

# FROM ALGORITHMS TO GATES: DEVELOPING A PEDAGOGICAL FRAMEWORK FOR DSP HARDWARE DESIGN

Tyson S. Hall and David V. Anderson

Georgia Institute of Technology, Atlanta, GA 30332-0250, tyson@ece.gatech.edu

## ABSTRACT

*In this paper, we present a pedagogical framework for teaching DSP hardware design and provide the necessary technical infrastructure for enabling this methodology. Many curricula include extensive training in DSP theory and in VHDL modeling of hardware; however, there are few opportunities given to the student to combine these two skills into a working knowledge of DSP hardware design. Our 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 systems. Even though many curricula include separate classes in both DSP theory and in VHDL modeling, there are few opportunities given to the student 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. Field-programmable gate arrays (FPGAs) are used to synthesize the DSP hardware, and MATLAB is used to do the system simulation, data acquisition (via a serial interface between the FPGA and the PC), and data analysis. This system provides a purely digital prototyping and testing platform for implementing a wide variety of DSP systems. MATLAB is used as the testing platform interface to ease the transition from a theoretical understanding of DSP systems to their hardware implementation. Using FPGAs as a synthesis platform provides a fast and cost-effective way of prototyping hardware systems in a laboratory environment. This paper presents the MATLAB functions and VHDL modules that provide a transparent connection between MATLAB and a DSP system implemented on an FPGA.

## 2. THEORY TO HARDWARE ... AND BACK

Our goal is to teach hardware implementation of DSP algorithms to engineering students at the senior level. Students at this level have had exposure to MATLAB in the curriculum at Georgia Tech and MATLAB is the logical choice for use as a prototyping environment. Students doing digital hardware design use a hardware description language such as VHDL and many engineering students are familiar with this VHDL synthesis. However, the translation of an algorithm from MATLAB to a hardware environment proves to be non-trivial.<sup>1</sup> One of the great difficulties is the problem of 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.

What is needed is an easy way for students to verify and debug hardware implementations of DSP algorithms using the tools that they already know. Our solution to this problem is a set of VHDL modules and MATLAB programs that can be used to pass signals to and 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 serial 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 Xilinx FPGAs

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

and ISE design software. The work presented in this paper provides an interface between these two environments. The connection is currently made up of a standard RS-232 serial connection between the PC and the FPGA. The FPGA board we use is a student board (Digilab 2E from Digilint, Inc.) provided by Xilinx, Inc. that is relatively inexpensive to acquire.

### 3.1.1. PC side

On the PC side, a toolbox of MATLAB functions provide the ability to do things such as 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 are a number of functions that handle the normalization, conversion, and byte encapsulation of samples for transmission, the sending and receiving of byte data, minimal error detection across the communication channel, and the conversion and decapsulation of bytes received from the FPGA back into valid MATLAB sample values.

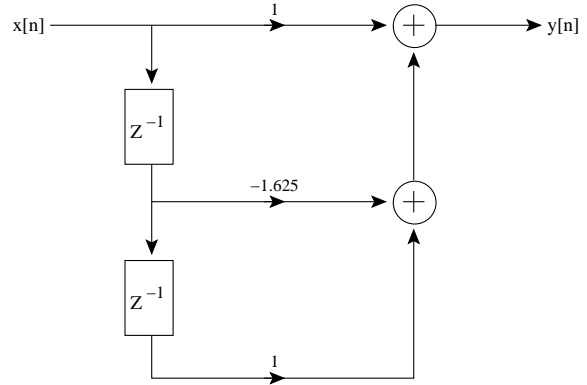
### 3.1.2. FPGA side

On the FPGA side, VHDL modules (and the Xilinx project directory) have been created that receive bytes from the serial connection, rebuild the multi-byte samples (currently supports up to 20-bit samples), issue the sample and a sample clock pulse to the students' DSP system, receive the output sample, break the sample up into multiple bytes and transmit them back to the PC through the serial connection. 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 will move from a (hopefully) familiar point and progress through the simulation and synthesis of the hardware. By leading them through their first complete system in a highly directed manner, students will be able to complete the entire design and implementation process within a few hours. The MATLAB–Xilinx 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 toolbox



**Fig. 1.** This simple filter design will be used to illustrate the functionality of the MATLAB–Xilinx interface described here.

also becomes useful for investigating engineering trade-offs such as the impact of coarser quantization.

