Implementation of a Real-Time Spectrum Analyzer Synthesized on FPGA

David Lavoie-Boutin, 260583602

December 18, 2016

Abstract

As a final project for the Signal Processing Hardware course (ECSE 436), I was tasked with designing, implementing and deploying a Real-Time spectrum analyzer. This project included the custom implementation of the FFT algorithm in Matlab, porting that algorithm to HDL code that would be synthesized for an FPGA, the acquisition of audio samples and display of the FFT results on the same FPGA. To achieve this goal, I implemented the common CooleyTukey algorithm for computing the FFT. I used Matlab and Simulink to generate the HDL code for the FFT algorithm based on the Simulink testbench created. I created the control logic modules to prepare the samples for the FFT and parse its output for the display module, which displays the magnitude of each FFT coefficient output.

Contents

1 Introduction & Background

The FFT algorithm is often considered as one of the algorithms that revolutionized the world, some even say, "the most important of our lifetime" [2]. Its applications are virtually endless and range from simple voice analysis to radar to every form of communication we know of. For this project, I had to create an audio spectrum analyzer that would employ the FFT to decompose the signal in its frequency components and display the magnitude of each component on a computer monitor.

In the end, this entire system was to run on an Altera DE-II development board which integrates the Cyclone II FGPA with a bunch of other dedicated controllers, including RAM, ADCs, USB, etc. First, I developed my custom implementation of the CooleyTukey FFT algorithm in Matlab. Then I created a Simulink testbench using Matlab's FFT function, optimized for HDL synthesis and used Matlab to generate the HDL code for the FFT.

I integrated the generated code the to rest of the FPGA project and linked it with some control logic to the acquisition system. The output of the FFT are then buffered until all the data is available and displayed on a VGA monitor. For this, I designed a simple module that plots an array of data in a bar chart.

2 Matlab Algorithm

The Matlab algorithm I implemented for the computation of the FFT follows very closely the CooleyTukey algorithm described on Wikipedia [3] and further developed in a paper by Stefan Worner [4]. My first implementation followed the recursive strategy and the second uses the iterative strategy:

Listing 1: "Recursive FFT implementation"

```
1  function X = myFFT(x)
2
3  N = numel(x);
4  x_even = x(1:2:end);
5  x_odd = x(2:2:end);
6
7  if N>=8
8      X_even = myFFT(x_even);
9      X_odd = myFFT(x_odd);
10
11     Wn = exp(-1i*2*pi*((0:N/2-1)')/N);
```

```
12      tmp = Wn .* X_odd;
13      X = [(X_even + tmp);(X_even -tmp)];
14
15  elseif N == 2
16      X = [1  1;1  -1]*x;
17  elseif N == 4
18      X = [1 0 1 0; 0 1 0 -1i; 1 0 -1 0;0 1 0 1i]*[1 0 1 0;1 0 -1
            0;0 1 0 1;0 1 0 -1]*x;
19  else
20      error('N not correct.');
21  end
22  end
```

Listing 2: "Iterative FFT implementation"

```
1  function d = fft_it(x)
2  % Cooley-Tukey flavor of the FFT algorithm
3  % Based on the implementation presented in Fast Fourier
       Transform
4  % by Stefan Worner of Swiss Federal Institute of Technology
       Zuric
5
6      N = length(x);
7      d = x(bitrevorder(1:N));
8      q = log2(N);
9
10     for j = 1:q
11        m = 2^(j-1);
12        d = exp(-2 *pi * i /m).^(0:m-1);
13        for k = 1:2^(q-j)
14            s = (k-1)*2*m+1; % start-index
15            e = k*2*m; % end-index
16            r = s + (e-s+1)/2; % middle-index
17            y_top = x(s:(r-1));
18            y_bottom = x(r:e);
19            z = d .* y_bottom;
20            y = [y_top + z, y_top - z];
21            x(s:e) = y;
22        end
23     end
24 end
```

After implementing my own version of FFT, I considered using Matlab to generate synthesizable HDL code. Using Simulink and the DSP toolbox, I used the HDL optimized FFT function in the example diagram shipped with Matlab and adapted it to my input specifications. The schematic is illustrated in figure 1.

