Wait for filter to take new data
4'h7:       if (address >= 16'hC350) begin   // If all data read, go back to
                lw = 1'b0;                      // initial state
                adv = 1'b0;
                mode = 1'b1;
                state_reg1 = 4'h0;
                g = 1'b1;
                clr = 1'b1;
                constantb = 3'b100;
                end
               else
                state_reg1 = 4'h8;
4'h8:   if (rfd && ~nd) begin indata = datab;  // If filter ready for data
                                                  // provide data
               rfda = 1'b1;
               rfdb = 1'b0;
               state_reg1 = 4'h9;
               end
             else state_reg1 = 4'h8;   // Wait until filter ready for data
4'h9:     begin
```

```
                    nd = 1'b1;              //Assert new data control
                    state_reg1 = 4'hA;
                    end
        4'hA:       if (~rfd) begin
                      nd = 1'b0;                        // Wait until filter ready for data
                      state_reg1 = 4'h4;  end
                    else
                      state_reg1 = 4'hA;
        default:  state_reg1 = 4'h0;
  endcase
end

// Same as above, but for the out of phase ADC channel
// except for address and advance, which is done above
always @ (negedge clk)
  if (reset) begin
    state_reg2 = 2'b00;
    adspb = 1'b1;
    adscb = 1'b1;  end
  else if (mode) case (state_reg2)
        2'b00:   begin  adspb = 1'b1;        // Switch back to burst mode
                   adscb = 1'b1;
                   state_reg2 = 2'b01;  end
        2'b01:     state_reg2 = 2'b10;  // Waste time
        2'b10:   state_reg2 = 2'b11;  // Increment to next four bit block
        2'b11:     begin  adspb = 1'b1;
                   adscb = 1'b0;
                   state_reg2 = 2'b00;  end
      default  state_reg2 = 2'b00;
  endcase
  else if (~mode) adspb = 1'b0;

// There seemed to be problems in getting address = address + 4
// to work fast enough so I used a core module hopefully get things
// faster
speedadd addressadd (
        .A(address),
        .B({13'b0, constantb}),
        .C(counter[1]),
        .CE(ce),
        .CI(ci),
        .CLR(clr),
        .S({carry, address}));


endmodule


//***********************************************************************
//* This file was created by the Xilinx CORE Generator tool, and     *
//* is (c) Xilinx, Inc. 1998, 1999. No part of this file may be      *
//* transmitted to any third party (other than intended by Xilinx)   *
//* or used without a Xilinx programmable or hardwire device without *
//* Xilinx's prior written permission.                          *
//***********************************************************************/
```

```
// The following line must appear at the top of the file in which
// the core instantiation will be made. Ensure that the translate_off/_on
// compiler directives are correct for your synthesis tool(s)

//----------- Begin Cut here for LIBRARY inclusion --------// LIB_TAG

// synopsys translate_off

`include "XilinxCoreLib/adreVHT.v"

// synopsys translate_on

// LIB_TAG_END ------- End LIBRARY inclusion --------------

// The following code must appear after the module in which it
// is to be instantiated. Ensure that the translate_off/_on compiler
// directives are correct for your synthesis tool(s).

//----------- Begin Cut here for MODULE Declaration -------// MOD_TAG
module speedadd (
        A,
        B,
        C,
        CE,
        CI,
        CLR,
        S);

input [15 : 0] A;
input [15 : 0] B;
input C;
input CE;
input CI;
input CLR;
output [16 : 0] S;

// synopsys translate_off

        ADREVHT #(
                16,
                0)
        inst (
                .A(A),
                .B(B),
                .C(C),
                .CE(CE),
                .CI(CI),
                .CLR(CLR),
                .S(S));

// synopsys translate_on

endmodule
// MOD_TAG_END ------- End MODULE Declaration -------------
```

```
module testsignal (clk, rfda, rfdb, reset, dataa, datab, mode) ;

// Produce sawtooth test signal to control to see if control
// works as intended

input clk ;
input reset;
input rfda, rfdb, mode;
output [9:0] dataa, datab ;

reg [9:0] dataa, datab;
reg state_reg, next_state ;

always @ (posedge clk)
  if (reset) begin
    state_reg = 2'b00;
  end
  else  if (~mode) state_reg = next_state;

