FPGA-based Direct Conversion Receiver with Continuous Acquisition to a PC

Qian Liu and Steven W. Ellingson

July 13, 2009

Contents

1 Introduction

This report describes a FPGA-based direct conversion receiver with high-speed continuous acquisition to a PC over Ethernet using user datagram protocol (UDP). The system is implemented on an Altera® Stratix® II EP2S60 DSP development board [1], shown in Figure 1. The on-board Analog Devices AD9433 A/D converter (ADC) samples data at 120 MSPS, and the output from the receiver has a sample rate of 3.75 MSPS. Each output sample has 14 bits, 7-bit "I" and 7-bit "Q", sent as a 16-bit word. The receiver output is sent to a PC over 100BaseT Ethernet at about 62.7 Mb/s, of which 60 Mb/s is used for samples, and the remaining 2.7 Mb/s is protocol overhead. The receiver can be tuned to any one of 16 center frequencies between 1.75 MHz and 54.25 MHz, selected using the push-button switches SW4 – SW7 shown in Figures 1 and 2. The output 1 dB bandwidth is 3.5 MHz. Full scale with 1/2 bit back-off varies from 662 mV_{pp} to 940 mV_{pp} with the increasing frequency.

The rest of this report is organized as follows. Section 2 ("System Architecture") describes the system architecture. Section 3 ("Firmware") presents the FPGA firmware. Section 4 ("Demonstration") addresses the system performance. Finally, the appendices give the source code used in this report: Appendix A ("FPGA Firmware") is Verilog HDL code for the implementation of the direct conversion receiver, Appendix B ("Data Acquisition Script") is Python code to capture and save the data, and Appendix C ("Data Analysis Scripts") is a set of MATLAB/Octave scripts for analyzing captured data.

2 System Architecture

Figure 2 illustrates the system architecture. This design is an extension of the direct sampling design described in [2]. All the components of the receiver are implemented in EP2S60F1020C3 using Verilog HDL, and Altera® MegaCore® IP functions provide most of the signal processing functionality. The swing buffer, Nios II CPU and Ethernet interface are used for data transfer between FPGA and PC; see [2] for additional details.

Either ADC-A or ADC-B can be used for input. The ADC selection depends on the firmware programmed to FPGA upon powered-up, which is controlled by a 8-pin DIP switch (SW2 in Figure 1) as shown in Table 1.

The ADC clock selectors (J3 and J4 in Figure 1) are used for ADC clock source settings. In our design, their pins 1 and 2 should be connected to choose Stratix II PLL circuitry as the ADC clock source.

