DESIGN, IMPLEMENTATION, AND CONSTRUCTION
OF RECONFIGURABLE HARDWARE FOR COMPUTER MUSIC

BY

CASEY JAMES SMITH

B.S., University of Illinois, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois
WE HEREBY RECOMMEND THAT THE THESIS BY

CASEY JAMES SMITH

ENTITLED DESIGN, IMPLEMENTATION, AND CONSTRUCTION
OF RECONFIGURABLE HARDWARE FOR COMPUTER MUSIC

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

Committee on Final Examination†

Chairperson

†Required for doctor's degree but not for master's.
© Copyright by Casey James Smith, 2001
For Kix.
ACKNOWLEDGMENTS

I would like to extend thanks to Professor Lippold Haken for his patience and guidance over the course of this project. Always willing to provide lunch, conversation, and advice, he offered a great opportunity to pursue work on a project that was an ideal mix of music and technology.

Thanks also to Chris Evans, Jake Janovetz, my family, and Wendy for technical advice as well as inspiration and motivation.

Finally, I would like to thank Professor Ricardo B. Uribe for providing an environment in which to educate myself as well as a healthy perspective on philosophy, life, and engineering.

In closing I would like to acknowledge the the position of privilege that I have been placed in. The opportunity to receive an education is not an opportunity everyone has. For this, I owe a debt of gratitude.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 General Description</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Organization</td>
<td>2</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Early Keyboard Instruments</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Analog and Digital Synthesizers</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Continuum</td>
<td>5</td>
</tr>
<tr>
<td>2.3.1 General description</td>
<td>5</td>
</tr>
<tr>
<td>2.3.2 Mechanical design</td>
<td>7</td>
</tr>
<tr>
<td>2.3.2.1 Computing finger location</td>
<td>7</td>
</tr>
<tr>
<td>3 PROPOSED METHODS</td>
<td>9</td>
</tr>
<tr>
<td>3.1 USB</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Integrated MIDI and Analog Board</td>
<td>11</td>
</tr>
<tr>
<td>4 HARDWARE</td>
<td>13</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>13</td>
</tr>
<tr>
<td>4.2 Input</td>
<td>13</td>
</tr>
<tr>
<td>4.3 Processing</td>
<td>14</td>
</tr>
<tr>
<td>4.4 Output</td>
<td>16</td>
</tr>
<tr>
<td>4.4.1 MIDI board</td>
<td>16</td>
</tr>
<tr>
<td>4.4.2 DAC board</td>
<td>17</td>
</tr>
<tr>
<td>5 SOFTWARE</td>
<td>22</td>
</tr>
<tr>
<td>5.1 Overview</td>
<td>22</td>
</tr>
<tr>
<td>5.2 ipEngine Code</td>
<td>22</td>
</tr>
<tr>
<td>5.2.1 Direct DAC updates</td>
<td>22</td>
</tr>
<tr>
<td>5.2.2 FIFO DAC updates</td>
<td>24</td>
</tr>
<tr>
<td>5.2.3 MIDI output</td>
<td>24</td>
</tr>
<tr>
<td>5.3 ipEngine VHDL</td>
<td>29</td>
</tr>
<tr>
<td>5.4 DAC Board VHDL</td>
<td>29</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Continuum and ipEngine Connections</td>
<td>14</td>
</tr>
<tr>
<td>4.2</td>
<td>ipEngine-1 Memory Map [13]</td>
<td>16</td>
</tr>
<tr>
<td>4.3</td>
<td>MIDI Board and ipEngine Connections</td>
<td>16</td>
</tr>
<tr>
<td>4.4</td>
<td>DAC Board and ipEngine Connections</td>
<td>18</td>
</tr>
<tr>
<td>4.5</td>
<td>DAC Board Registers</td>
<td>18</td>
</tr>
<tr>
<td>4.6</td>
<td>DAC Board Commands</td>
<td>19</td>
</tr>
<tr>
<td>4.7</td>
<td>DAC Serial Data (7..0)</td>
<td>20</td>
</tr>
<tr>
<td>4.8</td>
<td>DAC Serial Data (15..8)</td>
<td>20</td>
</tr>
<tr>
<td>4.9</td>
<td>DAC Serial Data (23..16)</td>
<td>20</td>
</tr>
<tr>
<td>4.10</td>
<td>DAC Status Register</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Alter FPGA Memory Map</td>
<td>29</td>
</tr>
<tr>
<td>B.1</td>
<td>MIDI Board Bill of Materials</td>
<td>46</td>
</tr>
<tr>
<td>D.1</td>
<td>DAC Board Version 1.0 Bill of Materials</td>
<td>53</td>
</tr>
<tr>
<td>F.1</td>
<td>DAC Board Version 2.0 Bill of Materials</td>
<td>66</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Continuum Fingerboard [7]</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Continuum Fingerboard Close-Up [7]</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Continuum Fingerboard Mechanics [7]</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Sensor Scan Screen Capture [7]</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>Sensor Parabolic Interpolation [7]</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>USB SOF Token</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Bode Plot of DAC output with 10-kΩ load</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>Bode Plot of DAC output without 10-kΩ load</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>ipEngine Block Diagram</td>
<td>15</td>
</tr>
<tr>
<td>5.1</td>
<td>Direct DAC Update Software Flow</td>
<td>23</td>
</tr>
<tr>
<td>5.2</td>
<td>FIFO DAC Update Software Flow</td>
<td>25</td>
</tr>
<tr>
<td>5.3</td>
<td>Direct DAC Update State Machine</td>
<td>30</td>
</tr>
<tr>
<td>5.4</td>
<td>Command Decode State Machine for Direct Update</td>
<td>32</td>
</tr>
<tr>
<td>5.5</td>
<td>DAC Loading State Machine</td>
<td>33</td>
</tr>
<tr>
<td>5.6</td>
<td>FIFO DAC Update State Machine</td>
<td>34</td>
</tr>
<tr>
<td>5.7</td>
<td>Command Decode State Machine for FIFO Update</td>
<td>35</td>
</tr>
<tr>
<td>5.8</td>
<td>DAC Consumer State Machine for FIFO Update</td>
<td>36</td>
</tr>
<tr>
<td>A.1</td>
<td>MIDI Board Schematic</td>
<td>40</td>
</tr>
<tr>
<td>A.2</td>
<td>MIDI Layout (All Layers)</td>
<td>41</td>
</tr>
<tr>
<td>A.3</td>
<td>MIDI Layout (Top Layer)</td>
<td>42</td>
</tr>
<tr>
<td>A.4</td>
<td>MIDI Layout (Bottom Layer)</td>
<td>43</td>
</tr>
<tr>
<td>A.5</td>
<td>MIDI Layout (Top Silk Screen Layer)</td>
<td>44</td>
</tr>
<tr>
<td>A.6</td>
<td>MIDI Layout (Drill Layer)</td>
<td>45</td>
</tr>
<tr>
<td>C.1</td>
<td>DAC Board Version 1.0 Schematic</td>
<td>49</td>
</tr>
<tr>
<td>C.2</td>
<td>DAC Board Version 1.0 Layout (All Layers)</td>
<td>50</td>
</tr>
<tr>
<td>C.3</td>
<td>DAC Board Version 1.0 Layout (Top Silk Screen Layer)</td>
<td>51</td>
</tr>
<tr>
<td>C.4</td>
<td>DAC Board Version 1.0 Layout (Drill Layer)</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>E.1</td>
<td>DAC Board Version 2.0 Schematic</td>
<td>57</td>
</tr>
<tr>
<td>E.2</td>
<td>DAC Board Version 2.0 Layout (All Layers)</td>
<td>58</td>
</tr>
<tr>
<td>E.3</td>
<td>DAC Board Version 2.0 Layout (Top Layer)</td>
<td>59</td>
</tr>
<tr>
<td>E.4</td>
<td>DAC Board Version 2.0 Layout (Bottom Layer)</td>
<td>60</td>
</tr>
<tr>
<td>E.5</td>
<td>DAC Board Version 2.0 Layout (Ground Layer)</td>
<td>61</td>
</tr>
<tr>
<td>E.6</td>
<td>DAC Board Version 2.0 Layout (Power Layer)</td>
<td>62</td>
</tr>
<tr>
<td>E.7</td>
<td>DAC Board Version 2.0 Layout (Silk Screen Top Layer)</td>
<td>63</td>
</tr>
<tr>
<td>E.8</td>
<td>DAC Board Version 2.0 Layout (Silk Screen Bottom Layer)</td>
<td>64</td>
</tr>
<tr>
<td>E.9</td>
<td>DAC Board Version 2.0 Layout (Drill Layer)</td>
<td>65</td>
</tr>
</tbody>
</table>

| G.1 | Continuum Control and Conversion Schematics [21]               | 69   |
| G.2 | Continuum Front Sensor Low Schematics [21]                    | 70   |
| G.3 | Continuum Front Sensor High Schematics [21]                   | 71   |
| G.4 | Continuum Back Sensor Low Schematics [21]                     | 72   |
| G.5 | Continuum Back Sensor High Schematics [21]                    | 73   |
| G.6 | Continuum Decoupling Schematics [21]                          | 74   |
LIST OF ABBREVIATIONS

BDM - background debug mode
BOM - bill of materials
CPU - central processing unit
DAC - digital-to-analog converter
DRAM - dynamic random access memory
DSA - dynamic signal analyzer
DSP - digital signal processing
FPGA - field programmable gate array
GPIO - general purpose input output
IC - integrated circuit
IDE - integrated development environment
I/O - input output
LCD - liquid crystal display
MB - megabyte
MIDI - musical instrument digital interface
MIPS - million instructions per second
NRZI - non return to zero inverted
PC - personal computer
PCB - printed circuit board
SOF - start of frame
UART - universal asynchronous receiver transmitter
UPM - user programmable machine
USB - universal serial bus
VCO - voltage controlled oscillator
VHDL - VHSIC hardware description language
VHSIC - very high speed integrated circuits
CHAPTER 1

INTRODUCTION

1.1 General Description

The hardware described in the following pages is an interface for a new computer music instrument, the Continuum Fingerboard. The Continuum is a new type of polyphonic music performance device with a continuous playing surface rather than discrete keys. In its current form, the Continuum is dependent on a PC for the scanning of its playing surface and has limited options for interfacing to synthesizers. The proposed hardware is designed to fulfill three major functions:

- Internalization of surface scanning hardware to remove the reliance on external PC.
- Provide MIDI input and output for interfacing with commercially available synthesizers.
- Provide analog output for “real-time” VCO based synthesizers.

The heart of the system centers around embedded PowerPC and FPGA technologies. An embedded PowerPC MPC823 microprocessor provides the processing power for scanning the keys and formatting the data for both the MIDI and analog interfaces. The FPGA provides flexible data buffering and formatting for the analog interface. All described hardware for the final design was designed, built, and tested.
1.2 Motivation

The motivation for the development of the proposed interface hardware for the Continuum was to expand its capabilities to include MIDI and analog outputs, thus allowing for connections to commercially available synthesizers. Currently, users of the Continuum keyboard are required to use a somewhat expensive (although extremely powerful) DSP engine called the Capybara from Symbolic Sound [1]. The most unique feature of the new interface is the ability to provide analog output. There are fairly few controllers available that provide analog control voltages as output. Having this capability makes the Continuum an attractive input device for experimental musicians using analog VCO based synthesizers. Attempts were also made to keep the analog interface standalone as well as configurable so as to expand its use to any application where there is a need for multichannel digital-to-analog conversion.

1.3 Organization

The remainder of this paper is organized as follows. Chapter 2 consists of general background information on previous and relevant instruments as well as information pertaining to the current architecture of the Continuum keyboard. Chapter 3 introduces the proposed directions for the continuum keyboard interface. Applicable work on proposed designs is discussed as well as why some development paths were abandoned in favor of the final implementation.

Chapter 4 covers the details of all the hardware designs for the interface. Chapter 5 details the software development involved with the final interface implementation. Chapter 6 contains concluding remarks.
CHAPTER 2

BACKGROUND

2.1 Early Keyboard Instruments

Interest in keyboard based instruments dates back to the fifteenth century with the development of the clavichord. The clavichord produces sound in a way fairly similar to a piano in that when a key is pressed a brass blade comes in contact with a string, causing it to vibrate. When the key is released the brass blade is lifted from the string and the vibration of the string is damped. As simple as this mechanism is, it provided the performer with a fair degree of dynamic control over individual notes in a chord. The clavichord could also produce vibrato in notes by varying pressure on the key after it was depressed. Despite the ability to provide responsive control over notes, the clavichord was not able to play loudly enough for most situations and thus gave way to the harpsichord in the sixteenth century.

The harpsichord had the advantage of being loud enough to fill a large room by plucking of strings when keys are depressed. However, the volume comes at a price since plucking necessitates the loss of dynamic control. Similarly, the organ of the seventeenth century also lacks dynamic control over individual notes in a chord.

The eighteenth and nineteenth centuries saw the development of what is still the most popular acoustic keyboard instrument, the piano. The piano was a culmination of previous efforts in that it provided the volume to fill a large hall as well as the ability to provide dynamic control over individual notes in a chord. However, the piano is not
without its limitations. The hammer and dampening system of the piano action make crescendo and vibrato impossible since notes begin to decay immediately after being struck [2].

2.2 Analog and Digital Synthesizers

Although there are far too many analog and digital performance synthesizers and input devices to be covered here, a few are worth mentioning due to their attempts at continuous control of pitch pertaining to the work documented in this thesis.

The first electronic instrument to allow continuous control of pitch was the Trautonium developed by Friedrich Trautwein in 1928 and later improved upon in the 1950s by Oskar Sala. Continuous pitch control was accomplished using a long resistive wire stretched across a metal plate. The resistive wire varied the voltage across a capacitive circuit which, in turn, discharged onto the grid of a vacuum tube periodically, changing the output frequency of the system. Continuous volume control was also provided through a pressure-sensitive resistor located underneath the plate. The main limitation of this method, and of the ribbon controllers of future analog synthesizers, was that these controllers provide only one dimension of continuous control and cannot track more than one finger [3], [4].

Another continuous pitch instrument from the 1920s was the Aetherphon, more commonly known as the Theremin, invented by Leon Theremin. The most striking feature of this instrument is in the way it is played. The performer plays the Theremin by moving her hands with respect to two antennae. By doing so, the performer changes the effective capacitances of the antennas which are components in a high-frequency oscillator. One hand controls amplitude while the other controls the frequency of a 170-kHz oscillator and through heterodyning, the audible beat frequency [3].

Modern electronic keyboards are able to use key velocity and aftertouch to control sound synthesis and in some cases provide a polyphonic aftertouch, allowing the performer continuous control over individual notes in a chord. The Clavier developed by Robert
Moog [5] in the 1980s extends these capabilities by measuring not only pressure aftertouch but exact vertical and horizontal location of the finger on the key as well.

The last keyboard of interest is the Rolky keyboard developed by researchers at McGill University and demonstrated in 1985. The Rolky provided a continuous playing surface which, unlike ribbon controllers, was two-dimensional and could track each finger's position, offering the performer a great deal of control over individual notes in a chord [6].

2.3 Continuum

As stated in Section 1.1, the Continuum, shown in Figure 2.1, is a new type of polyphonic music performance device with a continuous playing surface rather than discrete keys. It is capable of tracking up to 10 points on the x-, y-, and z-axes independently for up to 10 simultaneous notes, as shown in Figure 2.2. In order to have an understanding of the need for the interface discussed in this thesis, this section will provide background information on the continuum and its current and proposed interfaces [2].

![Figure 2.1 Continuum Fingerboard [7].](image)

2.3.1 General description

The outputs from the Continuum's playing surface are mapped to parameters corresponding to pitch, timbre, and dynamics. The x-axes position maps to pitch. Since the pitch control of this instrument is continuous, the performer must place the fingers quite accurately to play in tune. The performer is also able to slide or rock fingers after they are placed for pitch glide and vibrato. An octave on the Continuum is slightly smaller
than an octave on a standard piano keyboard with a 2.5-cm change in the x-direction corresponding to a pitch change of 200 cents [2].

The y-axis position maps to timbral control for each note in a chord. In this way the performer can bring out certain notes in a chord by playing them with different timbre in a fashion similar to that employed with the finger boards of classical string instruments. Since all axes offer continuous control, movement on the y-axis can be used to perform timbral glides as well [2].

The z-axis position, corresponding to the pressure placed on the playing surface, maps to dynamic control. By fluctuating pressure, a performer can create tremelo with an experienced performer being able to play a crescendo on one note while simultaneously playing a decrescendo on another [2].

It is important to note that although these are the current mappings used by the Continuum, they are completely arbitrary. The Continuum consists of two components: the mechanical playing surface connected to a control computer, and a synthesis system to produce sound samples. The control computer scans the mechanical surface and produces 10 sets of x-, y-, and z-coordinates. These coordinates may then be interpreted as desired by the users synthesis method. This makes the Continuum a very flexible input device since any change in synthesis technique does not affect the software of the control computer [2].
2.3.2 Mechanical design

The playing surface of the Continuum is constructed from multiple small rods with magnets on their ends. The rods are suspended by small pins and springs such that the magnets are near Hall effect sensors as shown in Figure 2.3. When the performer presses down on the Mylar playing surface the rods beneath the finger are depressed, moving the magnets on their ends closer to the Hall effect sensors. The two Hall effect sensors at either end of the rod detect the proximity of the magnets, and this information is used to calculate finger position as discussed in Section 2.3.2.1 [2].

![Figure 2.3 Continuum Fingerboard Mechanics [7].](image)

2.3.2.1 Computing finger location

The control hardware for the Continuum, further described in Chapter 4, scans all 512 Hall effect sensors and then normalizes their values using minimum and maximum readings. Minimum and maximum readings are obtained from the state where the rods are fully depressed \(^1\) and at rest, respectively.

The scanning software then detects fingers by looking for any bars whose normalized reading is greater than both its neighboring bars. This bar is labeled as the center, with its neighbors labeled as left and right. Raw data is shown in a screen capture in Figure 2.4. Pitch, in this case the x-axis, is computed from the location of the center bar. The y-axis position, timbre, is computed by summing the the back sensors of all three bars and dividing by the sum of all six sensors. The total finger pressure, the z-axis, is then just the sum of all six sensors. For vibrato the Continuum is able to detect exact

\(^1\)In practice this involves the performer running a finger at maximum pressure down the front and back of the playing surface for the full length of the Continuum.
pitch by using parabolic interpolation of the sensor values as illustrated in Figure 2.5. Software discussed in [2] then tracks the finger locations to determine when fingers are placed, removed, and moved and uses this information to generate and modify individual notes.

Figure 2.4 Sensor Scan Screen Capture [7].

Figure 2.5 Sensor Parabolic Interpolation [7].
CHAPTER 3

PROPOSED METHODS

As mentioned in Section 1.2, the current sound synthesis processor for the Continuum Fingerboard is the Capybara from Symbolic Sound [1]. The Capybara is a powerful signal processing engine based around multiple Motorola DSPs and a graphic programming environment called Kyma [8]. While this system is very powerful and useful for programming and testing new synthesis techniques as well as running computationally intensive algorithms, it all comes at a high dollar cost. As such, it becomes useful to provide the Continuum with other interface options to interface with a larger variety of commonly available synthesizers. In the following sections, methods for providing these interfaces are examined in various stages of development.

3.1 USB

At first glance a reasonable choice for a standardized interface for the Continuum was MIDI. Since its introduction in 1983 MIDI has become a fairly mature standard used in many existing synthesizers. Although the data from the Continuum is quite able to be conformed to the MIDI data protocol, the large amount of data produced by the Continuum when tracking 10 fingers is too much for the standard MIDI data rate of roughly 32-k baud to handle. Further analysis pointed to the development of a new USB-MIDI standard that proved to be of some interest. The USB-MIDI Class Definition defines protocols to transfer MIDI data at a rate hundreds of times faster than the original MIDI hardware [9].
The original MIDI specification has 16 channels available on a single cable [10]. The USB-MIDI standard expands this via use of "virtual" cables. By parsing standard MIDI events into virtual cables and 32-bit messages for standardized transfer over the USB, the MIDI data itself is transferred transparently.

In order to make the Continuum both independent of a PC and USB compatible, it was necessary to find a microprocessor with the processing power required for scanning the Continuum as well as a USB interface. The ipEngine, detailed in Section 4.3, was chosen for this task.

The first step was to create a BDM initialization file to configure the ipEngine memory for use with the Code Warrior IDE from Metrowerks [11]. Doing so required setting up the UPM on the ipEngine's Power MPC823 controller. The UPM is responsible for proper timing of read and write cycles to external RAM [12]. To speed the process the ipEngine's RS232 console bootstrap was used to read the UPM configuration registers [13]. The commands for this operation and the resulting BDM configuration are shown in Appendix K. Once the BDM was functional and the RS232 console I/O configured, the next task was to create low-level USB drivers based on the USB specifications [14]. At the level of development reached, the drivers were able to detect SOF, IN, and SETUP USB tokens. Code listings for the USB drivers are shown in Appendix H. Testing of the drivers was accomplished using a PC as a USB host and running the USB controller in test mode using USB Single Step Transaction software from Intel [15], [16]. This software allowed for stepping through the transmission of individual USB tokens. Despite some initial software compatibility issues, functioning of the software was determined by scoping the USB differential pair as shown in Figure 3.1 and interpreting the NRZI encoding. A log file showing the console I/O from the ipEngine running the basic USB driver and decoding SOF, IN, and SETUP tokens is shown in Appendix H. The next step in development would have been to set up the descriptor tables and USB command parsing for enumerating the ipEngine on a USB host bus. However, reports of timing problems with USB-MIDI as well as change in market interest prompted the discontinuance of this line of effort.
3.2 Integrated MIDI and Analog Board

As USB development was abandoned, interest arose to use the Continuum as an input device for VCO based analog synthesizers. Doing so required the Continuum to be able to output analog control voltages. To accomplish this schematics were completed and PCB layout initiated for a 12-channel analog output board with four MIDI outputs and one MIDI input as shown in Appendix C. The board has two basic sections: MIDI and analog. The MIDI portion was based around a PIC microcontroller and 8-channel UART. MIDI was to be transmitted in the I2C protocol to the PIC which would then pass it on to the UART. The analog portion consisted of three 4-channel serial DACs from Burr Brown capable of the 0-10V voltage swing and 16-bit resolution required. The decision was made early on to include differential line drivers on the output of the DACs. DSA analysis was done to ensure that the impedance characteristics of the DAC and line driver would not adversely affect the low-pass filter placed between them as shown in Figure 3.2 and Figure 3.3.

Although this design was progressing nicely it was also abandoned in favor of the current design presented in this thesis. The new design separates the MIDI from the analog and adds the ability to reconfigure the hardware for different applications.
Figure 3.2 Bode Plot of DAC output with 10-kΩ load

Figure 3.3 Bode Plot of DAC output without 10-kΩ load
CHAPTER 4

HARDWARE

4.1 Overview

The hardware is composed of three main functional components. The first is the input section, which pertains to the interface between the Continuum and ipEngine. The second component is the ipEngine itself and the associated processing involved in scanning the Continuum and formatting the data. The third and final is the DAC board itself and its associated interface.

4.2 Input

The main input into the system, aside from user interface buttons, is the Continuum itself. The Continuum has a fairly simple parallel port interface with which to control and read sensors. Schematics for the Continuum interface are shown in Appendix G.

Interfacing to the ipEngine is accomplished through the virtual I/O implemented with the onboard Altera FPGA [13]. Physical connections to the ipEngine are shown in Table 4.1. Access to the virtual I/O is handled through a straightforward memory mapping as discussed in Section 5.3.

The process for reading the Continuum is fairly simple. The data direction pins on the ipEngine interface are first set for input. The address to be read (see Appendix G for function table) is placed on the address pins and the corresponding register is read from the Continuum data lines.
Writing is also straightforward. The data direction pins on the ipEngine interface are set for output and the address pins set to the sensor number register. Next, the sensor number to be read is placed on the data lines. The data is then latched by toggling the hardware enable pin, low, then high.