Running the simulation compares the output of the built-in FFT function

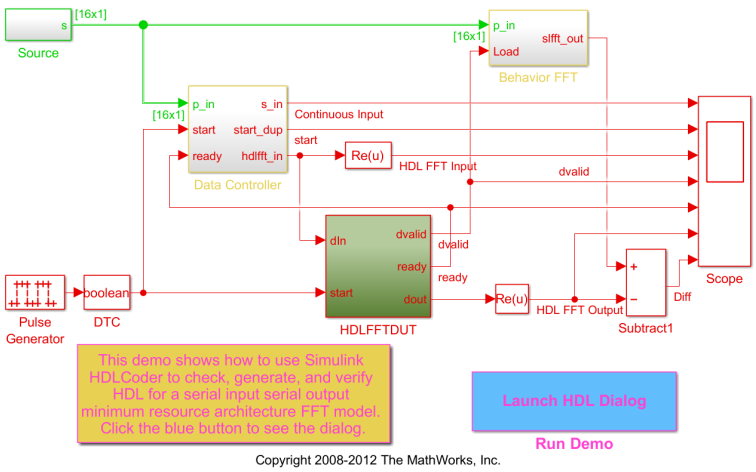

Figure 1: Simulink schematic of a HDL optimized FFT

to the HDL optimized FFT function and we can see that they do not match. This is due to the optimization strategy of the HDL block where the e^x function is replaced with a lookup table which is not as precise as the native function. With this schematic established, I was able to use the HD Coder function in Matlab to generate the VHDL code for the FFT component.

3 Display on the FPGA

Before implementing the full processing to display pipeline, I created and tested the display module on its own using the infrastructure provided in Lab 3. For this purpose, I designed a modular block that displays an array of values as a bar chart. There are several interesting points I want to outline in the implementation of this block.

3.1 Package Definitions

In order to maximize modularity of the code developed, I choose to define custom types and package constants to parameters that would be used in different modules and that were susceptible to change frequently in the life of the project. These include a specific data type to pass values to display as an array, the number of data samples and the number of bits used to represent those samples.

```
1 package graph_data_parameters is
```

```
2  constant SAMPLES_DATA_LENGTH: integer := 16;  --bits wide
3  constant NUMBER_OF_SAMPLES : integer := 6; --number of samples
       to analyse
4  type data_array is array(0 to NUMBER_OF_SAMPLES -1) of
       std_logic_vector (SAMPLES_DATA_LENGTH -1 downto 0);
5  end package graph_data_parameters;
```

3.2 Modular Bar Chart

Another interesting design point in this graphing module is the use of compilation time resolution of different parameters such as the number of bars and the width of each bars. This is done using simple arithmetic, but makes the design very modular simply using the constants described earlier.

```
1      col_width := 639 / NUMBER_OF_SAMPLES;
2      col_index := to_integer(unsigned(x)) / col_width;
```

3.3 Array Input

The last point I want to mention is the use of custom types to pass and manipulate the set of data to graph. The use of the array type make is easy to address the data elements using simple indexes and allows us to iterate on the data using a "for loop" construct in VHDL.

```
1  for j in 0 to NUMBER_OF_SAMPLES-1 loop
2      temp := pad & switch(j*switch_range_width+switch_range_width
           -1 downto j*switch_range_width);
3      data(j) <= temp(SAMPLES_DATA_LENGTH -5 downto 0) & "0000";
           -- convert range of switches to data element
4  end loop;
```

