Thursday May 17th 2007
TA: Amir Hirsch
Author I: Dimitri Podoliev
Author II: Will Buttinger

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.111 – Introductory Digital Systems Laboratory

FINAL PROJECT

DIGITAL AUDIO PROCESSOR

Abstract

A digital electronic system that processes audio signals was implemented using verilog on an FPGA labkit. The system is built from a number of individual modules that each perform a different audio effect, based on input control parameters. In particular the effects implemented were a variable gain signal mixer, an audio delay with feedback control, low pass, high pass and band pass frequency filters, and an audio compressor. Audio is passed in and out of the system by an AC97 analog-to-digital converter. The connection between these modules and any additional external inputs is controlled by a highly configurable routing interconnect architecture, which can be configured from a computer via RS232 serial cable with simple commands, or from a VGA visual interface coming directly from the unit. Using the router, the modules were interconnected in different ways, with each configuration providing a different total audio modification. The interconnect technology that was developed is highly generic and extendable, and can easily be used as a miniature digital lab kit to construct simple circuits. It can be used in any situation where it is desirable to allow a user to interconnect pre-defined modules without having to wait for hardware synthesis.

Table of Contents

Introduction

The processing of audio signals is a fundamental part of the music and entertainment industry. Although advanced audio-processing systems already exist, they cost tends to restrict their use to professional recording and editing studios. For the amateur instrumentalist, particularly the guitarist, who wishes to alter the way their instrument sounds, many individual effects units must be purchased and physically wired together. Some multi-effect processors exist, but these are expensive and none of these set-ups allow for true flexibility and creativity in the audio modification process. Also for the general music enthusiast, a lot of money can end up being spent on an amplifier that can perform the signal equalisations that may be desired. Because of these reasons, it is clear that there is a need for a highly configurable and customisable audio processing unit, which is contained within a single package and will allow the user to configure, control, and combine built-in audio effects in whatever way they wish.

The system was implemented in verilog, and synthesised on to an FPGA labkit. By using this development environment, a new module can be created for each distinct audio effect that was to be included in the system, and then instantiated as many times as desired. Each module has input and output audio signals, as well as parameter inputs which control the exact behaviour of that module instance. For this particular body of work, the modules that were implemented were a mixer, which combines audio signals, a delay with feedback control and various frequency filters. The labkit's built-in AC97 unit was used to convert an analog audio signal in to an 18-bit signed digital signal, and convert the final output digital signal in to an analog audio signal. The system would also make use of the labkit's buttons and switches, which could be used as a demonstration for the setting of module parameters. It is key part of the system that the connection between these modules, inputs, and outputs be specified by the user.

To allow the user this control, an interconnect architecture was developed that could be configured using simple commands sent from a computer via an RS232 serial cable, or by using a VGA interface with a mouse to point and click on the connections that are to be made. The basic overall block diagram for this system is shown in figure 1.

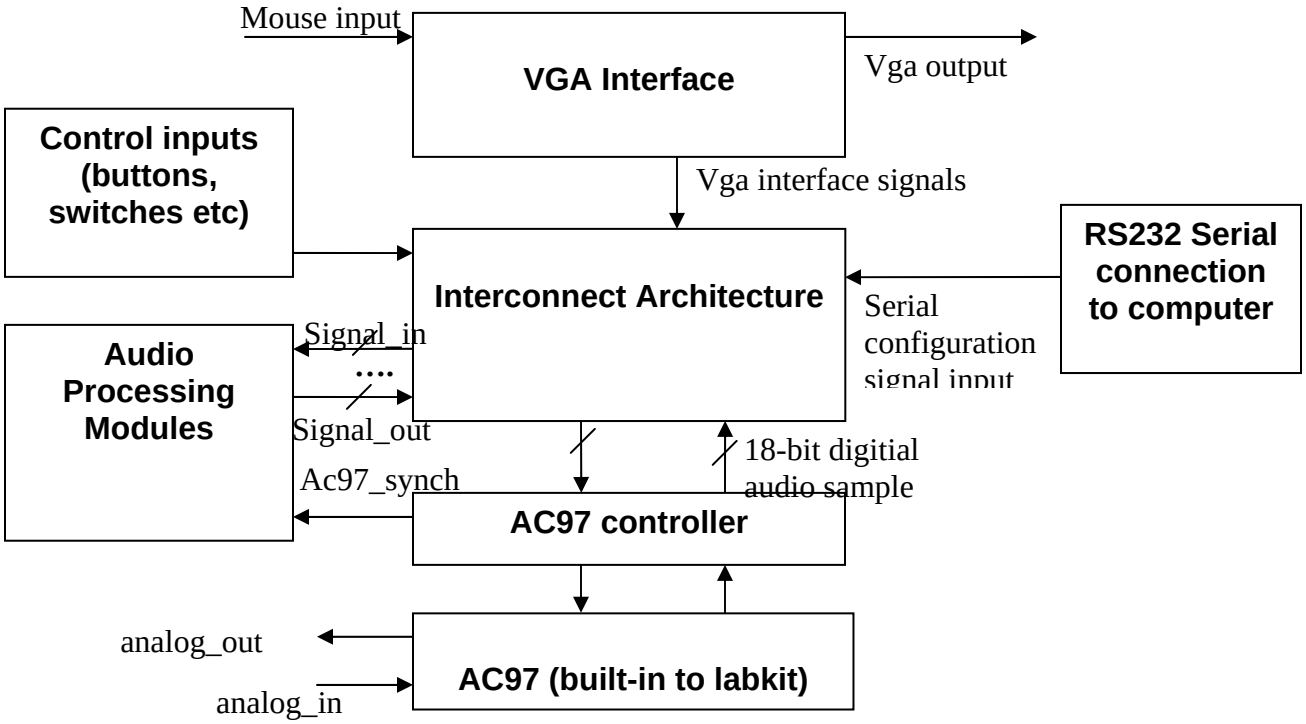

Figure 1 – General block diagram for system

The specific details of the modules within the system can be found in Section 2. The implementation and any simulated waveforms for each of these modules is given in Section 3, including any state transition diagrams. A discussion of how the system was tested and debugged is given in Section 4. A brief conclusion for the project is given in Section 5, where improvements and extensions to the project are discussed.