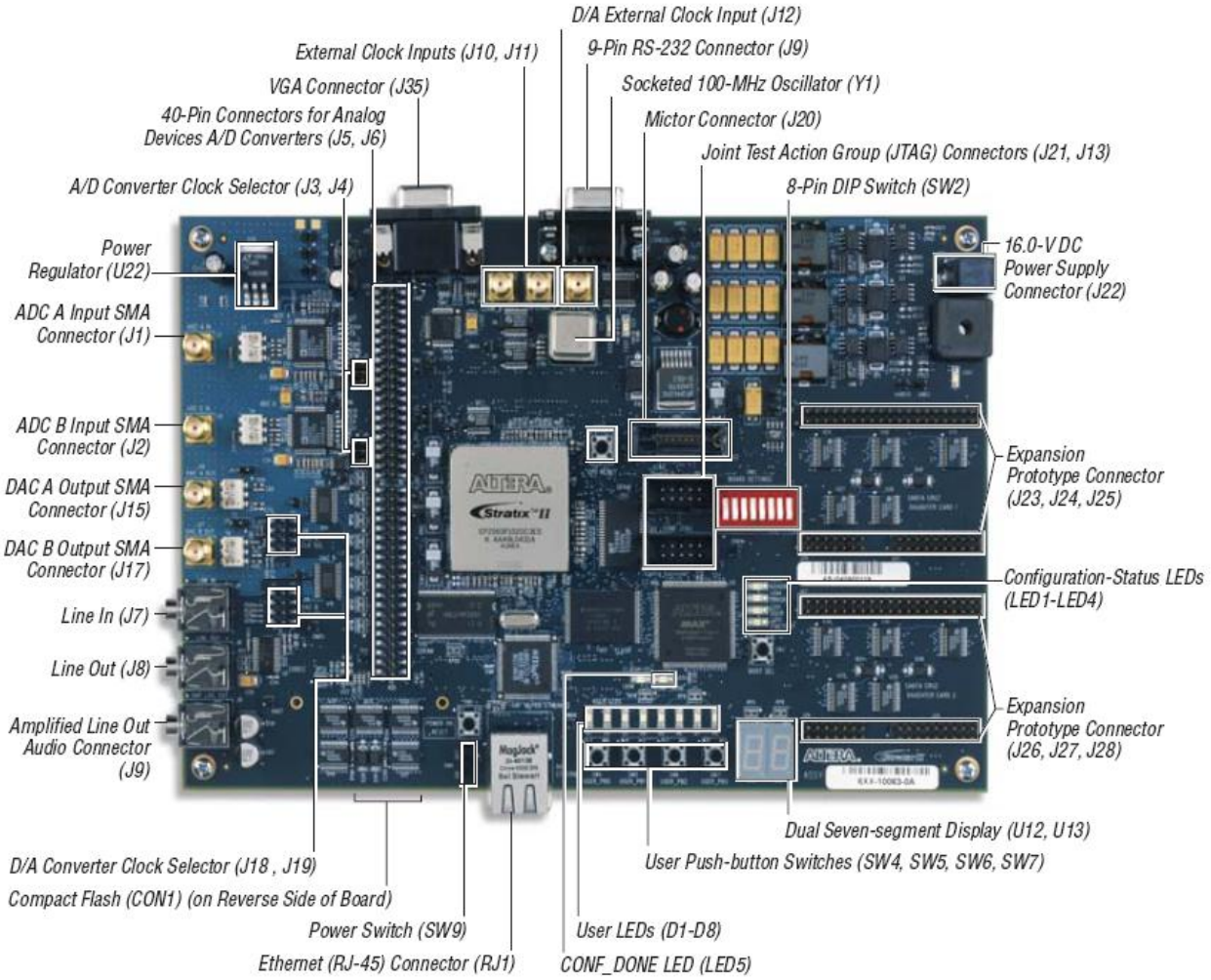


Figure 1: Stratix II EP2S60 DSP Development Board [1].

Table 1: Configuration DIP switch (SW2) state for firmware selection

SW2 State[1]				Seven-Segment Display U13	Input
Switch 1	Switch 2	Switch 3	Switch 4		
Open	Closed	Closed	Open	A	ADC-A
Closed	Open	Closed	Open	b	ADC-B

[1] The other switches of SW2 (switch 5, switch 6, switch 7 and switch 8) are not used and can be kept closed as default.

A sample is 14 bits: 7 bits for baseband "I" and another 7 bits for baseband "Q". 2-bit zeros are added to compose a 16-bit word, interfacing with the swing buffer. That is, the format of the 16-bit data is "$0Q_6Q_5Q_4Q_3Q_2Q_1Q_00I_6I_5I_4I_3I_2I_1I_0$", where I_i and Q_i denote the i-th bit of the output baseband "I" and "Q" respectively. A Verilog HDL script and a set of MATLAB/Octave scripts demonstrating the use of this format are shown in Appendix A and C.

3 Firmware

In Figure 2, the "Clock Synthesis" module uses the Altera® ALTPLL MegaCore® function to generate all the clocks for the components of the system: The 120 MHz clock is used as ADC sampling clock; the 3.75 MHz clock is used as the clock for the output register of receiver and the clock for data-width bridge of the swing buffer; the 1.875 MHz clock is used as the write clock for the swing buffer; and the 150 MHz clock is used for Nios II CPU and as the read clock for the swing buffer. The input to the "Clock Synthesis" module is currently the on-board 100 MHz crystal oscillator.

Figure 3 shows the architecture of the direct conversion receiver section of the FPGA firmware. The associated Verilog HDL source code is given in Appendix A.

