

# Merging Theory and Implementation: A Framework for Teaching DSP Hardware Design

Tyson S. Hall and David V. Anderson  
Georgia Institute of Technology, Atlanta, GA 30332-0250  
{tyson,dva}@ece.gatech.edu

## Abstract

*In this paper, we present a framework for teaching DSP hardware design and provide the necessary technical infrastructure for enabling this convergence of theory and implementation. Even though many curricula include separate classes in both DSP theory and VHDL modeling, there are few opportunities given to students to combine these two skills into a working knowledge of DSP hardware design. We have developed a pedagogical framework whereby students can leverage their previous knowledge of DSP theory and VHDL hardware design techniques to design, simulate, synthesize, and test digital signal processing systems. The synthesized hardware is implemented on FPGAs, which provides a fast and cost-effective way of prototyping hardware systems in a laboratory environment. This framework allows students to expand their previous knowledge into a more complete understanding of the entire design process from specification and simulation through synthesis and verification.*

## 1 Introduction

Students often struggle to bridge the gap between the theory and the hardware implementation of digital signal processing (DSP) systems. Even though many curricula include separate classes in both DSP theory and VHDL modeling, there are few opportunities given to students to combine these two skills into a working knowledge of DSP hardware design [1, 2]. We have developed a pedagogical framework whereby students can leverage their previous knowledge of DSP theory and VHDL hardware design techniques to design, simulate, synthesize, and test digital signal processing systems [3].

There are examples within the literature of DSP hardware courses being started [4, 5, 6, 7]. However, these programs rely on DSP microprocessors as their primary implementation medium. Thus, the emphasis is more on software programming than hardware design. By using FPGAs as the core technology, students can be given the opportunity to design custom hardware implementations [8]. In addition, FPGAs can synthesize microprocessor cores allowing students to investigate the trade-offs between hardware and software implementations.

This system provides a purely digital prototyping and testing platform for implementing

a wide variety of DSP systems. Field-programmable gate arrays (FPGAs) are used to synthesize the DSP hardware, which provides a fast and cost-effective way of prototyping hardware systems in a laboratory environment. On the PC side, MATLAB is used to do the system simulation, data acquisition (via a USB interface between the FPGA and the PC), and data analysis. Since MATLAB is familiar to many engineering students, it is used as the software platform to ease the transition from a theoretical understanding of DSP systems to their hardware implementation.

This paper also presents a toolbox of MATLAB functions and VHDL modules that provides a transparent connection between MATLAB and a DSP system implemented on an FPGA. The interface is made up of a standard USB connection between the PC and the FPGA, which provides a fast, robust communication link. By providing a reliable testing infrastructure up front, students can concentrate on implementing and optimizing the DSP systems and not the testing infrastructure.

## 2 Theory to Hardware

Our goal is to teach students at the senior or early graduate levels how to implement DSP algorithms in hardware. Our initial effort has been to create a lecture plus laboratory course that is taught within the School of Electrical and Computer Engineering at Georgia Tech under the title *DSP Chip Design*. Students taking this course have had exposure to MATLAB in the required curriculum, and given its widespread familiarity in the DSP community, MATLAB is the logical choice for use as a prototyping environment. On the hardware side, students have used VHDL and FPGAs in at least one prerequisite course. So, the students taking this course have a familiarity with the hardware and software tools they will use. In addition, the first two weeks of lecture and lab are used to review background information and step through tutorials to familiarize students with the specific software environment and FPGA development boards that are used throughout this course.

This course focuses on the transition from an algorithm that can be manipulated and computed in MATLAB to the hardware implementation of that algorithm. This translation proves to be non-trivial for most.<sup>1</sup> One of the greatest difficulties facing the students is hardware debugging. If students use a standard language such as VHDL for the hardware description, many tools exist for debugging the digital design but not the signal processing performance. Therefore, common and useful analyses such as frequency response, noise analysis, and system verification may require many steps and/or be impractical on the FPGA.

To facilitate this type of hardware analysis, a method is needed for students to verify and debug their hardware implementations of DSP algorithms. To make it easy to use, software tools that are already familiar to them should be used. Our solution to this problem is a set of VHDL modules and a toolbox of MATLAB functions that can be used to pass signals to and

---

<sup>1</sup>It should be noted here that the Mathworks, Inc. and Altera Corporation have developed software for converting Simulink models to FPGA-based implementations; but this abstracts the implementation step that we wish to teach, and it still does not provide a two-way communication mechanism between MATLAB (or Simulink) and the FPGA.