In order to interact with music, it was essential to use the AC97 audio codec available on the labkit. This codec facilitated the ADC (Analog-to-Digital Conversion) and DAC (Digital-to-Analog Conversion) between the analog signal received from any music source, such as an IPod or a microphone, and the processed digital signal sent back to the speakers or headset. Each of the effect's modules was unique in terms of the effect they have on the signal. The low-pass filter was designed to filter out the high frequency noise and to amplify the base and other low frequency sounds present in a given piece of music. The high-pass filter has exactly the opposite effect on the signal; it attenuates down the level of base, and amplifies the higher frequency sounds, such as the singer's voice. The band-pass filter on the other hand, incorporates both of the effects described above; it highlights the intermediate frequency range of the music. The compressor is only effect that is rarely applied to already made music. Its use is most rewarding during the music production, in cases where different instructs have different volume level, and when too loud, they can be appropriately compressed.

The division of the work was that Author 2 designed and implemented the interconnect architecture including the RS232 interface, the delay module and the mixer module. Author 1 designed and implemented the frequency filters, the compressor, the AC97 codec controller, and the VGA interface.

AC97 Interface

The AC97 audio codec which is integrated into the labkit provided high quality ADC and DAC of the music signal. The AC97 is a serial interface: all of the data is sent one bit at a time at a clock rate of 12.288MHz. The analog signal is sampled and each sample is decoded into a single frame of 256 bits. The sample rate of the codec is 48 KHz, which means that the AC97 can work with frequencies up to 24 KHz without introducing aliasing into the signal. This is more than required for this project, since the audible range of most humans does not exceed 22 KHz. Each digital sample was 18 bits long. Each frame has 12 10 bit long slots and a 16 bit tag. Two of the slots, slot 3 and 4, carry the left and right channel samples respectively. Even though the AC97 supports stereo sound, mono sound (left channel is the same as the right channel) was used through the course of this project. Each incoming frame was synchronized on the *ac97_synch* signal. This signal becomes high one clock cycle before the incoming frame, and remains high for the duration of the tag part of the frame (16 bits). In order not to mix different samples with each other, most of the audio modules were synchronized on the *ac97_synch* signal. Figure 2 shows the input and output signals of the AC97.

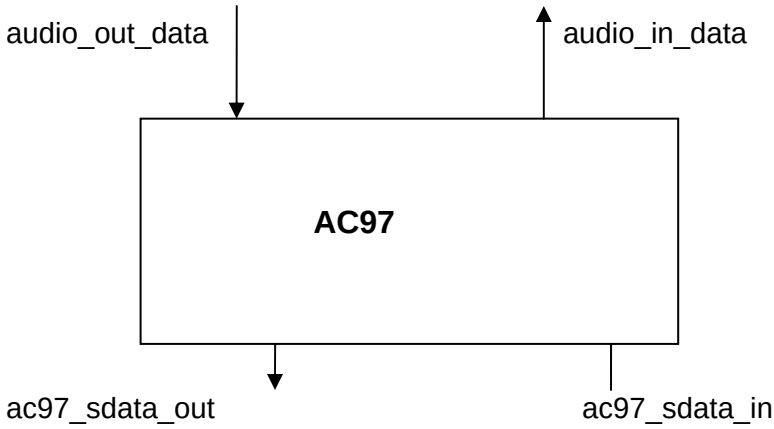

Figure 2 – AC97 Controller input/output signals

Filters

Digital Signal Processing Analysis

In general there are two types of digital filters: FIR (finite impulse response) and IIR (infinite impulse response). The difference between the two lies in the fact that IIR digital filters have internal feedback which potentially enables them to response to inputs for an indefinite amount of time. In contrast to IIR filters, FIR filters do not have internal feedback and their response to an impulse will eventually settle down to zero. All of the filters designed during the course of this project are FIR filters. In order to understand how to design a digital low pass filter, it is important to look through some digital signal processing math. The generic structure of a 2^{nd} order low pass filter is identical to the structure of an n^{th} order filter. Figure 3 shows a filter with its inputs and outputs expressed in time domain, frequency domain, and in discreet time domain.

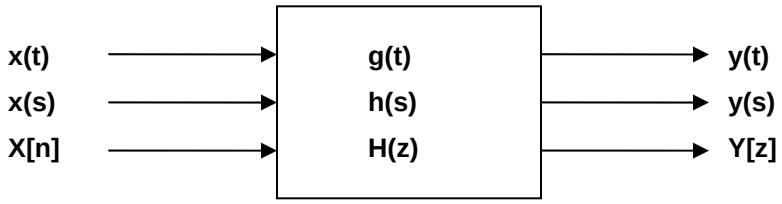

Figure 3 – Generic Filter diagram

Using Z-transform in discreet time domain, it is trivial to see that the output of a filter is related to its input according to the following equation:

$$y[n] = \sum_{k=0}^{N} a_k x[n-k] + \sum_{k=1}^{N} b_k y[n-k]$$

And a delay can easily be shown to be equivalent to: $z^{-1}y[n] = y[n-1]$. The following shows the general steps to be taken in order to obtain the discreet transfer function of the filter.

$$y[n] = \sum_{k=0}^{N} a_k z^{-k} x[n] + \sum_{k=1}^{N} b_k z^{-k} y[n]$$

$$H(z) = \frac{y[n]}{x[n]} = \frac{\sum_{k=0}^{N} a_k z^{-k}}{1 - \sum_{k=1}^{N} b_k z^{-k}}$$

With simple geometry, the poles' locations can be appropriately chosen, and the coefficients of the following difference equation can be found.

$$y[n] = \frac{n_0}{d_0} x[n] + \frac{n_1}{d_0} x[n-1] + \frac{n_1}{d_0} x[n-2] - \frac{d_1}{d_0} y[n-1] - \frac{d_2}{d_0} y[n-2]$$

Generic Architecture and Design

The difficulty with the above approach is that a second order filter will add a lot of noise to the filtered signal. In order to improve the signal-to-noise rejection ratio of the filter, the order of the filter should be increased. With increased order, the math becomes very tedious. That is why MatLab was used to generate the coefficients and to study the frequency response. In Verilog, this filter can be implemented in one of the following two ways. Figures 4 and 5 depict these methods.