The "Controllable Tuner" module uses the Altera® NCO MegaCore® function to generate the digital sine and cosine waveforms. The frequency of the waveforms, varying from 1.75 MHz to 54.25 MHz with 16 frequency steps, depends on the selected phase increment value which comes from the output of the 16:1 multiplexer, as Figure 3 shows. The state of the 4-bit control $s_3s_2s_1s_0$ is determined using the user push-button switches SW4, SW5, SW6, SW7. A logic "NOT" operation is performed on s_0 when the user pushes down SW4; the same operation will be performed on the other bits of the state register when their associated push-button switch is pushed down. Table 2 summarizes the center frequency selection scheme. The seven-segment display U12 (see Figure 1) indicates which center frequency is currently selected.

The filters that decimate and lowpass could have been implemented as either multirate FIR filters or cascaded integrator-comb (CIC) filters. Both can be implemented easily using Altera® MegaCore® functions [3]. However, with increasing decimation factor, the consumption of FPGA resources for the multirate FIR filter will increase faster. We implemented both types of filters with similar specifications and compared their resource utilization, illustrated in Table 3. The CIC filter has 4 stages, 1 differential delay, and decimator factor of 32. Their ideal responses are shown in Figure 4: Figure 4(a) is the ideal response of the

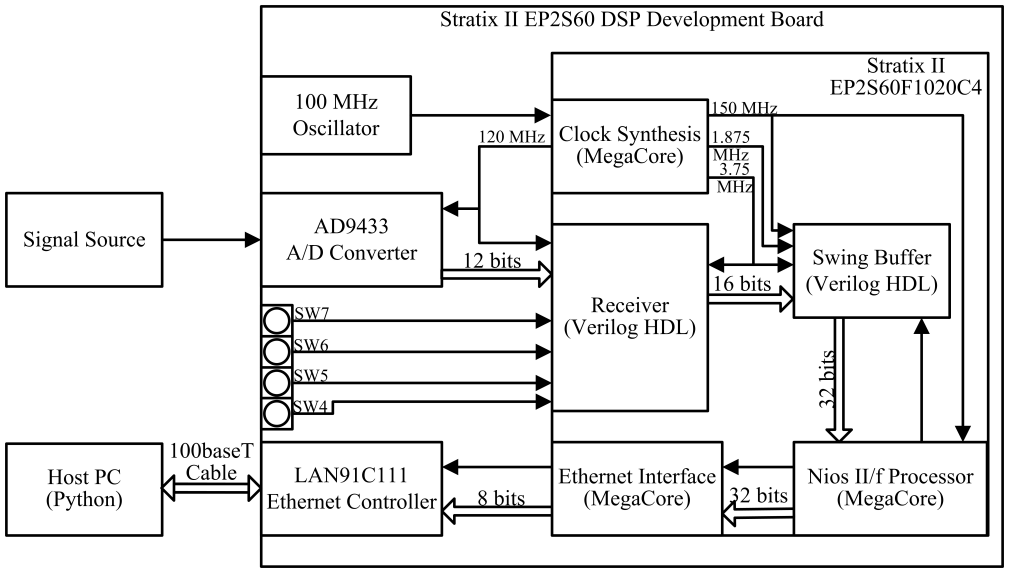


Figure 2: Block diagram of the system.