3.4 Switch to Test

In order to test this design, I used the switches on the board to emulate data and confirm the good behavior of the module. First I created a module that converts the vector of switch values to the data array type I defined.

```
1  entity switch_to_array is
2    port (
3      switch : in std_logic_vector(17 downto 0);
4      data : out data_array
5    ) ;
6  end entity ; -- switch_to_array
7
8  architecture arch of switch_to_array is
```

```
9
10 begin
11
12 switch_changes : process( switch )
13 variable switch_range_width : integer := 18/NUMBER_OF_SAMPLES;
14 variable pad : std_logic_vector(SAMPLES_DATA_LENGTH - 1 -
       switch_range_width downto 0);
15 variable temp : std_logic_vector(SAMPLES_DATA_LENGTH -1 downto
       0);
16 begin
17     pad := (others => '0');
18     for j in 0 to NUMBER_OF_SAMPLES-1 loop
19                             temp := pad & switch(j*
                                    switch_range_width+switch_range_width
                                    -1 downto j*switch_range_width);
20         data(j) <= temp(SAMPLES_DATA_LENGTH -5 downto 0) & "0000
               "; -- convert range of switches to data element
21     end loop;
22 end process ; -- switch_changes
23
24
25 end architecture ; -- arch
```

I then used Modelsim to simulate this module and confirm its good behavior which yielded figure 2. In the figure we can see that each sample is represented by a group of 4 switches and that switches 17 and 18 are not mapped to any sample as 18 is not divisible by 4.

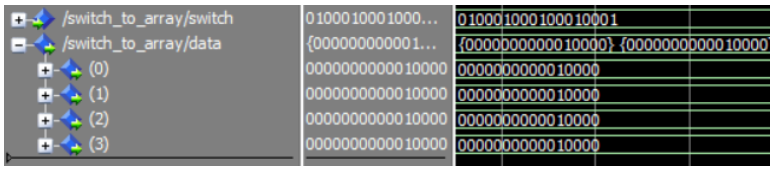

Figure 2: Modelsim simulation of the switch to array module

4 FFT on FPGA

After implementing and testing the graphing module, I used the infrastructure provided for lab 2 and modified it with the modules I needed to run the FFT on the audio samples. For this I created the block diagram shown in figure 3

This block diagram show several additional modules. The first block is the data controller. This block simply casts the 32 bit audio sample to a

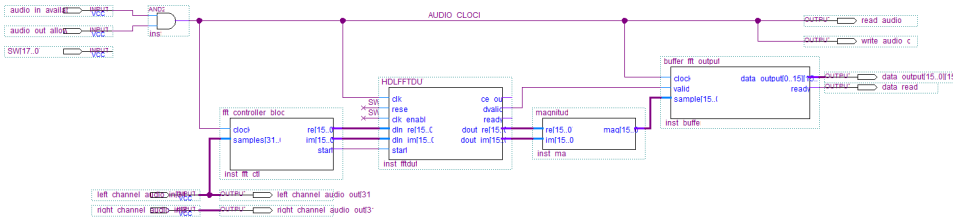

Figure 3: FFT integration in the audio processing unit

16 bits representation that the FFT block will use. It also drives the start signal which indicates a new set of inputs for the FFT computation. The data is then passed to the FFT block.

```
entity fft_controller_block is
  port (
    clock : in std_logic;
    samples : in std_logic_vector(31 downto 0) ;
    re : out std_logic_vector(15 downto 0) ;
    im : out std_logic_vector(15 downto 0) ;
    start : out std_logic
  ) ;
end entity ; -- fft_controller_block

architecture arch of fft_controller_block is
begin
    re <= samples(31 downto 16);
    im <= (others => '0');
    --start <= clock;

    counter : process( clock )
        variable count : integer := 0;
    begin
        if rising_edge(clock) then
            count := count + 1;
            if count = NUMBER_OF_SAMPLES then
                count := 0;
                start <= '1';
            else
                start <= '0';
            end if ;
        end if ;

    end process ; -- counter
end architecture ; -- arch
```

After the FFT, we compute the magnitude of the coefficients using the

traditional geometric norm. This poses the challenge of requiring us to compute the square root of the numbers which is not that trivial. For this task, I used an implementation of the Non-Restoring Square Root Algorithm [1].