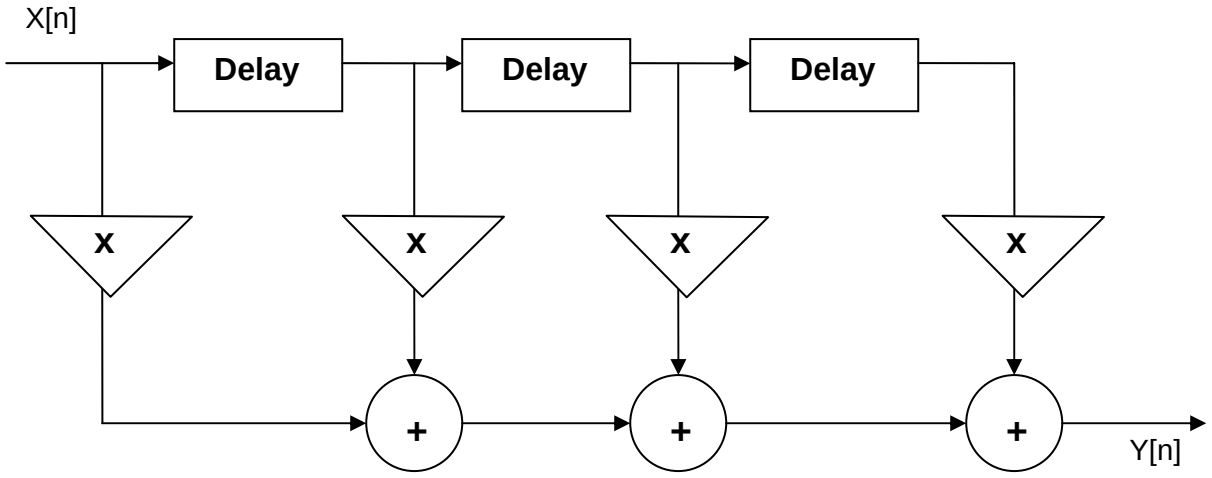

Figure 4 – Generic architecture of a digitally implemented filter 1

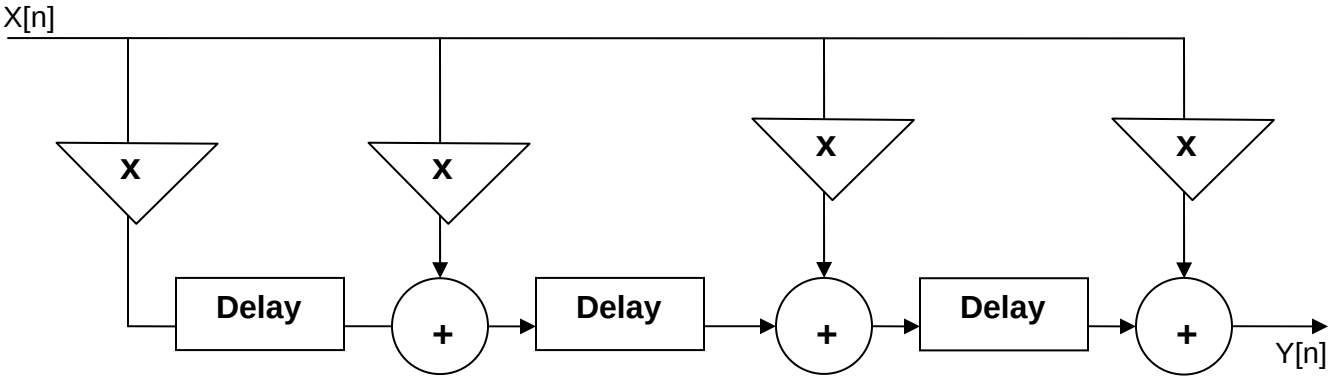

Figure 5 – Generic architecture of a digitally implemented filter 2

With some simple theory it is possible to prove that the two methods depicted above are equivalent. From the implementation point of view, the second (Figure 4), is computationally more efficient, because the longest critical path for the signal is smaller than in the case of Figure 3. In this project, the low-pass, high-pass and the band-pass filters were all implemented using the second generic architecture.

Low-Pass Filter

The purpose of a good low-pass filter is to rid the incoming signal of high frequency (quantization) noise, improving the signal-to-noise rejection ratio, and to take out all frequencies above a given threshold frequency.

Through a course of a lot of testing of various order low-pass filters, it was observed that 10^{th} order low pass filter provides excellent signal-to-noise rejection ratio with a -3dB cut-off frequency of around 0.3 in the normalized z-domain. This is equivalent to about 14.4 KHz in the frequency domain. Figure 6 shows the normalized frequency response of a 10^{th} order low-pass filter.

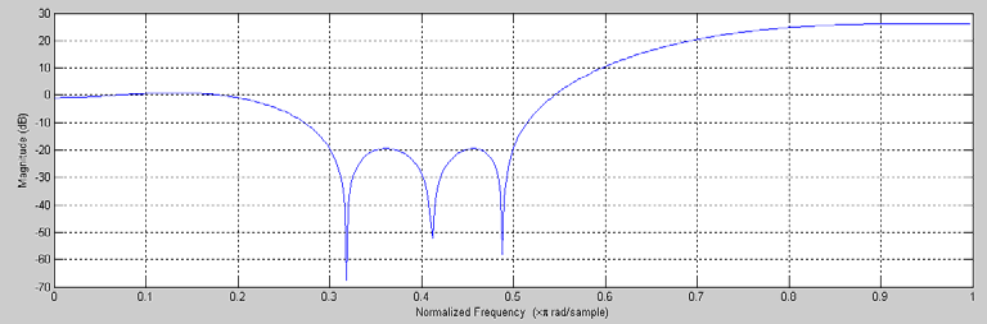

Figure 6 – Normalized frequency response of 10^{th} order low-pass filter

This low-pass filter attenuates the amplitudes of frequency components above 0.3 threshold by about -20dB.

When the design of this low-pass filter was completed, the filter exhibited good signal-to-noise rejection ratio, as well as nicely ridding the incoming signal from quantization noise which was added during the AC97 ADC.

High-Pass Filter

The design of the high-pass filter was similar to the design of the low-pass filter described above, although the multipliers' coefficients were different. Figure 7 shows the normalized frequency response of this high-pass filter.

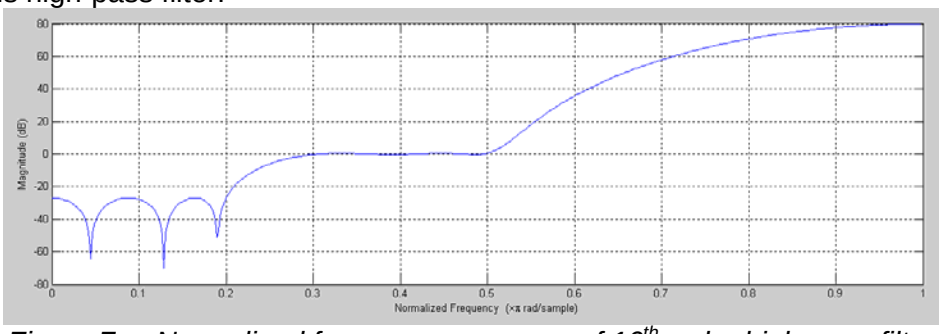

Figure 7 – Normalized frequency response of 16^{th} order high-pass filter

The goal of this filter is to filter out the frequencies below the specified threshold of 0.2 (9.6 KHz). Theoretically this filter has a very good signal-to-noise rejection ratio. But practically this filter can be

contaminated by high frequency noise. That is why before applying this filter, it is a good idea to filter low-pass the signal to get rid of the high frequency noise.