Table 4.1 Continuum and ipEngine Connections

<table>
<thead>
<tr>
<th>DB25</th>
<th>ipEngine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IA07</td>
<td>H7 (Continuum Data)</td>
</tr>
<tr>
<td>2</td>
<td>IA06</td>
<td>H6 (Continuum Data)</td>
</tr>
<tr>
<td>3</td>
<td>IA05</td>
<td>H5 (Continuum Data)</td>
</tr>
<tr>
<td>4</td>
<td>IA04</td>
<td>H4 (Continuum Data)</td>
</tr>
<tr>
<td>5</td>
<td>IA03</td>
<td>H3 (Continuum Data)</td>
</tr>
<tr>
<td>6</td>
<td>IA02</td>
<td>H2 (Continuum Data)</td>
</tr>
<tr>
<td>7</td>
<td>IA01</td>
<td>H1 (Continuum Data)</td>
</tr>
<tr>
<td>8</td>
<td>IA00</td>
<td>H0 (Continuum Data)</td>
</tr>
<tr>
<td>9</td>
<td>IA13</td>
<td>*RESET</td>
</tr>
<tr>
<td>10</td>
<td>IA12</td>
<td>*HEN (Hardware Enable)</td>
</tr>
<tr>
<td>11</td>
<td>IA11</td>
<td>R/W*</td>
</tr>
<tr>
<td>12</td>
<td>IA10</td>
<td>HA2 (Continuum Address)</td>
</tr>
<tr>
<td>13</td>
<td>IA9</td>
<td>HA1 (Continuum Address)</td>
</tr>
<tr>
<td>25</td>
<td>IA8</td>
<td>HA0 (Continuum Address)</td>
</tr>
<tr>
<td>14-24</td>
<td>GND</td>
<td>Ground</td>
</tr>
</tbody>
</table>

4.3 Processing

Processing is done by the ipEngine as shown in Figure 4.1. The ipEngine is responsible for all interfacing between components of the system as well as processing and preparing data. Key features of the ipEngine are:

- 66 MIPS Power PC CPU
- 16-MB DRAM
- 4-MB Flash
- 10Base-T Ethernet
Most of the description of what the ipEngine does is covered in the description of the software in Chapter 5. However, there is an interesting aspect of the ipEngine that is useful to cover as hardware, namely, the FPGA. The FPGA on the ipEngine is an Altera EPF6016QC used to implement memory mapped I/O. The advantage of this is that the user can configure the FPGA on the fly to perform a number of functions. To configure the device VHDL implementing the “virtual I/O” interface, provided by Brightstar, was compiled using free tools from Altera. The bit file generated by this process was then saved as an array of bytes separated by commas as shown in Appendix I. Programming the FPGA then simply involves clocking a serial bit stream out to the device using the FPGA Config register shown in Table 4.2 and the FPGA configuration code also in Appendix I.
Table 4.2 ipEngine-1 Memory Map [13]

<table>
<thead>
<tr>
<th>Address Range (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000.0000 - 0x00FF.FFFF</td>
<td>16MB DRAM</td>
</tr>
<tr>
<td>0xFC00.0000 - 0xFC7F.FFFF</td>
<td>8MB FPGA Space</td>
</tr>
<tr>
<td>0xFE00.0000 - 0xFE3F.FFFF</td>
<td>4MB Flash</td>
</tr>
<tr>
<td>0xFF00.0000 - 0xFF00.3FFF</td>
<td>MPC823 On-Chip Registers</td>
</tr>
<tr>
<td>0xFF01.0000 - 0xFF01.0000</td>
<td>FPGA Config Register</td>
</tr>
<tr>
<td>0xFF02.0000 - 0xFF02.0000</td>
<td>Clock Synth Register</td>
</tr>
</tbody>
</table>

4.4 Output

4.4.1 MIDI board

The MIDI board was designed and constructed according to the schematics and gerber plots shown in Appendix A to provide simple user I/O as well as MIDI in and out capability for the continuum [17]. Interfacing to the ipEngine uses GPIO as well as the ipEngine's second serial port as shown in Table 4.3. Reading buttons and writing LEDs simply involves reading or writing to the corresponding GPIO memory locations. LEDs need to be provided inverted input. Reading and writing MIDI simply involves transfers to the serial port at MIDI data rates.

Table 4.3 MIDI Board and ipEngine Connections

<table>
<thead>
<tr>
<th>7x2</th>
<th>ipEngine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>RSRX2</td>
<td>UART0_RX (MIDI IN)</td>
</tr>
<tr>
<td>3</td>
<td>RSTX2</td>
<td>UART0_TX (MIDI OUT)</td>
</tr>
<tr>
<td>5</td>
<td>IA34</td>
<td>UART0_DSR (Button 1)</td>
</tr>
<tr>
<td>6</td>
<td>IA35</td>
<td>UART0_CTS (Button 2)</td>
</tr>
<tr>
<td>7</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>8</td>
<td>IA32</td>
<td>UART0_DTR (LED 1)</td>
</tr>
<tr>
<td>9</td>
<td>IA33</td>
<td>UART0_RTS (LED 2)</td>
</tr>
<tr>
<td>14</td>
<td>Vcc</td>
<td>Vcc Offboard</td>
</tr>
</tbody>
</table>
4.4.2 DAC board

The DAC board performs the task of taking samples processed by the ipEngine and outputting them to one of 12 analog channels. The entire board was entered into Orcad Capture. The board layout was done with Orcad Capture as well.

There are four layers in the board: two signal layers and two power planes. Signals are routed on the outer two layers and power and ground layers are on the inner layers. Ground and power planes are separate for the digital and analog sections of the board. The digital and analog ground planes are connected beneath each of the three DACs to minimize ground differential at the devices. The power plane was separated into three regions to provide 5 V to the digital components as well as the +15 V and -15 V to the differential line drivers. Decoupling capacitors are used on all the power pins, as close as possible, to the ICs to ensure minimum noise.

Gerber files were produced and sent to Advanced Circuits for fabrication. The board was then assembled by hand by the author.

The heart of the DAC board is a Xilinx FPGA [18]. The use of an FPGA in the design allows the speed and capabilities of hardware with versatility of software. This allows the DAC board to be very flexible to adapt to the changing needs of the Continuum interface as well as those of unforeseen applications. In the case of the Continuum, the FPGA provides an identical parallel interface for communication with the ipEngine as shown in Table 4.4. Sample data is written to the FPGA, and based on the version of the VHDL installed, can be directly written out to the DAC output or placed in a FIFO to be read out by demand.

Communication and control of the DAC board is accomplished by reading and writing to a set of registers as shown in Table 4.5. The process for reading a DAC board register begins with setting the data direction pins on the ipEngine interface for input. The address to be read (see Appendix G for function table) is placed on the address pins and the hardware enable is brought low to enable the DAC board output. The corresponding register is read from the DAC board's data lines and the hardware enable set back to high.
Table 4.4 DAC Board and ipEngine Connections

<table>
<thead>
<tr>
<th>DB25</th>
<th>ipEngine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IA23</td>
<td>H7 (DAC Data)</td>
</tr>
<tr>
<td>2</td>
<td>IA22</td>
<td>H6 (DAC Data)</td>
</tr>
<tr>
<td>3</td>
<td>IA21</td>
<td>H5 (DAC Data)</td>
</tr>
<tr>
<td>4</td>
<td>IA20</td>
<td>H4 (DAC Data)</td>
</tr>
<tr>
<td>5</td>
<td>IA19</td>
<td>H3 (DAC Data)</td>
</tr>
<tr>
<td>6</td>
<td>IA18</td>
<td>H2 (DAC Data)</td>
</tr>
<tr>
<td>7</td>
<td>IA17</td>
<td>H1 (DAC Data)</td>
</tr>
<tr>
<td>8</td>
<td>IA16</td>
<td>H0 (DAC Data)</td>
</tr>
<tr>
<td>9</td>
<td>IA29</td>
<td>*RESET</td>
</tr>
<tr>
<td>10</td>
<td>IA28</td>
<td>*HEN (Hardware Enable)</td>
</tr>
<tr>
<td>11</td>
<td>IA27</td>
<td>R/W*</td>
</tr>
<tr>
<td>12</td>
<td>IA26</td>
<td>HA2 (DAC Address)</td>
</tr>
<tr>
<td>13</td>
<td>IA25</td>
<td>HA1 (DAC Address)</td>
</tr>
<tr>
<td>25</td>
<td>IA24</td>
<td>HA0 (DAC Address)</td>
</tr>
<tr>
<td>14-24</td>
<td>GND</td>
<td>Ground</td>
</tr>
</tbody>
</table>

Writing is also straightforward. The data direction pins on the ipEngine interface are set for output. Next, the address to be written to is placed on the address lines. The data is then latched by toggling the hardware enable pin, low, then high.

Table 4.5 DAC Board Registers

<table>
<thead>
<tr>
<th>HA2_0 (Hex)</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>ID</td>
</tr>
<tr>
<td>0x01</td>
<td>Command</td>
</tr>
<tr>
<td>0x02</td>
<td>Status</td>
</tr>
<tr>
<td>0x04</td>
<td>Data(7.0)</td>
</tr>
<tr>
<td>0x05</td>
<td>Data(15.8)</td>
</tr>
<tr>
<td>0x06</td>
<td>Data(23.16)</td>
</tr>
</tbody>
</table>

There are six registers in the current DAC board interface. The first is the ID register. The ID register is a read-only register that contains a hard coded device ID for version control and hardware identification. Next is the command register, which can be both read and written. Although reading the command register serves no real purpose, it is
useful for read-back and diagnostic purposes. Writing to the command register issues a
command. Commands are listed in Table 4.6.

Table 4.6 DAC Board Commands

<table>
<thead>
<tr>
<th>Hex</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>Load FIFO</td>
<td>Writes contents of data registers to FIFO.</td>
</tr>
<tr>
<td>0x02</td>
<td>Start Output</td>
<td>Starts DAC consumer process.</td>
</tr>
<tr>
<td>0x03</td>
<td>Stop Output</td>
<td>Stops DAC consumer process.</td>
</tr>
<tr>
<td>0x04</td>
<td>Clear FIFO</td>
<td>Flushes FIFO contents.</td>
</tr>
<tr>
<td>0x05</td>
<td>Change RSTSEL</td>
<td>Toggles DAC reset mode.</td>
</tr>
<tr>
<td>0x06</td>
<td>Read Buttons</td>
<td>Reads pushbutton data into register.</td>
</tr>
<tr>
<td>0x07</td>
<td>Output LEDs</td>
<td>Outputs data register to LEDs.</td>
</tr>
<tr>
<td>0x10</td>
<td>Reset DACs</td>
<td>Toggle DAC reset.</td>
</tr>
<tr>
<td>0x11</td>
<td>Read FIFO</td>
<td>Reads word from FIFO to data registers.</td>
</tr>
<tr>
<td>0x12</td>
<td>Write DAC</td>
<td>Writes word to DAC register.</td>
</tr>
<tr>
<td>0x13</td>
<td>Update DACs</td>
<td>Updates DAC outputs.</td>
</tr>
</tbody>
</table>

The commands that need data require that the data already be in the appropriate
data registers when the command is issued. Those that provide data place data in the
appropriate data registers after the command is issued to be read by subsequent read
commands from the ipEngine.

The first command, Load FIFO, expects data to be in all three data registers in the
DAC format shown in Table 4.7, Table 4.8, and Table 4.9. Bits 15 through 0 are the 16
bits of the sample to be output. Bits 17 and 16 select one of the three DACs, while bits
23 and 22 select one of the four channels on the selected DAC. Bit 21 writes the data
to all four channels of a DAC if selected. Finally, bit 20 is set to indicate that the given
sample is the last sample of a series to be updated for the given period of the update
clock. Upon issue the data is placed into the FIFO. Read FIFO reads the FIFO data
into the three data registers.

The Start Output command sets the run flag high, thus starting the DAC consumer
process, which reads data from the FIFO and updates the DAC output. The Stop Output
command clears the run flag high, thus stopping the consumer process.
Table 4.7 DAC Serial Data (7:0)

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 7</td>
<td>Data 6</td>
<td>Data 5</td>
<td>Data 4</td>
<td>Data 3</td>
<td>Data 2</td>
<td>Data 1</td>
<td>Data 0</td>
</tr>
</tbody>
</table>

Table 4.8 DAC Serial Data (15:8)

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 15</td>
<td>Data 14</td>
<td>Data 13</td>
<td>Data 12</td>
<td>Data 11</td>
<td>Data 10</td>
<td>Data 9</td>
<td>Data 8</td>
</tr>
</tbody>
</table>

Table 4.9 DAC Serial Data (23:16)

<table>
<thead>
<tr>
<th>Bit 23</th>
<th>Bit 22</th>
<th>Bit 21</th>
<th>Bit 20</th>
<th>Bit 19</th>
<th>Bit 18</th>
<th>Bit 17</th>
<th>Bit 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr 1</td>
<td>Addr 0</td>
<td>Q_Load</td>
<td>Last</td>
<td>NC</td>
<td>NC</td>
<td>Addr 3</td>
<td>Addr 2</td>
</tr>
</tbody>
</table>
When issued, the Clear FIFO command simply toggles the reset pin of the FIFO to flush the FIFO contents.

The Read Buttons command writes the button data into the least significant four bits of the Data(7:0) register. Similarly, the Output LED command outputs the least significant four bits of the Data(7:0) register to the LEDs.

Finally, the Write DAC command simply writes the three data registers directly to DAC selected by the contents of data word. Update DAC forces the DACs to update their output when issued.

The status register, shown in Table 4.10, contains flags for detecting errors in the state machine as well as determining the current state of the FIFO and the DAC consumer process.

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>Button</td>
<td>Rstsel</td>
<td>Run</td>
<td>Empty</td>
<td>Full</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The error flag indicates that the DAC consumer process that reads data from the FIFO has run out of data. If the button flag is set, it indicates that one, or several, buttons have been pressed since the last time the buttons were checked. Button presses are sticky, meaning that the button status flag and the button presses are latched until serviced by the ipEngine. Rstsel indicates the current setting of the DAC reset select bit, which determines if the DAC resets to 0 or to the middle of range. The run bit indicates if the consumer process is running. When the FIFO is empty or full, the empty and full flags are set appropriately.

The data registers are fairly self explanatory from the above descriptions; they are both readable and writable.
5.1 Overview

The software is composed of three main functional components: C code running on the ipEngine, VHDL in the Altera FPGA on the ipEngine, and VHDL in the Xilinx FPGA on the DAC Board.

5.2 ipEngine Code

5.2.1 Direct DAC updates

In order to test the functioning of the DACs without yet implementing the producer and consumer routines to provide data, the code in Appendix I was written in conjunction with VHDL to use the ipEngine to do direct writes and updates to the DAC registers. Figure 5.1 shows the high-level overview of the software flow. The test program simply outputs a square wave to the first DAC channel.

The software starts by initializing the system. This involves toggling the reset bit to the DAC and setting up proper data directions for the DAC board interface.

Next the program writes 0x00.FFFF to the data registers. This implies that DAC channel one gets full scale output. The write command is then issued, latching the data into the DAC. The update command is issued to update the DACs output.
**Figure 5.1** Direct DAC Update Software Flow
After waiting a set amount of time, the above process repeats, this time with 0x00.0000 in the data register. After waiting again, the whole cycle repeats.

5.2.2 FIFO DAC updates

The next code segment presented in Appendix I updates the DAC through a FIFO based producer-consumer process. The FIFO is intended for use in waveform smoothing. Two different cores were tried, one from Xilinx and the one currently implemented from the Free-IP Project [19], [20]. Figure 5.2 shows the high-level overview of the software flow.

The code starts by initializing the system as discussed earlier. It then proceeds to prefill the FIFO with waveform data. The start command is then issued, starting the consumer process and DAC output. If the consumer runs out of data, the program halts. Otherwise, so long as the consumer keeps requesting data, the program checks the status register on the DAC board to see if the FIFO is full. If it is, the program waits for the consumer to catch up. Otherwise, if it is not empty, sample data is placed into the data registers and a Load FIFO command is issued. The sample data is then incremented and the process repeats until the consumer stops running. The resulting output is a ramped waveform.

5.2.3 MIDI output

In the final system the ipEngine would be responsible for encoding the Continuum data into MIDI and transmitting via the serial port. The second serial port was configured for testing and the Continuum MIDI encoding established [7].

The MIDI encoding used by the Continuum Fingerboard assigns each finger on the playing surface its own MIDI channel. It tracks up to 10 fingers, and uses MIDI channels 1-9 (n=0..8 in the lists below) and channel 12 (n=11 below). The exact pitch of the finger is encoded as the nearest MIDI note number to the initial position of the finger, plus the Pitch Bend value. Note numbers range from 15 to 109 on a full-size Continuum, and from 40 to 85 on a half-size version.
Initialize System

Pre-fill FIFO:
Write Ramp Sample to Data Register then Issue Load Fifo Command for X Samples

Issue Start Output Command

Consumer Process Running?

Yes

Is FIFO Full?

Yes

No

Write Sample to Data register

Issue Load FIFO Command

Yes

No

Increment Sample

Done

Figure 5.2 FIFO DAC Update Software Flow
The Pitch Bend range is two octaves. The pressure of the finger is encoded (by default) using the Channel Volume controller, and the front-to-back position is encoded (by default) using the Cutoff Frequency controller. The Continuum Fingerboard can be configured to round off pitches to the nearest half step, to use fixed pressure (volume) values, or to round the front-to-back position of fingers. It can also be configured to encode finger pressure as Polyphonic Key Pressure, Channel Pressure, or Expression Controller. A variety of controllers and value ranges are configurable for the encoding of the front-to-back position of each finger.

When the Continuum Fingerboard is idle (no fingers on the playing surface), it transmits the following data once per second to set the pitch bend range on each MIDI channel (n is the MIDI channel, MIDI data are shown in hex):

- \( B_n 65\ 00 \)  RPN low for Set Pitch Bend range
- \( B_n 64\ 00 \)  RPN high for Set Pitch Bend range
- \( B_n 06\ 18 \)  Pitch Bend range is 24 half steps
- \( B_n 65\ 7F \)  RPN low null
- \( B_n 64\ 7F \)  RPN high null

The Continuum Fingerboard assigns each note its own MIDI channel. When a finger first touches the playing surface, it transmits a Note On:

- \( 9n\ rr\ 7F \)  Note On, \( rr \) is nearest MIDI note number

When a finger is lifted from the playing surface it transmits a Note Off:

- \( 8n\ rr\ 00 \)  Note Off, \( rr \) is same value as in Note On

The following three MIDI messages are used to track finger movements. These three messages are always transmitted in the same order. They are transmitted continually between Note On and Note Off, and once immediately before the Note On and immediately after the Note Off:
Bn \text{ mm kk} \quad \text{Front-to-back position}

Default controller is Cutoff (mm=4A).
Default range is kk=00..40

Bn 07 vv \quad \text{Finger pressure, vv=00..7F}

This is the default pressure encoding; see below for other encodings.

En pp pp \quad \text{Pitch Bend, pppp=0000..3FFF}

Pitch Bend uses a 24 half-step range

The Continuum Fingerboard's configuration can be changed using MIDI, by sending it Note On messages on channel 16. The MIDI pitch in the Note On messages will configure the Continuum Fingerboard according to the lists below. It is harmless to send other MIDI data to the Continuum Fingerboard, but only Note On messages on channel 16 will have any effect. Select the encoding for finger pressure using pitches two octaves below middle C:

<table>
<thead>
<tr>
<th>PITCH</th>
<th>SELECTS THIS PRESSURE ENCODING</th>
<th>PRESSURE BYTE CODES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ab2</td>
<td>volume controller (default)</td>
<td>Bn 07 vv</td>
</tr>
<tr>
<td>A2</td>
<td>expression controller</td>
<td>Bn 0B vv</td>
</tr>
<tr>
<td>Bb2</td>
<td>channel pressure</td>
<td>Dn vv</td>
</tr>
<tr>
<td>B2</td>
<td>polyphonic key pressure</td>
<td>An rr vv</td>
</tr>
</tbody>
</table>

Select front-to-back controller number (mm in the description above) using the octave below middle C:

<table>
<thead>
<tr>
<th>PITCH</th>
<th>SELECTS FRONT-BACK ENCODING</th>
<th>CONTROLLER NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>modulation depth controller</td>
<td>mm=01</td>
</tr>
<tr>
<td>C#3</td>
<td>pan controller</td>
<td>mm=0A</td>
</tr>
<tr>
<td>D3</td>
<td>filter resonance</td>
<td>mm=47</td>
</tr>
<tr>
<td>Eb3</td>
<td>attack time</td>
<td>mm=49</td>
</tr>
<tr>
<td>E3</td>
<td>filter cutoff (default)</td>
<td>mm=4A</td>
</tr>
<tr>
<td>F3</td>
<td>decay time</td>
<td>mm=4B</td>
</tr>
</tbody>
</table>
F#3  reverb
G3  chorus

Select rounding options and the front-to-back value range using the middle C octave:

**PITCH**  SELECTS THIS OPTION
C4  round pitches to nearest half step
C#4  exact pitches (default)
D4  use pp dynamic, round to front or back
Eb4  use mf dynamic, round to front or back
E4  use ff dynamic, round to front or back
F4  exact dynamic and front-back (default)
F#4  front-back range kk=00..40 (default)
G4  front-back range kk=40..7F
Ab4  front-back range kk=00..7F
A4  front-back range kk=7F..00 (reverse)

Select the polyphony (number of simultaneous notes) using the octave above the middle C octave:

**PITCH**  SELECTS THIS POLYPHONY
C5  one note at a time (use n=0 only)
C#5  maximum 2 simultaneous notes (n=0..1)
D5  maximum 3 simultaneous notes (n=0..2)
Eb5  maximum 4 simultaneous notes (n=0..3)
E5  maximum 5 simultaneous notes (n=0..4)
F5  maximum 6 simultaneous notes (n=0..5)
F#5  maximum 7 simultaneous notes (n=0..6)
G5  maximum 8 simultaneous notes (n=0..7)
Ab5  maximum 9 simultaneous notes (n=0..8)
A5  maximum 10 notes (default; n=0..8 and n=11)
5.3 ipEngine VHDL

The VHDL code running on the Altera FPGA is sample code provided by Brightstar Engineering to implement "virtual I/O." Table 5.1 shows the resulting memory map for the GPIO used in this thesis.

Table 5.1 Alter FPGA Memory Map

<table>
<thead>
<tr>
<th>Address Range (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFC00.0000</td>
<td>GPIO Data Reg IA15_00</td>
</tr>
<tr>
<td>0xFC00.0002</td>
<td>GPIO Data Reg IA31_16</td>
</tr>
<tr>
<td>0xFC00.0004</td>
<td>GPIO Data Reg IA43_32</td>
</tr>
<tr>
<td>0xFC00.0006</td>
<td>GPIO Data Reg IB15_00</td>
</tr>
<tr>
<td>0xFC00.0008</td>
<td>GPIO Data Reg IB31_16</td>
</tr>
<tr>
<td>0xFC00.000A</td>
<td>GPIO Data Reg IB43_32</td>
</tr>
<tr>
<td>0xFC00.000C</td>
<td>GPIO Data Dir Reg IA15_00</td>
</tr>
<tr>
<td>0xFC00.000E</td>
<td>GPIO Data Dir Reg IA31_16</td>
</tr>
<tr>
<td>0xFC00.0010</td>
<td>GPIO Data Dir Reg IA43_32</td>
</tr>
<tr>
<td>0xFC00.0012</td>
<td>GPIO Data Dir Reg IB15_00</td>
</tr>
<tr>
<td>0xFC00.0014</td>
<td>GPIO Data Dir Reg IB31_16</td>
</tr>
<tr>
<td>0xFC00.0016</td>
<td>GPIO Data Dir Reg IB43_32</td>
</tr>
</tbody>
</table>

5.4 DAC Board VHDL

5.4.1 Direct DAC updates

The VHDL listed in Appendix J implements a state machine on the DAC board’s FPGA for implementing the parallel interface, command parsing, and direct DAC updates. Figure 5.3 shows the high-level overview of the software flow.

The state machine starts the process by clearing all registers and setting flag and synch bits to their default values. If there is no activity on the parallel port (hardware enable is high) then the status bits are checked to see if a button was pressed. If yes, the button register is ORed with the button value and the button flag is set in the status register. If no, the state returns to idle. If there is port activity (hardware enable is low),
Figure 5.3 Direct DAC Update State Machine
then the machine proceeds to the decode port address state. Here the address is decoded and commands parsed as shown in Figure 5.4 and discussed in Chapter 4.

If the parsed command is for a DAC load, the state machine in Figure 5.5 is executed. The DAC loading machine starts by enabling the chip select of the DAC as selected by the address bits of the DAC data. The data is then shifted out, serially, to all the DACs since only the selected one will accept the data. After the data is entirely shifted out, the address pins are again used to disable the proper chip select.

After the state machine has returned from address decoding or command parsing, the state machine waits until the port's hardware enable returns high before going back to the idle state.

5.4.2 FIFO DAC updates

Again, the VHDL listed in Appendix J implements a state machine on the DAC board's FPGA for implementing the parallel interface, command parsing. However, this version updates the DACs through a FIFO based producer-consumer process. Figure 5.6 shows the high-level overview of the software flow.

The flow of the software is much the same as before, only this time the command decoding has some new command as shown in Figure 5.7 and discussed in Chapter 4, and the machine checks to see if the consumer process in Figure 5.8 has been started and is requesting data before checking the buttons.

The consumer signals a request for data by toggling its synch bit. When this is detected by the producer process it sets its own synch bit to match. The producer will then only provide data from the FIFO to the consumer process when the synch bits are no longer equal. If the FIFO is empty, the error flag is set and the run flag is cleared, thus stopping the consumer. Otherwise, if there is data in the FIFO, the FIFO data is shifted out to the DAC by the DAC loading state machine. This process repeats until data from the FIFO has the last update bit set, or until the FIFO empties itself. After the last update bit is reached, the state machine returns to the idle state and awaits another request.
Figure 5.4 Command Decode State Machine for Direct Update
Figure 5.5 DAC Loading State Machine.
Figure 5.6 FIFO DAC Update State Machine
**Figure 5.7 Command Decode State Machine for FIFO Update**
Figure 5.8 DAC Consumer State Machine for FIFO Update.
Finally, the consumer process is very simple. If the run bit is set, then the machine waits until the DAC update clock goes high. On the rising edge of the clock, it toggles the DAC's update pins to update all the DACs. Next, it toggles its synch bit to request more data from the consumer before returning to the idle state to wait for the next update clock.
CHAPTER 6

CONCLUSION

A system for providing MIDI and reconfigurable analog output to a new musical instrument, the Continuum Fingerboard, has been presented. The proposed hardware fulfills the initial requirements of internalization of the surface scanning functions as well as MIDI and analog output.

All hardware for the final DAC board was successfully designed, built, and tested. Initial testing shows that both the MIDI board and DAC board function as designed.