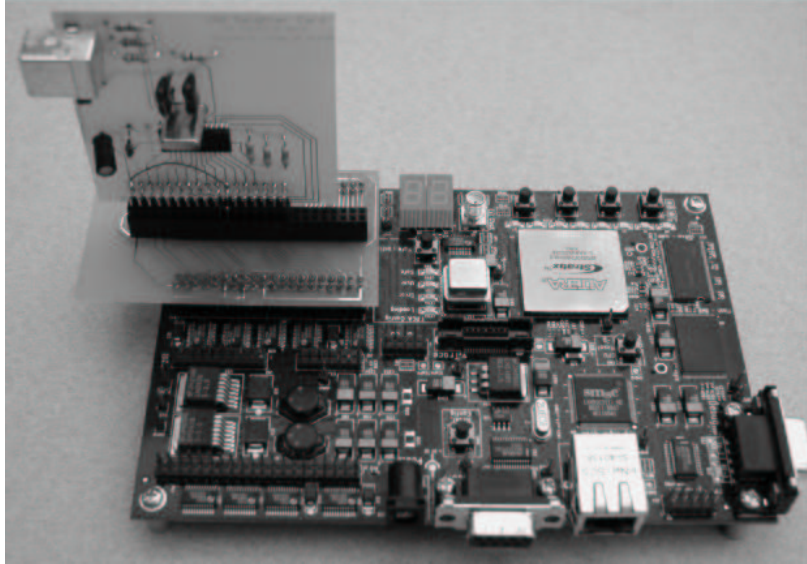


Figure 1: The FPGA board that comes with the Altera Nios Development Kit-Stratix Edition is shown with our custom USB daughter card attached to it. The USB daughter card is built around the Philips PDIUSB12 chip with implements the physical-level interface of the USB protocol.

from MATLAB and an FPGA. By using this infrastructure, students can pass signals to or retrieve signals from the FPGA and create or analyze the signals using MATLAB commands and a simple USB interface.

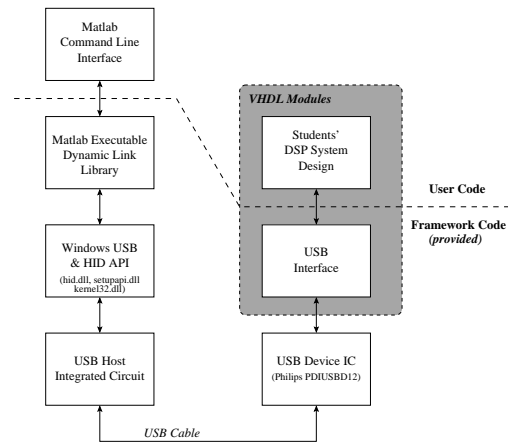
### 3 Teaching Framework and Application

#### 3.1 The Prototyping Station

Having a straight-forward prototyping system is very important to the success of this teaching framework. At the highest level, we need an environment that contains both theoretical analysis and hardware implementation capabilities. Currently, these two environments exist separately in the forms of the MATLAB software and the Altera FPGAs and Quartus II design software. The work presented in this paper provides an interface between these two environments. The connection is currently made up of a plug-and-play USB connection between the PC and the FPGA. The FPGA board we use is a development board (Nios Development Kit-Stratix Edition) provided by the Altera Corporation. A custom daughter card was then developed for the FPGA to provide the USB capabilities. Figure 1 shows the Altera Nios Development board with our custom USB daughter card attached.

The prototyping system requires a similar USB protocol stack to be implemented on both the PC and the FPGA as shown in Fig. 2. On the PC side, most of the USB protocol is already implemented in Windows. On the FPGA side, however, much of the USB and HID protocols had to be synthesized on the FPGA.

Figure 2: This diagram illustrates the datapath for the MATLAB–FPGA interface. On the PC side, only the MATLAB Executable module had to be created since most of the USB protocol is already implemented in Windows. On the FPGA side, however, much of the USB and HID protocols had to be implemented in synthesized hardware on the FPGA.