## 3.3. Example

To illustrate the use of our MATLAB–Xilinx system, the FIR filter illustrated in Figure 1 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 Figure 2, which shows the output of the hardware filter for sine wave input of 600 Hz with a 8 kHz sampling frequency. Figure 3 shows the theoretical frequency response obtained from the *freqz* function in MATLAB's Signal Processing toolbox plotted in blue, and the experimental frequency response measured from the hardware filter itself using the *freq\_response* function plotted in red. In this case, the results are seemingly identical, because the coefficients were chosen to minimize quantization error.

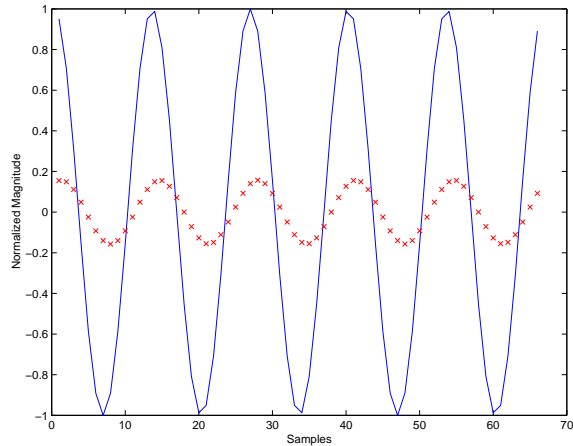
## 4. IMPLEMENTATION

### 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 and used to send and receive data from the FPGA. These functions provide the ability for students to quickly process data using the FPGA and return the results. Those needing additional functionality could easily modify the MATLAB code but that has not been needed with our situation.

#### 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



**Fig. 2.** This illustrates the output of a simple FIR filter implemented on a Xilinx 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.

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 scaling factor to remove its effect on the gain of the overall system.

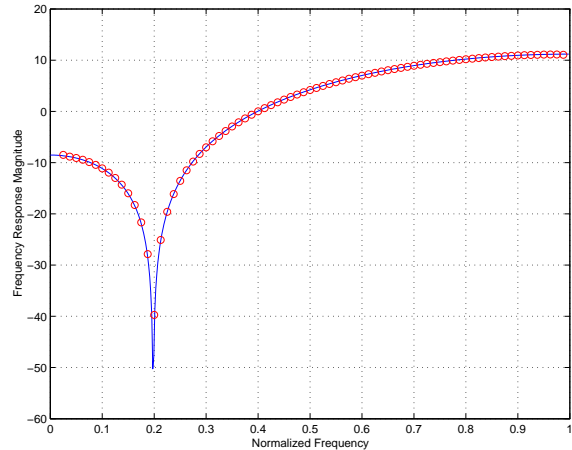
The scaling factor does, however, have an effect on the output signal. By varying the resolution, the amount of quantization error is also varied. Additionally, choosing a scaling factor that is too large can cause overflows to occur. By adjusting the scaling factor, overflow conditions can be eliminated,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

Figure 4.1 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 20 bits and prepended with a 4-bit header to give three bytes per sample, but with only minor modifications, these functions can be customized to use any desired length.

The input waveform generated by this function,  $x$ , and output waveform returned from it,  $y$ , are plotted in Figure 3 when the FPGA is configured to implement the system in Figure 1.

#### 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



**Fig. 3.** 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.

---

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

---

**Fig. 4.** 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 Figure 1) are plotted in Figure 2.

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 (i.e., resolution) of the input samples. Currently, the input is limited to 20 bits, but with only minor modifications, these functions can be extended. Figure 4.1 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 Figure 4.1 assuming the FPGA were configured identically.