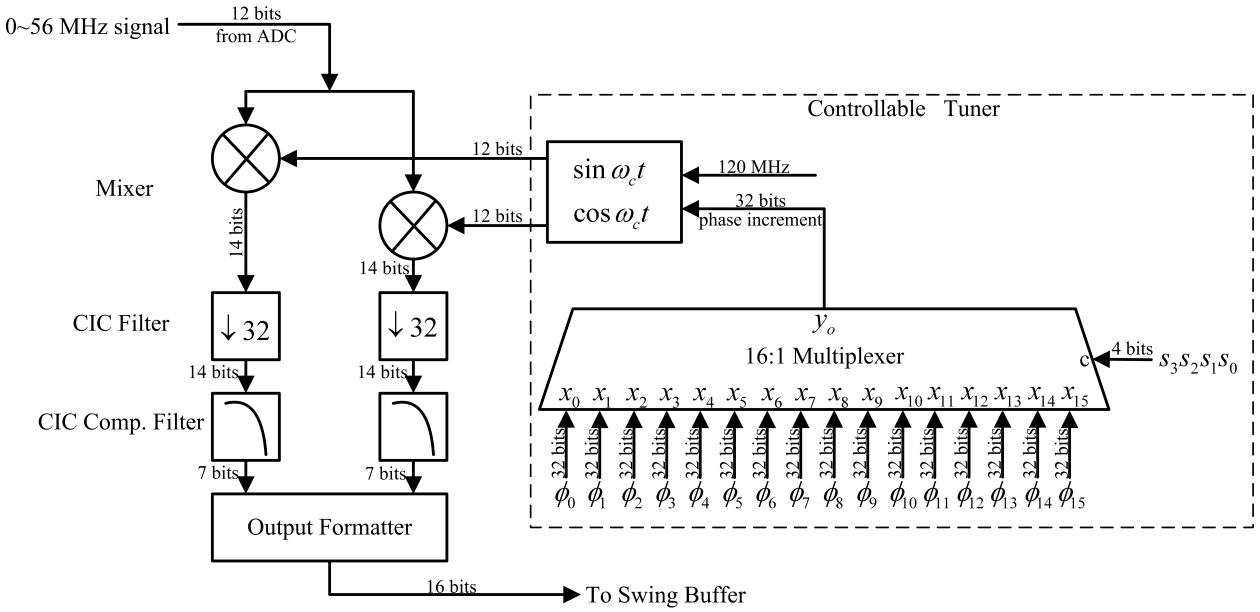


Figure 3: Block diagram of the direct down conversion receiver FPGA firmware. The input signals ϕ_0 to ϕ_{15} denote the phase increment values associated with the 16 possible center frequencies. The 4-bit control $s_3 s_2 s_1 s_0$ is the value of the state register, which is determined using push-button switches as explained in the text.

CIC filter method, and Figure 4(b) is the ideal response of the multirate FIR filter method. Both Table 3 and Figure 4 show that we can implement the CIC filter scheme with better performance and less resource utilization in FPGA when the decimation factor is 32. We also tried to implement a multirate FIR filter with the same performance as that of the CIC filter and CIC Compensation filter, but its order would be up to 7920 and could not be implemented in Stratix II EP2S60F1020C3 FPGA. Therefore, we selected the CIC filter approach for this system.

The passband of the CIC filter is not flat. This issue can be alleviated using a compensation filter [4], whose coefficients are generated automatically by the Altera® CIC Compiler MegaCore function and can be loaded to the Altera® FIR Compiler MegaCore function for the implementation. Figure 4(a) gives the ideal frequency responses of the CIC filter and CIC Compensation filter which are designed for the FPGA implementation in this report. In this case, the combination of the desired CIC filter and CIC Compensation filter provides very flat magnitude response between -1.75 MHz and $+1.75$ MHz at baseband.

4 Demonstration

To test and verify the receiver performance, we have done the following three experiments. For each one, we transferred 8000 packets of length 734 samples/packet (1468 bytes per packet [2]) for a total of $5,872,000$ samples (about 1.57 s of acquisition).

The first experiment is a "glitch" test to validate data integrity. In the test, channel 0 is selected; that is, the center frequency of the receiver is 1.75 MHz. Using a 1.7125 MHz tone with amplitude of 622 mV_{pp} as the input signal, there are 100 samples per period at output of the receiver. A MATLAB/Octave program (see listing 3 in Appendix C) is used to read the data from the file generated by the Python capture code (see listing 2 in Appendix B), and generates an "eye diagram" of the output. Figures 5(a) and 5(b) are the eye diagrams for 200 periods. Note that no errors or discontinuities are visible.