### 3.1.1 PC side

On the PC side, a toolbox of MATLAB functions provide the ability to send and receive arbitrary waveforms; generate, send, and receive sinusoidal waveforms; measure the frequency response of the hardware system; and generate signed, decimal, fixed-point coefficients. Supporting the public interface for this toolbox is a MATLAB Executable (MEX) dynamic link library (DLL) that handles communication with the Windows USB and Human Interface Device (HID) API.

The FPGA implements a USB device that is natively supported by Microsoft Windows XP. Specifically, the FPGA board appears to be a generic HID device. When it is connected to the USB port of a PC, it will be automatically recognized, and the appropriate Windows drivers will be loaded (e.g., it is a plug-and-play device).

### 3.1.2 FPGA side

On the FPGA side, VHDL modules have been created that implement the USB and HID protocols, receive packets of data from the USB connection, rebuild the individual samples (currently supports up to 32-bit samples), issue the sample and a sample clock pulse to the students' DSP system, receive the output sample, encapsulate multiple samples into data packets, and transmit them back to the PC through the USB connection. The FPGA communicates with the USB bus via a custom daughter card built for the FPGA board. The daughter card contains a Philips PDIUSB12 chip that handles the physical communication link with the USB connection. For interrupt-based transfers, this chip has a maximum throughput of 512 Kbps. By providing all of this functionality up front, students can concentrate on implementing and optimizing the DSP systems and not the testing infrastructure.

## 3.2 A Pedagogical Approach

The teaching approach that we take is to start with a simple DSP system such as a small FIR filter; students move from a (hopefully) familiar point and progress through the simulation and synthesis of the hardware. By leading them through their first complete system

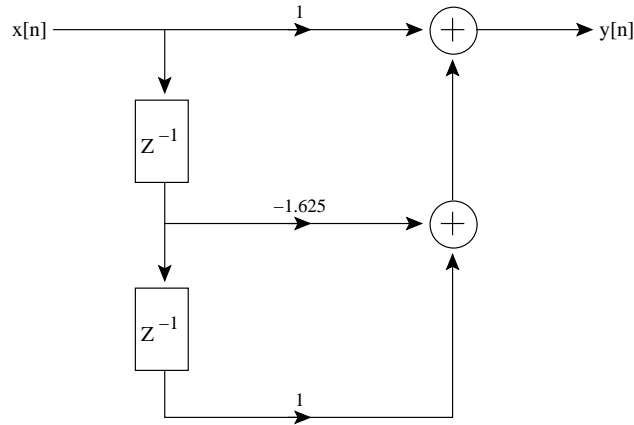


Figure 3: This simple filter design will be used to illustrate the functionality of the MATLAB–FPGA interface described here.

in a highly directed manner, students will be able to complete the entire design and implementation process within a few hours. The MATLAB–FPGA interface toolbox is useful at this stage for allowing students to test the results of the hardware filter against those of a software implementation to verify correctness.

Once students understand the entire process (i.e., the global perspective), the concentration can shift towards studying characteristics of hardware implementations such as quantization effects and resolution, area, power, and implementation optimizations. At this stage, verification of correctness becomes even more important but the USB toolbox also becomes useful for investigating engineering trade-offs such as the impact of increasing the level of quantization.

### 3.3 Example

To illustrate the use of our MATLAB–FPGA system, the FIR filter illustrated in Fig. 3 has been described using VHDL and synthesized along with the interfacing infrastructure on an FPGA. The *sendcos* function was used to generate the data plotted in Fig. 4a, which shows the output of the hardware filter for sine wave input of 600 Hz with an 8 kHz sampling frequency. Figure 4b shows the theoretical frequency response obtained from the *freqz* function in MATLAB’s Signal Processing toolbox plotted as a solid line, and the experimental frequency response measured from the hardware filter itself using the *freq\_response* function plotted with “x” marks. In this case, the results are seemingly identical, because the coefficients were chosen to minimize quantization error.