Use of reconfigurable hardware for the DAC board makes future revisions to the design a simple process of updating the VHDL of the DAC board FPGA. Future improvements might include asynchronous FIFOs, extended command sets including user I/O, higher resolution, and higher output rates.
APPENDIX A

MIDI BOARD SCHEMATIC

Figure A.1 shows the MIDI board schematic.
Figure A.2 shows the MIDI layout (all layers).
Figure A.3 shows the MIDI layout (top layer).
Figure A.4 shows the MIDI layout (bottom layer).
Figure A.5 shows the MIDI layout (top silk screen layer).
Figure A.6 shows the MIDI layout (drill layer).
Figure A.1 MIDI Board Schematic
**Figure A.2 MIDI Layout (All Layers)**

<table>
<thead>
<tr>
<th>SYM</th>
<th>DIAM</th>
<th>TOL</th>
<th>QTY</th>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.015</td>
<td></td>
<td>30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.037</td>
<td></td>
<td>28</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.055</td>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.075</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.110</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td></td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>SYM</td>
<td>DIAM</td>
<td>TOL</td>
<td>QTY</td>
<td>NOTE</td>
</tr>
<tr>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td>•</td>
<td>0.015</td>
<td></td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>×</td>
<td>0.037</td>
<td></td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0.055</td>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>♦</td>
<td>0.075</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>☰</td>
<td>0.110</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td>89</td>
<td></td>
</tr>
</tbody>
</table>

Figure A.6 MIDI Layout (Drill Layer)
APPENDIX B

MIDI BOARD BOM

Table B.1 provides the bill of materials for the schematic shown in Appendix A, Figure A.1.

Table B.1: MIDI Board Bill of Materials

<table>
<thead>
<tr>
<th>RefDes</th>
<th>Manufacturer ID: Distributor ID:</th>
<th>Description</th>
<th>QTY</th>
<th>Unit Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>U7</td>
<td>NEC:PS9711 Digikay: PS9711-ND</td>
<td>Hi Speed Optocoupler (SOP)</td>
<td>1</td>
<td>$2.35</td>
</tr>
<tr>
<td>U2, U6</td>
<td>Fairchild: 74ACTQ245SC Digikay: 74ACTQ245SC-ND</td>
<td>IC Fast Trans Octal Bidir (S0IC)</td>
<td>2</td>
<td>$1.73</td>
</tr>
<tr>
<td>JP2</td>
<td>Molex(GC/Walcom): 26-60-4050 Digikay: WM4623-ND</td>
<td>Conn Hous 5 Pos .156&quot; w/Polar</td>
<td>1</td>
<td>$0.52</td>
</tr>
<tr>
<td></td>
<td>Molex(GC/Walcom): 09-50-8053 Digikay: WM2114-ND</td>
<td>Conn Hous 5 Pos .156&quot; w/Polar</td>
<td>1</td>
<td>$0.42</td>
</tr>
<tr>
<td></td>
<td>Molex(GC/Walcom): 08-52-0113 Digikay: WM2313-N</td>
<td>Term Crimp .156&quot; Tin Trifurcon</td>
<td>5</td>
<td>$0.09</td>
</tr>
<tr>
<td>U3</td>
<td>Texas Instruments: SN74LS05D Digikay: 296-1630-6-ND</td>
<td>IC Hex Inverter w/oc (14-SOP)</td>
<td>1</td>
<td>$0.60</td>
</tr>
<tr>
<td>U1, U4, U5</td>
<td>CUI Stack: SDF-503 Digikay: CP-7050-ND</td>
<td>Conn DIN 5Pos Female Shielded</td>
<td>3</td>
<td>$0.78</td>
</tr>
<tr>
<td>SW1, SW2</td>
<td>CK Components: EP115D1ABE Digikay: CKN4007-ND</td>
<td>Tiny Pushbutton SPST R/H/Angle</td>
<td>2</td>
<td>$1.92</td>
</tr>
<tr>
<td></td>
<td>CK Components: 465802000 Digikay: CKN1111-ND</td>
<td>Swit Cap Blk for..100 PB</td>
<td>2</td>
<td>$0.45</td>
</tr>
<tr>
<td>D1</td>
<td>Chicago Miniature Lamp: 5302H5-5V Digikay: L20025-ND</td>
<td>PC Board Mat LED Green</td>
<td>1</td>
<td>$0.57</td>
</tr>
<tr>
<td>D2</td>
<td>Chicago Miniature Lamp: 5302HJ-5V Digikay: L20021-ND</td>
<td>PC Board Mat LED Red</td>
<td>1</td>
<td>$0.57</td>
</tr>
<tr>
<td>R1-R9, R12</td>
<td>220 Ohm Resistor (1206 SMD)</td>
<td>10</td>
<td>$0.00</td>
<td></td>
</tr>
<tr>
<td>R10, R11</td>
<td>1k Ohm Resistor (1206 SMD)</td>
<td>2</td>
<td>$0.00</td>
<td></td>
</tr>
<tr>
<td>C2-C6</td>
<td>0.1 Microfarad Capacitor (1206 SMD)</td>
<td>5</td>
<td>$0.00</td>
<td></td>
</tr>
</tbody>
</table>

46
Table B.1: (Continued)

<table>
<thead>
<tr>
<th>RefDes</th>
<th>Manufacturer ID: Distributor ID:</th>
<th>Description (Package)</th>
<th>QTY</th>
<th>Unit Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td></td>
<td>1 Microfarad Capacitor (1206 SMD)</td>
<td>1</td>
<td>$0.00</td>
</tr>
<tr>
<td>JP1</td>
<td></td>
<td>7x2 .100 Header</td>
<td>1</td>
<td>$0.00</td>
</tr>
<tr>
<td>D3</td>
<td></td>
<td>Small Signal Diode</td>
<td>1</td>
<td>$0.00</td>
</tr>
</tbody>
</table>
APPENDIX C

DAC BOARD VERSION 1.0 SCHEMATIC

Figure C.1 shows the DAC board version 1.0 schematic.
Figure C.2 shows the DAC board version 1.0 layout (all layers).
Figure C.3 shows the DAC board version 1.0 layout (top silk screen layer).
Figure C.4 shows the DAC board version 1.0 layout (drill layer).
Table D.1 provides the bill of materials for the schematic shown in Appendix C, Figure C.1.

Table D.1: DAC Board Version 1.0 Bill of Materials

<table>
<thead>
<tr>
<th>RefDes</th>
<th>Manufacturer ID:</th>
<th>Description (Package)</th>
<th>QTY</th>
<th>Unit Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>U32</td>
<td>NEC-PS9711</td>
<td>Hi Speed Optocoupler (SOP)</td>
<td>1</td>
<td>$2.35</td>
</tr>
<tr>
<td>U1, U8</td>
<td>Fairchild: 74ACTQ245SC</td>
<td>IC Fast Trans Octal Bidir (SOIC)</td>
<td>2</td>
<td>$1.73</td>
</tr>
<tr>
<td>JPT1</td>
<td>Molex(GC/Walcom): 26-60-4050</td>
<td>Conn: Hous 5 Pos .156&quot; w/Polar</td>
<td>1</td>
<td>$0.42</td>
</tr>
<tr>
<td></td>
<td>Molex(GC/Walcom): 09-50-8053</td>
<td>Conn Hous 5 Pos .156&quot; w/Polar</td>
<td>1</td>
<td>$0.42</td>
</tr>
<tr>
<td></td>
<td>Molex(GC/Walcom): 08-52-0113</td>
<td>Term Crimp .156&quot; Tin Trifurcon</td>
<td>5</td>
<td>$0.09</td>
</tr>
<tr>
<td>U18, U26</td>
<td>Texas Instruments: SN74LS05D</td>
<td>IC Hex Inverter w/oc (14-SOP)</td>
<td>1</td>
<td>$0.60</td>
</tr>
<tr>
<td></td>
<td>CUI Stack: SDF-50J</td>
<td>Conn DIN 5Pos Female Shielded</td>
<td>3</td>
<td>$0.78</td>
</tr>
<tr>
<td>D1, D2</td>
<td>Diodes Inc.: SMCJ15A-13</td>
<td>TVS 1500W 15.0V Uni-Dir (SMC)</td>
<td>2</td>
<td>$1.52</td>
</tr>
<tr>
<td>D3</td>
<td>Diodes Inc.: SMCJ5.0CA-13</td>
<td>TVS 1500W 5.0V Bi-Dir (SMC)</td>
<td>1</td>
<td>$1.68</td>
</tr>
<tr>
<td>U31</td>
<td>National: LM3405S-3.3</td>
<td>IC +5.0V 1.5A VREG (TO-263)</td>
<td>1</td>
<td>$1.65</td>
</tr>
<tr>
<td>P1</td>
<td>AMP: 750801-4</td>
<td>Stacked D-Sub Conn 25Pin Plug</td>
<td>1</td>
<td>$14.79</td>
</tr>
<tr>
<td>R26</td>
<td>BC Components: STSTP20</td>
<td>POT 20k 6mm, Cerm Sq S/T (SMD)</td>
<td>1</td>
<td>$4.19</td>
</tr>
<tr>
<td></td>
<td>Burr Brown: DRV135UA</td>
<td>IC Audio Dir Line Driver (8-SOIC)</td>
<td>12</td>
<td>$</td>
</tr>
<tr>
<td>U2, U3, U15</td>
<td>Burr Brown: DAC7734EC</td>
<td>IC Quad Ser 16B Vout D/A (48-SSOP)</td>
<td>3</td>
<td>$57.00</td>
</tr>
<tr>
<td>J10</td>
<td>AMP: 555163-1</td>
<td>Mod Jack 6-6 Low Profile w/Sto</td>
<td>1</td>
<td>$0.91</td>
</tr>
<tr>
<td>J1, J9, J11-13</td>
<td>Switchcraft: SN40B-14B</td>
<td>Phone Jack Rf Angle</td>
<td>12</td>
<td>$1.26</td>
</tr>
</tbody>
</table>

1U14, U19, U24, U27, U29, U30
2U4, U7, U9, U10, U12, U13, U16, U20, U21, U25

53
<table>
<thead>
<tr>
<th>RefDes</th>
<th>Manufacturer ID:</th>
<th>Description (Package)</th>
<th>QTY</th>
<th>Unit Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>U28</td>
<td>Burr Brown: REF102AU</td>
<td>Precision 10V Reference (8-SSOP)</td>
<td>1</td>
<td>$5.28</td>
</tr>
<tr>
<td></td>
<td>Digikey: REF102AU-ND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U11</td>
<td>Microchip: PIC16F877-20/L</td>
<td>Micro Ctrl Flash 8K (44-PDIP)</td>
<td>1</td>
<td>$70.05</td>
</tr>
<tr>
<td></td>
<td>Digikey: PIC16F877-20/L-ND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U22, U23</td>
<td>Texas Instruments: SN74F374DBR</td>
<td>Oct D Flip Flop (20-SSOP)</td>
<td>2</td>
<td>$0.63</td>
</tr>
<tr>
<td></td>
<td>Digikey: 296-3387-1-ND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U17</td>
<td>Cirrus Logic: CL-CD1865-10QC-B</td>
<td>8 Channel UART</td>
<td>1</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>220 Ohm Resistor (1206 SMD)</td>
<td>15</td>
<td>$0.00</td>
</tr>
<tr>
<td>R35-R42</td>
<td></td>
<td>230 Ohm Resistor (1206 SMD)</td>
<td>8</td>
<td>$0.00</td>
</tr>
<tr>
<td>R47-R55</td>
<td></td>
<td>1k Ohm Resistor (1206 SMD)</td>
<td>8</td>
<td>$0.00</td>
</tr>
<tr>
<td>R1-R8, R13-R20</td>
<td></td>
<td>22 Ohm Resistor (1206 SMD)</td>
<td>15</td>
<td>$0.00</td>
</tr>
<tr>
<td>R25-R53</td>
<td></td>
<td>1M Ohm Resistor (1206 SMD)</td>
<td>2</td>
<td>$0.00</td>
</tr>
<tr>
<td>R43-R45</td>
<td></td>
<td>1.5k Ohm Resistor (1206 SMD)</td>
<td>3</td>
<td>$0.00</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Filter Resistor (1206 SMD)</td>
<td>12</td>
<td>$0.00</td>
</tr>
<tr>
<td>C1-C8, C11-C14</td>
<td></td>
<td>Filter Capacitor (1206 SMD)</td>
<td>12</td>
<td>$0.00</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0.1 Microfarad Capacitor (1206 SMD)</td>
<td>4</td>
<td>$0.00</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1 Microfarad Capacitor (1206 SMD)</td>
<td>3</td>
<td>$0.00</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>0.01 Microfarad Capacitor (1206 SMD)</td>
<td>52</td>
<td>$0.00</td>
</tr>
<tr>
<td>C47</td>
<td></td>
<td>0.22 Microfarad Capacitor (1206 SMD)</td>
<td>1</td>
<td>$0.00</td>
</tr>
<tr>
<td>C9,C10</td>
<td></td>
<td>15 Picofarad Capacitor (1206 SMD)</td>
<td>1</td>
<td>$0.00</td>
</tr>
<tr>
<td>JP1</td>
<td></td>
<td>8x2 .100 Header</td>
<td>1</td>
<td>$0.00</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>10x2 .100 Header</td>
<td>3</td>
<td>$0.00</td>
</tr>
</tbody>
</table>

3 R27, R28, R31, R32, R33, R46, R56-R61, R64-R67
4 R9, R10, R11, R12, R21, R22, R23, R24, R26, R30, R34, R47
5 C28, C31, C48, C50
6 C27, C30, C39, C49
7 C29, C32, C51, C15-C26, C33-C38, C40-C46, C52-C63, C64-C75
8 JP2, JP3, JP7

54
### Table D.1: (Continued)

<table>
<thead>
<tr>
<th>RefDes</th>
<th>Manufacturer ID: Distributor ID:</th>
<th>Description (Package)</th>
<th>QTY</th>
<th>Unit Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td></td>
<td>5x2 .100 Header</td>
<td>3</td>
<td>$0.00</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1x2 .100 Header</td>
<td>6</td>
<td>$0.00</td>
</tr>
<tr>
<td>D4</td>
<td></td>
<td>Small Signal Diode</td>
<td>1</td>
<td>$0.00</td>
</tr>
<tr>
<td>Y1</td>
<td></td>
<td>10Mhz Crystal</td>
<td>1</td>
<td>$0.00</td>
</tr>
</tbody>
</table>

APPENDIX E

DAC BOARD VERSION 2.0 SCHEMATIC

Figure E.1 shows the DAC board version 2.0 schematic.
Figure E.2 shows the DAC board version 2.0 layout (all layers).
Figure E.3 shows the DAC board version 2.0 layout (top layer).
Figure E.4 shows the DAC board version 2.0 layout (bottom layer).
Figure E.5 shows the DAC board version 2.0 layout (ground layer).
Figure E.6 shows the DAC board version 2.0 layout (power layer).
Figure E.7 shows the DAC board version 2.0 layout (silk screen top layer).
Figure E.8 shows the DAC board version 2.0 layout (silk screen bottom layer).
Figure E.9 shows the DAC board version 2.0 layout (drill layer).
Figure E.3 DAC Board Version 2.0 Layout (Top Layer)
Figure E.7 DAC Board Version 2.0 Layout (Silk Screen Top Layer)
Figure E.8 DAC Board Version 2.0 Layout (Silk Screen Bottom Layer)
Figure E.9 DAC Board Version 2.0 Layout (Drill Layer)
APPENDIX F

DAC BOARD VERSION 2.0 BOM

Table F.1 provides the bill of materials for the schematic shown in Appendix E, Figure E.1.

Table F.1: DAC Board Version 2.0 Bill of Materials.

<table>
<thead>
<tr>
<th>RefDes</th>
<th>Manufacturer ID:</th>
<th>Description (Package)</th>
<th>QTY</th>
<th>Unit Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>U4, U9</td>
<td>Fairchild: 74ACTQ245SC</td>
<td>IC Fast Trans Octal Bidir (50IC)</td>
<td>2</td>
<td>$1.73</td>
</tr>
<tr>
<td></td>
<td>Digikey: 74ACTQ245SC-ND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JP6</td>
<td>Molex(GC/Walcom): 26-60-4050 Digikey: WM4623-ND</td>
<td>Conn Hous 5 Pos .156'' w/Polar</td>
<td>3</td>
<td>$0.42</td>
</tr>
<tr>
<td></td>
<td>Molex(GC/Walcom): 08-52-8053 Digikey: WM2114-ND</td>
<td>Conn Hous 5 Pos .156'' w/Polar</td>
<td>1</td>
<td>$0.42</td>
</tr>
<tr>
<td></td>
<td>Molex(GC/Walcom): 08-52-0113 Digikey: WM2313-N</td>
<td>Term.Crimp .156'' Tin Trifurcon</td>
<td>5</td>
<td>$0.09</td>
</tr>
<tr>
<td>D5, D6</td>
<td>Diodes Inc.: SMCJ15A-13 Digikey: SMCJ15ADICT-ND</td>
<td>TVS 1500W 15.0V Uni-Dir (SMC)</td>
<td>2</td>
<td>$1.52</td>
</tr>
<tr>
<td></td>
<td>Diodes Inc.: SMCJ15.0OA-13 Digikey: SMCJ15.0CADICT-ND</td>
<td>TVS 1500W 5.0V Bi-Dir (SMC)</td>
<td>1</td>
<td>$1.68</td>
</tr>
<tr>
<td>U23</td>
<td>National: LM340S-5.0 Digikey: LM340S-5.0-ND</td>
<td>IC +5.0V 1.5A VREG (TO-263)</td>
<td>5</td>
<td>$1.65</td>
</tr>
<tr>
<td>P1</td>
<td>AMP: 7478424-4 Digikey: A23285-ND</td>
<td>25 MSL Plug RA 318</td>
<td>1</td>
<td>$3.73</td>
</tr>
<tr>
<td>R45</td>
<td>BC Components: STSTP20 Digikey: STSTP30CT-ND</td>
<td>POT 20k 6mm Cerm Sq S/T (SMD)</td>
<td>1</td>
<td>$4.19</td>
</tr>
<tr>
<td>2</td>
<td>Burr Brown: DAC774EC Digikey: DAC774EC-</td>
<td>IC Quad Ser 16B-Vout D/A (48-SSOP)</td>
<td>12</td>
<td>$57.00</td>
</tr>
<tr>
<td>J1-J12</td>
<td>Switchcraft: SN49B-14B Switching Devices: SN49-14B</td>
<td>Phone Jack Rt Angle</td>
<td>12</td>
<td>$12.26</td>
</tr>
<tr>
<td>U22</td>
<td>Burr Brown: REF102AU Digikey: REF102AU-ND</td>
<td>Prec. 10V Reference (8-SOIC)</td>
<td>1</td>
<td>$5.28</td>
</tr>
<tr>
<td>3</td>
<td>220 Ohm Resistor (1206 SMD)</td>
<td>4</td>
<td>$0.00</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.7k Ohm Resistor (1206 SMD)</td>
<td>13</td>
<td>$0.00</td>
<td></td>
</tr>
</tbody>
</table>

1 U2, U6, U8, U10, U12, U13, U14, U16, U19, U20, U21, U24
2 U1, U11, U18
3 R26, R28, R29, R30
4 R18, R22, R25, R32-R37, R40, R41

66
Table F.1: (Continued)

<table>
<thead>
<tr>
<th>RefDes</th>
<th>Manufacturer ID:</th>
<th>Description (Package)</th>
<th>QTY</th>
<th>Unit Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>R46</td>
<td>Panasonic: EXC-ML20A390U Digikey: P10191CT-ND</td>
<td>22 Ohm Resistor (1206 SMD)</td>
<td>16</td>
<td>$0.00</td>
</tr>
<tr>
<td>R5</td>
<td>Panasonic: EXC-ML20A390U Digikey: P10191CT-ND</td>
<td>1M Ohm Resistor (1206 SMD)</td>
<td>1</td>
<td>$0.00</td>
</tr>
<tr>
<td>R6</td>
<td>Digikey: P10191CT-ND</td>
<td>Filter Resistor (1206 SMD)</td>
<td>12</td>
<td>$0.00</td>
</tr>
<tr>
<td>R7</td>
<td>Digikey: P10191CT-ND</td>
<td>Filter Capacitor (1206 SMD)</td>
<td>12</td>
<td>$0.00</td>
</tr>
<tr>
<td>R8</td>
<td>Panasonic: EXC-ML20A390U Digikey: P10191CT-ND</td>
<td>0.1 Microfarad Capacitor (1206 SMD)</td>
<td>3</td>
<td>$0.00</td>
</tr>
<tr>
<td>R9</td>
<td>Panasonic: EXC-ML20A390U Digikey: P10191CT-ND</td>
<td>1 Microfarad Capacitor (1206 SMD)</td>
<td>4</td>
<td>$0.00</td>
</tr>
<tr>
<td>R10</td>
<td>Panasonic: EXC-ML20A390U Digikey: P10191CT-ND</td>
<td>0.01 Microfarad Capacitor (1206 SMD)</td>
<td>52</td>
<td>$0.00</td>
</tr>
<tr>
<td>C56</td>
<td>Panasonic: EXC-ML20A390U Digikey: P10191CT-ND</td>
<td>0.22 Microfarad Capacitor (1206 SMD)</td>
<td>1</td>
<td>$0.00</td>
</tr>
<tr>
<td>JP1,JP5</td>
<td>Panasonic: EXC-ML20A390U Digikey: P10191CT-ND</td>
<td>Bead Core 4A 100Mhz (1206 SMD)</td>
<td>3</td>
<td>$0.26</td>
</tr>
<tr>
<td>JP1,JP5</td>
<td>Panasonic: EXC-ML20A390U Digikey: P10191CT-ND</td>
<td>7x2 .100 Header (1206 SMD)</td>
<td>2</td>
<td>$0.00</td>
</tr>
<tr>
<td>JP4</td>
<td>Digikey: AT17C256-10PI-ND</td>
<td>10x2 .100 Header (1206 SMD)</td>
<td>5</td>
<td>$0.00</td>
</tr>
<tr>
<td>JP1,JP5</td>
<td>Panasonic: EXC-ML20A390U Digikey: P10191CT-ND</td>
<td>5x2 .100 Header (1206 SMD)</td>
<td>3</td>
<td>$0.00</td>
</tr>
<tr>
<td>U17</td>
<td>Digikey: AT17C256-10PI-ND</td>
<td>Serial FPGA PROM 256k (8-DIP)</td>
<td>1</td>
<td>$15.00</td>
</tr>
<tr>
<td>U5,U7</td>
<td>Digikey: AT17C256-10PI-ND</td>
<td>Tiny Pushbutton SFSI R/Angle</td>
<td>2</td>
<td>$0.00</td>
</tr>
<tr>
<td>SW1, SW2, SW3, SW4</td>
<td>CK Components: EP11S5D1ABE Digikey: CKN4007-ND</td>
<td>Tiny Pushbutton SFSI R/Angle</td>
<td>4</td>
<td>$4.92</td>
</tr>
<tr>
<td>D1,D3</td>
<td>Digikey: CKN1113-ND</td>
<td>Swit Cap Blk for .100 PB</td>
<td>4</td>
<td>$0.45</td>
</tr>
<tr>
<td>D2,D4</td>
<td>Chicago Miniature Lamp: S302H5-5V Digikey: L20025-ND</td>
<td>PC Board Mnt LED Green</td>
<td>2</td>
<td>$0.57</td>
</tr>
<tr>
<td>D2,D4</td>
<td>Chicago Miniature Lamp: S302H1-5V Digikey: L20021-ND</td>
<td>PC Board Mnt LED Red</td>
<td>2</td>
<td>$0.57</td>
</tr>
<tr>
<td>U3</td>
<td>Xilinx: XC4013E-5PQ208I</td>
<td>FPGA</td>
<td>1</td>
<td>$0.00</td>
</tr>
</tbody>
</table>

5 R2-R9, R12-R20
6 R1,R10,R11,R21,R27,R31,R38,R39,R42,R43,R44,R47
7 C1-C9,C22,C42,C61
8 C30,C33,C57,C59
9 C29,C32,C58,C43
10 C31,C34,C60,C10-C21,C23-C28,C35-C41,C44-C55,C62-C73
11 C74,C75,C76
12 JP2,JP3, U15
APPENDIX G

CONTINUUM SCHEMATICS

Figure G.1 shows the Continuum control and conversion schematics [21].
Figure G.2 shows the Continuum front sensors low schematics [21].
Figure G.3 shows the Continuum front sensors high schematics [21].
Figure G.4 shows the Continuum back sensors low schematics [21].
Figure G.5 shows the Continuum back sensors high schematics [21].
Figure G.6 shows the Continuum decoupling schematics [21].
Figure G.1 Continuum Control and Conversion Schematics [21]
Figure G.2 Continuum Front Sensor Low Schematics [21]
Figure G.3 Continuum Front Sensor High Schematics [21]
Figure G.4 Continuum Back Sensor Low Schematics [21]
The following code listing is C language code for the IpEngine from Brightstar Engineering. The code was compiled using CodeWarrior for Embedded Power PC.