// Need to alternate the simulated ADCs that the signal is
// coming from in such a way as to produce a sawtooth wave

always @ (posedge clk)  begin
  if (mode == 0) case (state_reg)
    1'b0:  begin  if (rfda) begin
                    dataa = dataa + 2'b10;
                    next_state = 1'b1;  end
                  else
                    next_state = 1'b0;
                  end
    1'b1:     begin  if (rfdb) begin
                datab = datab + 2'b10;
                next_state = 1'b0;  end
              else
                next_state = 1'b1;
              end
    default:    begin  next_state = 1'b0;
                  dataa = 10'b0000000000;
                  datab = 10'b0000000001;
                  end
    endcase
end

endmodule
```

Appendix E

Two FPGA Design Verilog

Memory Controller

```
module control (clk, reset, clka, clkb, clkmema, clkmemb, adv, adspa, adsca, adspb, adscb, g, lw, address,
adc_choice, nd, rfd, mode) ;

input reset;
input clk ;
input rfd;
output clka, clkb, clkmema, clkmemb;
output adv, adspa, adsca, adspb, adscb, g, lw, adc_choice, nd, mode;
output [15:0] address;


wire rfd ;
wire adv, adspa, adsca, adspb, adscb, g, lw, nd;
wire address;

reg mode;
reg adc_choice ;
reg clkb1, clkb2, clkb3, clkb4, clka1, clka2, clka3 ;

// Using buffers to attempt to get better delay control for the clock
// output

assign clkmema = clk;
assign clkmemb = ~clk;

buf       adcb1 (clkb1, clkmemb && mode);
buf     adca1 (clka1, clkmema && mode);
buf     adcb2 (clkb2, clkb1);
buf     adcb3 (clkb3, clkb2);
buf     adcb4 (clkb4, clkb3);
buf       adcb5 (clkb, clkb4);
buf     adca2 (clka2, clka1);
buf     adca3 (clka3, clka2);
buf     adca4 (clka, clka3);

// Memory controller instantiation
memfsm     memcontrol(clk, mode, rfd, reset, adv, adspa, adsca, adspb, adscb, g, lw, nd, address,
adc_choice);


endmodule
```

```
module memfsm (clk, mode, rfd, reset, adv, adspa, adsca, adspb, adscb, g, lw, nd, address, adc_choice) ;

input clk ;
input rfd ;
input reset ;
output mode;
output adv ;
output adspa, adsca ;
output adspb, adscb ;
output g ;
output lw ;
output nd ;
output [15:0] address ;
output adc_choice;

reg [15:0] address;
reg [1:0]  counter;
reg [3:0]  state_reg1;
reg [2:0] constantb;  // In phase ADC Channel Write Control
reg [1:0]  state_reg2;  // Out of phase ADC Channel Write Control
reg state_reg3;          // Read control
reg lw, g, adv, adspa, adsca, adspb, adscb, nd, mode, ce, ci, clr, carry, adc_choice;

// Although there are a several states, this FSM can be broken into two main parts
// the sample part and the filter part.  The first four states provide control
// signals to the RAM which needs a new address every four cycles and needs adsca
// low for one cycle to take in that address.  The rest of the states are used
// during the filter stage and consist mainly of waiting for the filter to be
// ready for new data, getting that data, and then telling the filter that the
// new data is ready.

always @ (posedge clk)
  if (reset) begin
    state_reg1 = 3'b000;      // Initialize variables for write mode
    mode = 1'b1;
    adspa = 1'b1;
    adsca = 1'b1;
    g = 1'b1;
    counter = 2'b01;
    ce = 1'b1;
    ci = 1'b0;
    constantb = 3'b100;
    clr = 1'b0;
    end
  else begin
    if (mode) counter = counter + 1'b1;           // While during write mode, want a slower clock to adder
    case (state_reg1)
          4'h0:    begin
                      clr = 1'b0;
                      adspa = 1'b1;        // Switch back to burst mode
                      adsca = 1'b1;
                      state_reg1 = 4'h1; end
          4'h1:       state_reg1 = 4'h2;  // Waste time
          4'h2:    if (address >= 16'hC350) // If have taken 50k samples
                        begin
                        state_reg1 = 4'h4;  // to go into read mode
```