Listing 3: "Square-Root function"

```
package sqrt_p is
    function  sqrt  ( d : UNSIGNED ) return UNSIGNED;
end package ; -- sqrt_p

package body sqrt_p is
function  sqrt  ( d : UNSIGNED ) return UNSIGNED is
    variable a : unsigned(31 downto 0):=d;  --original input.
    variable q : unsigned(15 downto 0):=(others => '0');  --
        result.
    variable left,right,r : unsigned(17 downto 0):=(others =>
        '0');  --input to adder/sub.r-remainder.
    variable i : integer:=0;

    begin
        for i in 0 to 15 loop
            right(0):='1';
            right(1):=r(17);
            right(17 downto 2):=q;

            left(1 downto 0):=a(31 downto 30);
            left(17 downto 2):=r(15 downto 0);
            a(31 downto 2):=a(29 downto 0);  --shifting by 2 bit
                .

            if ( r(17) = '1') then
                r := left + right;
            else
                r := left - right;
            end if;

            q(15 downto 1) := q(14 downto 0);
            q(0) := not r(17);
        end loop;
    return q;

end sqrt;
end sqrt_p;
```

Listing 4: "Magnitude module"

```
entity magnitude is
  port (
    re : in std_logic_vector(15 downto 0) ;
```

```
4       im : in std_logic_vector(15 downto 0) ;
5       mag : out std_logic_vector(15 downto 0)
6     ) ;
7   end entity ; -- magnitude
8
9   architecture arch of magnitude is
10      signal square : unsigned(31 downto 0) ;
11  begin
12      square <= ((unsigned(re)*unsigned(re))+(unsigned(im)*
            unsigned(im)));
13      mag <= std_logic_vector(sqrt(square));
14  end architecture ; -- arch
```

The last step before displaying the data is to buffer a set of outputs since the FFT outputs the coefficients in series and we want to display all the data at once. For this purpose I implemented a simple buffer module that build an array with the samples and published it once it is filled.

Listing 5: "Buffer module for the FFT output"

```
1   entity buffer_fft_output is
2     port (
3       clock : in std_logic;
4       valid : in std_logic;
5       sample : in std_logic_vector(SAMPLES_DATA_LENGTH - 1 downto
            0) ;
6       data_output : out data_array;
7       ready : out std_logic
8     ) ;
9   end entity ; -- buffer_fft_output
10
11  architecture arch of buffer_fft_output is
12
13  begin
14
15  identifier : process( valid, clock, sample )
16      variable count : integer := 0;
17      variable last_valid : std_logic;
18  begin
19      if (rising_edge(clock)) then
20          if (valid = '0') then
21              -- rising edge of valid
22              count := 0;
23              ready <= '0';
24          end if ;
25
26          if (valid = '1') then
27              -- iterate over count
28              data_output(count) <= sample;
```

```
29              count := count + 1;
30          end if;
31
32          if (count = NUMBER_OF_SAMPLES) then
33              -- falling edge valid
34              ready <= '1';
35          end if ;
36
37          last_valid := valid;
38      end if ;
39  end process ; -- identifier
40
41  end architecture ; -- arch
```

And in order to test the proper behavior of this module, I also created a Modelsim testbench and simulated its behavior. In the simulation results in figure 4, notice how each cell is loaded sequentially and how the "valid" signal is set once all the cells are loaded.