## 4 Implementation

The infrastructure needed on the hardware side included VHDL modules to implement the USB protocol/interface as well as a custom PCB board with a USB device chip and USB connector on it. The USB device implemented on the FPGA uses native Windows drivers, so the software requirements were reduced to a toolbox of MATLAB functions that communicate

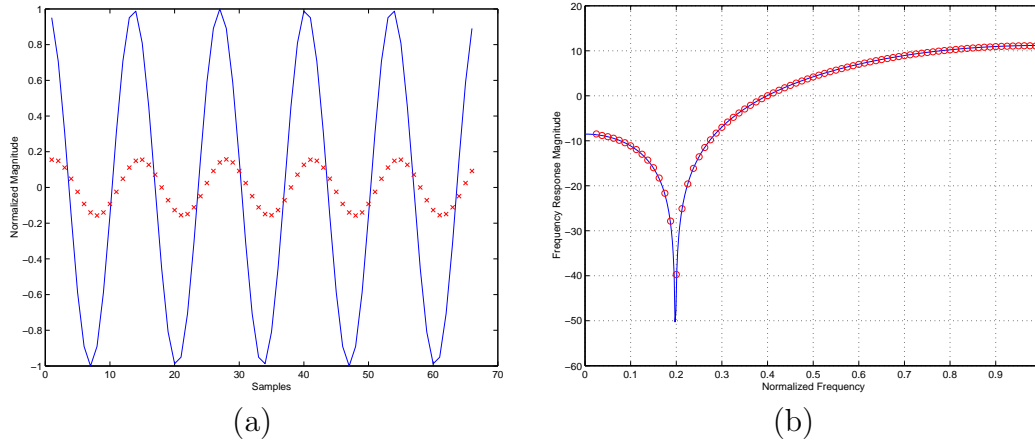


Figure 4: (a) This illustrates the output of a simple FIR filter implemented on an FPGA. The MATLAB toolbox of functions provides the interface to the FPGA. In this case, the input (denoted by the solid line) is a 600 Hz sine wave with a 8 kHz sample frequency. As expected, the output is a scaled and shifted version of the input. (b) The frequency response for a filter implemented on an FPGA can be generated by plotting the output magnitudes from input sine waves. Our MATLAB toolbox encapsulates this functionality into a single function whose output is similar to the `freqz()` function found in MATLAB's Signal Processing toolbox. The theoretical frequency response (generated by MATLAB's `freqz()` function) is shown with a solid line. The frequency response measured from the FPGA implementation of the filter is shown with the circles.

with the Windows USB subsystem. The MATLAB toolbox, VHDL files, and example Altera project files are freely available online [9].

#### 4.1 MATLAB Toolbox

On the PC side, a toolbox of MATLAB functions provides access to the hardware system. In particular, three functions have been developed to send data to and receive data from the FPGA. These functions provide the ability for students to quickly process data using the FPGA and return the results.

---

```
>> [x, y] = sendcos(600, 10, 65535, 8000);
```

---

Figure 5: In this example, 10 ms of a 600 Hz sinusoidal waveform are sent to the FPGA using 17 bits of resolution and an 8 kHz sampling frequency. The vectors, `x` and `y`, returned from this function (when the FPGA is configured to implement the system in Fig. 3) are plotted in Fig. 4a.

##### 4.1.1 `sendcos()`

The `sendcos` function sends a sinusoidal waveform to the FPGA and returns the output from the FPGA. This function takes the frequency, duration of the waveform, scaling factor (resolution), and sampling frequency as input arguments. Internally, `sendcos` generates a cosine waveform of the specified frequency and length with an amplitude of 1. To convert this floating-point value into a binary, fixed-point number, the input samples are multiplied by a scaling factor and then rounded to the nearest integer. Upon return from the FPGA,

the samples are divided by the scaling factor to remove its effect on the gain of the overall system.

The scaling factor does, however, have an effect on the output signal. By varying the resolution, the amount of quantization error is also varied. Additionally, choosing a scaling factor that is too large can cause overflows to occur. By adjusting the scaling factor, overflow conditions can be eliminated, or they can be induced to study their effects on the system.

Figure 5 shows a sample call of the *sendcos* function. For this example, a scaling factor of 65535 is used, so the input samples will be 17 bits long (remember the sign bit). Currently, the inputs are always sign-extended to 32 bits.

The input waveform generated by this function,  $x$ , and output waveform returned from it,  $y$ , are plotted in Fig. 4a when the FPGA is configured to implement the system in Fig. 3.

---

```
>> [y] = sendwave(x, 65535);
```

---

Figure 6: In this example, the input waveform  $x$  is sent to the FPGA using 17 bits of resolution. The scaling factor is removed from the output from the FPGA system and returned in  $y$ .