H.1 main.c

The code below, main.c, is the main program loop for initial testing of low-level USB drivers for the IpEngine.

```c
#include "mpc823.h"
#include "UART.h"
#include "serial_config.h"
#include "usb.h"

void main()
{
    /* USB Variable Setup */
    unsigned short temp1=0,temp2=1;
    char c;
```
UARTError err;

/* MPC8xx internal register map */
EPPC *imwr = (EPPC *)(GetIMMR() & 0xffffffff);
/* Buffer descriptors */
RXTXBD *rxtxbd[NUM_ENDPOINTS];
/* Endpoint Parameters */
USEP *endpoint_params[NUM_ENDPOINTS];
/* Device descriptor */
DD *dd = (DD *)calloc(sizeof(DD), sizeof(char));
/* Configuration descriptor(s) */
CD *cd[NUM_CONFIG_DESC];
/* Interface descriptors(s) */
ID *id[NUM_INTERFACE_DESC];
/* Endpoint descriptors(s) */
ED *ed[NUM_ENDPOINT_DESC];
/* Request Structure */
DEV_REQUEST *dev_request;

/* Serial Setup */
InitializeUART(TEST_BAUD_RATE);

for (c = 32; c < 127; c++) {
  err = WriteUART1(c);
  if (err != kUARTNoError)
    break;
}
WriteUARTString( "\r\n" );

/* Initialize USB */
USB_Initialize(imwr, rxtxbd, endpoint_params);
USB_Desc_Init(dd, cd, id, ed);

/* Wait for Events */
printf("Wait for Events\r\n");
while(1){
  if (imwr->usb_regs.usber & USBER_RES){
    printf("\r\n\nUSBER: Reset Detected");
    imwr->usb_regs.usber = USBER_RES; // Clear RESET Bit
    print_mem(imwr, rxtxbd, endpoint_params);
    USB_Reset(imwr, rxtxbd, endpoint_params);
  }
  else if (imwr->usb_regs.usber & USBER_RXB){
    printf("\r\nUSBER: Buffer Received");
    imwr->usb_regs.usber = USBER_RXB; // Clear RXB Bit
    print_mem(imwr, rxtxbd, endpoint_params);
    USB_Receive(imwr, dd, cd, id, ed, rxtxbd, dev_request);
  }
  else if (imwr->usb_regs.usber & USBER_TXB){
    printf("\r\nUSBER: Buffer Transmitted");
    imwr->usb_regs.usber = USBER_TXB; // Clear TXB Bit
  }
print_mem(immr, rxtxbd, endpoint_params);
}
else if (immr->usb_regs.usber & USBER_SOF){
    printf("\r\nUSBER: SOF Recieved\n");
    immr->usb_regs.usber = USBER_SOF; // Clear SOF Bit
    print_mem(immr, rxtxbd, endpoint_params);
}

/**------------------------------------------**/
asm GetIMMR()
{
    nofralloc
    mfspr r3, 638
    blr
}

/**------------------------------------------**/
void print_mem(EPPC *immr, RXTXBD **rxtxbd, USEP **endpoint_params){
    int i=0;

    /* printf("\nMEMORY POINTERS:\n") ;
    printf("immr: %08x rxtxbd: %08x endpoint_params: %08x\n", immr, rxtxbd[0], endpoint_params[0]);
    */

    printf("\nPORTS:\n");
    printf("A: padir: %04x papar: %04x paodr: %04x padat: %04x\n", immr->pio.padir, immr->pio.papar, immr->pio.paodr, immr->pio.padat);
    printf("B: pbdir: %08x pbpar: %08x pbodr: %08x pbdat: %08x\n", immr->pb.pbdir, immr->pb.pbpar, immr->pb.pbodr, immr->pb.pbdat);
    printf("C: pcdir: %04x pccar: %04x pcso: %04x pcdat: %04x pcint: %04x\n", immr->pio.pcdir, immr->pio.pccar, immr->pio.pcso, immr->pio.pcdat, immr->pio.pcint);

    */
printf("D: pddir: %04x pdpar: %04x pddat: %04x\r\n", 
immr->pio.pddir, immr->pio.pdpar, immr->pio.pddat); */

printf("\r\nUSB REGS:\r\n");
printf("usmod: %02x usadr: %02x uscom: %02x ", 
immr->usb_regs.usmod, immr->usb_regs.usadr, immr->usb_regs.uscom);
printf("usepO: %04x usepi: %04x usep2: %04x usep3: %04x\r\n", 
immr->usb_regs.usep[0], immr->usb_regs.usep[1], 
immr->usb_regs.usep[2], immr->usb_regs.usep[3]);
printf("usep: %04x usbmr: %04x usbs: %02x\r\n", 
immr->usb_regs.usber, immr->usb_regs.usbmr, immr->usb_regs.usbs);

/* printf("\r\nBRGC:\r\n");
printf("brgci: %08x brgc2: %08x brgc3: %08x brgc4: %08x\r\n", 
immr->brgc1, immr->brgc2, immr->brgc3, immr->brgc4); 

printf("\r\nCommunication Processor:\r\n");
printf("cpcr: %04x rccr: %04x rmds: %04x rter: %04x rtmr: %04x\r\n", 
immr->cp.cpcr, immr->cp.rccr, immr->cp.rmds, immr->cp.rter, immr->cp.rttr);

printf("\r\nSerial Interface:\r\n");
printf("simode: %08x sigmr: %02x sistr: %02x\r\n", 
immr->si.simode, immr->si.sigmr, immr->si.sistr);
printf("sicmr: %02x sicr: %08x sirp: %08x\r\n", 
immr->si.sicmr, immr->si.sicr, immr->si.sirp);

printf("\r\nDPRAM:\r\n");
printf("Buffer Descriptors:\r\n");
for(i=0; i<=0; i++)
printf("%08x: %08x %08x %08x %08x %08x %08x %08x\r\n", 
&immr->udata.bd_ucode[i], immr->udata.bd_ucoce[i], 
immr->udata.bd_ucoce[i+1], immr->udata.bd_ucoce[i+2], 
immr->udata.bd_ucoce[i+3], immr->udata.bd_ucoce[i+4], 
immr->udata.bd_ucoce[i+5], immr->udata.bd_ucoce[i+6], 
immr->udata.bd_ucoce[i+7], immr->udata.bd_ucoce[i+8], 
immr->udata.bd_ucoce[i+9], immr->udata.bd_ucoce[i+10], 
immr->udata.bd_ucoce[i+11], immr->udata.bd_ucoce[i+12], 
immr->udata.bd_ucoce[i+13], immr->udata.bd_ucoce[i+14], 
immr->udata.bd_ucoce[i+15]);

printf("USB Params:\r\n");
printf("%08x: %04x%04x %04x%04x %04x%04x %04x%04x \r\n\n" , 
&immr->pram.usb.ep_ptr[0], immr->pram.usb.ep_ptr[0], 
immr->pram.usb.ep_ptr[1], immr->pram.usb.ep_ptr[2], 
immr->pram.usb.ep_ptr[3], immr->pram.usb.rstate, 
immr->pram.usb.rptr, &immr->pram.usb.frame_n, immr->pram.usb.frame_n, 
immr->pram.usb.rcnt, immr->pram.usb.rtemp);
printf("Endpoint Params:
\n");
for(i=0; i<=16; i+=16)
printf(
"%08x: %02x%02x%02x%02x %02x%02x%02x%02x %02x%02x%02x%02x %02x%02x%02x%02x
”,
&immr->udata.bd_ucode1[i], immr->udata.bd_ucode1[i],
&immr->udata.bd_ucode1[i+1], immr->udata.bd_ucode1[i+1],
&immr->udata.bd_ucode1[i+3], immr->udata.bd_ucode1[i+3],
&immr->udata.bd_ucode1[i+5], immr->udata.bd_ucode1[i+5],
&immr->udata.bd_ucode1[i+7], immr->udata.bd_ucode1[i+7],
&immr->udata.bd_ucode1[i+9], immr->udata.bd_ucode1[i+9],
&immr->udata.bd_ucode1[i+11], immr->udata.bd_ucode1[i+11],
&immr->udata.bd_ucode1[i+13], immr->udata.bd_ucode1[i+13],
&immr->udata.bd_ucode1[i+15], immr->udata.bd_ucode1[i+15]);
}

printf("DRAM:\n");
printf("RX Data Buffer:\n");
for(i=0; i<=16; i+=16)
printf(
"%08x: %02x%02x%02x%02x %02x%02x%02x%02x %02x%02x%02x%02x %02x%02x%02x%02x
”,
&rxtxbd[0]->rxbd.data_buffer_pointer[i],
rxtxbd[0]->rxbd.data_buffer_pointer[i],
rxtxbd[0]->rxbd.data_buffer_pointer[i+1],
rxtxbd[0]->rxbd.data_buffer_pointer[i+2],
rxtxbd[0]->rxbd.data_buffer_pointer[i+3],
rxtxbd[0]->rxbd.data_buffer_pointer[i+4],
rxtxbd[0]->rxbd.data_buffer_pointer[i+5],
rxtxbd[0]->rxbd.data_buffer_pointer[i+6],
rxtxbd[0]->rxbd.data_buffer_pointer[i+7],
rxtxbd[0]->rxbd.data_buffer_pointer[i+8],
rxtxbd[0]->rxbd.data_buffer_pointer[i+9],
rxtxbd[0]->rxbd.data_buffer_pointer[i+10],
rxtxbd[0]->rxbd.data_buffer_pointer[i+11],
rxtxbd[0]->rxbd.data_buffer_pointer[i+12],
rxtxbd[0]->rxbd.data_buffer_pointer[i+13],
rxtxbd[0]->rxbd.data_buffer_pointer[i+14],
rxtxbd[0]->rxbd.data_buffer_pointer[i+15]);
}

printf("TX Data Buffer:\n");
for(i=0; i<=16; i+=16)
printf(
"%08x: %02x%02x%02x%02x %02x%02x%02x%02x %02x%02x%02x%02x %02x%02x%02x%02x
”,
&rxtxbd[0]->txbd.data_buffer_pointer[i],
rxtxbd[0]->txbd.data_buffer_pointer[i],
rxtxbd[0]->txbd.data_buffer_pointer[i+1],
rxtxbd[0]->txbd.data_buffer_pointer[i+2],
rxtxbd[0]->txbd.data_buffer_pointer[i+3],
rxtxbd[0]->txbd.data_buffer_pointer[i+4],
rxtxbd[0]->txbd.data_buffer_pointer[i+5],
rxtxbd[0]->txbd.data_buffer_pointer[i+6],
rxtxbd[0]->txbd.data_buffer_pointer[i+7],

79
H.2 usb.c

The code below, usb.c, contains low-level routines for initializing, configuring, reading from, and writing to the USB port on the IpEngine.

```c
#include "mpc823.h"
#include "usb.h"

/*-------1---------2---------3---------4---------5---------6---------7---------
123456789-123456789-123456789-123456789-123456789-123456789-123456789-123456789
* *
* Function name : USB_Initialize()
* Description : Does the initial setup of the USB controller including
*               CLK source selection, baudrate generator setup,
*               port setup, parameter/data memory allocation, and
*               USB register initialization.
*----------------------------------------------------------------------------*/

void USB_Initialize(EPPC *immr, RXTXED **rxtxbd, USEP **endpoint_params){
    int a;
    /* USB Setup */
    /* Configure SICR (p16-134) for R1CS=001=USB clock is BRG2 */
    immr->si.sicr |= 0x0008;
    
    
    rxtxbd[0]->txbd.data_buffer_pointer[i+8],
    rxtxbd[0]->txbd.data_buffer_pointer[i+9],
    rxtxbd[0]->txbd.data_buffer_pointer[i+10],
    rxtxbd[0]->txbd.data_buffer_pointer[i+11],
    rxtxbd[0]->txbd.data_buffer_pointer[i+12],
    rxtxbd[0]->txbd.data_buffer_pointer[i+13],
    rxtxbd[0]->txbd.data_buffer_pointer[i+14],
    rxtxbd[0]->txbd.data_buffer_pointer[i+15]);
}
```
/** Configure BRGC2 for division factor 1 (48MHz rom 48MHz CPUClk) */
immr->brgc2 = 0x00010000;

/* Allocate space for rx and tx buffer descriptors */
rxtxbd[0] = (RXTXBD *) immr->udata.bd_ucode0;

/* Allocate space for rx and tx data buffers */
rxtxbd[0]->rxbd.data_buffer_pointer =
(unsigned char *) calloc(RX_BUF_SIZE,sizeof(long));
rxtxbd[0]->txbd.data_buffer_pointer =
(unsigned char *) calloc(TX_BUF_SIZE,sizeof(long));

/* Allocate space for endpoint parameter blocks */
endpoint_params[0] = (USEP *) immr->udata.bd_ucode1;

/* Initialize RX and TX Buffer Descriptors */
rxtxbd[0]->rxbd.control_status = RXBD_E | RXBD_W;
rxtxbd[0]->rxbd.data_length = 0;
rxtxbd[0]->txbd.control_status =
TXBD_R | TXBD_W | TXBD_I | TXBD_L | TXBD_TC | TXBD_PID_DATA0;
rxtxbd[0]->txbd.data_length = 0;

/* Initialize Endpoint parameter blocks p16-352*/
endpoint_params[0]->rbase =
endpoint_params[0]->rbptr = (unsigned short) & rxtxbd[0]->rxbd;
endpoint_params[0]->tbbase =
endpoint_params[0]->tbptr = (unsigned short) & rxtxbd[0]->txbd;
endpoint_params[0]->rfcr = endpoint_params[0]->tfcr = 0x18;
endpoint_params[0]->mrblr = MAX_RX_TX_BYTES;
endpoint_params[0]->tstate = 0;

/* Initialize Endpoint Parameter Base Addresses */
immr->pram.usb.ep_ptr[0] = (unsigned short) endpoint_params[0];

/* Clear Frame number entry */
immr->pram.usb.frame_n = 0;

/* Initialize USB Endpoint registers p16-365*/
/* immr->usb_regs.uep[0] = 0x0200; BULK Mode*/
/* immr->usb_regs.uep[0] = 0x0000; //Control Mode */

/* Configure USMODE for normal USB function operation p16-357 */
immr->usb_regs.usmod = 0x00;

/* Initialize USB address */
immr->usb_regs.usadr = 0x00;

/* Configure Port A Registers for USB Operation */
/* Select USBRXD,USBE(USBTXD) Pin Functions */
Configure Port C Registers for USB Operation
* Select USBTXP, USBTXN Pin Functions, and
* Select USBRXP, USBRXN Pin Functions.
* USBSPD (Brightstar) = 1 for 12MBit/s
* PDN (Brightstar) = 0 for NO POWERDOWN/SUSPEND

imr->pio.papar |= 0x0003;
imr->pio.padir &= ~(0x0003);

/*
 * Configure Port C Registers for USB Operation
 * Select USBTXP, USBTXN Pin Functions, and
 * Select USBRXP, USBRXN Pin Functions.
 * USBSPD (Brightstar) = 1 for 12MBit/s
 * PDN (Brightstar) = 0 for NO POWERDOWN/SUSPEND
 */
imr->pio.pcdir |= 0x0708;
imr->pio.pcpar |= 0x0000;
imr->pio.pcsel |= 0x0003;
imr->pio.pcdat = 0x0000;

/* CPM Init */
imr->cp.cpcr = 0x0001; // init RX/TX params
while(imr->cp.cpcr & 0x0001){} // Wait for FLG bit to clear

/* Clear USCOM */
imr->usb_regs.uscom = 0x00;

/* Enable USB Operation */
imr->usb_regs.usmod |= 1;

/* Clear Event register */
imr->usb_regs.usber = 0xffff;

}

/******************
 * Function name : USB_Desc_Init()
 * Description : Allocates memory for and initializes USB descriptor
 * tables.
 ******************/
void USB_Desc_Init(DD *dd, CD **cd, ID **id, ED **ed){
    int i;

    /* Allocate space for Descriptors */
    for(i=0; i<NUM_CONFIG_DESC; i++){
        cd[i] = (CD*) calloc(sizeof(CD),sizeof(char));
    }
    for(i=0; i<NUM_INTERFACE_DESC; i++){
        id[i] = (ID*) calloc(sizeof(ID),sizeof(char));
    }
    for(i=0; i<NUM_END_DESC; i++){
        ed[i] = (ED*) calloc(sizeof(ED),sizeof(char));
    }
    }

82
/ * Initialize Device Descriptor */
    dd->bLength = sizeof(DD);
    dd->bDescriptorType = DESC_TYPE_DEVICE;
    dd->bDeviceClass = 0x00;
    dd->bDeviceSubClass = 0x00;
    dd->bDeviceProtocol = 0x00;
    dd->bcdDevice = 0x0110; // USB REV 1.1 in BCD
    dd->bMaxPacketSize0 = 0x40;
    dd->idVendor = 0x4242; // USB Vendor ID (GET ONE!)
    dd->idProduct = 0x1234;
    dd->bcdDevice = 0x1122;
    dd->iManufacturer = 0x00;
    dd->iProduct = 0x00;
    dd->iSerialNumber = 0x00;
    dd->bNumConfigurations = 0x01;

    /* Initialize Configuration Descriptor(s) */
    cd[0]->bLength = sizeof(CD);
    cd[0]->bDescriptorType = DESC_TYPE_CONFIG;
    cd[0]->wTotalLength = 0x09 + 0x09 + 0x07; // CD+ID+ED sizes
    cd[0]->bNumInterfaces = 0x01;
    cd[0]->bConfigurationValue = 0x01;
    cd[0]->iConfiguration = 0x00; // Index to string descriptor
    cd[0]->bmAttributes
        = CONF_ATTR_RES | CONF_ATTR_SELFPOWER | CONF_ATTR_REMOTE;
    cd[0]->MaxPower = 0x00;

    /* Initialize Interface Descriptor(s) */
    id[0]->bLength = sizeof(ID);
    id[0]->bDescriptorType = DESC_TYPE_INTER;
    id[0]->bInterfaceNumber = 0x00;
    id[0]->bAlternateSetting = 0x00; // No alternate setting
    id[0]->bNumEndpoints = 0x01;
    id[0]->bInterfaceClass = CONF_CLASS_VENDOR;
    id[0]->bInterfaceSubClass = 0x00;
    id[0]->bInterfaceProtocol = 0x00;
    id[0]->iInterface = 0x00;

    /* Initialize Endpoint Descriptor(s) */
    ed[0]->bLength = sizeof(ED);
    ed[0]->bDescriptorType = DESC_TYPE_END;
    ed[0]->bEndpointAddress = END_ADDR_EP | END_ADDR_IN;
    ed[0]->bmAttributes = END_ATTR_BULK;
    ed[0]->wMaxPacketSize = 0x40; // 64 bytes
    ed[0]->bInterval = 0x64; // Poll every 0.1 sec

    /**************************************************************************/
    * Function name : USB_Recieve()
void USB_Receive(EPPC *immr, DD *dd, CD **cd, ID **id, ED **ed, RXTXBD **rxtxbd, DEV_REQUEST *dev_request){
    /* Detect Packet type */
    if (rxtxbd[0]->rxbd.control_status & RXBD_PID_SETUP){
        printf(" SETUP Packet\r\n");
        USB_Request_Decode(immr, dd, cd, id, ed, rxtxbd, dev_request);
    }
    else if(rxtxbd[0]->rxbd.control_status & RXBD_PID_DATA0){
        printf(" OUT (DATA 0) Packet\r\n");
    }
    else if(rxtxbd[0]->rxbd.control_status & RXBD_PID_DATA1){
        printf(" OUT (DATA 1) Packet\r\n");
    }
}

void USB_Request_Decode(EPPC *immr, DD *dd, CD **cd, ID **id, ED **ed, RXTXBD **rxtxbd, DEV_REQUEST *dev_request){
    /* Map RX buffer onto Request structure */
    dev_request = (DEV_REQUEST *) rxtxbd[0]->rxbd.data_buffer_pointer;
    switch(dev_request->bmRequestType & 0x60){
        case REQ_TYPE_STANDARD:
            switch (dev_request->bRequest){
                case DESC_REQ_GET_DESCRIPTOR:
                    printf(" Get Device Descriptor\r\n");
                    /* Reset RX Buffer Descriptor */
                    rxtxbd[0]->rxbd.control_status = RXBD_E | RXBD_W;
                    USB_Send(immr, rxtxbd, (unsigned char *) dd, 0, TXBD_PID_DATA1, dd->bLength); // Change to wLength
                    break;
                case DESC_TYPE_DEVICE:
                    printf(" Get Device Descriptor\r\n");
                    break;
            }
            break;
        case DESC_TYPE_CONFIG:

    }
}
printf(" Get Config Descriptor\r\n");
/* Reset RX Buffer Descriptor */
rxtxbd[0]->rxbd.control_status = RXBD_E | RXBD_W;
USB_Send(immr, rxtxbd, (unsigned char *) dd,
0, TXBD_PID_DATA1, dd->bLength); // Change to uLength
break;

break;

case DESC_REQ_SET_ADDRESS:
break;
}
break;

case REQ_TYPE_CLASS:
break;

case REQ_TYPE_VENDOR:
break;
};
}/*-----------------------------1---------2---------3---------4---------5---------6---------7---------
123456789-123456789-123456789-123456789-123456789-123456789-123456789-123456789
*/
* Function name : USB_Send()
* Description : Sends USB data block.
*-------------------------------------------------------------------------*/

void USB_Send(EPPC *immr, RXTXBD **rxtxbd, unsigned char *tx_data,
int endpoint, short pid, int count){
    int i;

    printf("txdata: %02x\r\n", tx_data[0]);
    printf("endpoint: %d\r\n", endpoint);
    printf("pid: %04x\r\n", pid);
    printf("count: %d\r\n", count);

    for(i=0; i<count; i++){
        rxtxbd[endpoint]->txbd.data_buffer_pointer[i] = tx_data[i];
    }

    rxtxbd[endpoint]->txbd.control_status =
    TXBD_R | TXBD_W | TXBD_I | TXBD_L | TXBD_TC | pid;
    rxtxbd[endpoint]->txbd.data_length = count;