#### 4.1.3. *freq\_response()*

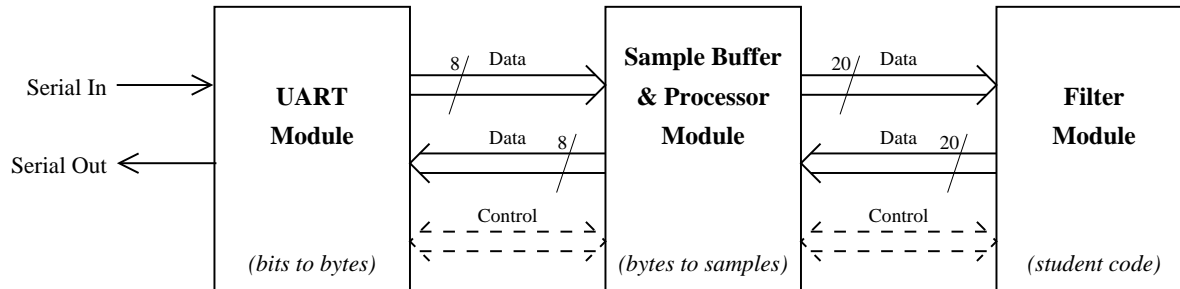
The *freq\_response* function uses the *sendcos* function to send a range of sinusoidal waveforms to the FPGA and returns the magnitudes of the respective outputs. In this way, an experimental frequency response magnitude vector

---

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

---

**Fig. 5.** 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  $y$ .



**Fig. 6.** The FPGA interface is comprised of three basic blocks. The UART module communicates directly with the PC's serial port and converts the serial bit stream into bytes. The Sample Buffer and Processor module buffers the bytes and provides a sample-level interface to the students' code, which resides in the Filter module.

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

**Fig. 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.

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,  $X[k]$ .

#### 4.1.4. Other Functions

In addition to the functions mentioned above, our MATLAB toolbox has a number of smaller utilities. One such function is *float2bin*, which converts a signed, decimal number into a signed, fixed-point, binary number. This is useful for generating the coefficient memories when implementing filters on FPGAs. Other functions and scripts have been generated to automate many of the repetitive tasks involved in testing and verification of hardware systems using our MATLAB-Xilinx infrastructure.

## 4.2. FPGA Interface

On the FPGA side, two drop-in VHDL modules provide the complete interface to the serial port and the Matlab toolbox described in previous section. As shown in Figure 6, the UART module communicates directly to the serial port on the PC, while the SAMPLE\_PROC (sample buffer and processor) module controls the UART module and provides a straight-forward interface to the user's design. Except for adding the components and respective signal routing code for these modules to their top-level VHDL design file, the student only needs to be concerned with the data and control signals presented at the interface to the SAMPLE\_PROC module.

### 4.2.1. VHDL Code

The top-level VHDL file must contain the component declarations and routing logic for at least three basic modules: the UART, Sample Buffer and Processor, and DSP System. Figure 4.2.2 shows the component declarations for these three modules. The FILTER module is a sample DSP system (the one illustrated in Figure 1). It is the only module that needs to be modified by the students to implement a different system. Figure 4.2.3 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. Sample Buffer and Processor

The SAMPLE\_PROC module generates the control signals for the UART module and buffers the bytes so as to provide a sample-level interface to the students' code. It also checks for the header nibble and uses this information to determine the first byte of each sample. Figure 4.2.2 shows the component description of this module and the signal routing necessary for implementing the filter shown in Figure 1.

### 4.2.3. Data and Control Signals

The interface to the students' code is comprised of two data busses and five control signals. The data busses, SAMPLE\_IN and SAMPLE\_OUT, are both 20 bits long, but only those bits which are used need to be routed to the students' module. 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' module.