#### 4.1.2 sendwave()

The *sendwave* function sends any arbitrary waveform to the FPGA and returns the output from the FPGA. This function takes the input waveform and scaling factor (resolution) as input arguments. The samples of the input waveform are expected to be  $1 > x > -1$ . Similarly to the *sendcos* function, the scaling factor determines the length (e.g., resolution) of the input samples. Again, the input length is limited to 32 bits. Figure 6 shows a sample call of the *sendwave* function. If the input waveform,  $x$ , is a 600 Hz cosine signal with a sampling frequency of 8 kHz, then the output would be identical to that of the *sendcos* function in Fig. 5 assuming the FPGA is configured identically.

---

```
>> freq = [100 : 100 : 4000];  
>> [mag, x, y] = freq_response(freq, 30, 65535, 8000);
```

---

Figure 7: This code generates the magnitudes of a discrete freq. response,  $X[k]$ , where the discrete frequencies are specified in the input vector  $freq$ . The duration, scaling factor, and sampling frequency arguments are used in the calls to *sendcos* that are made internally.

#### 4.1.3 freq\_response()

The *freq\_response* function uses the *sendcos* function to send a range of sinusoidal waveforms to the FPGA and returns the magnitudes of the respective outputs. In this way, an experimental frequency response magnitude vector is generated. If the input frequency vector contains equidistant values of the form  $freq(i + 1) = freq(i) + constant$ , then the returned vector  $mag$  represents the magnitudes of the discrete frequency response,  $X[k]$ .

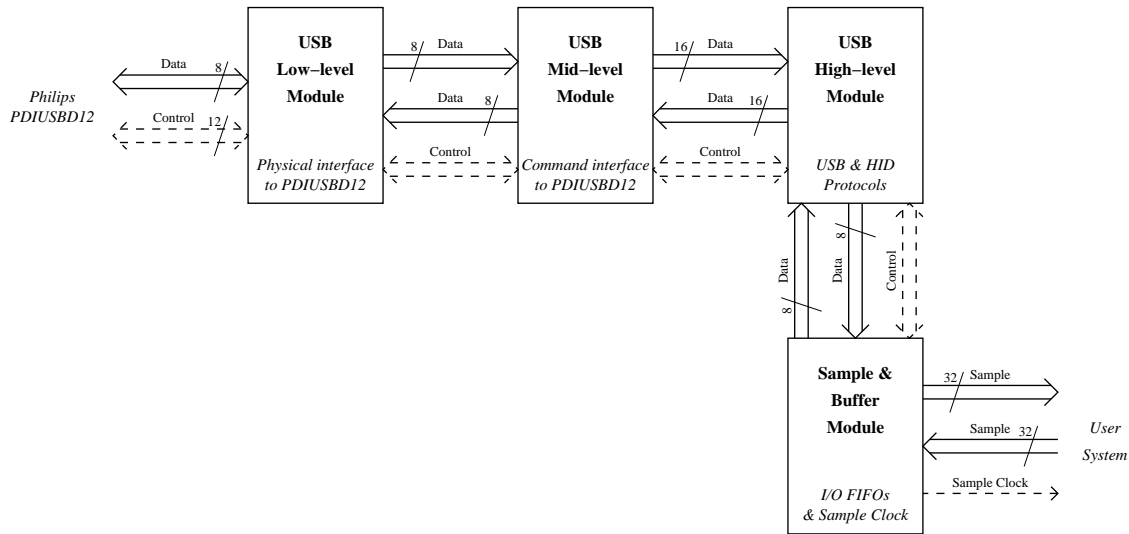


Figure 8: The USB interface on the FPGA is comprised of four basic blocks. The low-level USB module communicates directly with the PDIUSB12 chip. The mid-level USB module implements the command interface protocol for the PDIUSB12 chip. The high-level USB module implements the USB and HID protocols, and the Sample & Buffer module buffers the USB packets and provides a sample-level interface to the students' code, which is designated as User System here.

A sample call of the *freq\_response* function is shown in Fig. 8. For this example, the discrete frequency response will be sampled at 100 Hz intervals over the valid frequency range with an 8 KHz sampling frequency. 30 ms of data will be measured for each sample frequency, and as in the previous examples, 17 bits of resolution is selected by using a scaling factor of 65535. The output of this function is plotted in Fig. 4b along with the theoretical frequency response generated in MATLAB.