    /* Start transaction for Endpoint 0 */
    immr->usb_regs.uscom = 0x80;
}

/*-----------------------------1---------2---------3---------4---------5---------6---------7---------
Function name: USB_Reset()
Description: Resets the USB controller.

```
void USB_Reset(EPPC *immr, RXTXBD **rxtxbd, USEP **endpoint_params){
    int i;

    /* Initialize RX and TX Buffer Descriptors */
    rxtxbd[0]->rxbd.control_status = RXBD_E | RXBD_W;
    rxtxbd[0]->rxbd.data_length = 0;
    rxtxbd[0]->txbd.control_status =
        TXBD_R | TXBD_W | TXBD_I | TXBD_L | TXBD_TC | TXBD_PID_DATA0;
    rxtxbd[0]->txbd.data_length = 0;

    /* Initialize Endpoint parameter blocks */
    endpoint_params[0]->rbase = (unsigned short) & rxtxbd[0]->rxbd;
    endpoint_params[0]->tbase = (unsigned short) & rxtxbd[0]->txbd;
    endpoint_params[0]->rfcr = endpoint_params[0]->tfcr = Ox18;
    endpoint_params[0]->mrblr = MAX_RX_TX_BYTES;
    endpoint_params[0]->tstate = 0;

    /* Clear Buffers */
    for(i=0; i<RX_BUF_SIZE; i++){
        rxtxbd[0]->rxbd.data_buffer_pointer[i] = 0x00;
    }
    for(i=0; i<TX_BUF_SIZE; i++){
        rxtxbd[0]->txbd.data_buffer_pointer[i] = 0x00;
    }

    /* Initialize Endpoint Parameter Base Addresses */
    immr->pram.usb.ep_ptr[0] = (unsigned short) endpoint_params[0];

    /* Clear Frame number entry */
    immr->pram.usb.frame_n = 0;

    /* Initialize USB Endpoint registers */
    immr->usb_regs.usep[0] = Ox0200; BULK Mode*/
    immr->usb_regs.usep[0] = Ox0000; //Control Mode

    /* Configure USMODE for normal USB function operation */
    immr->usb_regs.usmode = 0;

    /* Initialize USB address */
    immr->usb_regs.usadr = 0x00;

    /* CPN Init */
    immr->cp.cpcr = Ox0001; // init RX/TX params
    while((immr->cp.cpcr & Ox0001){} // Wait for FLG bit to clear
```
The code below, usb.h, contains function prototypes for usb.c as well type definitions and constants for USB registers and data structures.

```c
typedef struct BufferDescriptor {
    unsigned short control_status;
    unsigned short data_length;
    unsigned char *data_buffer_pointer;
} BD;

typedef struct BufferDescriptor_TX_RX {
    BD rxbd;
    BD rxbd;
} RXTXBD;

#pragma options align=packed

typedef struct DeviceDescriptor {
    char bLength;
    char bDescriptorType;
    short bcdUSB;

87
char bDeviceClass;
char bDeviceSubClass;
char bDeviceProtocol;
char bMaxPacketSize0;
short idVendor;
short idProduct;
short bcdDevice;
char iManufacturer;
char iProduct;
char iSerialNumber;
char bNumConfigurations;
} DD;

typedef struct ConfigurationDescriptor{
    char bLength;
    char bDescriptorType;
    short wTotalLength;
    char bNumInterfaces;
    char bConfigurationValue;
    char iConfiguration;
    char bmAttributes;
    char MaxPower;
} CD;

typedef struct InterfaceDescriptor{
    char bLength;
    char bDescriptorType;
    char bInterfaceNumber;
    char bAlternateSetting;
    char bNumEndpoints;
    char bInterfaceClass;
    char bInterfaceSubClass;
    char bInterfaceProtocol;
    char iInterface;
} ID;

typedef struct EndpointDescriptor{
    char bLength;
    char bDescriptorType;
    char bEndpointAddress;
    char bmAttributes;
    short wMaxPacketSize;
    char bInterval;
} ED;

/* Control Transfer Request Structure (USB 1.1 Spec p183 Table 9-2) */
typedef struct DeviceRequest{
    char bmRequestType;
    char bRequest;
    short wValue;
    short wIndex;
    short wLength;
} DEV_REQUEST;
#pragma options align=reset

void USB_Initialize(EPPC *immr, RXTXBD **rxtxbd, USEP **endpoint_params);
void USB_Desc_Init(DD *dd, CD **cd, ID **id, ED **ed);
void USB_Reset(EPPC *immr, RXTXBD **rxtxbd, USEP **endpoint_params);
void USB_Receive(EPPC *immr, RXTXBD **rxtxbd, USEP **endpoint_params);

ID **id, ED **ed, RXTXBD **rxtxbd, DEV_REQUEST *dev_request);
void USB_Send(EPPC *immr, RXTXBD **rxtxbd, unsigned char *tx_data,
int endpoint, short pid, int count);
void USB_Request_Decode(EPPC *immr, DD *dd, CD **cd, ID **id,
ED **ed, RXTXBD **rxtxbd, DEV_REQUEST *dev_request);

#define NUM_ENDPOINTS (1) // Number of active endpoints */
#define RX_BUF_SIZE (1024) /* RX Buffer size */
#define TX_BUF_SIZE (1024) /* TX Buffer size */
#define MAX_RX_TX_BYTES (512) /* Max Bytes < TX/RX_BUF_SIZE */
#define NUM_CONFIG_DESC (1) /* Num of Config Descriptors */
#define NUM_INTERFACE_DESC (1) /* Num of Interface Descriptors */
#define NUM_END_DESC (1) /* Num of Endpoint Descriptors */

/* USB Event Register Bit Definitions (USBER) 16.10.8.8 p16-369 */
#define USBER_RXB (0x0001)
#define USBER_TXB (0x0002)
#define USBER_BSY (0x0004)
#define USBER_SOF (0x0008)
#define USBER_TXE0 (0x0010)
#define USBER_TXE1 (0x0020)
#define USBER_TXE2 (0x0040)
#define USBER_TXE3 (0x0080)
#define USBER_IDLE (0x0100)
#define USBER_RES (0x0200)

/* Device Request Type (USB 1.1 Spec p183 Table 9-2) */
#define REQ_DIR_OUT (0x00)
#define REQ_DIR_IN (0x08)
#define REQ_TYPE_STANDARD (0x00)
#define REQ_TYPE_CLASS (0x20)
#define REQ_TYPE_VENDOR (0x40)
#define REQ_RECIP_DEV (0x00)
#define REQ_RECIP_INTER (0x01)
#define REQ_RECIP_END (0x02)
#define REQ_RECIP_OTHER (0x03)

/* Descriptor Type Definitions (USB 1.1 Spec p187 Table 9-5) */
#define DESC_TYPE_DEVICE (0x01)
#define DESC_TYPE_CONFIG (0x02)
#define DESC_TYPE_STRING (0x03)
#define DESC_TYPE_INTER (0x04)
#define DESC_TYPE_END (0x05)

89
/* Descriptor Standard Request Codes (USB 1.1 Spec p187 Table 9-4) */
#define DESC_REQ_GET_STATUS (0x00)
#define DESC_REQ_CLEAR_FEATURE (0x01)
#define DESC_REQ_SET_FEATURE (0x03)
#define DESC_REQ_SET_ADDRESS (0x05)
#define DESC_REQ_GET_DESCRIPTOR (0x06)
#define DESC_REQ_SET_DESCRIPTOR (0x07)
#define DESC_REQ_GET_CONFIG (0x08)
#define DESC_REQ_SET_CONFIG (0x09)
#define DESC_REQ_GET_INTERFACE (0xa)
#define DESC_REQ_SET_INTERFACE (0xb)
#define DESC_REQ_SYNCH_FRAME (0xc)

/* Config Descriptor attributes (USB 1.1 Spec p200 Table 9-8) */
#define CONF_ATTR_RES
#define CONF_ATTR_SELFPOWER
#define CONF_ATTR_REMOTE

/* Interface Descriptor Class/Subclass/Protocol */
#define CONF_CLASS_HID (0x03)
#define CONF_CLASS_VENDOR (0xff)

/* Endpoint Descriptor Address / Attributes (USB 1.1 Spec p203 Table 9-10) */
#define END_ADDR_EP0 (0x00)
#define END_ADDR_EP1 (0x01)
#define END_ADDR_EP2 (0x02)
#define END_ADDR_EP3 (0x03)
#define END_ADDR_OUT (0x00)
#define END_ADDR_IN (0x80)
#define END_ATTR_CONTROL (0x00)
#define END_ATTR_ISO (0x01)
#define END_ATTR_BULK (0x02)
#define END_ATTR_INTER (0x03)

/* RX Buffer Descriptor Control/Status Definitions p16-359*/
#define RXBD_E (0x8000) /* Empty */
#define RXBD_W (0x2000) /* Wrap */
#define RXBD_I (0x1000) /* Interrupt */
#define RXBD_L (0x0800) /* Last */
#define RXBD_F (0x0400) /* First */
#define RXBD_PID_DATA0 (0x0000) /* DATA0 PID */
#define RXBD_PID_DATA1 (0x0040) /* DATA1 PID */
#define RXBD_PID_SETUP (0x0080) /* SETUP PID */
#define RXBD_NO (0x0010) /* RX non octet aligned */
#define RXBD_AB (0x0008) /* Frame Aborted */
#define RXBD_CR (0x0004) /* CRC Error */
#define RXBD_OV (0x0002) /* Overrun */

90
H.4 capture.txt

The capture file below, capture.txt, contains a log file of the console output generated by main.c test functions. The capture sequence shows import USB registers and buffers before and after receiving SOF and IN USB tokens.

Before IN Token

USB REGS:
usmod: 01 usadr: 05 uscom: 00 usep0: 0200 usep1: 0000 usep2: 0000 usep3: 0000
usber: 0100 usbr: 0000 usbs: 01

DPRAM:
Buffer Descriptors:
ff002000: a0000000 003b0008 bc800004 003b1010
USB Params:
ff003c00: 2200feef c2b0b916 00000000 0a12d012
ff003c10: 2200f6fe 00000ae3
Endpoint Params:
ff002200: 20002008 18180200 20002008 18890eac
ff002210: 003b1014 ffff0000 ffff3d82 71041800
DRAM:
RX Data Buffer:
003b0008: 00000000 00000000 00000000 00000000
003b0018: 00000000 00000000 00000000 00000000
TX Data Buffer:
003b1010: abcddef12 00000000 00000000 00000000
003b1020: 00000000 00000000 00000000 00000000
AFTER SOF Token

USB REGS:
usmod: 01 usadr: 05 uscom: 00 usep0: 0200 usep1: 0000 usep2: 0000 usep3: 0000
usber: 0308 usbmr: 0000 usbs: 01
DPRAM:
Buffer Descriptors:
ff002000: a0000000 003b0008 bc800004 003b1010
USB Params:
ff003c00: 2200feef c2b0b916 00000000 0a12d012
ff003c10: 8002f6fe 000000ae3
Endpoint Params:
ff002200: 20002008 18180200 20002008 18890eac
ff002210: 003b1014 ffff0000 ffff3d82 71041800
DRAM:
RX Data Buffer:
003b0008: 00000000 00000000 00000000 00000000
003b0018: 00000000 00000000 00000000 00000000
TX Data Buffer:
003b1010: abcdef12 00000000 00000000 00000000
003b1020: 00000000 00000000 00000000 00000000

AFTER IN TOKEN

USB REGS:
usmod: 01 usadr: 05 uscom: 00 usep0: 0200 usep1: 0000 usep2: 0000 usep3: 0000
usber: 0308 usbmr: 0000 usbs: 01
DPRAM:
Buffer Descriptors:
ff002000: a0000000 003b0008 3e800004 003b1010
USB Params:
ff003c00: 2200feef c2b0b916 00000000 0a12d012
ff003c10: 8003f6fe 00000ae3
Endpoint Params:
ff002200: 20002008 18180200 20002008 18000eac
ff002210: 003b1014 ffff0000 ffff3d82 71041800
DRAM:
RX Data Buffer:
003b0008: 00000000 00000000 00000000 00000000
003b0018: 00000000 00000000 00000000 00000000
TX Data Buffer:
003b1010: abcdef12 00000000 00000000 00000000
003b1020: 00000000 00000000 00000000 00000000

AFTER SETUP TOKEN (GET_DEVICE_DESCRIPTOR)
  <-inc address max length = 8
  00 18 00 00 01 00 06 80
SETUP : index: 0 length: 18 language ID: 0 type: device  (hex)

USB REGS:
usmod: 01 usadr: 05 uscom: 00 usep0: 0000 usep1: 0000 usep2: 0000 usep3: 0000
usber: 0309 usbmr: 0000 usbs: 01
DPRAM:
Buffer Descriptors:
ff002000: 2c80000a 003b0008 bc800004 003b1010
USB Params:
ff003c00: 2200feef c2b0b916 18000ef6 003b0010
ff003c10: 800301f8 00002200
Endpoint Params:
ff002200: 20002008 18180200 20020008 18890eac
ff002210: 003b1014 fff00000 fff3d82 71041800
DRAM:
RX Data Buffer:
003b0008: 80060001 00001800 e65454f6 00000000
003b0018: 00000000 00000000 00000000 00000000
TX Data Buffer:
003b1010: abcdef12 00000000 00000000 00000000
003b1020: 00000000 00000000 00000000 00000000
APPENDIX I

DAC CODE

The following code listing is C language code for the IpEngine from Brightstar Engineering. The code was compiled using CodeWarrior for Embedded Power PC.

1.1 main_dac.c

The code below, main_dac.c, is a main program loop for testing of the DAC board. In this version of the code the DAC updates are handled through direct writes to the DAC registers.

```c
#include "mpc823.h"
#include "UART.h"
#include "serial_config.h"
#include "util.h"
#include "mpc_uart.h"

void main()
{
    char c;
    UARTError err;
```
unsigned char status_data;
unsigned char data0 = 0, data1 = 0;
unsigned long data;

/* MPC8xx internal register map */
EPPC *immr = (EPPC *) (GetIMMR() & 0xffff0000);

/* FPGA register map */
FPGA *fpga = (FPGA *) 0xff010000;
FPGA_MAP *fpga_ports = (FPGA_MAP *) 0xfc000000;

// Multiport Midi Test
MPCUARTInit(kBaud9600, kSMC1);
MPCUARTPutChar(‘z’);
MPCUARTInit(kBaud9600, kSMC2);
MPCUARTPutChar2(‘z’);

/* Serial Setup */
InitializeUART(TEST_BAUD_RATE);
WriteUARTString(“\r\n”);
for (c = 32; c < 127; c++) {
  err = WriteUART1(c);
  if (err != kUARTNoError)
    break;
}
WriteUARTString(“\r\n”);

/* Timer Config */
immr->timer.tgcr = 0x0000; // Timers Disabled
immr->timer.tmr2 = 0x002a; // Toggle TOUT with Timer CLK input as sys CLK.
immr->timer.tgcr |= 0x0010; // Start Timer 2
immr->timer.tcn2 = 0;
immr->timer.trr2 = 0;
immr->timer.ter2 = 0xffff;

/* Setup Port A (PA4) as BCLK */
immr->pio.papar = 0x0800; // Select Clock Functions
immr->pio.padir = 0x0800; // Select TOUT2.

FPGA_Config(immr, fpga);

MIDI_Initialize(fpga_ports);
Continuum_Initialize(fpga_ports);
MIDI_Led(fpga_ports, 0);
MIDI_Led(fpga_ports, 1);
MIDI_Led(fpga_ports, 2);
MIDI_Led(fpga_ports, 3);

Dac_Initialize(fpga_ports);
while(1){

    Write_Dac_Byte(fpga_ports,0x04,0xff);
    Write_Dac_Byte(fpga_ports,0x05,0xff);
    Write_Dac_Byte(fpga_ports,0x06,0x00);

    // Write to Dac
    Write_Dac_Byte(fpga_ports,0x01,0x12);
    // Update Dac Outout
    Write_Dac_Byte(fpga_ports,0x01,0x13);

    Wait(10000);

    Write_Dac_Byte(fpga_ports,0x04,0x00);
    Write_Dac_Byte(fpga_ports,0x05,0x00);
    Write_Dac_Byte(fpga_ports,0x06,0x00);

    // Write to Dac
    Write_Dac_Byte(fpga_ports,0x01,0x12);
    // Update Dac Outout
    Write_Dac_Byte(fpga_ports,0x01,0x13);

    Wait(10000);

}

/***************************************
* Function name  : GetIMMR()
* Description    : Uses mfspr (Move From Special Register) instruction to get
*                 the base address for the internal memory map stored in
*                 the Internal Memory Map Register (IMMR).
*                 
*----------------------------------------------------------------------------*

asm GetIMMR()
{
    nofralloc
    mfspr r3, 638;
    blr
}
I.2 main_fifo.c

The code below, main_fifo.c, is a main program loop for testing of the DAC board. In this version of the code, the DAC updates are handled through a FIFO based producer/consumer scheme.

```c
#include "mpc823.h"
#include "UART.h"
#include "serial_config.h"
#include "util.h"
#include "mpc_uart.h"

void main()
{
    char c;
    UARTError err;
    unsigned char status_data;
    unsigned char data0 = 0, data1 = 0;

    /* MPC8xx internal register map */
    EPPC *immr = (EPPC *) (GetIMMR() & 0xffff0000);

    /* FPGA register map */
    FPGA *fpga = (FPGA *) 0xff010000;
    FPGA_MAP *fpga_ports = (FPGA_MAP *) 0xfc000000;

    // Midi Test
    MPCUARTInit(kBaud9600, kSMC1);
    MPCUARTPutChar('z');

    MPCUARTInit(kBaud9600, kSMC2);
    MPCUARTPutChar2('z');

    /* Serial Setup */
    InitializeUART(TEST_BAUD_RATE);
    WriteUARTString( "\r\n" );

    for (c = 32; c < 127; c++) {
```
err = WriteUART1(c);
if (err != kUARTNoError)
    break;
}
WriteUARTString("\r\n");

/* Timer Config */
immr->timer.tgcr = 0x0000; // Timers Disabled
immr->timer.tmr2 = 0x002a; // Toggle TOUT with Timer CLK input as sys CLK
immr->timer.tgcr |= 0x0010; // Start Timer 2
immr->timer.tcn2 = 0;
immr->timer.trr2 = 0;
immr->timer.ter2 = 0xffff;

/* Setup Port A (PA4) as BCLK */
immr->pio.papar |= 0x0800; // Select Clock Functions
immr->pio.padir |= 0x0800; // Select TOUT2

FPGA_Config(immr, fpga);
MIDI_Initialize(fpga_ports);
Continuum_Initialize(fpga_ports);

MIDI_Led(fpga_ports,0);
MIDI_Led(fpga_ports,1);
MIDI_Led(fpga_ports,2);
MIDI_Led(fpga_ports,3);
Dac_Initialize(fpga_ports);

// Output to LEDs
Write_Dac_Byte(fpga_ports,0x04, "(0x0a)");
Write_Dac_Byte(fpga_ports,0x01,0x07);

// Read Buttons
Write_Dac_Byte(fpga_ports,0x01,0x06);
Read_Dac_Byte(fpga_ports,0x04);

// Read Status
Read_Dac_Byte(fpga_ports,0x02);

// Read Buttons
Write_Dac_Byte(fpga_ports,0x01,0x06);
Read_Dac_Byte(fpga_ports,0x04);

Dac_Initialize(fpga_ports);

// Read Status
printf("DATA: %02x \r\n", Read_Dac_Byte(fpga_ports,0x02));
// Write fifo
Write_Dac_Byte(fpga_ports, Ox04, Oxaa);
Write_Dac_Byte(fpga_ports, Ox05, Oxbb);
Write_Dac_Byte(fpga_ports, Ox06, Oxcc);
Write_Dac_Byte(fpga_ports, Ox01, Ox01);

// Read Status
printf("DATA: %02x \r\n", Read_Dac_Byte(fpga_ports, Ox02));
Write_Dac_Byte(fpga_ports, Ox04, Ox00);
Write_Dac_Byte(fpga_ports, Ox05, Ox00);
Write_Dac_Byte(fpga_ports, Ox06, Ox00);

// Read Fifo
Write_Dac_Byte(fpga_ports, Ox01, Ox11);

// Read Status
printf("DATA: %02x \r\n", Read_Dac_Byte(fpga_ports, Ox02));

if((Read_Dac_Byte(fpga_ports, Ox04) == Oxaa) &
   (Read_Dac_Byte(fpga_ports, Ox05) == Oxbb) &
   (Read_Dac_Byte(fpga_ports, Ox06) == Oxcc))
   printf("");
else
   printf("*");

Dac_Initialize(fpga_ports);

// Reset Dacs
Write_Dac_Byte(fpga_ports, Ox01, Ox10);

// Read Status
printf("DATA: %02x \r\n", Read_Dac_Byte(fpga_ports, Ox02));

// Fill FIFO
Write_Dac_Byte(fpga_ports, Ox04, Ox11);
Write_Dac_Byte(fpga_ports, Ox05, Ox00);
Write_Dac_Byte(fpga_ports, Ox06, Ox10);
Write_Dac_Byte(fpga_ports, Ox01, Ox01);

Write_Dac_Byte(fpga_ports, Ox04, Ox11);
Write_Dac_Byte(fpga_ports, Ox05, Ox08);
Write_Dac_Byte(fpga_ports, Ox06, Ox10);
Write_Dac_Byte(fpga_ports, Ox01, Ox01);

Write_Dac_Byte(fpga_ports, Ox04, Ox11);
Write_Dac_Byte(fpga_ports, Ox05, Ox10);
Write_Dac_Byte(fpga_ports, Ox06, Ox10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x18);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x20);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x28);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x30);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x38);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x40);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x48);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x50);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x58);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x60);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x68);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x05,0x68);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x70);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x78);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x80);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x88);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x90);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0x98);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0xa0);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0xa8);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);
Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0xc0);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);

Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0xc8);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);

Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0xd0);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);

Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0xd8);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);

Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0xe0);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);

Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0xe8);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);

Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0xf0);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);

Write_Dac_Byte(fpga_ports,0x04,0x11);
Write_Dac_Byte(fpga_ports,0x05,0xff);
Write_Dac_Byte(fpga_ports,0x06,0x10);
Write_Dac_Byte(fpga_ports,0x01,0x01);

// Read_Status
printf("DATA: %02x \r\n",Read_Dac_Byte(fpga_ports,0x02));

// Start Output
Write_Dac_Byte(fpga_ports,0x01,0x02);
status_data = Read_Dac_Byte(fpga_ports,0x02);

while((status_data & 0x04) == 0x04){
    status_data = Read_Dac_Byte(fpga_ports,0x02);
    if((status_data & 0x01) != 0x01){
        Write_Dac_Byte(fpga_ports,0x04,data0);
Write_Dac_Byte(fpga_ports,0x05,data1);
Write_Dac_Byte(fpga_ports,0x06,10);
// Write Fifo
Write_Dac_Byte(fpga_ports,0x01,0x01);
data0 = data0 + 100;
data1 = data1 + 100;
status_data = Read_Dac_Byte(fpga_ports,0x02);
}

// Read_Status
printf("DATA: \x02x \r\n",Read_Dac_Byte(fpga_ports,0x02));

// Read Fifo
Write_Dac_Byte(fpga_ports,0x01,0x11);

// Read_Status
printf("DATA: \x02x \r\n",Read_Dac_Byte(fpga_ports,0x02));
printf("DATA: \x02x \r\n",Read_Dac_Byte(fpga_ports,0x04));
printf("DATA: \x02x \r\n",Read_Dac_Byte(fpga_ports,0x05));
printf("DATA: \x02x \r\n",Read_Dac_Byte(fpga_ports,0x06));

while(1){}
}

/*-----------1---------2---------3---------4---------5---------6---------7---------
123456789-123456789-123456789-123456789-123456789-123456789-123456789-123456789
* *
* Function name : GetIMMR()
* Description : Uses mf spr (Move From Special Register) instruction to get
* the base address for the internal memory map stored in
* the Internal Memory Map Register (IMMR).
* *
*----------------------------------------------------------------------------*/
asm GetIMMR()
{
    nofralloc
    mf spr r3, 638
    blr
}
1.3 fpga.c

The code below, fpga.c, contains functions for reseting and configuring the Altera FPGA on the ipEngine.

#include "mpc823.h"
#include "fpga.h"

FPGA_Status(EPPC *immr) {
    int i;
    if (i=((immr->pcmcia.pipr) & OxCO00) != OxCO00) {
        printf("FPGA Load Error \n", i);
        return -1;
    }
    printf("FPGA Load OK \n", i);
    return 0;
}

FPGA_Reset(EPPC *immr) {
    int i;
    volatile static int dummy = 0;
    //Set PC13 (FPGA's nCONFIG) to output
    immr->pio.pcdir |= Ox0004;
    immr->pio.pcdat &= ~ (Ox0004);
    for(i=0; i<1000; i++) dummy++;
    immr->pio.pcdat |= Ox0004;
    return 0;
}

FPGA_Config(EPPC *immr, FPGA *fpga) {
    int n, t, i, j;
    char b;
    FPGA_Reset(immr);
The code below, fpga.h, contains function prototypes for fpga.c as well as type definitions for the FPGA memory structure.

```
/*-------1---------2---------3---------4---------5---------6---------7---------
123456789-123456789-123456789-123456789-123456789-123456789-123456789-123456789
* 
*/
#include "mpc823.h"

typedef struct fpga_config_register {
  unsigned long config_reg;
} FPGA;
```

## 1.4 fpga.h

The code below, fpga.h, contains function prototypes for fpga.c as well as type definitions for the FPGA memory structure.

```
/*-------1---------2---------3---------4---------5---------6---------7---------
123456789-123456789-123456789-123456789-123456789-123456789-123456789-123456789
* 
*/
```

```
#include "mpc823.h"

typedef struct fpga_config_register {
  unsigned long config_reg;
} FPGA;
```

```
typedef struct fpga_memory_map{
    /* FPGA GPIO Port A Data Register */
    unsigned short ia15_00;
    unsigned short ia31_16;
    unsigned short ia43_32;
    /* FPGA GPIO Port B Data Register */
    unsigned short ib15_00;
    unsigned short ib31_16;
    unsigned short ib43_32;
    /* FPGA GPIO Port A Data Dir */
    unsigned short ia15_dir;
    unsigned short ia31_dir;
    unsigned short ia43_dir;
    /* FPGA GPIO Port B Data Dir */
    unsigned short ib15_dir;
    unsigned short ib31_dir;
    unsigned short ib43_dir;
}FPGA_MAP;

char fpga_image[];

int FPGA_Status(EPPC *immr);
int FPGA_Reset(EPPC *immr);
int FPGA_Config(EPPC *immr, FPGA *fpga);

I.5 util.c

The code below, util.c, contains all of the utility functions for reading buttons, outputing to LEDs, initializing, reading, and writting to the registers of both the MIDI board and the DAC board.

errorCode = FPGA_Reset(immr);
errorCode = FPGA_Config(immr, fpga);
errorCode = FPGA_Status(immr);

#include "mpc823.h"
#include "util.h"
//include "fpga.h"

int WAIT_TIME = 600;

Led(EPPC *immr, int led, int val){
  if ((led < 0) || (led > 3)) return -1;
  // enable led output
  if ((val < 0) || (val > 1)) {
    immr->pio.pddir &= ~(1<<(led+5));
    return 0;
  }
  else
    immr->pio.pddir |= 1<<(led+5); // enable pin as output
  if (val == 0)
    immr->pio.pddat |= 1<<(led+5); // turn off -- (set pin high)
  else
    immr->pio.pddat &= ~(1<<(led+5)); // turn on -- (set pin low)
  return 0;
}

MIDI_Led(FPGA_MAP *fpga_ports, int val){
  switch (val){
    case 0:
      fpga_ports->ia43_32 = 0x0003;
      //printf("case 0 \r\n");
      break;
    case 1:
      fpga_ports->ia43_32 = 0x0002;
      //printf("case 1 \r\n");
      break;
    case 2:
      fpga_ports->ia43_32 = 0x0001;
      //printf("case 2 \r\n");
      break;
    case 3:
      fpga_ports->ia43_32 = 0x0000;
      //printf("case 3 \r\n");
      break;
    default:
      fpga_ports->ia43_32 = 0x0003;
      //printf("case 4 \r\n");
      break;
  }
  return 0;
}

MIDI_Button(FPGA_MAP *fpga_ports){
  int button;
  button = (fpga_ports->ia43_32 & 0x000c) >> 2;
  //printf("Button: %i \r\n",button);
  return button;
}
MIDI.Initialize(FPGA_MAP *fpga_ports){
  /* Config MIDI Brd Leds */
  fpga_ports->ia43_dir = 0x0003;

  return 0;
}

Continuum.Initialize(FPGA_MAP *fpga_ports){
  fpga_ports->ia15_dir = 0x0000; // All input for testing
  return 0;
}

char Read_Cont_Byte(FPGA_MAP *fpga_ports, int port){
  char data;

  // Set Data Direction 13-8 Out, 7-0 In;
  fpga_ports->ia15_dir = 0x3f00;
  // *HEN = *RESET = 1, R/*W = 1
  fpga_ports->ia15_00 = 0x3800;
  // Set Address HA2-0
  switch(port){
    case 0:
      fpga_ports->ia15_00 = 0x3800;
      break;
    case 1:
      fpga_ports->ia15_00 = 0x3900;
      break;
    case 2:
      fpga_ports->ia15_00 = 0x3a00;
      break;
    case 3:
      fpga_ports->ia15_00 = 0x3b00;
      break;
    default:
      fpga_ports->ia15_00 = 0x3800;
      break;
  }

  data = fpga_ports->ia15_00;
  printf("DATA: \%01x \r\n",data);
  return data;
}

Write_Cont_Byte(FPGA_MAP *fpga_ports, char data){
  // Set Data Direction 13-8 Out, 7-0 In;
  fpga_ports->ia15_dir = 0x3f00;
  // *HEN = *RESET = 1, R/*W = 0, HA2-0 = 000
  fpga_ports->ia15_00 = 0x3800;
  // Set Data Direction 13-8 Out, 7-0 Out;
  fpga_ports->ia15_dir = 0x3fff;
  
  data = fpga_ports->ia15_00;
  printf("DATA: \%01x \r\n",data);
  return data;
}
// Output data
fpga_ports->ia15_00 |= data;

// Toggle *HEN
fpga_ports->ia15_00 &= ~(0x1000);
fpga_ports->ia15_00 |= 0x1000;
// Set Data Direction 13-8 Out, 7-0 In;
fpga_ports->ia15_dir = 0x3f00;
return 0;
}

Dac_Initialize(FPGA_MAP *fpga_ports){

    // Set Data Direction 13-8 Out, 7-0 In;
fpga_ports->ia31_dir = 0x3f00;
    Wait(WAIT_TIME);
    // *HEN = *RESET = 1, R/*W = 0
fpga_ports->ia31_16 = 0x3000;
    Wait(WAIT_TIME);
    // *HEN = 1 *RESET = 0, R/*W = 0
fpga_ports->ia31_16 = 0x1000;
    Wait(WAIT_TIME);
    // Clear FIFO
    Write_Dac_Byte(fpga_ports,0x01,0x04);
    Wait(WAIT_TIME);
    // Reset Dacs
    Write_Dac_Byte(fpga_ports,0x01,0x10);
    Wait(WAIT_TIME);
    return 0;
}

unsigned char Read_Dac_Byte(FPGA_MAP *fpga_ports, unsigned char addr){
    unsigned char data;

    // Set Data Direction 13-8 Out, 7-0 In;
fpga_ports->ia31_dir = 0x3f00;
    // *HEN = *RESET = 1, R/*W = 1
fpga_ports->ia31_16 = 0x3800;
    // Set Address HA2-0
fpga_ports->ia31_16 |= (addr & 0x07)<<8;
    // Toggle *HEN (Enable DAC Output)
    fpga_ports->ia31_16 &= ~(0x1000);
    Wait(WAIT_TIME);

data = fpga_ports->ia31_16;
    // Toggle *HEN (Disable DAC Output)
fpga_ports->ia31_16 |= 0x1000;
    //printf("DATA: %02x \r\n",data);
return data;
}
Write_Dac_Byte(FPGA_MAP *fpga_ports, unsigned char addr, unsigned char data) {
    // Set Data Direction 13-8 Out, 7-0 In;
    fpga_ports->ia31_dir = 0x3f00;
    // *HEN = *RESET = 1, R/W = 0
    fpga_ports->ia31_16 = 0x3000;
    // Set Address HA2-0
    fpga_ports->ia31_16 |= (addr & 0x07)<<8;
    // Set Data Direction 13-8 Out, 7-0 Out;
    fpga_ports->ia31_dir = 0x3fff;
    // Output data
    fpga_ports->ia31_16 &= 0x3f00;
    fpga_ports->ia31_16 |= ((long) data & 0x00ff);
    // Toggle *HEN
    fpga_ports->ia31_16 &= ~(0x1000);
    Wait(WAIT_TIME);
    fpga_ports->ia31_16 |= 0x1000;
    // Set Data Direction 13-8 Out, 7-0 In;
    fpga_ports->ia31_16 = 0x3f00;
    return 0;
}

Wait(int cycles){
    int i;
    //Burning Cycles
    for(i=0; i<cycles; i++){};
    return 0;
}

1.6 util.h

The code below, util.h, contains the function prototypes for util.c.