Band-Pass Filter

A band-pass filter is effectively a combination of a low-pass filter together with a high pass filter. A band-pass filter admits only a specified band of frequencies, rejecting everything else. The band-pass filter designed for this project that the following frequency response characteristics (Figure 8).

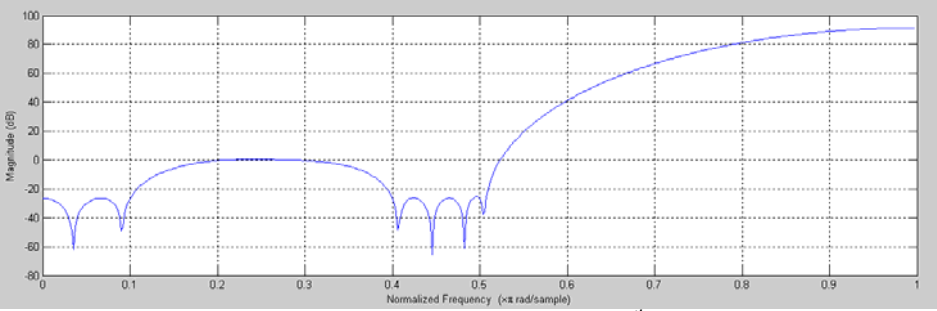

Figure 8 – Normalized frequency response of 16th order band-pass filter

Compressor

The compressor compresses the louder sounds, while leaving the quieter ones unchanged. The input to output amplitude characteristics of the audio compressor are depicted on Figure 9.

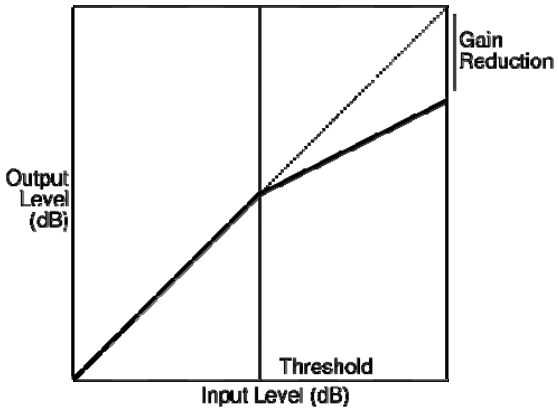

Figure 9 – Audio Compressor, Source: Wikipedia

Graphical User Interface

A graphical user interface has been developed to ease the use of the system. With this interface, the user no longer has to type in specific string in the HyperTerminal to interconnect the router. Instead the user can make the needed interconnections with the help of a mouse. This part of the project has been broken down into several sections. The following block diagram show the whole system, with all the modules and interconnections.

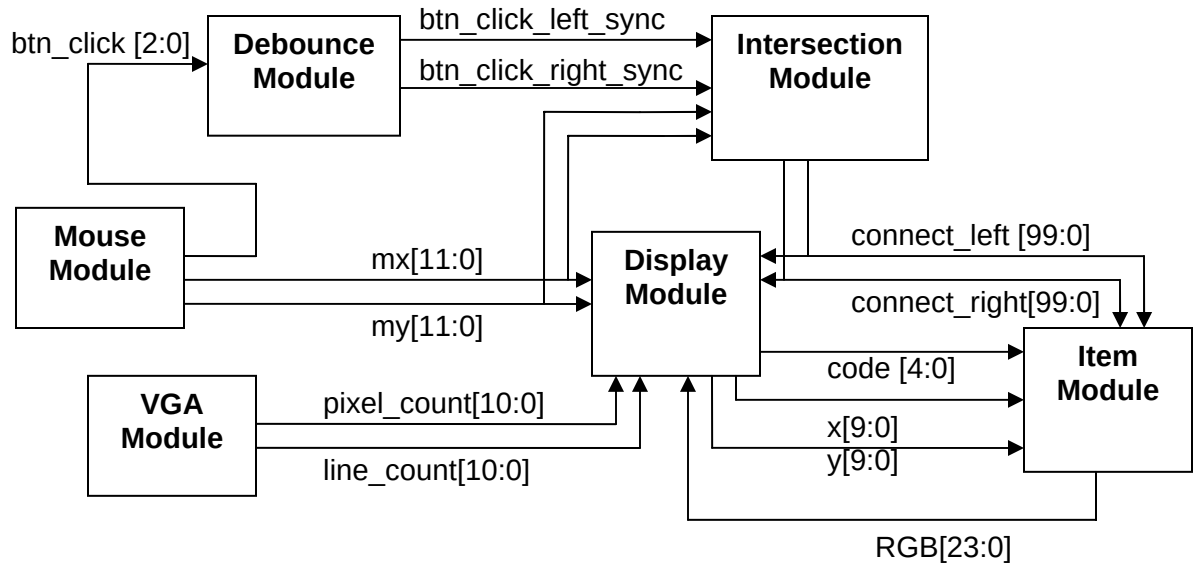

Figure 10 – Detailed block diagram for VGA interface

DCM

DCM or Digital Clock Manager was used to generate *pixel_clock* of 31.5 MHz from the 27 MHz labkit clock. *pixel_clock* was used to clock the VGA, Display, and Item modules.

VGA Controller

The VGA comptroller which was used in this project is similar to the one designed during lab 4. The resolution of the screen is 640 x 480. Each pixel on the screen has 24 bits which determine its color; eight bits for red, green and blue respectively.

Mouse Controller

The mouse controller module was used to interface with a PS2 mouse. The module provided the x[11:0] and y[11:0] coordinates of the mouse position, and three bits signal btn_click[2:0] which determined which button (left/right/center) was pressed.