```
                    lw = 1'b1;
                    mode = 1'b0;
                    g = 1'b0;
                    clr = 1'b1;
                    adv = 1'b1;
                    constantb = 1'b1;  end
                  else                    // If haven't taken 50k samples
                    begin
                    mode = 1'b1;
                    state_reg1 = 4'h3;  end // Increment to next four bit address
4'h3:             begin  adspa = 1'b1;        // Switch to suspend burst for new address
                  adsca = 1'b0;
                  state_reg1 = 4'h0; end
4'h4:         begin  clr = 1'b0;
                  adspa = 1'b0;
            if (rfd && ~nd) begin
              adc_choice = 1'b1;
              state_reg1 = 4'h5;     // Provided data from one ADC, now
              end                                 // go to next one for data
            else state_reg1 = 4'h4;  // Wait for rfd from filter
                  end
4'h5:             begin
                    nd = 1'b1;
                    state_reg1 = 4'h6;
                  end
4'h6:             if (~rfd)  begin
                    nd = 1'b0;                                  // Deassert new data pin
                    counter = counter + 2'b10;    // Increment to next address
                    state_reg1 = 4'h7;  end
                  else
              state_reg1 = 4'h6;                 // Wait for filter to take new data
4'h7:            if (address >= 16'hC350) begin   // If all data read, go back to
                      lw = 1'b0;                 // initial state
                      adv = 1'b0;
                      mode = 1'b1;
                      state_reg1 = 4'h0;
                      g = 1'b1;
                      clr = 1'b1;
                      constantb = 3'b100;
                      end
                    else
                      state_reg1 = 4'h8;
4'h8:   if (rfd && ~nd) begin                          // Choose to get next sample from
                   adc_choice = 1'b0;                  // other ADC
                   state_reg1 = 4'h9;
                    end
                  else state_reg1 = 4'h8;
4'h9:     begin
                  nd = 1'b1;                             // Transfer data
                  state_reg1 = 4'hA;
                  end
4'hA:             if (~rfd) begin
                    nd = 1'b0;
                    state_reg1 = 4'h4;  end
                  else
                    state_reg1 = 4'hA;
            default:  state_reg1 = 4'h0;
  endcase
end

// Same as above, but for the out of phase ADC channel
// except for address and advance, which is done above
always @ (negedge clk)
  if (reset) begin
    state_reg2 = 2'b00;
    adspb = 1'b1;
    adscb = 1'b1;  end
  else if (mode) case (state_reg2)
            2'b00:   begin  adspb = 1'b1;         // Switch back to burst mode
                       adscb = 1'b1;
                       state_reg2 = 2'b01;  end
            2'b01:     state_reg2 = 2'b10;  // Waste time
            2'b10:   state_reg2 = 2'b11;  // Increment to next four bit block
            2'b11:     begin  adspb = 1'b1;
                       adscb = 1'b0;
                       state_reg2 = 2'b00;  end
        endcase
  else if (~mode) adspb = 1'b0;

speedadd addressadd (
            .A(address),
            .B({13'b0, constantb}),
            .C(counter[1]),
            .CE(ce),
            .CI(ci),
            .CLR(clr),
            .S({carry, address}));


endmodule


//*******************************************************************
//* This file was created by the Xilinx CORE Generator tool, and      *
//* is (c) Xilinx, Inc. 1998, 1999. No part of this file may be       *
//* transmitted to any third party (other than intended by Xilinx)    *
//* or used without a Xilinx programmable or hardwire device without *
//* Xilinx's prior written permission.                                *
//*******************************************************************/

// The following line must appear at the top of the file in which
// the core instantiation will be made. Ensure that the translate_off/_on
// compiler directives are correct for your synthesis tool(s)

//----------- Begin Cut here for LIBRARY inclusion --------// LIB_TAG

// synopsys translate_off

`include "XilinxCoreLib/adreVHT.v"

// synopsys translate_on
// LIB_TAG_END ------- End LIBRARY inclusion --------------

// The following code must appear after the module in which it
// is to be instantiated. Ensure that the translate_off/_on compiler
// directives are correct for your synthesis tool(s).

//----------- Begin Cut here for MODULE Declaration -------// MOD_TAG
module speedadd (
        A,
        B,
        C,
        CE,
        CI,
        CLR,
        S);

input [15 : 0] A;
input [15 : 0] B;
input C;
input CE;
input CI;
input CLR;
output [16 : 0] S;

// synopsys translate_off

        ADREVHT #(
                16,
                0)
        inst (
                .A(A),
                .B(B),
                .C(C),
                .CE(CE),
                .CI(CI),
                .CLR(CLR),
                .S(S));

// synopsys translate_on

endmodule
// MOD_TAG_END ------- End MODULE Declaration -------------
```

Filter Control and Interface with RAM

```
module control (clk, data_out, reset, dataa, datab) ;

input reset;
input clk ;
output [15:0] data_out ;
input [7:0] dataa, datab;

wire [7:0] dataa, datab ;
wire [15:0]  data ;

reg rdy, nd1;
reg [15:0] data_out;
reg sinr, sinf, soutr, soutf, state_reg;
reg [7:0]  signal;