#include "mpc823.h"
#include "fpga.h"
int Led(EPPC *immr, int, int);
int MIDI_Led(FPGA_MAP *fpga_ports, int);
int MIDI_Button(FPGA_MAP *fpga_ports);
int MIDI_Initialize(FPGA_MAP *fpga_ports);

int Continuum_Initialize(FPGA_MAP *fpga_ports);
char Read_Cont_Byte(FPGA_MAP *fpga_ports, int);
int Write_Cont_Byte(FPGA_MAP *fpga_ports, char);

int Dac_Initialize(FPGA_MAP *fpga_ports);
unsigned char Read_Dac_Byte(FPGA_MAP *fpga_ports, unsigned char);
int Write_Dac_Byte(FPGA_MAP *fpga_ports, unsigned char, unsigned char);
int Wait(int);

I.7 fpga_image.c

The code below, fpga_image.c, contains the bitstream data for programming the Al­
tera FPGA on the ipEngine with sample memmory mapped I/O code. In order to
conserve space the body of the data is omitted.

char fpga_image[] = {
255,255, 98,125, 20,255,255,255,255, ...

    /* In the interest of space the actual contents
        of the FPGA bitfile have been omitted */

    ... , 4, 8, 0, 0,179, 255,255
};
APPENDIX J

DAC BOARD VHDL

The following code listing is VHDL language code for the Xilinx XC4013E FPGA used on the DAC board. All code was compiled using Foundation 3.1i tools from Xilinx.

J.1 dac_dac.vhd

The code below, dac_dac.vhd, is the VHDL for implementing direct writes to the DAC registers on the DAC board.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

-- Dac Board VHDL for Xilinx XC4013E FPGA
-- KZ 11/2001
-- This code version updates DACs through direct register commands from PowerPC.
entity dacboard is