Debounce Module

In this project there were three asynchronous inputs, namely: *reset, btn_click{2] and btn_click[0]*. Whenever an asynchronous input is present in a sequential system, setup and hold times can't always be guaranteed because the user may change any of these inputs at any time. A second problem occurs due to the mechanical bounce that is inherent in buttons and switches. As a switch or a button is pressed, the mechanical contact can jump up and down a few times, creating a sequence of on and off states. By using the debounce module provided, it is required of the input to be stable for 0.01 seconds before reporting a change on its output. The synchronized and de-bounced output signals are: *reset_sync, btn_click_left_sync* and *btn_click_right_sync.*

Display Module

The display module is the main module that generates the RGB values to be generated on the monitor screen. It interacts with all the other assisting modules. This module takes in as inputs the *pixel_count* and *line_count* from the VGA module, the *x[11:0]* and *y[11:0]* position of the mouse controller module and the signals *connect_left[99:0]* and *connect_right[99:0]* which determine which connections are connected and weather it's a horizontal or a vertical drive. The monitor screen is composed of pre-made blocks, which are defined in the item module. Using the signal *code[4:0]* and *color[23:0]* the display module determines which of the three pre-made structures to display and in what color. There are three pre-made structures: the horizontal blocks, the vertical blocks and the intersections. By specifying the x and y position of the top left corner of each of these structures, display module is able to replicate them as many times as required.

Item Module

This module communicates with the display module. It receives the code[4:0], the x[9:0] and y[9:0] position values of the top left corner, the *color[23:0]*, the position current position of the mouse, mx[11:0] and my[11:0], and the *connect_left[99:0]* and *connect_right[99:0]* signals which determine weather or not there is a connected join at a given intersection. Using these inputs, it outputs the RGB values of the region of a specified width and height, starting at the inputted x and y position. This module is also responsible for the changing color of the intersection lines when the mouse is over a given intersection region.

Intersection Module

This module is an intermediate module between the router and the display module. It facilitates the router interconnection based on the mouse position and clicks. It takes as input the mouse position and the mouse click, and outputs a number of outputs which tell the router what changes to be made. The first output is a single clock period pulse that tells the router that a click was made. Then there are two four bit numbers which determine at which row and column the click was made. And the last signal tells the router weather it was a right click or a left click, hence if it was a horizontal or vertical connect.

Modular Design

The interconnect architecture is made up of several modules. A module called *rs232_controller* exists to interface with the RS232 serial cable. It uses the labkit's built-in 27-mhz clock, and is configured to work with a 115200 bit rate. The module monitors the serial port for an incoming byte, and when one is detected it must ignore the start bit, sample the next 8 bits at approximately the mid-point of the bit (to ensure accuracy), and then handle the parity bit. This module does no parity checking so it will simply ignore this parity bit. The module also has an output *byte_ready* signal, which is high when the byte being output is valid, and low when the controller is currently updating the byte bus.

A module called *program_controller* is used to count through bytes. Each join in the interconnect architecture requires 8 bytes to define, so keeping count of which byte is currently being input is important. This module also detects if a reset signal is sent from the computer (which is a lower case "r"), or if a finish signal is sent (which is a lower case "f"). The reset and finish signals will reset the interconnections and disable any changes to the interconnections respectively. A reset will override the disabled state that is triggered by the finish signal.

The main module for the interconnect is the *router*. This module uses verilog-defined parameters to determine the bus-width for each line, the number of rows in the router, and the number of columns. It is also assigned an identifier, and is told which rows and columns are inputs. This is required by verilog because the development environment will not allow for the possibility of an input line being driven by

another line. We are required to always have input lines assigned to high impedance to prevent synthesis problems. The joins for each line are determined by a single register per line, since only one join is allowed per line. These joins are asynchronous, so that there are minimal delays for signal propagation between modules. For the rows, a join is called a “horizontal join”, which drives the row by the specified column. If the column number is set to zero, then the row is unjoined and defaulted to 0. A similar structure exists for the “vertical joins” between columns and rows, with the rows driving the columns. The limit for the number of rows or columns is restricted to 999, which is a restriction caused by the nature of the configuration commands from the computer. The 8 bytes which are used to program a join are broken down as follows: the first byte references the router identifier. Only the matching router will utilise the following bytes, the other routers will ignore the bytes. Because the router number is drawn from the last 4 bits of this first byte, this restricts us to a maximum of 16 routers. The second byte in the sequence specifies either a “horizontal join”, which is when an “h” is sent from the computer, or a “vertical join”, which is when any other byte is sent. The next 3 bytes represent the 3 digits of the decimal number for the row or column that is being driven, and the final 3 bytes represent the decimal number for the row or column that is doing the driving. So setting these last three bytes to 0 (the last 4 bits of each byte are what determines the number) will unjoin the specified line.

The system implemented in this body of work uses two instances of this router. The first is an 18-bit bus-width router, which is where the audio signals travel between modules. The second router is an 8-bit bus-width router, which is where the parameter values are connected. This implementation means that parameters by default in this system are 8-bits long. A visualization of this setup is given in figure 11.

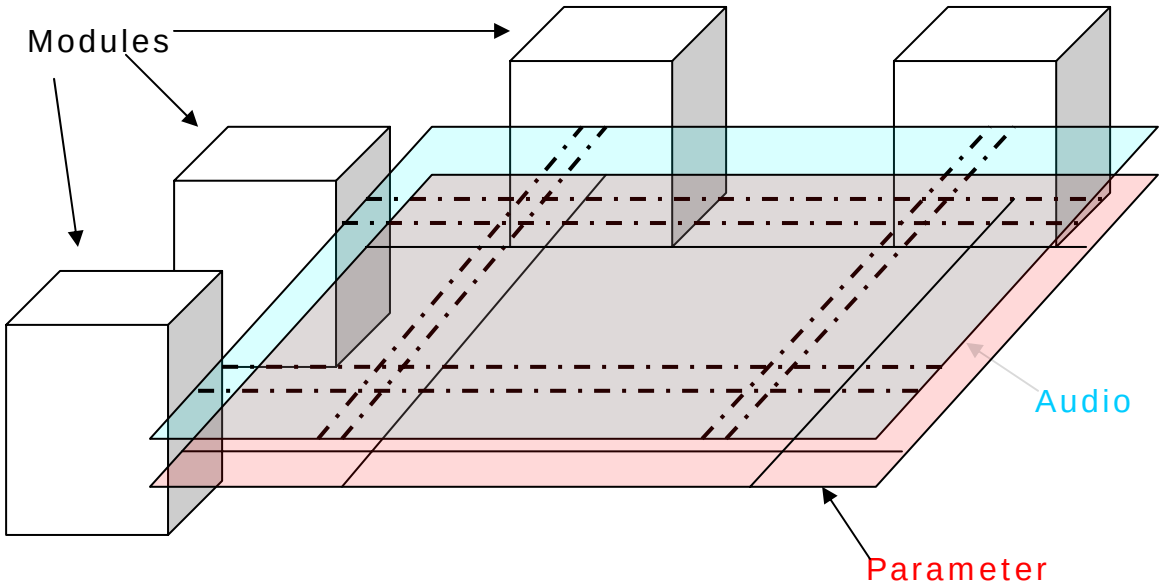

Figure 11 – visualization of the two router set up

The *mixer* is a module that takes in two audio signals and multiplies them by a coefficient, which ranges from –1 to approximately +1. This imperfect range is due to the nature of the coefficients being supplied, which are 8-bit signals, where the most significant bit represents a –1, and the next most significant bit represent +1/2, then the next bit represents +1/4, and so on. So when all the bits are on, the coefficient is a small negative value. The signals are also checked for clipping, which is where the combined signal crosses the upper or lower limit of the 18-bit signed format for the audio signal. If such a situation occurs, then the mixer will clip the output signal to the limit positive or negative value.