## 4.2 FPGA Interface

On the FPGA side, five drop-in VHDL modules provide the complete interface to the USB port and the MATLAB toolbox described in the previous section. As shown in Fig. 8, the low-level USB module communicates directly with the Philips PDIUSB12 integrated chip, which handles the physical-level USB connection to the USB port on the PC. The mid-level USB module translates the commands from the high-level module into multiple command and data transfers with the PDIUSB12 chip. The high-level USB module implements the USB and Human Interface Device (HID) protocols. The bulk of this module is involved with the automatic enumeration of the USB device when it is plugged into a PC as required by the USB specification. Finally, the sample and buffer module stores the incoming and outgoing data in FIFO buffers, provides the conversion between bytes and samples, and generates the sample clock, which can be used by the students' code to read the next sample. Except for adding the components and respective signal routing code for these modules to a top-level VHDL design file, the students need only be concerned with the data and control signals presented at the interface to the sample and buffer module.

---

```

COMPONENT usb_clk IS
  PORT( inclk0 : IN  STD_LOGIC;
        c0     : OUT STD_LOGIC;
        c1     : OUT STD_LOGIC;
        locked : OUT STD_LOGIC
      );
END COMPONENT usb_clk;

COMPONENT usb_interface IS
  PORT( clock_45MHz, reset      : IN  STD_LOGIC;
        clock_90Mhz           : IN  STD_LOGIC;
        vbus, dmreq_n         : IN  STD_LOGIC;           -- PDIUSB12 control signals
        int_n, suspend        : IN  STD_LOGIC;           -- PDIUSB12 control signals
        sample_in             : IN  STD_LOGIC_VECTOR(31 DOWNTO 0); -- Data from user module
        usb_data              : INOUT STD_LOGIC_VECTOR( 7 DOWNTO 0); -- PDIUSB12 data bus
        sample_out            : OUT  STD_LOGIC_VECTOR(31 DOWNTO 0); -- Data to user module
        sample_clk            : OUT  STD_LOGIC;
        a0, reset_n, dmack_n  : OUT  STD_LOGIC;           -- PDIUSB12 control signals
        wr_n, rd_n, ale, cs_n : OUT  STD_LOGIC;           -- PDIUSB12 control signals
      );
END COMPONENT usb_interface;

COMPONENT filter IS
  PORT( reset : IN  STD_LOGIC;
        clock : IN  STD_LOGIC;
        x     : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
        y     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
      );
END COMPONENT filter;

```

---

Figure 9: The component declarations of the three modules here are needed to implement the FPGA interface. The system logic needed to implement the system in Fig. 3 is located in the “filter” module.

#### 4.2.1 Top-level Project File

The top-level project file typically contains the components and routing logic for at least three modules: the top-level USB interface (this contains the five modules described in the previous section), clock PLL (to generate the 45 MHz and 90 MHz clocks), and the student’s DSP system. Figure 9 shows the component declarations for these three modules. The “filter” module is a sample DSP system (the one illustrated in Fig. 3). It is the only module that needs to be modified by the students to implement a different system. Figure 10 shows the VHDL code for implementing the signal routing between the modules. Signal names beginning with *int* are internal signals defined in the local architecture statement, and signal names beginning with *ext* are external input/output signals that are defined in the top-level entity statement.

#### 4.2.2 Data and Control Signals

The interface to the filter code is comprised of two data busses and five control signals. The data busses, *sample\_in* and *sample\_out*, are both 32 bits long, but only those bits which are used need to be routed to the students’ modules. The remaining bits of *sample\_out* can be left unattached, and the remaining bits of *sample\_in* should be tied to the sign bit of the data coming from the students’ modules.

The control signal *sample\_clock* provides a clock that advances a single period each time

---

```

clk1: usb_clk
  PORT MAP ( inclk0      => ext_clock,
             c0         => int_clk_45mhz,
             c1         => int_clk_90mhz
           );

usb1: usb_interface
  PORT MAP ( reset      => ext_reset,
             clock_45MHz => int_clk_45mhz,
             clock_90MHz => int_clk_90mhz,
             vbus       => ext_vbus,
             dmreq_n    => ext_dmreq_n,
             int_n      => ext_int_n,
             suspend    => ext_suspend,
             sample_in  => int_x,
             usb_data   => ext_usb_data,
             sample_out => int_y,
             sample_clk => int_sample_clock,
             a0         => ext_a0,
             reset_n    => ext_reset_n,
             dmack_n    => ext_dmack_n,
             wr_n       => ext_wr_n,
             rd_n       => ext_rd_n,
             ale        => ext_ale,
             cs_n       => ext_cs_n
           );

filt1: filter
  PORT MAP ( reset      => ext_reset,
             clock      => int_sample_clock,
             x          => int_x,
             y          => int_y
           );