-- PIN/Signal Definitions
port (  
  LEDS : out std_logic_vector (3 downto 0);
  UIOA : out std_logic_vector (7 downto 0);
  UIOB : out std_logic_vector (7 downto 0);
  UIOC : out std_logic_vector (7 downto 0);
  UIOD : out std_logic_vector (3 downto 0);
  SDI_0 : out std_logic;
  CLK_0 : out std_logic;
  LDAC_0 : out std_logic;
  LOAD_0 : out std_logic;
); 
end dacboard;
```

112
architecture arch_dacboard of dacboard is

-- Internal Signal Definitions
signal Clk: std_logic;
signal RST: std_logic;
signal Internal_counter: integer range 0 to 127;
signal Test_counter: integer range 0 to 255;
signal Toggle: std_logic;
signal Button_reg: std_logic_vector (3 downto 0);
signal Data_7_O_reg: std_logic_vector (7 downto 0);
signal Data_15_8_reg: std_logic_vector (7 downto 0);
signal Data_23_16_reg: std_logic_vector (7 downto 0);
signal Control_reg: std_logic_vector (7 downto 0);
signal Status_reg: std_logic_vector (7 downto 0);
signal Data_reg: std_logic_vector (7 downto 0);
signal Control_handshake: std_logic;
signal State: std_logic_vector (7 downto 0);
signal Dac_state: std_logic_vector (3 downto 0);
signal Dac_data: std_logic_vector (23 downto 0);

-- Misc Constants
constant board_id : std_logic_vector (7 downto 0) := x"a1";

-- Address State Constants
constant id : std_logic_vector (2 downto 0) := "000";
constant command : std_logic_vector (2 downto 0) := "001";
constant status : std_logic_vector (2 downto 0) := "010";
constant unused0 : std_logic_vector (2 downto 0) := "011";
constant data_7_0 : std_logic_vector (2 downto 0) := "100";
constant data_15_8 : std_logic_vector (2 downto 0) := "101";
constant data_23_16 : std_logic_vector (2 downto 0) := "110";
constant unused1 : std_logic_vector (2 downto 0) := "111";

-- Command Register Constants
constant change_rstsel : std_logic_vector (7 downto 0) := x"05";
constant read_buttons : std_logic_vector (7 downto 0) := x"06";
constant write_leds : std_logic_vector (7 downto 0) := x"07";
constant reset_dacs : std_logic_vector (7 downto 0) := x"10";
constant write_dac : std_logic_vector (7 downto 0) := x"12";
constant update_dacs : std_logic_vector (7 downto 0) := x"13";

-- Control State Constants
constant state_idle : std_logic_vector (7 downto 0) := x"00";
constant state_parse : std_logic_vector (7 downto 0) := x"11";
constant state_wait : std_logic_vector (7 downto 0) := x"12";
constant state_decode_addr : std_logic_vector (7 downto 0) := x"13";
constant state_change_rstsel : std_logic_vector (7 downto 0) := x"05";
constant state_read_buttons : std_logic_vector (7 downto 0) := x"30";
constant state_read_buttons_1 : std_logic_vector (7 downto 0) := x"31";
constant state_write_leds : std_logic_vector (7 downto 0) := x"40";
constant state_reset_dacs : std_logic_vector (7 downto 0) := x"50";
constant state_reset_dacs_1 : std_logic_vector (7 downto 0) := x"51";
constant state_write_dac : std_logic_vector (7 downto 0) := x"e0";
constant state_write_dac_1 : std_logic_vector (7 downto 0) := x"e1";
constant state_write_dac_2 : std_logic_vector (7 downto 0) := x"e2";
constant state_write_dac_3 : std_logic_vector (7 downto 0) := x"e3";
constant state_write_dac_4 : std_logic_vector (7 downto 0) := x"e4";
constant state_write_dac_5 : std_logic_vector (7 downto 0) := x"e5";
constant state_update_dacs : std_logic_vector (7 downto 0) := x"f0";
constant state_update_dacs_1 : std_logic_vector (7 downto 0) := x"f1";
constant state_update_dacs_2 : std_logic_vector (7 downto 0) := x"f2";
constant state_update_dacs_3 : std_logic_vector (7 downto 0) := x"f3";
constant state_update_dacs_4 : std_logic_vector (7 downto 0) := x"f4";
constant state_update_dacs_5 : std_logic_vector (7 downto 0) := x"f5";

-- Dac Output State Constants
constant msb23 : std_logic_vector (7 downto 0) := x"a2";
constant bit22 : std_logic_vector (7 downto 0) := x"a3";
constant bit21 : std_logic_vector (7 downto 0) := x"a4";
constant bit20 : std_logic_vector (7 downto 0) := x"a5";
constant bit19 : std_logic_vector (7 downto 0) := x"a6";
constant bit18 : std_logic_vector (7 downto 0) := x"a7";
constant bit17 : std_logic_vector (7 downto 0) := x"a8";
constant bit16 : std_logic_vector (7 downto 0) := x"a9";
constant bit15 : std_logic_vector (7 downto 0) := x"aa";
constant bit14 : std_logic_vector (7 downto 0) := x"ab";
constant bit13 : std_logic_vector (7 downto 0) := x"ac";
constant bit12 : std_logic_vector (7 downto 0) := x"ad";
constant bit11 : std_logic_vector (7 downto 0) := x"ae";
constant bit10 : std_logic_vector (7 downto 0) := x"af";
constant bit9 : std_logic_vector (7 downto 0) := x"b0";
constant bit8 : std_logic_vector (7 downto 0) := x"b1";
constant bit7 : std_logic_vector (7 downto 0) := x"b2";
constant bit6 : std_logic_vector (7 downto 0) := x"b3";
constant bit5 : std_logic_vector (7 downto 0) := x"b4";
constant bit4 : std_logic_vector (7 downto 0) := x"b5";
constant bit3 : std_logic_vector (7 downto 0) := x"b6";
constant bit2 : std_logic_vector (7 downto 0) := x"b7";
constant bit1 : std_logic_vector (7 downto 0) := x"b8";
constant lsb0 : std_logic_vector (7 downto 0) := x"b9";
constant dataLatch : std_logic_vector (7 downto 0) := x"ba";
constant dac_update : std_logic_vector (7 downto 0) := x"bb";
constant dac_load_0 : std_logic_vector (7 downto 0) := x"bc";
constant dac_load_1 : std_logic_vector (7 downto 0) := x"bd";

-- Dac state constants
constant dac_out_0 : std_logic_vector (3 downto 0) := x"0";
constant dac_out_1 : std_logic_vector (3 downto 0) := x"1";
constant dac_out_2 : std_logic_vector (3 downto 0) := x"2";
constant dac_out_3 : std_logic_vector (3 downto 0) := x"3";
constant dac_out_4 : std_logic_vector (3 downto 0) := x"4";
constant dac_out_5 : std_logic_vector (3 downto 0) := x"5";
constant dac_idle : std_logic_vector (3 downto 0) := x"6";

component OSC4
port ( F15: out std_logic;
F16K: out std_logic;
F490: out std_logic;
F500K: out std_logic;
FSM: out std_logic);
end component;

COMPONENT ibuf PORT(i: IN STD_LOGIC; o: OUT STD_LOGIC); END COMPONENT;
COMPONENT bufg PORT(i: IN STD_LOGIC; o: OUT STD_LOGIC); END COMPONENT;
SIGNAL buf_clkO: STD_LOGIC;
begin
osc: OSC4 port map (FBM => Clk);

b0: ibuf PORT MAP(i=>RST_IN,o=>buf_clkO);
b1: bufg PORT MAP(i=>buf_clkO,o=>RST);

process (Clk) is
begin
if (Clk'event AND Clk='O') then
  Internal_counter <= Internal_counter + 1;
  if(Toggle='l') then
    Toggle <= '0';
  else
    Toggle <= '1';
  end if;
end if;
end process;

-- Port Interface and command parsing

process(RST,Clk) is
begin
if (RST = '0') then
  Data_7_0_reg <= x"00";
  Data_15_8_reg <= x"00";
  Data_23_16_reg <= x"00";
  Control_reg <= x"00";
  Status_reg <= x"00";
  Data_reg <= x"00";
  Button_reg <= x"0000";
  Dac_data <= x"0000000";
  State <= state_idle;
  LDAC_0 <= '1';
  LDAC_1 <= '1';
  LDAC_2 <= '1';
  Test_counter <= 0;
  LEDS <= not("0000");
  UIOC <= x"00";
  SDI_0 <= '0';
  LOAD_0 <= '1';
  CS_0 <= '1';
  RSTSEL_0 <= '0';
  RST_0 <= '1';
  SDI_1 <= '1';
  LOAD_1 <= '1';
  CS_1 <= '1';
  RSTSEL_1 <= '0';
  RST_1 <= '1';
  SDI_2 <= '1';
end if;
end process;
LOAD_2 <= '1';
CS_2 <= '1';
RSTSEL_2 <= '0';
RST_2 <= '1';

elseif (Clk'event and Clk='0') then
   -- Latch Variables so they don't change
   Data_7_0_reg <= Data_7_0_reg;
   Data_15_8_reg <= Data_15_8_reg;
   Data_23_16_reg <= Data_23_16_reg;
   Control_reg <= Control_reg;
   Status_reg <= Status_reg;
   Data_reg <= Data_reg;
   Button_reg <= Button_reg;
   Dac_data <= Dac_data;

   case State is

      when state_idle =>
         if(HEN = '0') then
            State <= state_decode_addr_0;
         else
            case BUTTONS is
               when "0001" =>
                  Button_reg <= Button_reg or "0001";
                  Status_reg(4) <= '1';
               when "0010" =>
                  Button_reg <= Button_reg or "0010";
                  Status_reg(4) <= '1';
               when "0100" =>
                  Button_reg <= Button_reg or "0100";
                  Status_reg(4) <= '1';
               when "1000" =>
                  Button_reg <= Button_reg or "1000";
                  Status_reg(4) <= '1';
               when others =>
                  Button_reg <= Button_reg or "0000";
            end case;
            State <= state_idle;
         end if;

      when state_decode_addr_0 =>
         case HA2_0 is
            when id =>
               Data_reg <= board_id;
               State <= state_hen_wait_0;

      when others =>
         State <= state_idle;
   end case;

end if;

end when;

when state_hen_wait_0 =>
   State <= state_decode_addr_1;
when command =>
   if (RW = '1') then
      Data_reg <= Control_reg;
      State <= state henne_wait_0;
   else
      Control_reg <= H8_0;
      State <= state_parse;
   end if;

when status =>
   if (RW = '1') then
      Data_reg <= Status_reg;
   else
      Status_reg <= 'H8_0;
   end if;
   State <= state henne_wait_0;

when data_7_0 =>
   if (RW = '1') then
      Data_reg <= Data_7_0_reg;
   else
      Data_7_0_reg <= H8_0;
   end if;
   State <= state henne_wait_0;

when data_15_8 =>
   if (RW = '1') then
      Data_reg <= Data_15_8_reg;
   else
      Data_15_8_reg <= H8_0;
   end if;
   State <= state henne_wait_0;

when data_23_16 =>
   if (RW = '1') then
      Data_reg <= Data_23_16_reg;
   else
      Data_23_16_reg <= H8_0;
   end if;
   State <= state henne_wait_0;

when others =>
   Data_reg <= x"ff"
   State <= state henne_wait_0;

end case;

when state_parse =>
   case Control_reg is
when change_rstsel =>
  State <= state_change_rstsel_0;
when read_buttons =>
  State <= state_read_buttons_0;
when write_leds =>
  State <= state_write_leds_0;
when reset_dacs =>
  State <= state_reset_dacs_0;
when write_dac =>
  State <= state_write_dac_0;
when update_dacs =>
  State <= state_update_dacs_0;
when others =>
  State <= state_hen_wait_0;
end case;

when state_change_rstsel_0 =>
  RSTSEL_0 <= not RSTSEL_0;
  RSTSEL_1 <= not RSTSEL_1;
  RSTSEL_2 <= not RSTSEL_2;
  State <= state_hen_wait_0;
when state_read_buttons_0 =>
  Data_7_0_reg <= ("0000" & Button_reg);
  State <= state_read_buttons_1;
when state_read_buttons_1 =>
  Button_reg <= "0000";
  Status_reg(4) <= '0' ;
  State <= state_hen_wait_0;
when state_write_leds_0 =>
  LEDs <= Data_7_0_reg(3 downto 0);
  State <= state_hen_wait_0;
when state_reset_dacs_0 =>
  RST_0 <= '0';
  RST_1 <= '0';
  RST_2 <= '0';
  State <= state_reset_dacs_1;
when state_reset_dacs_1 =>
  RST_0 <= '1';
  RST_1 <= '1';
RST_2 <= '1';
State <= state_hen_wait_O;

when state_hen_wait_O =>
if (HEN = '1') then
State <= state_idle;
else
State <= state_hen_wait_O;
end if;

when state_write_dac_0 =>
Dac_Data(23 downto 16)<= Data_23_16_reg;
Dac_Data(15 downto 8)<= Data_15_8_reg;
Dac_Data(7 downto 0)<= Data_7_0_reg;
State <= msb23;

when state_update_dacs_0 =>
LDAC_0 <= '0';
LDAC_1 <= '0';
LDAC_2 <= '0';
State <= state_update_dacs_1;

when state_update_dacs_1 =>
LDAC_0 <= '1';
LDAC_1 <= '1';
LDAC_2 <= '1';
State <= state_update_dacs_2;

when state_update_dacs_2 =>
Test_counter <= Test_counter + 1;
State <= state_idle;

-- Dac Loading Section

when msb23 =>
case (Dac_data(17 downto 16)) is
when "00" =>
SDI_0 <= Dac_data(23);
CS_0 <= '0';
when "01" =>
SDI_1 <= Dac_data(23);
CS_1 <= '0';
when "10" =>
SDI_2 <= Dac_data(23);
CS_2 <= '0';
when others =>
end case;
State <= bit22;

when bit22 =>
SDI_0 <= Dac_data(22);
SDI_1 <= Dac_data(22);
SDI_2 <= Dac_data(22);

120
State <= bit21;

when bit21 =>
    SDI_0 <= Dac_data(21);
    SDI_1 <= Dac_data(21);
    SDI_2 <= Dac_data(21);
    State <= bit20;

when bit20 =>
    SDI_0 <= '0';
    SDI_1 <= '0';
    SDI_2 <= '0';
    State <= bit19;

when bit19 =>
    SDI_0 <= '0';
    SDI_1 <= '0';
    SDI_2 <= '0';
    State <= bit18;

when bit18 =>
    SDI_0 <= '0';
    SDI_1 <= '0';
    SDI_2 <= '0';
    State <= bit17;

when bit17 =>
    SDI_0 <= '0';
    SDI_1 <= '0';
    SDI_2 <= '0';
    State <= bit16;

when bit16 =>
    SDI_0 <= '0';
    SDI_1 <= '0';
    SDI_2 <= '0';
    State <= bit15;

when bit15 =>
    SDI_0 <= Dac_data(15);
    SDI_1 <= Dac_data(15);
    SDI_2 <= Dac_data(15);
    State <= bit14;

when bit14 =>
    SDI_0 <= Dac_data(14);
    SDI_1 <= Dac_data(14);
    SDI_2 <= Dac_data(14);
    State <= bit13;

when bit13 =>
    SDI_0 <= Dac_data(13);
    SDI_1 <= Dac_data(13);
SDI_2 <= Dac_data(13);
State <= bit12;

when bit12 =>
SDI_0 <= Dac_data(12);
SDI_1 <= Dac_data(12);
SDI_2 <= Dac_data(12);
State <= bit11;

when bit11 =>
SDI_0 <= Dac_data(11);
SDI_1 <= Dac_data(11);
SDI_2 <= Dac_data(11);
State <= bit10;

when bit10 =>
SDI_0 <= Dac_data(10);
SDI_1 <= Dac_data(10);
SDI_2 <= Dac_data(10);
State <= bit9;

when bit9 =>
SDI_0 <= Dac_data(9);
SDI_1 <= Dac_data(9);
SDI_2 <= Dac_data(9);
State <= bit8;

when bit8 =>
SDI_0 <= Dac_data(8);
SDI_1 <= Dac_data(8);
SDI_2 <= Dac_data(8);
State <= bit7;

when bit7 =>
SDI_0 <= Dac_data(7);
SDI_1 <= Dac_data(7);
SDI_2 <= Dac_data(7);
State <= bit6;

when bit6 =>
SDI_0 <= Dac_data(6);
SDI_1 <= Dac_data(6);
SDI_2 <= Dac_data(6);
State <= bit5;

when bit5 =>
SDI_0 <= Dac_data(5);
SDI_1 <= Dac_data(5);
SDI_2 <= Dac_data(5);
State <= bit4;

when bit4 =>
SDI_0 <= Dac_data(4);
SDI_1 <= Dac_data(4);
SDI_2 <= Dac_data(4);
State <= bit3;

when bit3 =>
SDI_0 <= Dac_data(3);
SDI_1 <= Dac_data(3);
SDI_2 <= Dac_data(3);
State <= bit2;

when bit2 =>
SDI_0 <= Dac_data(2);
SDI_1 <= Dac_data(2);
SDI_2 <= Dac_data(2);
State <= bit1;

when bit1 =>
SDI_0 <= Dac_data(1);
SDI_1 <= Dac_data(1);
SDI_2 <= Dac_data(1);
State <= lsb0;

when lsb0 =>
SDI_0 <= Dac_data(0);
SDI_1 <= Dac_data(0);
SDI_2 <= Dac_data(0);
State <= data_latch;

when data_latch =>
case (Dac_data(17 downto 16)) is
  when "00" =>
    CS_0 <= '1';
  when "01" =>
    CS_1 <= '1';
  when "10" =>
    CS_2 <= '1';
  when others =>
end case;
State <= dac_load_0;

when dac_load_0 =>
case (Dac_data(17 downto 16)) is
  when "00" =>
    LOAD_0 <= '0';
  when "01" =>
    LOAD_1 <= '0';
  when "10" =>
    LOAD_2 <= '0';
  when others =>
end case;
State <= dac_load_1;

when dac_load_1 =>

123
case (Dac_data(17 downto 16)) is
  when "00" =>
    LOAD_0 <= '1';
  when "01" =>
    LOAD_1 <= '1';
  when "10" =>
    LOAD_2 <= '1';
  when others =>
end case;
State <= state_hen_wait_0;

when others =>
  State <= state_idle;
end case;
end if;
end process;

-- Tri-State
  H8_0 <= Data_reg when (HEN = '0' and RW = '1') else (others => 'Z');
  TR <= '0' when (HEN = '0' and RW = '1') else '1';

-- Clock Routing
  CLK_0 <= Clk;
  CLK_1 <= Clk;
  CLK_2 <= Clk;

-- Test Outputs
  UIOD(0) <= SDO_0;
  UIOD(1) <= SDO_1;
  UIOD(2) <= SDO_2;
  UIOD(3) <= RST;
  UIOA <= CONV_STD_LOGIC_VECTOR(Test_counter, 8);
  UIOB <= ("00000000" & HEN);
end arch_dacboard;

J.2 dac_fifo.vhd

The code below, dac_fifo.vhd, is the VHDL for implementing writes to the DAC registers on the DAC Board via a FIFO based producer/consumer process.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library work;
use work.ram_lib.all;
use work.free_fifo.all;

-- Dac Board VHDL for Xilinx XC4013E FPGA
-- KZ 11/2001
-- This code version updates DACs through FIFO controlled
-- by register commands from PowerPC.

-- synopsys translate_off

--Library xilinxcorelib;

--configuration cfg_dacboard of dacboard is
-- for arch_dacboard
--for all : dac_fifo use entity XilinxCoreLib.fifosyncVHT(behavioral)
--generic map(
--Depth => 64,
--Create_RLOCs_for_TBUFs => 0,
--Dual_Port => 0,
--Port_Width => 24,
--Address_Width => 6);
--end for;
-- end for;
--end cfg_dacboard;

-- synopsys translate_on

entity dacboard is

-- PIN/Signal Definitions:
port (
  LEDS : out std_logic_vector (3 downto 0);
  UIDA : out std_logic_vector (7 downto 0);
  UIDB : out std_logic_vector (7 downto 0);
  UIDC : out std_logic_vector (7 downto 0);
  UIOD : out std_logic_vector (3 downto 0);
  SDI_0 : out std_logic;
  CLK_0 : out std_logic;
  LDAC_0 : out std_logic;
  LOAD_0 : out std_logic;
  CS_0 : out std_logic;
  SDO_0 : in std_logic;
  RSTSEL_0 : inout std_logic;
  RST_0 : out std_logic;
  SDI_1 : out std_logic;
  CLK_1 : out std_logic;
  LDAC_1 : out std_logic;
  LOAD_1 : out std_logic;
  CS_1 : out std_logic;
);
architecture arch_dacboard of dacboard is

-- Internal Signal Definitions

signal Clk: std_logic;
signal RST: std_logic;
signal Run_bit: std_logic;
signal Sync_bit_consume: std_logic;
signal Sync_bit_provide: std_logic;
signal Internal_counter : integer range 0 to 127;
signal Test_counter : integer range 0 to 255;
--signal Clk_reg : std_logic_vector (6 downto 0);
signal Toggle: std_logic;
signal Button_reg : std_logic_vector (3 downto 0);
signal Data_7_0_reg: std_logic_vector (7 downto 0);
signal Data_15_8_reg: std_logic_vector (7 downto 0);
signal Data_23_16_reg: std_logic_vector (7 downto 0);
signal Control_reg: std_logic_vector (7 downto 0);
signal Status_reg: std_logic_vector (7 downto 0);
signal Data_reg: std_logic_vector (7 downto 0);
signal Control_handshake: std_logic;
signal State: std_logic_vector (7 downto 0);
signal Return_state: std_logic_vector (7 downto 0);
signal Dac_state: std_logic_vector (3 downto 0);

signal Dac_data: std_logic_vector (23 downto 0);
-- Fifo Signals
signal d: std_logic_vector(23 downto 0);
signal we: std_logic;
signal re: std_logic;
signal reset: std_logic;
signal c: std_logic;
signal full: std_logic;
signal empty: std_logic;
--signal bufctr_ce: std_logic;
--signal bufctr_updn: std_logic;
signal q: std_logic_vector(23 downto 0);

signal er: std_logic;

-- Hise Constants
constant board_id : std_logic_vector (7 downto 0) := x"a111";
-- Address State Constants
constant id std_logic_vector (2 downto 0) := x"000";
constant command std_logic_vector (2 downto 0) := x"001";
constant status std_logic_vector (2 downto 0) := x"010";
constant unused0 std_logic_vector (2 downto 0) := x"011";
constant data_7_0 std_logic_vector (2 downto 0) := x"0100";
constant data_15_8 std_logic_vector (2 downto 0) := x"0101";
constant data_23_16 std_logic_vector (2 downto 0) := x"0110";
constant unused1 std_logic_vector (2 downto 0) := x"0111";

-- Command Register Constants
constant load_fifo : std_logic_vector (7 downto 0) := x"01";
constant start_output : std_logic_vector (7 downto 0) := x"02";
constant stop_output : std_logic_vector (7 downto 0) := x"03";
constant clear_fifo : std_logic_vector (7 downto 0) := x"04";
constant change_rstsel : std_logic_vector (7 downto 0) := x"05";
constant read_buttons : std_logic_vector (7 downto 0) := x"06";
constant write_leds : std_logic_vector (7 downto 0) := x"07";
--constant read_uioa : std_logic_vector (7 downto 0) := x"08";
--constant read_uiob : std_logic_vector (7 downto 0) := x"09";
--constant read_uioc : std_logic_vector (7 downto 0) := x"0a";
--constant write_uioa : std_logic_vector (7 downto 0) := x"0b";
--constant write_uiob : std_logic_vector (7 downto 0) := x"0c";
--constant write_uioc : std_logic_vector (7 downto 0) := x"0d";
--constant write_uioa : std_logic_vector (7 downto 0) := x"0f";
constant reset_dacs : std_logic_vector (7 downto 0) := x"10";
constant read_fifo : std_logic_vector (7 downto 0) := x"11";

-- Control State Constants
constant state_idle : std_logic_vector (7 downto 0) := x"00";
constant state_start_output: std_logic_vector (7 downto 0) := x"02";
constant state_stop_output: std_logic_vector (7 downto 0) := x"03";
--constant state_read_uioa : std_logic_vector (7 downto 0) := x"08";
--constant state_read_uiob : std_logic_vector (7 downto 0) := x"09";
--constant state_read_uioc : std_logic_vector (7 downto 0) := x"0a";
--constant state_read_uiod : std_logic_vector (7 downto 0) := x"0b";
--constant state_write_uioa : std_logic_vector (7 downto 0) := x"0c";
--constant state_write_uiob : std_logic_vector (7 downto 0) := x"0d";
--constant state_write_uioc : std_logic_vector (7 downto 0) := x"0e";
--constant state_write_uiod : std_logic_vector (7 downto 0) := x"0f";

constant state_parse : std_logic_vector (7 downto 0) := x"11";
constant state_hen_wait_0 : std_logic_vector (7 downto 0) := x"12";
constant state_decode_addr_0 : std_logic_vector (7 downto 0) := x"13";
constant state_load_fifo_0 : std_logic_vector (7 downto 0) := x"20";
constant state_load_fifo_1 : std_logic_vector (7 downto 0) := x"21";
constant state_load_fifo_2 : std_logic_vector (7 downto 0) := x"22";
constant state_read_fifo_0 : std_logic_vector (7 downto 0) := x"23";
constant state_read_fifo_1 : std_logic_vector (7 downto 0) := x"24";
constant state_read_fifo_2 : std_logic_vector (7 downto 0) := x"25";
constant state_read_fifo_3 : std_logic_vector (7 downto 0) := x"26";
constant state_clear_fifo_0 : std_logic_vector (7 downto 0) := x"27";
constant state_clear_fifo_1 : std_logic_vector (7 downto 0) := x"28";
constant state_clear_fifo_2 : std_logic_vector (7 downto 0) := x"29";
constant state_update_flags_0 : std_logic_vector (7 downto 0) := x"2a";
constant state_update_flags_1 : std_logic_vector (7 downto 0) := x"2b";
constant state_update_flags_2 : std_logic_vector (7 downto 0) := x"2c";
constant state_read_fifo_4 : std_logic_vector (7 downto 0) := x"2d";
constant state_load_fifo_3 : std_logic_vector (7 downto 0) := x"2e";
constant state_change_rstsel_0 : std_logic_vector (7 downto 0) := x"05";
constant state_read_buttons_0 : std_logic_vector (7 downto 0) := x"30";
constant state_read_buttons_1 : std_logic_vector (7 downto 0) := x"31";
constant state_write_leds_0 : std_logic_vector (7 downto 0) := x"40";
constant state_reset_dacs_0 : std_logic_vector (7 downto 0) := x"50";
constant state_reset_dacs_1 : std_logic_vector (7 downto 0) := x"51";
constant state_provider_0 : std_logic_vector (7 downto 0) := x"d0";
constant state_provider_1 : std_logic_vector (7 downto 0) := x"d1";
constant state_provider_2 : std_logic_vector (7 downto 0) := x"d2";
constant state_provider_3 : std_logic_vector (7 downto 0) := x"d3";
constant state_provider_4 : std_logic_vector (7 downto 0) := x"d4";
constant state_provider_5 : std_logic_vector (7 downto 0) := x"d5";

128
-- Dac Output State Constants

--constant dac_idle : std_logic_vector (7 downto 0) := x"a0";
cconstant msb23 : std_logic_vector (7 downto 0) := x"a2";
cconstant bit22 : std_logic_vector (7 downto 0) := x"a3";
cconstant bit21 : std_logic_vector (7 downto 0) := x"a4";
cconstant bit20 : std_logic_vector (7 downto 0) := x"a5";
cconstant bit19 : std_logic_vector (7 downto 0) := x"a6";
cconstant bit18 : std_logic_vector (7 downto 0) := x"a7";
cconstant bit17 : std_logic_vector (7 downto 0) := x"a8";
cconstant bit16 : std_logic_vector (7 downto 0) := x"a9";
cconstant bit15 : std_logic_vector (7 downto 0) := x"aa";
cconstant bit14 : std_logic_vector (7 downto 0) := x"ab";
cconstant bit13 : std_logic_vector (7 downto 0) := x"ac";
cconstant bit12 : std_logic_vector (7 downto 0) := x"ad";
cconstant bit11 : std_logic_vector (7 downto 0) := x"ae";
cconstant bit10 : std_logic_vector (7 downto 0) := x"af";
cconstant bit9 : std_logic_vector (7 downto 0) := x"b0";
cconstant bit8 : std_logic_vector (7 downto 0) := x"b1";
cconstant bit7 : std_logic_vector (7 downto 0) := x"b2";
cconstant bit6 : std_logic_vector (7 downto 0) := x"b3";
cconstant bit5 : std_logic_vector (7 downto 0) := x"b4";
cconstant bit4 : std_logic_vector (7 downto 0) := x"b5";
cconstant bit3 : std_logic_vector (7 downto 0) := x"b6";
cconstant bit2 : std_logic_vector (7 downto 0) := x"b7";
cconstant bit1 : std_logic_vector (7 downto 0) := x"b8";
cconstant lsb0 : std_logic_vector (7 downto 0) := x"b9";
cconstant data_latch : std_logic_vector (7 downto 0) := x"ba";
cconstant dac_update : std_logic_vector (7 downto 0) := x"bb";
cconstant dac_load_0 : std_logic_vector (7 downto 0) := x"bc";
cconstant dac_load_1 : std_logic_vector (7 downto 0) := x"bd";

-- Dac state constants

cconstant dac_out_0 : std_logic_vector (3 downto 0) := x"0";
cconstant dac_out_1 : std_logic_vector (3 downto 0) := x"1";
cconstant dac_out_2 : std_logic_vector (3 downto 0) := x"2";
cconstant dac_out_3 : std_logic_vector (3 downto 0) := x"3";
cconstant dac_out_4 : std_logic_vector (3 downto 0) := x"4";
cconstant dac_out_5 : std_logic_vector (3 downto 0) := x"5";
cconstant dac_idle : std_logic_vector (3 downto 0) := x"6";

component OSC4
port ( F15:out std_logic;
F16K:out std_logic;
F490:out std_logic;
F500K:out std_logic;
F8M:out std_logic);
end component;

--component dac_fifo
--port (d IN std_logic_VECTOR(23 downto 0);
--we: IN std_logic;
--re: IN std_logic;
--reset: IN std_logic;
--c: IN std_logic;
--full: OUT std_logic;
--empty: OUT std_logic;
--bufctr_ce: OUT std_logic;
--bufctr_updn: OUT std_logic;
--q: OUT std_logic_VECTOR(23 downto 0));
--end component;

component fifo_async
  generic (data_bits : integer := 24;
            addr_bits : integer := 6;
            block_type : integer := 1;
            fifo_arch : integer := 0); -- 0=Generic architecture,
            -- 1=Xilinx XAPP131,
            -- 2=Xilinx XAPP131 w/carry mux
port (reset :in std_logic;
      wr_clk :in std_logic;
      wr_en :in std_logic;
      wr_data :in std_logic_vector (data_bits downto 0);
      rd_clk :in std_logic;
      rd_en :in std_logic;
      rd_data :out std_logic_vector (data_bits downto 0);
      full :out std_logic;
      empty :out std_logic
    );
end component;

COMPONENT ibuf PORT(i: IN STD_LOGIC; o: OUT STD_LOGIC); END COMPONENT;
COMPONENT bufg PORT(i: IN STD_LOGIC; o: OUT STD_LOGIC); END COMPONENT;
SIGNAL buf_clkO: STD_LOGIC;
begin
osc: OSC4 port map (FSM => Clk);
b0: ibuf PORT MAP(i=>RST_IN,o=>buf_clkO); -- buffer the clock from
b1: bufg PORT MAP(i=>buf_clk0,o=>RST); -- the keyboard

-- U1: BUFGS port map (I => RST_IN, O => RST);

--single_port_dac_fifo : dac_fifo
--port map (  
--d => d,  
--we_en => we,  
--rd_en => re,  
--reset => reset,  
--c => c,  
--full => full,  
--empty => empty,  

130
async_dac_fifo : fifo_async
port map (
  wr_data => d,
  wr_en => we,
  rd_en => re,
  reset => reset,
  wr_clk => c,
  full => full,
  empty => empty,
  rd_clk => er,
  rd_data => q);

process (Clk) is
begin
  if (Clk'event AND Clk='0') then
    Internal_counter <= Internal_counter + 1;
    if (Toggle='1') then
      Toggle <= '0';
    else
      Toggle <= '1';
    end if;
  end if;
end process;

-- Port Interface and command parsing

process(RST, Clk) is
begin
  if (RST = '0') then
    Data_7_0_reg <= x"00";
    Data_15_8_reg <= x"00";
    Data_23_16_reg <= x"00";
    Control_reg <= x"00";
    Status_reg <= x"00";
    Data_reg <= x"00";
    Button_reg <= x"0000";
    Dac_data <= x"000000";
    State <= state_idle;
    Return_state <= state_idle;
    -- Fifo Signals
    d <= x"000000";
  end if;
end process;
we <= '0';
re <= '0';
reset <= '0';
c <= '0';

LEDS <= not("0000");
--UIOA <= x"00";
--UIOB <= x"00";
UIOC <= x"00";
--UIDD <= x"00";
SDI_0 <= '0';
LOAD_0 <= '1';

CS_0 <= '1';
RSTSEL_0 <= '0';
RST_0 <= '1';
SDI_1 <= '0';
LOAD_1 <= '1';
CS_1 <= '1';
RSTSEL_1 <= '0';
RST_1 <= '1';
SDI_2 <= '0';
LOAD_2 <= '1';
CS_2 <= '1';
RSTSEL_2 <= '0';
RST_2 <= '1';
Run_bit <= '0';
Sync_bit_provide <= '0';

elsif (Clk'event and Clk='0') then
  -- Latch Variables so they don't change
  Data_7_0_reg <= Data_7_0_reg;
  Data_15_8_reg <= Data_15_8_reg;
  Data_23_16_reg <= Data_23_16_reg;
  Control_reg <= Control_reg;
  Status_reg <= Status_reg;
  Data_reg <= Data_reg;
  Button_reg <= Button_reg;
  Dac_data <= Dac_data;
  d <= d;
  we <= we;
  re <= re;
  reset <= reset;
  c <= c;
  Run_bit <= Run_bit;
  Sync_bit_provide <= Sync_bit_provide;
  Return_state <= Return_state;

  case State is
    when state_idle =>
if (HEN = '0') then
    State <= state_decode_addr_O;
elsif (Run_bit = '1' and (Sync_bit_provide /= Sync_bit_consume)) then
    State <= state_provider_O;
else
    case BUTTONS is
        when "0001" =>
            Button_reg <= Button_reg or "0001"
            Status_reg(4) <= '1';
        when "0010" =>
            Button_reg <= Button_reg or "0010"
            Status_reg(4) <= '1';
        when "0100" =>
            Button_reg <= Button_reg or "0100"
            Status_reg(4) <= '1';
        when "1000" =>
            Button_reg <= Button_reg or "1000"
            Status_reg(4) <= '1';
        when others =>
            Button_reg <= Button_reg or "0000";
    end case;
    Status_reg(2) <= Run_bit;
    Status_reg(3) <= RSTSEL_O;
    Status_reg(1) <= empty;
    Status_reg(0) <= full;
    State <= state_idle;
end if;
when state_decode_addr_O =>
case HA2_0 is
    when id =>
        Data_reg <= board_id;
        State <= state_hen_wait_O;
        when command =>
            if (RW = '1') then
                Data_reg <= Control_reg;
                State <= state_hen_wait_O;
            else
                Control_reg <= H8_0;
                State <= state_parse;
            end if;
        when status =>
            if (RW = '1') then
                Data_reg <= Status_reg;
            else
                Status_reg <= H8_0;
            end if;
            State <= state_hen_wait_O;
        when data_7_0 =>
if (RW = '1') then
  Data_reg <= Data_7_0_reg;
else
  Data_7_0_reg <= H8_0;
end if;
State <= state_hen_wait_0;

when data_15_8 =>
  if (RW = '1') then
    Data_reg <= Data_15_8_reg;
  else
    Data_15_8_reg <= H8_0;
  end if;
  State <= state_hen_wait_0;
when data_23_16 =>
  if (RW = '1') then
    Data_reg <= Data_23_16_reg;
  else
    Data_23_16_reg <= H8_0;
  end if;
  State <= state_hen_wait_0;
when others =>
  Data_reg <= x"ff";
  State <= state_hen_wait_0;
end case;

when state_parse =>
case Control_reg is
  when load_fifo =>
    State <= state_load_fifo_0;
  when read_fifo =>
    State <= state_read_fifo_0;
  when start_output =>
    State <= state_start_output_0;
  when stop_output =>
    State <= state_stop_output_0;
  when clear_fifo =>
    State <= state_clear_fifo_0;
      when change_rstsel =>
        State <= state_change_rstsel_0;
    when read_buttons =>
      State <= state_read_buttons_0;
when write_leds =>
  State <= state_write_leds_0;

when reset_dacs =>
  State <= state_reset_dacs_0;

when others =>
  State <= state_hen_wait_0;
end case;

when state_start_output_0 =>
  Run_bit <= '1';
  State <= state_hen_wait_0;

when state_stop_output_0 =>
  Run_bit <= '0';
  State <= state_hen_wait_0;

when state_load_fifo_0 =>
  d <= (Data_23_16_reg & Data_15_8_reg & Data_7_0_reg);
  we <= '1';
  State <= state_load_fifo_1;

when state_load_fifo_1 =>
  c <= '1';
  State <= state_load_fifo_2;

when state_load_fifo_2 =>
  we <= '0';
  State <= state_load_fifo_3;

when state_load_fifo_3 =>
  c <= '0';
  Return_state <= state_hen_wait_0;
  State <= state_update_flags_0;
  when state_update_flags_0 =>
    c <= '1';
    cr <= '1';
    State <= state_update_flags_1;
  when state_update_flags_1 =>
    c <= '0';
    cr <= '0';
    State <= state_update_flags_2;
  when state_update_flags_2 =>
    Status_reg(2) <= Run_bit;
    Status_reg(3) <= RSTSEL_O;
    Status_reg(1) <= empty;
    Status_reg(0) <= full;
State <= Return_state;

when state_read_fifo_0 =>
  re <= '1';
  State <= state_read_fifo_1;

when state_read_fifo_1 =>
  cr <= '1';
  State <= state_read_fifo_2;

when state_read_fifo_2 =>
  Data_23_16_reg <= q(23 downto 16);
  Data_15_8_reg <= q(15 downto 8);
  Data_7_0_reg <= q(7 downto 0);
  State <= state_read_fifo_3;

when state_read_fifo_3 =>
  re <= '0';
  State <= state_read_fifo_4;

when state_read_fifo_4 =>
  cr <= '0';
  Return_state <= state_hen_wait_0;
  State <= state_update_flags_0;

when state_clear_fifo_0 =>
  reset <= '1';
  State <= state_clear_fifo_1;

when state_clear_fifo_1 =>
  c <= '1';
  State <= state_clear_fifo_2;

when state_clear_fifo_2 =>
  reset <= '0';
  c <= '0';
  Return_state <= state_hen_wait_0;
  State <= state_update_flags_0;

when state_change_rstsel_0 =>
  RSTSEL_0 <= not RSTSEL_0;
  RSTSEL_1 <= not RSTSEL_1;
  RSTSEL_2 <= not RSTSEL_2;
  State <= state_hen_wait_0;

when state_read_buttons_0 =>
  Data_7_0_reg <= ("0000" & Button_reg);
  State <= state_read_buttons_1;

when state_read_buttons_1 =>
  Button_reg <= "0000";
  Status_reg(4) <= '0';
  State <= state_hen_wait_0;
when state_write_leds_0 =>
  LEDs <= Data_7_0_reg(3 downto 0);
  State <= state_hen_wait_0;

when state_reset_dacs_0 =>
  RST_0 <= '0';
  RST_1 <= '0';
  RST_2 <= '0';
  State <= state_reset_dacs_1;

when state_reset_dacs_1 =>
  RST_0 <= '1';
  RST_1 <= '1';
  RST_2 <= '1';
  State <= state_hen_wait_0;

when state_hen_wait_0 =>
  if (HEN = '1') then
    State <= state_idle;
  else
    State <= state_hen_wait_0;
  end if;

--- Dac Data Provider Section

when state_provider_0 =>
  Sync_bit_provide <= Sync_bit_consume;
  if (empty /= '1') then
    State <= state_provider_1;
  else
    Run_bit <= '0';
    Status_reg(7) <= '1';
    State <= state_idle;
  end if;

when state_provider_1 =>
  re <= '1';
  State <= state_provider_2;

when state_provider_2 =>
  cr <= '1';
  State <= state_provider_3;

when state_provider_3 =>
  Dac_Data <= q;
  State <= state_provider_4;

when state_provider_4 =>
  cr <= '0';
  re <= '0';
  Return_state <= msb23;
State <= state_update_flags_0;

when state_provider_5 =>
  if(Dac_Data(20) = '1') then
    State <= state_idle;
  else
    State <= state_provider_0;
  end if;

-- Dac Loading Section

when msb23 =>
  case (Dac_data(17 downto 16)) is
    when "00" =>
      SDI_0 <= Dac_data(23);
      CS_0 <= '0';
    when "01" =>
      SDI_1 <= Dac_data(23);
      CS_1 <= '0';
    when "10" =>
      SDI_2 <= Dac_data(23);
      CS_2 <= '0';
    when others =>
      end case;
    State <= bit22;
  end when;

when bit22 =>
  SDI_0 <= Dac_data(22);
  SDI_1 <= Dac_data(22);
  SDI_2 <= Dac_data(22);
  State <= bit21;

when bit21 =>
  SDI_0 <= Dac_data(21);
  SDI_1 <= Dac_data(21);
  SDI_2 <= Dac_data(21);
  State <= bit20;

when bit20 =>
  SDI_0 <= '0';
  SDI_1 <= '0';
  SDI_2 <= '0';
  State <= bit19;   -- Don't Care

when bit19 =>
  SDI_0 <= '0';
  SDI_1 <= '0';
  SDI_2 <= '0';
  State <= bit18;

when bit18 =>
  SDI_0 <= '0';
  SDI_1 <= '0';
```
SDI_2 <= '0'; -- Don't Care
State <= bit17;

when bit17 =>
  SDI_0 <= '0';
  SDI_1 <= '0';
  SDI_2 <= '0'; -- Don't Care
  State <= bit16;

when bit16 =>
  SDI_0 <= '0';
  SDI_1 <= '0';
  SDI_2 <= '0'; -- Don't Care
  State <= bit15;

when bit15 =>
  SDI_0 <= Dac_data(15);
  SDI_1 <= Dac_data(15);
  SDI_2 <= Dac_data(15);
  State <= bit14;

when bit14 =>
  SDI_0 <= Dac_data(14);
  SDI_1 <= Dac_data(14);
  SDI_2 <= Dac_data(14);
  State <= bit13;

when bit13 =>
  SDI_0 <= Dac_data(13);
  SDI_1 <= Dac_data(13);
  SDI_2 <= Dac_data(13);
  State <= bit12;

when bit12 =>
  SDI_0 <= Dac_data(12);
  SDI_1 <= Dac_data(12);
  SDI_2 <= Dac_data(12);
  State <= bit11;

when bit11 =>
  SDI_0 <= Dac_data(11);
  SDI_1 <= Dac_data(11);
  SDI_2 <= Dac_data(11);
  State <= bit10;

when bit10 =>
  SDI_0 <= Dac_data(10);
  SDI_1 <= Dac_data(10);
  SDI_2 <= Dac_data(10);
  State <= bit9;