The *delay* module uses a block-ram in the FPGA to store up to 48,000 audio samples, which represents one second of audio. The stored samples are cycled through, up to a maximum number that is scaled by the 8-bit delay parameter (where a delay of 255 represents close to a full one second delay). As the

samples are cycled through, they are mixed back in to the original signal after having been scaled by a decay parameter, which determines how much of the sample to mix in. The larger the decay parameter, the more which is fed back, so the longer the echo will appear to occur for. The decay parameter is between 0 and approximately 1, and clipping checks are also performed, just like with the mixer. A simple audio flow diagram for the delay is shown in figure 12.

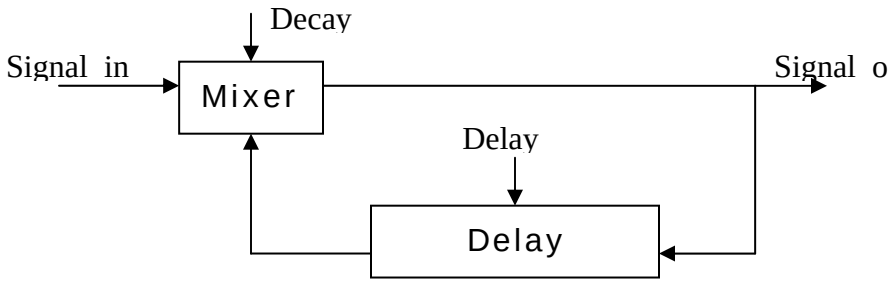

Figure 12 – Flow diagram for a delay module

Finally, a standard debounce module is used to debounce the global reset button, and any other unsynchronized signals that are required to be synchronized.

A detailed connection diagram between the various modules of this specific implementation of the system is given in figure 13. This includes the router connections made to the switches, buttons and LEDs of the labkit.

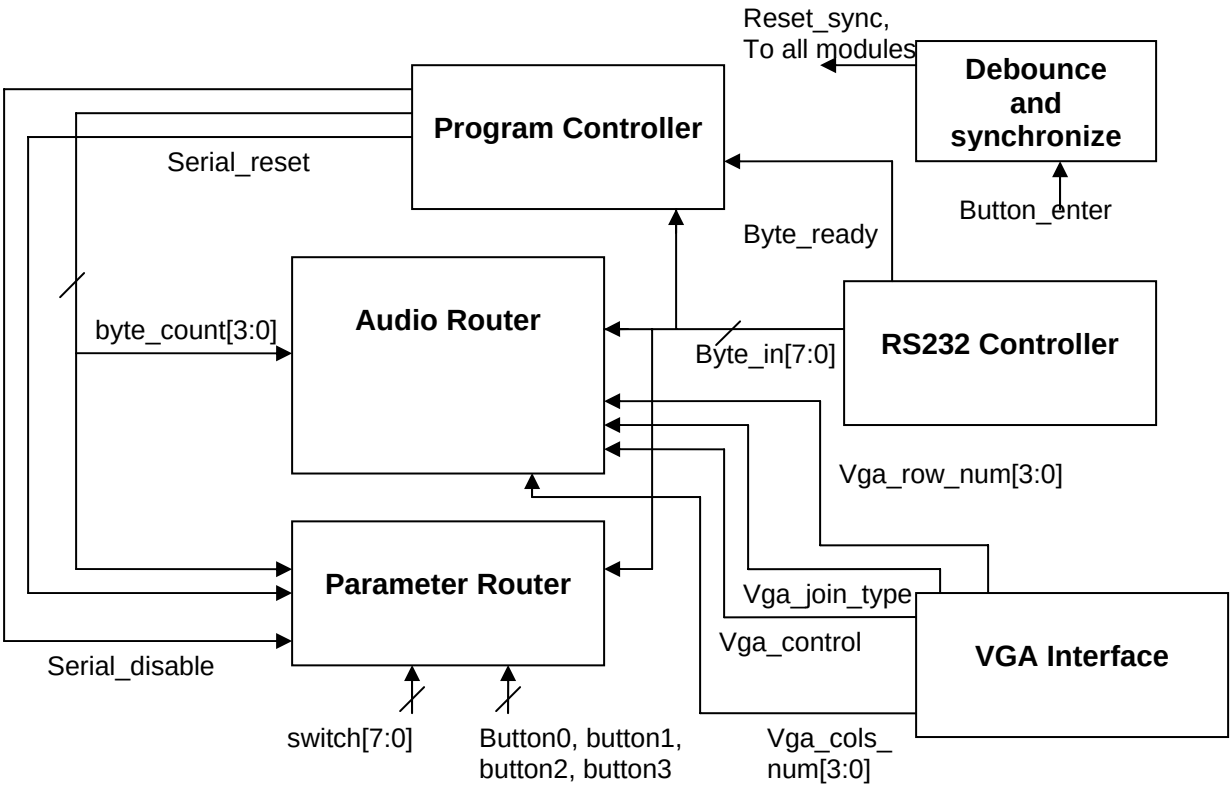

Figure 13 – Overall block diagram of interconnections between the router and the main control modules. The connection to the audio modules is not shown, but these connect directly to the audio router or parameter router.