```

---

Figure 10: The port map declarations of the three modules describe the signal routing necessary for the FPGA interface. The system logic needed to implement the system in Fig. 3 is located in the filter module.

a new data sample is output from the `usb_interface` module. This signal can be used to clock the delay registers or otherwise control the data flow for the filter module.

In addition, the control signal `sample_reset` provides a reset that can be triggered by the on-board reset signal or a reset command from the MATLAB-FPGA toolbox. All filter delay registers should be cleared on reset to allow accurate system analysis for successive sine wave inputs.

## 5 Conclusions

The toolbox of MATLAB functions and VHDL code has been built and successfully used by students. The toolbox has proved very helpful in debugging, testing, and verifying hardware implementations of multipliers, FIR and IIR filters, and transform systems. Because of their familiarity with the MATLAB environment, students have been able to learn and use this testing platform very quickly. These materials—including the VHDL modules, MATLAB toolbox, and daughter card PCB files—are available on our website [9] for general use.

## References

- [1] L. S. DeBrunner and V. DeBrunner, “The case for teaching DSP algorithms in conjunction with implementations,” in *Proc. IEEE Signal Processing Society’s 2nd Signal Processing Education Workshop*, Pine Mountain, GA, Sept. 2002.

- [2] A. J. S. Ferreira and F. J. O. Restivo, "Grasping the potential of digital signal processing through real-time DSP laboratory experiments," in *Proc. IEEE Signal Processing Society's 2nd Signal Processing Education Workshop*, Pine Mountain, GA, Sept. 2002.
- [3] T. S. Hall and D. V. Anderson, "From algorithms to gates: Developing a pedagogical framework for DSP hardware design," in *Proc. IEEE Signal Processing Society's 2nd Signal Processing Education Workshop*, Pine Mountain, GA, Sept. 2002.
- [4] W. T. Padgett, "An undergraduate fixed point DSP course," in *Proc. IEEE Signal Processing Society's 2nd Signal Processing Education Workshop*, Pine Mountain, GA, Sept. 2002.
- [5] J. Vieira, A. Tome, and J. Rodrigues, "Providing an environment to teach DSP algorithms," in *Proc. IEEE Signal Processing Society's 2nd Signal Processing Education Workshop*, Pine Mountain, GA, Sept. 2002.
- [6] A. J. Kornecki, "Real-time systems course in undergraduate cs/ce programs," *CD-ROM Supplement, IEEE Transactions on Education*, vol. 40, pp. 9, Nov. 1997.
- [7] C. H. G. Wright, T. B. Welch, D. M. Etter, and M. G. Morrow, "Teaching hardware-based DSP: theory to practice," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing*, Orlando, FL, May 2002, vol. 4, pp. 4148–4151.
- [8] S. L. Wood, "Signal processing and architecture in the lower division electrical engineering core," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing*, May 2001, vol. 5, pp. 2713–2716.
- [9] T. S. Hall, *FPGA Resource Page*, School of Electrical and Computer Engineering at Georgia Institute of Technology, <http://www.ece.gatech.edu/academic/courses/fpga/>, Jan. 2004.

TYSON S. HALL is a PhD candidate in Electrical and Computer Engineering at Georgia Tech. His current research interests include rapid prototyping of mixed-signal systems, cooperative analog/digital signal processing, reconfigurable computing, and embedded systems. Mr. Hall received an MSECE from Georgia Tech in '01 and a BSCMPE from Georgia Tech in '99.

DAVID V. ANDERSON is an Assistant Professor in Electrical and Computer Engineering at Georgia Tech. He received a PhD from Georgia Tech in '99, an MSEE from Brigham Young University (BYU) in '94, and a BSEE from BYU in '93. Dr. Anderson's research interests include audition and psycho-acoustics, signal processing in the context of human auditory characteristics, and the real-time application of such techniques.