The SAMPLE\_OREADY and SAMPLE\_IREADY output control signals indicate when the output or input buffer respectively are ready for the next byte. The students' module needs to wait for the respective signal to go High be-

```

component UART
port(OREADY : inout std_logic;
SDATAOUT : inout std_logic;
SDATAIN : in std_logic;
IREADY : inout std_logic;
CHARIN : inout std_logic_vector(7 downto 0);
CHAROUT : inout std_logic_vector(7 downto 0);
CLOCK : in std_logic;
RESET : in std_logic;
WRITE : inout std_logic;
READ : inout std_logic);
end component UART;

component SAMPLE_PROC
port(CLOCK, RESET : in std_logic;
OREADY : in std_logic;
IREADY : in std_logic;
CHARIN : in std_logic_vector( 7 downto 0);
SAMPLE_IN : in std_logic_vector(19 downto 0);
READ, WRITE : out std_logic;
SAMPLE_CLOCK : out std_logic;
SAMPLE_OUT : out std_logic_vector(19 downto 0);
CHAROUT : out std_logic_vector( 7 downto 0);
SAMPLE_OREADY : out std_logic;
SAMPLE_IREADY : out std_logic;
SAMPLE_READ : in std_logic;
SAMPLE_WRITE : in std_logic);
end component SAMPLE_PROC;

component FILTER
port(CLOCK : in std_logic;
SAMPLE_CLOCK : in std_logic;
X : in std_logic_vector(19 downto 0);
SAMPLE_OREADY : in std_logic;
SAMPLE_IREADY : in std_logic;
READ, WRITE : out std_logic;
Y : out std_logic_vector(19 downto 0));
end component FILTER;

```

**Fig. 8.** The component declarations of the three modules here are needed to implement the FPGA interface. The system logic needed to implement the system in Figure 1 is located in the FILTER module.

fore the next data sample is read from or written to the SAMPLE\_PROC busses. After the SAMPLE\_OUT bus is read, the students' code needs to send a High pulse on the SAMPLE\_READ signal so the SAMPLE\_PROC module can clear the buffer and clear the SAMPLE\_OREADY signal in preparation for receiving the next sample from the UART. When writing a signal to the SAMPLE\_IN bus, the students' code must pulse the SAMPLE\_WRITE signal while the data is still valid to so the SAMPLE\_PROC module can write the data to its buffer and begin sending the data to the UART. The final control signal, SAMPLE\_CLOCK, provides a clock that advances a single period each time a new data sample is output from the SAMPLE\_PROC module. This signal can be used to clock the delay registers or otherwise control the data flow for the students' module.

#### 4.2.4. USB Interface

A USB daughter board and accompanying VHDL interface code is currently being developed to replace the RS-232 serial interface. The USB interface is advantageous primarily because it is much faster (on the order of 100x faster than the current scheme) and includes error correction and multi-byte packet transmission inherent in the protocol. The dis-

```

uart1: UART
port map (
OREADY => intOREADY,
SDATAOUT => extSDATAOUT,
SDATAIN => extSDATAIN,
IREADY => intIREADY,
CHARIN => intCHARIN,
CHAROUT => intCHAROUT,
CLOCK => extCLOCK,
RESET => extRESET,
WRITE => intWRITE,
READ => intREAD );

sample_proc1: SAMPLE_PROC
port map (
CLOCK => intBAUD_CLOCK,
RESET => intRESET,
OREADY => intOREADY,
IREADY => intIREADY,
SAMPLE_OUT => intSAMPLE_OUT,
CHARIN => intCHARIN,
READ => intREAD,
WRITE => intWRITE,
SAMPLE_CLOCK => intSAMPLE_CLOCK_OUT,
CHAROUT => intCHAROUT,
SAMPLE_OREADY => intSAMPLE_OREADY,
SAMPLE_IREADY => intSAMPLE_IREADY,
SAMPLE_READ => intSAMPLE_READ,
SAMPLE_WRITE => intSAMPLE_WRITE,
SAMPLE_IN => intSAMPLE_IN );

filter1: FILTER
PORT MAP(
CLOCK => extCLOCK,
SAMPLE_CLOCK => intSAMPLE_CLOCK,
X => intSAMPLE_OUT,
SAMPLE_OREADY => intSAMPLE_OREADY,
SAMPLE_IREADY => intSAMPLE_IREADY,
READ => intSAMPLE_READ,
WRITE => intSAMPLE_WRITE,
Y => intSAMPLE_IN );

```

**Fig. 9.** 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 Figure 1 is located in the FILTER module.

advantage is that it is a more complicated interface and requires additional logic gates to implement in VHDL. These problems are being mitigated by using a USB device IC that implements much of the protocol and by building interfacing modules on the FPGA that will provide an interface to the students' modules of similar complexity to the current one.

## 5. CONCLUSIONS

The toolbox of MATLAB functions and VHDL code has been built and successfully used by several undergraduate research 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.