// Need to set some of the inputs to the filter to high
always @ (posedge clk)
begin
  sinr = 1'b1;
  sinf = 1'b1;
end

// Need to offset the input signal due to the format that
// the ADC produces the output
always @ (posedge clk)
  case (state_reg)
  1'b0:     begin
                signal = datab + 9'd512;
           state_reg = 1'b1;
           end
  1'b1:     begin
                signal = dataa + 9'd512;
                state_reg = 1'b1;  end
  default:  state_reg = 1'b0;
  endcase


// 16 tap FIR filter
filt            filter (clk, nd1, sinr, sinf, signal, rdy, rfd, soutr, soutf, data);

always @ (posedge rdy)
  data_out = data;

endmodule
```

```
module filt (ck, nd, sinr, sinf, indata, rdy, rfd, soutr, soutf, outdata) ;

input ck ;
input nd ;
input sinr ;
input sinf ;
input [7:0] indata ;
output rdy ;
output rfd ;
output soutr ;
output soutf ;
output [15:0] outdata ;

// Used the Xilinx Core Generator to create a 16 tap FIR Filter
// Basically just need to cut and paste.

filter1 filter2 (
            .DATA(indata),
            .ND(nd),
            .RFD(rfd),
            .SINF(sinf),
            .SINR(sinr),
            .SOUTF(soutf),
            .SOUTR(soutr),
            .CK(ck),
            .RSLT(outdata),
            .RDY(rdy));
// INST_TAG_END ------ End INSTANTIATION Template ---------

endmodule



/**********************************************************************
/* This file was created by the Xilinx CORE Generator tool, and      *
/* is (c) Xilinx, Inc. 1998, 1999. No part of this file may be       *
/* transmitted to any third party (other than intended by Xilinx)    *
/* or used without a Xilinx programmable or hardwire device without *
/* Xilinx's prior written permission.                                *
/*********************************************************************/

// The following line must appear at the top of the file in which
// the core instantiation will be made. Ensure that the translate_off/_on
// compiler directives are correct for your synthesis tool(s)

//----------- Begin Cut here for LIBRARY inclusion --------// LIB_TAG

// synopsys translate_off

`include "XilinxCoreLib/sdafirVHT.v"

// synopsys translate_on

// LIB_TAG_END ------- End LIBRARY inclusion -------------

// The following code must appear after the module in which it
// is to be instantiated. Ensure that the translate_off/_on compiler
// directives are correct for your synthesis tool(s).

//----------- Begin Cut here for MODULE Declaration -------// MOD_TAG
module filter1 (
            DATA,
            ND,
            RFD,
            SINF,
            SINR,
            SOUTF,
            SOUTR,
            CK,
            RSLT,
            RDY);

input [7 : 0] DATA;
input ND;
output RFD;
input SINF;
input SINR;
output SOUTF;
output SOUTR;
input CK;
output [15 : 0] RSLT;
output RDY;

// synopsys translate_off

            SDAFIRVHT #(
                    0,
                    0,
                    -2,
                    3,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    2,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    5,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    26,
                    -58,
                    -92,
                    115,
                    0,
                    0,
                    10,
                    8,
                    16,
                    16,
                    0,
                    1,
                    1,
                    0)
            inst (
                    .DATA(DATA),
                    .ND(ND),
                    .RFD(RFD),
                    .SINF(SINF),
                    .SINR(SINR),
                    .SOUTF(SOUTF),
                    .SOUTR(SOUTR),
                    .CK(CK),
                    .RSLT(RSLT),
                    .RDY(RDY));

// synopsys translate_on

endmodule
// MOD_TAG_END ------- End MODULE Declaration -------------
```

Appendix F

HC11 Design Verilog

```
module control (clk, data_out, reset, dataa, datab, clka, clkb) ;

input reset;
input clk ;
output [15:0] data_out ;
output clka, clkb;
input [7:0] dataa, datab;

wire [7:0] dataa, datab ;
wire [15:0]  data ;
wire clka, clkb, slowclk, prevclktemp;

reg rdy, nd1, rfd, prevclk;
reg [15:0] data_out;
reg sinr, sinf, soutr, soutf;
reg [1:0] state_reg;
reg [7:0]  signal;
reg [6:0]  timecount;


// Clock generation
assign clka = clk;
assign clkb = ~clk;

// Need to set some of the inputs to the filter to high
always @ (posedge clk)
if (reset) begin
  timecount = 6'b0;
  sinr = 1'b1;
  sinf = 1'b1;
  end
else
begin
  timecount = timecount + 1;
  sinr = 1'b1;
  sinf = 1'b1;
end