Modular Implementation

The rs232_controller module uses the 27-mhz clock of the labkit for synchronization. It forms a finite state machine (FSM) with four states. The IDLE state is where the system waits for the incoming serial signal to transition from high to low, which signifies the start of an incoming packet from the RS232, and the byte_ready signal goes low. The state moves to START_BIT where the controller waits half a bit transmit period (which is 234 clock cycles for 115200 bit rate). This helps time the controller so that it samples the incoming signal on the centre point of each incoming bit. The next state is READ_WAIT, where the controller counts up the necessary number of clock cycles, and then samples the signal_in, and sets the appropriate bit of the signal_out. The bit number is kept track of with a register called bit_count, which when equal to eight signifies the completion of the byte read. At this point the byte_ready signal goes high, and the controller state changes to STOP_WAIT. This state simply waits for two full periods before returning the controller to the IDLE state. This wait period is necessary to ignore the parity bit that is added at the end of the byte, so that we are sure the signal_in will be in the high state when we return to IDLE. The controller does not expect any stop bits in this implementation, however handling these would simply involve the extension of the STOP_WAIT state by additional periods. The controller passes the signal_in through two synchronized registers at all times, to prevent metastable states. The state transition diagram for this controller is given in figure 14.

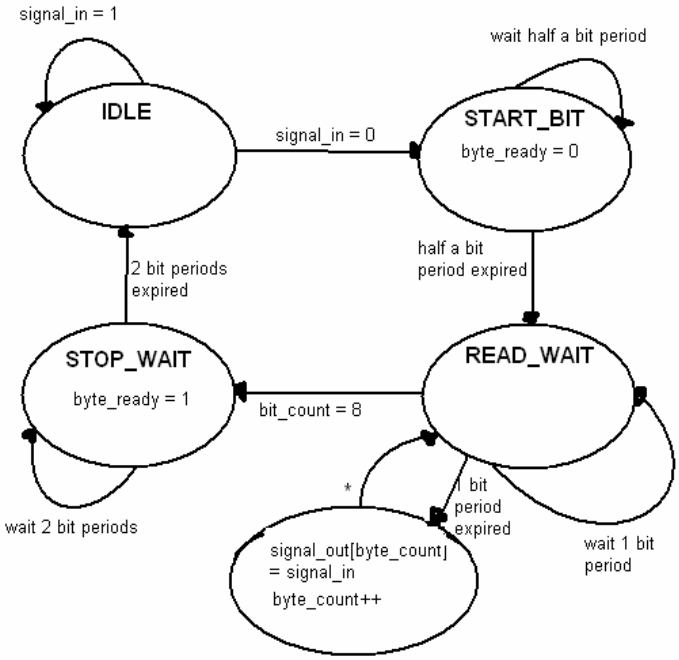

Figure 14 – State Transition Diagram for the RS232 controller. Module is timed from the 27-mhz clock.

The program_controller module is synchronized with the 27-mhz clock, and monitors the byte_ready signal from the rs232_controller for transitions from low to high. At this point the byte_in bus is checked. If this bus is found to be the byte representation for the “r” character then the controller will send the serial_reset signal high. Also the internal byte_count is reset to seven (so that the next byte coming in is the 0-byte), and the serial_disable signal is sent low. If otherwise the byte represents the “f” character, then the serial_disable signal is sent high. For all other bytes the byte_count is incremented by one. Being a 3 bit number, this byte_count register will automatically cycle to 0 after the 8th byte. The simulation waveform for this controller is shown in figure 15.

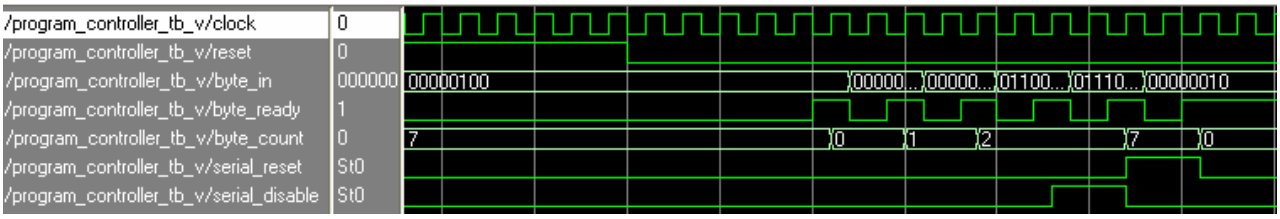

Figure 15 - Simulation waveform showing the controller coming out of reset state, byte_count defaults to 7. The first three bytes are regular bytes, the 4th byte is the "f" character the triggers the serial_disable. The 5th byte is the serial_reset which resets the byte_count to 7, triggers the serial_reset and turns the serial_disable low. The last byte is normal.

The router module is a series of asynchronous wire assignments, which are controlled by the verilog parameters and by the internal registers that represent the various joins. If a line is labelled as an input by the parameters, then it is always set to high impedance. Otherwise the corresponding register is checked, and if found to be non-zero, the join to the specified perpendicular line is performed. If the line is unjoined, it is assigned to zero. A series of for-loops are used in verilog to implement these wire assignments, because the width, height and bus-width of the router are specified externally by the parameters. This module also contains a 27-mhz clock synchronized block that monitors the control signals. If there is a global reset, or a serial_reset, then the join registers are cleared to zero, along with the other internal registers. Otherwise, if a change in the byte_count is detected, and the serial_disable is low, then the byte is processed according to it's byte number. The first byte in an 8-byte sequence is checked for a match against the routers identifier. If the match is positive, a join will occur after the 8th byte. The second byte is checked for the join type, and the remaining bytes are multiplied by the necessary power of 10 to make up the row and column number. Only the 4 least significant bits in each of these bytes are used to identify the line numbers.
The router also monitors the signals coming from the VGA interface section of the system. If a control pulse is detected, then a join is performed based on the signals from other signals from the interface.

The mixer module (also called an adder) takes in two audio signals, and using the 27-mhz clock the ac97_synch signal from the ac97 is monitored (after having been synchronized to the clock with multiple registers to prevent metastable states). When a rising edge is detected on this signal, the input signals are multiplied by the coefficient inputs. This multiplication is done via a series of bit shifts and adds (or one subtraction in the case of the most significant bit of the coefficient). The signals are added, and on the next clock cycle the combined signal is checked for clipping, which is where the combined signal has crossed the maximum or minimum value for the 18-bit signed audio signal. The clip can be detected by checking the changes of the MSB between combined and uncombined signals. If a clip is detected, the output signal is saturated to the maximum or minimum value for the 18-bit signal.
A simulation waveform for the mixer is shown in figure 16.