when bit9 =>
  SDI_0 <= Dac_data(9);
```
SDI_1 <= Dac_data(9);
SDI_2 <= Dac_data(9);
State <= bit8;

when bit8 =>
SDI_0 <= Dac_data(8);
SDI_1 <= Dac_data(8);
SDI_2 <= Dac_data(8);
State <= bit7;

when bit7 =>
SDI_0 <= Dac_data(7);
SDI_1 <= Dac_data(7);
SDI_2 <= Dac_data(7);
State <= bit6;

when bit6 =>
SDI_0 <= Dac_data(6);
SDI_1 <= Dac_data(6);
SDI_2 <= Dac_data(6);
State <= bit5;

when bit5 =>
SDI_0 <= Dac_data(5);
SDI_1 <= Dac_data(5);
SDI_2 <= Dac_data(5);
State <= bit4;

when bit4 =>
SDI_0 <= Dac_data(4);
SDI_1 <= Dac_data(4);
SDI_2 <= Dac_data(4);
State <= bit3;

when bit3 =>
SDI_0 <= Dac_data(3);
SDI_1 <= Dac_data(3);
SDI_2 <= Dac_data(3);
State <= bit2;

when bit2 =>
SDI_0 <= Dac_data(2);
SDI_1 <= Dac_data(2);
SDI_2 <= Dac_data(2);
State <= bit1;

when bit1 =>
SDI_0 <= Dac_data(1);
SDI_1 <= Dac_data(1);
SDI_2 <= Dac_data(1);
State <= lsb0;

when lsb0 =>
SDI_0 <= Dac_data(0);
SDI_1 <= Dac_data(0);
SDI_2 <= Dac_data(0);
State <= data_latch;
when data_latch =>
case (Dac_data(17 downto 16)) is
  when "00" =>
    CS_0 <= '1';
  when "01" =>
    CS_1 <= '1';
  when "10" =>
    CS_2 <= '1';
  when others =>
end case;
State <= dac_load_0;
when dac_load_0 =>
case (Dac_data(17 downto 16)) is
  when "00" =>
    LOAD_O <= '0';
  when "01" =>
    LOAD_1 <= '0';
  when "10" =>
    LOAD_2 <= '0';
  when others =>
end case;
State <= dac_load_1;
when dac_load_1 =>
case (Dac_data(17 downto 16)) is
  when "00" =>
    LOAD_0 <= '1';
  when "01" =>
    LOAD_1 <= '1';
  when "10" =>
    LOAD_2 <= '1';
  when others =>
end case;
State <= state_provider_5;
when others =>
  State <= state_idle;
end case;
end if;
end process;

-- Dac Update / Consumer

process(Clk,RST) is
begin
if (RST = '0') then
    LDAC_0 <= '1';
    LDAC_1 <= '1';
    LDAC_2 <= '1';
    Dac_state <= dac_idle;
    Sync_bit_consume <= '0';
    Test_counter <= 0;
elsif (Clk'event and Clk='1') then
    Sync_bit_consume <= Sync_bit_consume;
    if (Run_bit = '1') then
        case Dac_state is
            when dac_idle =>
                if (Internal_counter = 127) then
                    Dac_state <= dac_out_0;
                else
                    Dac_state <= dac_idle;
                end if;
            when dac_out_0 =>
                LDAC_0 <= '0';
                LDAC_1 <= '0';
                LDAC_2 <= '0';
                Dac_state <= dac_out_1;
            when dac_out_1 =>
                LDAC_0 <= '1';
                LDAC_1 <= '1';
                LDAC_2 <= '1';
                Dac_state <= dac_out_2;
            when dac_out_2 =>
                Sync_bit_consume <= not Sync_bit_consume;
                Test_counter <= Test_counter + 1;
                Dac_state <= dac_idle;
            when others =>
                Dac_state <= dac_idle;
        end case;
    end if;
end if;
end process;
-- Tri-State
HS_0 <= Data_reg when (HEN = '0' and RW = '1') else (others => 'Z');
TR <= '0' when (HEN = '0' and RW = '1') else '1';

-- Clock Routing
    CLK_0 <= Clk;
    CLK_1 <= Clk;
    CLK_2 <= Clk;
-- Test Outputs
UIOD(0) <= SDO_0;
UIOD(1) <= SDO_1;
UIOD(2) <= SDO_2;
UIOD(3) <= RST;
UIOA <= CONV_STD_LOGIC_VECTOR(Test_count, 8);
UIOB <= (we & re & reset & c & Run_bit & Sync_bit_provide & Sync_bit_consume & HEN);

end arch_dacboard;

J.3 dac.ucf

The code below, dac.ucf, is the constraints file for mapping VHDL port definitions to the physical pins of the device.

*******************************************************************************
# DAC Board UCF Listings KZ 11/2001 #
*******************************************************************************
NET "LEDS<0>" LOC ="P68";
NET "LEDS<1>" LOC ="P69";
NET "LEDS<2>" LOC ="P70";
NET "LEDS<3>" LOC ="P71";

NET "UIOA<0>" LOC ="P86";
NET "UIOA<1>" LOC ="P83";
NET "UIOA<2>" LOC ="P82";
NET "UIOA<3>" LOC ="P81";
NET "UIOA<4>" LOC ="P80";
NET "UIOA<5>" LOC ="P76";
NET "UIOA<6>" LOC ="P75";
NET "UIOA<7>" LOC ="P74";

NET "UIOB<0>" LOC ="P42";
NET "UIOB<1>" LOC ="P43";
NET "UIOB<2>" LOC ="P44";
NET "UIOB<3>" LOC ="P45";
NET "UIOB<4>" LOC ="P46";
NET "UIOB<5>" LOC ="P59";
NET "UIOB<6>" LOC ="P60";
NET "UIOB<7>" LOC ="P61";

NET "UIOC<0>" LOC ="P141";
NET "UIOC<1>" LOC ="P140";
NET "UIOC<2>" LOC ="P139";

143
NET "UIOC<3>" LOC = "P138";
NET "UIOC<4>" LOC = "P123";
NET "UIOC<5>" LOC = "P122";
NET "UIOC<6>" LOC = "P121";
NET "UIOC<7>" LOC = "P120";
NET "UIOC<0>" LOC = "P172";
NET "UIOC<1>" LOC = "P173";
NET "UIOC<2>" LOC = "P174";
NET "UIOC<3>" LOC = "P175";
NET "SDI_0" LOC = "P166";
NET "CLK_0" LOC = "P165";
NET "LDAC_0" LOC = "P164";
NET "LOAD_0" LOC = "P163";
NET "CS_0" LOC = "P161";
NET "SDO_0" LOC = "P150";
NET "RSTSEL_0" LOC = "P149";
NET "RST_0" LOC = "P147";
NET "SDI_1" LOC = "P135";
NET "CLK_1" LOC = "P134";
NET "LDAC_1" LOC = "P133";
NET "LOAD_1" LOC = "P132";
NET "CS_1" LOC = "P129";
NET "SDO_1" LOC = "P128";
NET "RSTSEL_1" LOC = "P127";
NET "RST_1" LOC = "P126";
NET "SDI_2" LOC = "P112";
NET "CLK_2" LOC = "P111";
NET "LDAC_2" LOC = "P109";
NET "LOAD_2" LOC = "P99";
NET "CS_2" LOC = "P98";
NET "SDO_2" LOC = "P97";
NET "RSTSEL_2" LOC = "P96";
NET "RST_2" LOC = "P95";
NET "TR" LOC = "P180";
NET "H8_0<0>" LOC = "P21";
NET "H8_0<1>" LOC = "P22";
NET "H8_0<2>" LOC = "P23";
NET "H8_0<3>" LOC = "P24";
NET "H8_0<4>" LOC = "P27";
NET "H8_0<5>" LOC = "P28";
NET "H8_0<6>" LOC = "P29";
NET "H8_0<7>" LOC = "P30";
NET "HA2_0<0>" LOC = "P6";
NET "HA2_0<1>" LOC = "P5";
NET "HA2_0<2>" LOC = "P7";
NET "RW" LOC = "P15";
NET "HEN" LOC = "P16";
NET "RST_IN" LOC = "P18";

NET "BUTTONS<0>" LOC = "P33";
NET "BUTTONS<1>" LOC = "P34";
NET "BUTTONS<2>" LOC = "P35";
NET "BUTTONS<3>" LOC = "P36";
APPENDIX K

IPENGINE CONFIGURATION FILES

The following code listings are configuration files for the IpEngine from Brightstar Engineering. The code was compiled using CodeWarrior for embedded Power PC.

K.1 upm.txt

The capture file below, upm.txt, contains a log file of the IpEngine console session used to determine UPM (user programmable machine) settings for properer timing of IpEngine RAM.

```
>ww ff000168 40000000; rw ff000168; rw ff00017c
FF000168 : 40000000
FF00017C : 0FFFFFC24

>ww ff000168 40000001; rw ff000168; rw ff00017c
FF000168 : 40000001
FF00017C : 0FFFFFC04

>ww ff000168 40000002; rw ff000168; rw ff00017c
FF000168 : 40000002
FF00017C : 08F3FC00

>ww ff000168 40000003; rw ff000168; rw ff00017c
FF000168 : 40000003
FF00017C : 17F7FC07

>ww ff000168 40000004; rw ff000168; rw ff00017c
FF000168 : 40000004
FF00017C : FFFFFFC07

>ww ff000168 40000005; rw ff000168; rw ff00017c
FF000168 : 40000005
```
FF00017C : FFFFFC05
>ww ff000168 40000006; rw ff000168; rw ff00017c
FF000168 : 40000006
FF00017C : FFFFFC05

>ww ff000168 40000007; rw ff000168; rw ff00017c
FF000168 : 40000007
FF00017C : FFFFFC05

>ww ff000168 40000008; rw ff000168; rw ff00017c
FF000168 : 40000008
FF00017C : 0FFFFFFC24

>ww ff000168 40000009; rw ff000168; rw ff00017c
FF000168 : 40000009
FF00017C : 0FFFFFFC04

>ww ff000168 4000000a; rw ff000168; rw ff00017c
FF000168 : 4000000A
FF00017C : 08F3FC00

>ww ff000168 4000000b; rw ff000168; rw ff00017c
FF000168 : 4000000B
FF00017C : 07F3FC00

>ww ff000168 4000000c; rw ff000168; rw ff00017c
FF000168 : 4000000C
FF00017C : 08F3FC00

>ww ff000168 4000000d; rw ff000168; rw ff00017c
FF000168 : 4000000D
FF00017C : 07F3FC00

>ww ff000168 4000000e; rw ff000168; rw ff00017c
FF000168 : 4000000E
FF00017C : 08F3FC00

>ww ff000168 4000000f; rw ff000168; rw ff00017c
FF000168 : 4000000F
FF00017C : 07F3FC00

>ww ff000168 40000010; rw ff000168; rw ff00017c
FF000168 : 40000010
FF00017C : 08F3FC00

>ww ff000168 40000011; rw ff000168; rw ff00017c
FF000168 : 40000011
FF00017C : 17F7FC07

>ww ff000168 40000012; rw ff000168; rw ff00017c
FF000168 : 40000012
FF00017C : FFFFFC05.
>ww ff000168 40000013; rw ff000168; rw ff00017c
FF000168 : 40000013
FF00017C : FFFFFC04

>ww ff000168 40000014; rw ff000168; rw ff00017c
FF000168 : 40000014
FF00017C : FFFFFC04

>ww ff000168 40000015; rw ff000168; rw ff00017c
FF000168 : 40000015
FF00017C : FFFFFC04

>ww ff000168 40000016; rw ff000168; rw ff00017c
FF000168 : 40000016
FF00017C : FFFFFC04

>ww ff000168 40000017; rw ff000168; rw ff00017c
FF000168 : 40000017
FF00017C : FFFFFC05

>ww ff000168 40000018; rw ff000168; rw ff00017c
FF000168 : 40000018
FF00017C : 0FFFFC24

>ww ff000168 40000019; rw ff000168; rw ff00017c
FF000168 : 40000019
FF00017C : 0FFFFC04

>ww ff000168 4000001a; rw ff000168; rw ff00017c
FF000168 : 4000001A
FF00017C : 0FFFFC00

>ww ff000168 4000001b; rw ff000168; rw ff00017c
FF000168 : 4000001B
FF00017C : 17FFFFC07

>ww ff000168 4000001c; rw ff000168; rw ff00017c
FF000168 : 4000001C
FF00017C : FFFFFC05

>ww ff000168 4000001d; rw ff000168; rw ff00017c
FF000168 : 4000001D
FF00017C : FFFFFC04

>ww ff000168 4000001e; rw ff000168; rw ff00017c
FF000168 : 4000001E
FF00017C : FFFFFC07

>ww ff000168 4000001f; rw ff000168; rw ff00017c
FF000168 : 4000001F
FF00017C : FFFFFC07

148
>ww ff000168 40000020; rw ff000168; rw ff00017c
FF000168 : 40000020
FF00017C : 0FFFFFC24

>ww ff000168 40000021; rw ff000168; rw ff00017c
FF000168 : 40000021
FF00017C : 0FFFFFC04

>ww ff000168 40000022; rw ff000168; rw ff00017c
FF000168 : 40000022
FF00017C : 08FFFC00

>ww ff000168 40000023; rw ff000168; rw ff00017c
FF000168 : 40000023
FF00017C : 08FFFC00

>ww ff000168 40000024; rw ff000168; rw ff00017c
FF000168 : 40000024
FF00017C : 08FFFC00

>ww ff000168 40000025; rw ff000168; rw ff00017c
FF000168 : 40000025
FF00017C : 07FFFC0C

>ww ff000168 40000026; rw ff000168; rw ff00017c
FF000168 : 40000026
FF00017C : 08FFFC00

>ww ff000168 40000027; rw ff000168; rw ff00017c
FF000168 : 40000027
FF00017C : 07FFFC0C

>ww ff000168 40000028; rw ff000168; rw ff00017c
FF000168 : 40000028
FF00017C : 08FFFC00

>ww ff000168 40000029; rw ff000168; rw ff00017c
FF000168 : 40000029
FF00017C : 37FFFC07

>ww ff000168 4000002a; rw ff000168; rw ff00017c
FF000168 : 4000002A
FF00017C : FFFFFFC04

>ww ff000168 4000002b; rw ff000168; rw ff00017c
FF000168 : 4000002B
FF00017C : FFFFFFC04

>ww ff000168 4000002c; rw ff000168; rw ff00017c
FF000168 : 4000002C
FF00017C : FFFFFFC04

>ww ff000168 4000002d; rw ff000168; rw ff00017c

149
FF000168 : 4000002D
FF00017C : FFFFFC04

>ww ff000168 4000002e; rw ff000168; rw ff00017c
FF000168 : 4000002E
FF00017C : FFFFFC07

>ww ff000168 4000002f; rw ff000168; rw ff00017c
FF000168 : 4000002F
FF00017C : FFFFFC07

>ww ff000168 40000030; rw ff000168; rw ff00017c
FF000168 : 40000030
FF00017C : FFFFFC04

>ww ff000168 40000031; rw ff000168; rw ff00017c
FF000168 : 40000031
FF00017C : FFFFFC04

>ww ff000168 40000032; rw ff000168; rw ff00017c
FF000168 : 40000032
FF00017C : FFFFFC04

>ww ff000168 40000033; rw ff000168; rw ff00017c
FF000168 : 40000033
FF00017C : FFFFFC04

>ww ff000168 40000034; rw ff000168; rw ff00017c
FF000168 : 40000034
FF00017C : FFFFFC04

>ww ff000168 40000035; rw ff000168; rw ff00017c
FF000168 : 40000035
FF00017C : FFFFFC04

>ww ff000168 40000036; rw ff000168; rw ff00017c
FF000168 : 40000036
FF00017C : FFFFFC04

>ww ff000168 40000037; rw ff000168; rw ff00017c
FF000168 : 40000037
FF00017C : FFFFFC04

>ww ff000168 40000038; rw ff000168; rw ff00017c
FF000168 : 40000038
FF00017C : FFFFFC04

>ww ff000168 40000039; rw ff000168; rw ff00017c
FF000168 : 40000039
FF00017C : FFFFFC07

>ww ff000168 4000003a; rw ff000168; rw ff00017c
FF000168 : 4000003A

150
K.2 ipEngine_823_init.txt

The BDM initialization file below, ipEngine.823.init.txt, contains configuration information to run the ipEngine in debug mode and configure memory based on configuration data in upm.txt.

;*-------1---------2---------3---------4---------5---------6---------7---------
;------------------------------------------------------------------------
; IpEngine configuration for MPC 823. (ipEngine from Brightstar Engineering)
; Written by : KZ (Based on RPX Lite Configuration file by Collin T. Chan)
; Date : 4/28/00
;
; Note: The following Exceptions must be caught.
; (For earlier IDE set the DER register)
; Ox40000000 System Reset
; Ox20000000 Check Stop
; Ox00800000 Program
; Ox00020000 Trace
; Ox00004000 Software Emulation
; Ox00000001 Development Port
;------------------------------------------------------------------------

; IMMR - Internal Memory Map Register (p12-34) set to Oxff000000
; corresponding to ipEngine’s MPC823 On-chip registers.
MSR - Machine State Register (p6-20) configured with Machine Check Enable (ME) and Recoverable Interrupt (RI) bits set
SRR1 - Save/Restore Register (p7-9) 1 set to current machine status (ie SRR1 = MSR)
DER - Debug Enable Register (p20-57) left at default
SYPCR - System Protection Control Register (p12-35) configured with watchdog disabled
ICTRL - Instruction Support Control Register (p20-45) configured for ISCT_SER = 110 (Core not serialized and show cycle performed for all indirect changes in program flow)
SIUMCR - System Configuration and Protection Register (p12-30) set to configure pin functionality

writespr 638 0xFF000000 # IMMR
writespr MSR 0x00001002 # MSR
writespr 27 0x00001002 # SRR1
writespr 149 0x00824000 # DER
writespr 158 0x00000007 # ICTRL
writespr 168 0x00000007 # SRR1
writespr 158 0x00000007 # SYPCR
writespr 158 0x00000007 # SYPCR
writespr 168 0x00000007 # SYPCR

writesmem 1 0xFF000280 0x01000000 # SCCR
writesmem 1 0xFF000284 0x00804080 # PLPRCR

; Set up the chip selects for ipEngine Memory Map
; 0000.0000 - 00FF.FFFF 16MB DRAM (4Mx32 60ns EDD)
; FC00.0000 - FC7F.FFFF 8MB FPGA Space
; FE00.0000 - FE3F.FFFF 4MB Flash (2Mx16 90ns)
; FF00.0000 - FF00.3FFF MPC823 On-Chip Registers
; FF01.0000 - FF01.0000 FPGA Config Register
; FF02.0000 - FF02.0000 Clock Synth RegA
; BR(0-7) - Base Registers (p15-9) set to base addresses of memories.
; OR(0-7) - Option Registers (p15-11)
; MPTPR - Memory Periodic Timer Prescaler Register (p15-28) set devisior of BRGCLK to divide by 32
; MAMR - Machine A Mode Register (p15-19) (DRAM Control)
; MBMR - Machine D Mode Register (p15-23) Disabled

Memory dump from ipEngine after firmware boot.
; B0R1 OR0 BR1 OR1
; FF000100 FF000801 FF000F84 FC000801 FF000934
; BR2 OR2 BR3 OR3
; FF000110 00000001 FF000800 FF010000 FFF0F24
writemem.w 0xFF00017A 0x0200  # MPTPR
writemem.l 0xFF000170 0x17A20000  # MAMR
writemem.l 0xFF000174 0x00001000  # MBMR

writemem.l 0xFF00011B 0xFF010001  # BR3 - FPGA Config Reg
writemem.l 0xFF00011C 0xFF000080  # OR3

writemem.l 0xFF00011D 0x00000081  # BR2 - DRAM
writemem.l 0xFF00011E 0xFF000934  # OR1 - (8MB)

writemem.l 0xFF000118 0xFF010001  # BR1 FPGA Space
writemem.l 0xFF00010C 0xFF000934  # OR1 - (8MB)

writemem.l 0xFF000100 0x00000081  # OR2 - (16M)

; UPMA Initialization for DRAM - EDQ, 16MB, 60ns
; MCR - Memory Command Register (p15-17)
; MDR - Memory Data Register (p15-26)

# Single Read
writemem.l 0xFF00017C 0x0FFFFFFC24  # MDR
writemem.l 0xFF000168 0x00000000  # MCR
writemem.l 0xFF00017C 0x0FFFFFFC04
writemem.l 0xFF000168 0x00000001
writemem.l 0xFF00017C 0x08F3FC00
writemem.l 0xFF000168 0x00000002
writemem.l 0xFF00017C 0x17F7FC07
writemem.l 0xFF000168 0x00000003
writemem.l 0xFF00017C 0x0FFFFFFC07
writemem.l 0xFF000168 0x00000004
writemem.l 0xFF00017C 0x0FFFFFFC05
writemem.l 0xFF000168 0x00000005
writemem.l 0xFF00017C 0x0FFFFFFC05
writemem.l 0xFF000168 0x00000006
writemem.l 0xFF00017C 0x0FFFFFFC05
writemem.l 0xFF000168 0x00000007

# Burst Read
writemem.l 0xFF00017C 0x0FFFFFFC24
writemem.l 0xFF000168 0x00000008
writemem.l 0xFF00017C 0x0FFFFFFC04
writemem.l 0xFF000168 0x00000009
writemem.l 0xFF00017C 0x08F3FC00
writemem.l 0xFF000168 0x0000000A
writemem.l 0xFF00017C 0x07F3FC0C
writemem.l 0xFF000168 0x0000000B
writemem.l 0xFF00017C 0x08F3FC00
writemem.1 0xFF000168 0x0000000C
writemem.1 0xFF00017C 0x07F3FC0C
writemem.1 0xFF000168 0x0000000D
writemem.1 0xFF00017C 0x08F3FC00
writemem.1 0xFF000168 0x0000000E
writemem.1 0xFF00017C 0x07F3FC0C
writemem.1 0xFF000168 0x0000000F
writemem.1 0xFF00017C 0x08F3FC00
writemem.1 0xFF000168 0x00000010
writemem.1 0xFF00017C 0x17F7FC07
writemem.1 0xFF000168 0x00000011
writemem.1 0xFF00017C 0xFFFFFC05
writemem.1 0xFF000168 0x00000012
writemem.1 0xFF00017C 0xFFFFFC04
writemem.1 0xFF000168 0x00000013
writemem.1 0xFF00017C 0xFFFFFC04
writemem.1 0xFF000168 0x00000014
writemem.1 0xFF00017C 0xFFFFFC04
writemem.1 0xFF000168 0x00000015
writemem.1 0xFF00017C 0xFFFFFC05
writemem.1 0xFF000168 0x00000016
writemem.1 0xFF00017C 0xFFFFFC07
writemem.1 0xFF000168 0x00000017

# Single Write.
writemem.1 0xFF00017C 0x0FFFFFFC24
writemem.1 0xFF000168 0x00000018
writemem.1 0xFF00017C 0x0FFFFFFC04
writemem.1 0xFF000168 0x00000019
writemem.1 0xFF00017C 0x08FFFFFFC00
writemem.1 0xFF000168 0x0000001A
writemem.1 0xFF00017C 0x17FFFFFFC07
writemem.1 0xFF000168 0x0000001B
writemem.1 0xFF00017C 0xFFFFFFFFFFC05
writemem.1 0xFF000168 0x0000001C

writemem.1 0xFF00017C 0x0FFFFFFC04
writemem.1 0xFF000168 0x0000001D
writemem.1 0xFF00017C 0xFFFFFFFFFFC07
writemem.1 0xFF000168 0x0000001E
writemem.1 0xFF00017C 0xFFFFFFFFFFC07
writemem.1 0xFF000168 0x0000001F

# Burst Write
writemem.1 0xFF00017C 0x0FFFFFFC24
writemem.1 0xFF000168 0x00000020
writemem.1 0xFF00017C 0x0FFFFFFC04
writemem.1 0xFF000168 0x00000021
writemem.1 0xFF00017C 0x08FFFFFFC00
writemem.l 0xFF000168 0x00000022
writemem.l 0xFF00017C 0x07FFFC0C
writemem.l 0xFF000168 0x00000023
writemem.l 0xFF00017C 0x08FFFC00
writemem.l 0xFF000168 0x00000024
writemem.l 0xFF00017C 0x07FFFC0C
writemem.l 0xFF000168 0x00000025
writemem.l 0xFF00017C 0x07FFFC0C
writemem.l 0xFF000168 0x00000026
writemem.l 0xFF00017C 0x07FFFC0C
writemem.l 0xFF000168 0x00000027
writemem.l 0xFF00017C 0x08FFFC00
writemem.l 0xFF000168 0x00000028
writemem.l 0xFF00017C 0x07FFFC07
writemem.l 0xFF000168 0x00000029
writemem.l 0xFF00017C 0x08FFFC04
writemem.l 0xFF000168 0x0000002A
# Refresh
writemem.l 0xFF00017C 0x0FOFFFC04
writemem.l 0xFF000168 0x00000030
writemem.l 0xFF00017C 0x080FFFC04
writemem.l 0xFF000168 0x0000002C
writemem.l 0xFF00017C 0x07FFFC04
writemem.l 0xFF000168 0x0000002D
writemem.l 0xFF00017C 0x08FFFC07
writemem.l 0xFF000168 0x0000002E
writemem.l 0xFF00017C 0x08FFFC07
writemem.l 0xFF000168 0x0000002F
writemem.l 0xFF00017C 0x0FFFC04
writemem.l 0xFF000168 0x00000030
writemem.l 0xFF00017C 0x0FFFC04
writemem.l 0xFF000168 0x00000031
writemem.l 0xFF00017C 0x0FFFC04
writemem.l 0xFF000168 0x00000032
writemem.l 0xFF00017C 0x0FFFC04
writemem.l 0xFF000168 0x00000033
writemem.l 0xFF00017C 0x0FFFC04
writemem.l 0xFF000168 0x00000034
writemem.l 0xFF00017C 0x0FFFC04
writemem.l 0xFF000168 0x00000035
writemem.l 0xFF00017C 0x0FFFC04
writemem.l 0xFF000168 0x00000036
writemem.l 0xFF00017C 0x0FFFC04
writemem.l 0xFF000168 0x00000037
writemem.l 0xFF00017C 0x0FFFC04
writemem.l 0xFF000168 0x00000038
writemem.l 0xFF00017C 0x0FFFC07
writemem.l 0xFF000168 0x00000039
writemem.l 0xFF00017C 0x0FFFC07
writemem.l 0xFF000168 0x0000003A
writemem.l 0xFF00017C 0xFFFFFC07
writemem.l 0xFF000168 0x0000003B

# Exception
writemem.l 0xFF00017C 0xFFFFFC07
writemem.l 0xFF000168 0x0000003C
writemem.l 0xFF00017C 0xFFFFFC07
writemem.l 0xFF000168 0x0000003D
writemem.l 0xFF00017C 0xFFFFFC07
writemem.l 0xFF000168 0x0000003E
writemem.l 0xFF00017C 0xFFFFFC07
writemem.l 0xFF000168 0x0000003F
REFERENCES