// Need to get output from HC11s at a much slower rate than the
// FPGA clock.  Also need to provide more control over the control
// signals to the filter since the period between samples is going
// to be longer than the time needed to process one sample and I don't
// want filter taking in data until it is ready.

always @ (posedge clk)
  if (reset) begin state_reg = 1'b0;
    prevclk = 1'b0; end
  else
  case (state_reg)
  2'b00:     if (rfd && (prevclk ^ timecount[6]))   // Take in data from ADC A
                  begin
                  signal <= dataa;
            state_reg = 2'b01;
            nd1 <= 1'b1;
            prevclk = timecount[6];
            end
           else
            begin
            state_reg = 2'b00;//Wait until time for next sample from ADC A
            nd1 <= 1'b0;
           end
  2'b01:     begin                          //Deassert new data signal
               nd1 <= 1'b0;
               state_reg = 2'b10;
               end
  2'b10:     if (rfd && (prevclk ^ timecount[6]))  //  Take in data from ADC B
                 begin
                 signal <= datab;
                 state_reg = 2'b11;
                 nd1 <= 1'b1;
                 prevclk = timecount[6];
                 end
               else
                 begin
                   state_reg = 2'b10;     // Wait until proper time to take in ADC B
                   nd1 <= 1'b0;                        // sample
                 end
  2'b11:     begin
               nd1 <= 1'b0;               // Deassert new data signal
               state_reg = 2'b00;
               end
  default:  state_reg = 2'b00;
  endcase


// 16 tap FIR filter
filt          filter (clk, nd1, sinr, sinf, signal, rdy, rfd, soutr, soutf, data);

always @ (posedge rdy)
  data_out = data;

endmodule
```

```
module filt (ck, nd, sinr, sinf, indata, rdy, rfd, soutr, soutf, outdata) ;

input ck ;
input nd ;
input sinr ;
input sinf ;
input [7:0] indata ;
output rdy ;
output rfd ;
output soutr ;
output soutf ;
output [15:0] outdata ;

// Used Xilinx Core Generator to make a 16-tap FIR filter.

//----------- Begin Cut here for INSTANTIATION Template ---// INST_TAG
filter1 filter2 (
        .DATA(indata),
        .ND(nd),
        .RFD(rfd),
        .SINF(sinf),
        .SINR(sinr),
        .SOUTF(soutf),
        .SOUTR(soutr),
        .CK(ck),
        .RSLT(outdata),
        .RDY(rdy));
// INST_TAG_END ------ End INSTANTIATION Template ---------

endmodule

/*******************************************************************
/* This file was created by the Xilinx CORE Generator tool, and     *
/* is (c) Xilinx, Inc. 1998, 1999. No part of this file may be      *
/* transmitted to any third party (other than intended by Xilinx)   *
/* or used without a Xilinx programmable or hardwire device without *
/* Xilinx's prior written permission.                               *
/*******************************************************************/

// The following line must appear at the top of the file in which
// the core instantiation will be made. Ensure that the translate_off/_on
// compiler directives are correct for your synthesis tool(s)

//----------- Begin Cut here for LIBRARY inclusion --------// LIB_TAG

// synopsys translate_off

`include "XilinxCoreLib/sdafirVHT.v"

// synopsys translate_on

// LIB_TAG_END ------- End LIBRARY inclusion --------------

// The following code must appear after the module in which it
// is to be instantiated. Ensure that the translate_off/_on compiler
// directives are correct for your synthesis tool(s).
```

```
//----------- Begin Cut here for MODULE Declaration -------// MOD_TAG
module filter1 (
        DATA,
        ND,
        RFD,
        SINF,
        SINR,
        SOUTF,
        SOUTR,
        CK,
        RSLT,
        RDY);

input [7 : 0] DATA;
input ND;
output RFD;
input SINF;
input SINR;
output SOUTF;
output SOUTR;
input CK;
output [15 : 0] RSLT;
output RDY;

// synopsys translate_off

        SDAFIRVHT #(
                0,
                0,
                -2,
                3,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                2,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                5,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                0,
                26,
                -58,
                -92,
                115,
                0,
                0,
                10,
                8,
                16,
                16,
                0,
                0,
                1,
                0)
        inst (
                .DATA(DATA),
                .ND(ND),
                .RFD(RFD),
                .SINF(SINF),
                .SINR(SINR),
                .SOUTF(SOUTF),
                .SOUTR(SOUTR),
                .CK(CK),
                .RSLT(RSLT),
                .RDY(RDY));

// synopsys translate_on

endmodule
// MOD_TAG_END ------- End MODULE Declaration -------------
```