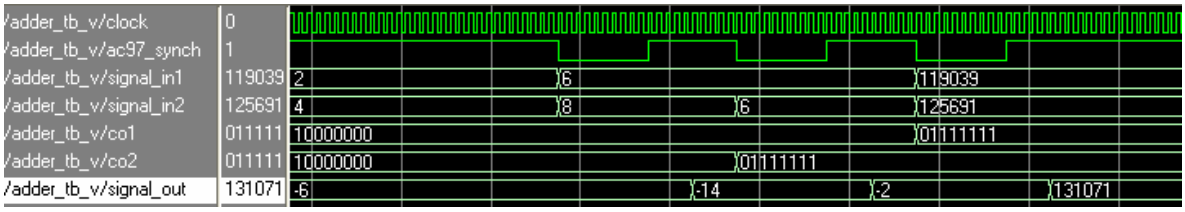

Figure 16 - Simulation waveform for the mixer. Initially the coefficients are set to -1. Then 2nd coefficient changes to almost +1, the result is a slightly negative number. Finally, both coefficients are changed to positive, and two large numbers are mixed. The result is a clipped number, at the maximum value for the 18-bit signal

The delay module is effectively a mixer that uses a block-ram as the source for it's second signal. It performs the same functions as the mixer module, however restricts the coefficient of the first signal to be 1 (unmodified), and the coefficient of the second signal (the delayed signal) to between 0 and approximately 1, which is implemented with bit-shifts similarly to the mixer. On a rising edge of the ac97_synch, the current sample from the block-ram (which is selected by a pointer register) is combined with the incoming signal according to the decay parameter. However, if the pointer is currently greater than a maximum pointer value, then no mixing occurs, because this state represents an increase in the delay parameter, and to prevent samples from further back in the audio signal being combined, they are ignored in the first cycle through the ram, which effectively deletes these old samples. The state of the module then moves to PRE_WRITE, where a clip-check is performed on the signal, just like in the mixer. The module then moves to the WRITE_START state, where the new signal_out sample is written to the current memory location, overwriting the old one. The system moves to WRITE_END1 state, where the writing to the ram stops, but the data in (din) and address (pointer) lines remain fixed, which ensures that the write has completed. The module then moves to the WRITE_END2, where the pointer value is checked against the specified delay parameter (multiplied by 188 to scale up to almost 48000 possible addresses given the 8-bit delay parameter). If required, the pointer is reset to 0 and the max_pointer register updated to signify the completion of the cycle, or otherwise the pointer is simple incremented.
The state transition diagram for this module is given in figure 17.

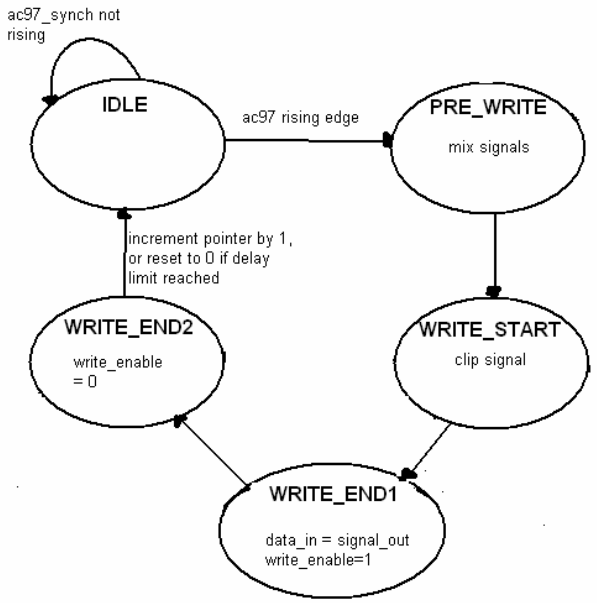

Figure 17 – State transition diagram for the delay.

Testing and Debugging

Although initial development and testing of the interconnect architecture is done using the ModelSim simulations on computer, it was found that in many situations the simulation would work, however various errors would occur when the code was synthesized, such as possible multi-source errors on the lines. It was quickly found that testing was more effective by simply synthesizing each change to the code and testing it. This was feasible given that initially only a single line join was being created, and the whole synthesis procedure took only a couple of minutes. This testing methodology actually

triggered design changes for the routing module, based on the restrictions that the software synthesizer was imposing. In particular, the initial design was intended to allow potentially multiple joins for one line, even though a user would never want to do this. The software detected the fact that multiple joins were possible, even if they were never going to happen in the configurations, and so rejected the code. This led to the development of storing the line number of the corresponding perpendicular line that needed to be joined. Furthermore, the addition of the verilog parameters that specified which lines were inputs was a requirement to get the code to compile, since the software detected the possibility that an input line could end up being driven by another input line. So the verilog parameters ensure that input lines are always set to high-impedance, regardless of what joins are made to them.

The audio modules were much more testable using the simulations on the computer, since they only involved various sequences of arithmetic. These simulations proved valuable to development because by the time the router was ready to interconnect audio signals, the synthesis time had already increased substantially. However, the simulations did not help identify a problem with noise appearing in some of the modules (particularly the delay and mixer). The source of the problem was eventually identified to be the unsynchronized ac97_synch signal which was being used in the modules to trigger signals in and out. Catching the signal in a metastable state caused certain signal updates to be missed, and noise appeared in the signal. Once the ac97_synch signal was passed through registers within the modules, this noise went away.

Also, obviously, testing audio effects is best done with an audio output, it is often hard to get a good idea of what is going on with an audio signal from a simulation waveform. This proved to be the case in some of the additional modules which were being developed but were not included in this body of work. For example a time-averaging module was being developed that could time-average the audio signal, and output an 8-bit number that can be used to automatically control the parameter of another audio module. This module appeared to work well in simulation, but did not function correctly after synthesis, and the module was not fixed in time for inclusion in this particular system implementation.

Conclusion

The system that was finally constructed forms a good basis for where the project can be extended. The interconnect architecture is highly flexible and extendable, and can be used in many other systems, not just audio processors. This was obvious from the testing where the router was connected to some basic digital logic gates, and the configuration of basic digital logic blocks, such as a latch, were easily determined and saved on the computer. The system can also be extended to include controlling inputs from analog signals, such as a potentiometer, as long as the signal is passed through an analog to digital converter. The input ports to the labkit can easily be added to the parameter router within the system.

In general this system has demonstrated it has the potential to be very useful. For example, using the low pass filter and the mixer with one of the coefficients set to invert, a high pass filter was created. This demonstrates how different effects can be built up from the basic modules which the system can provide. As more modules are develops, they can be connected in to the router. The user has the power to save their configurations on the computer, and change them at will using the VGA interface. As an extension to the project, adding a memory block for storing multiple configurations would be useful, as it would allow the user of the system to take their configurations away with them in the unit, without the need to connect it to a computer.

We, the authors, are pleased with the outcome of this work, and believe that the system produced can be adapted by others and incorporated in to various other digital electronic applications.