Listing 6: "Buffer module testbench"

```
1
2  proc GenerateCPUClock {} {
3      force -deposit /buffer_fft_output/clock 1 0 ns, 0 0.5 ns -
           repeat 1 ns
4  }
5
6  proc AddWaves {} {
7      add wave -position end  sim:/buffer_fft_output/clock
8      add wave -position end  sim:/buffer_fft_output/valid
9      add wave -position end -radix unsigned sim:/
           buffer_fft_output/sample
10     add wave -position end  sim:/buffer_fft_output/ready
11     add wave -position end -radix unsigned sim:/
           buffer_fft_output/data_output
12 }
13
14 proc Init {} {
15     vlib work
16     vcom graph_display.vhd
17     vcom buffer_fft_output.vhd
18
19     vsim buffer_fft_output
20     AddWaves
21     GenerateCPUClock
22
23     force -deposit sim:/buffer_fft_output/valid 0 0
24 }
25
```

```
Init


force -deposit sim:/buffer_fft_output/valid 1 0
force -deposit sim:/buffer_fft_output/sample 0001110001110001 0
run 6ns
force -deposit sim:/buffer_fft_output/valid 0 0
force -deposit sim:/buffer_fft_output/sample UUUUUUUUUUUUUUU 0
run 3ns
force -deposit sim:/buffer_fft_output/valid 1 0
force -deposit sim:/buffer_fft_output/sample 0000000000000000 0
run 1ns
force -deposit sim:/buffer_fft_output/sample 0000000000000001 0
run 1ns
force -deposit sim:/buffer_fft_output/sample 0000000000000010 0
run 1ns
force -deposit sim:/buffer_fft_output/sample 0000000000000011 0
run 1ns
force -deposit sim:/buffer_fft_output/sample 0000000000000100 0
run 1ns
force -deposit sim:/buffer_fft_output/sample 0000000000000101 0
run 1ns
force -deposit sim:/buffer_fft_output/valid 0 0
force -deposit sim:/buffer_fft_output/sample UUUUUUUUUUUUUUU 0
run 3ns
```

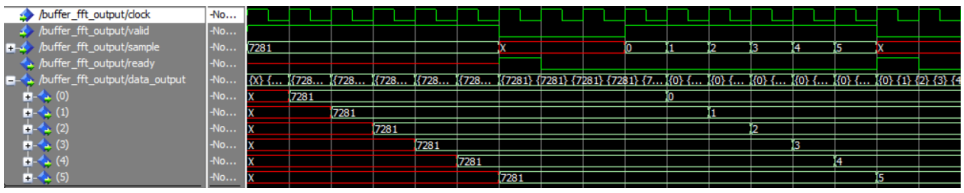

Figure 4: Simulation results for the buffer module

5 Conclusion

For this final project, I designed, implemented, integrated and tested a real-time spectrum analyzer which uses the FFT to decompose a signal in its frequency components which are then plotted on a computer monitor. This implementation was synthesized from Verilog and VHDL code and loaded on a Cyclone II FPGA development board.

Preliminary steps in the project included a Matlab and Simulink imple-

mentation and simulation of the FFT algorithm, generation of HDL code for the FFT using Matlab, Modelsim simulations of different HDL modules and finally a full deployment and test of the processing pipeline.

During the tests with the FPGA, the data observed on the screen was very noisy. I must point out that signal acquired by the FPGA itself was very noisy to begin with so many parasitic frequency components are expected on the output and this is what I observed. When sitting idle, the bars shown on screen oscillated very much very fast, to the point where many samples aliased together and gave the impression the full bars were displayed.

When playing a tone, this parasitic noise behavior was still observed, but some bars clearly had more consistent values and we could see a clear value for that bar. Additionally, the position of that bar on the frequency scale matched the frequency of the tone, which leads me to believe this implementation is generally functional. With extra noise filtering efforts, I believe this project to turn to a fully functional analyzer.

6 References

References

[1] Vipin Lal. A vhdl function for finding square root. `http://vhdlguru.blogspot.ca/2010/03/vhdl-function-for-finding-square-root.html`. Visited: Dec. 12, 2016.

[2] Gilbert Strang. Wavelets. *American Scientist*, 3(82):253, 1994.

[3] Wikipedia. Cooleytukey fft algorithm. `https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm`. Visited: Dec. 12, 2016.

[4] Stefan Worner. Fast fourier transform. Master's thesis, Swiss Federal Institute of Technology Zurich.