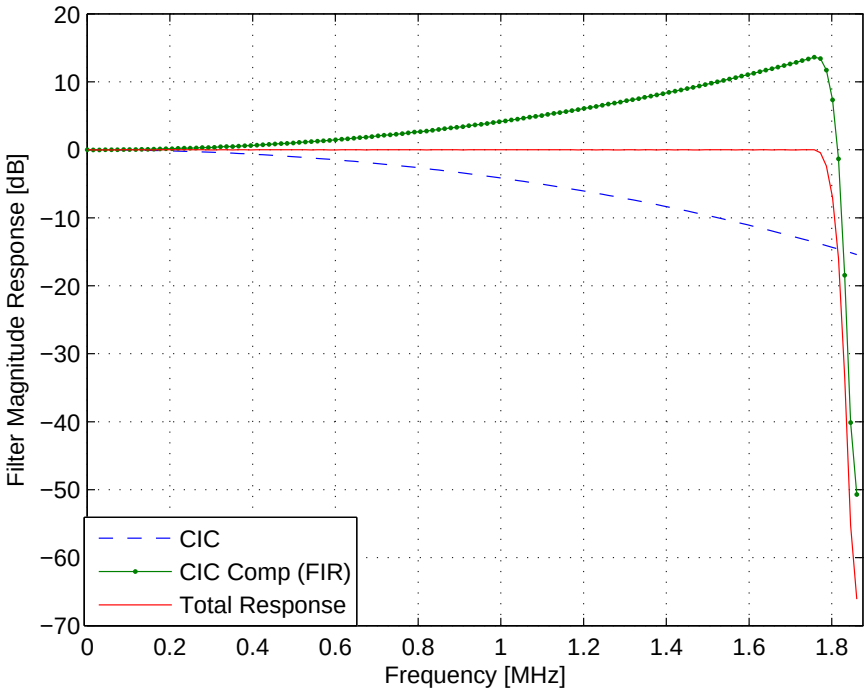
In the second experiment, DDS-synthesized white noise having 10 MHz (low pass) bandwidth and 210 mV_{rms} amplitude is used as the input signal. The output spectrum is then averaged to trace out the frequency response of the receiver. Since output data are 7-bit baseband "I" and 7-bit baseband "Q", the quantization noise density is about 44 dB below the input noise power density. For the sampling rate of 3.75 MSPS and using a 2048-point FFT, the frequency resolution is about 1.8 kHz/bin. By selecting 1.75 MHz center frequency and averaging 2867 FFTs, the computed PSD of the receiver is plotted in Figure 6. Note that the receiver response is approximately flat in the passband between -1.75 MHz and

Table 2: Center frequency selection scheme.

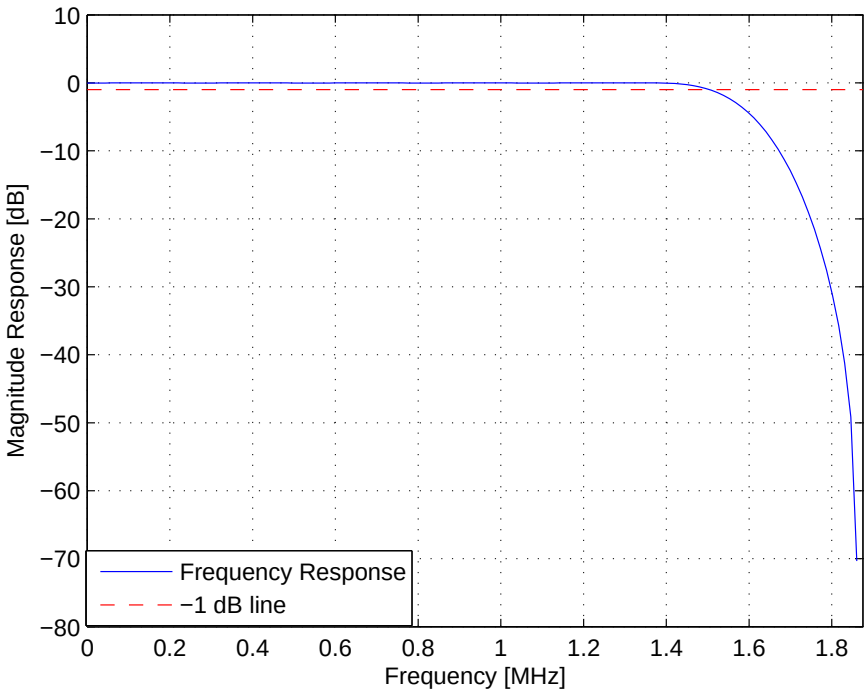
Chan. No.	State Register				Center Freq. (MHz)	1 dB Band Range (MHz)	3 dB Band Range (MHz)	7-Segment Display U12
	s_3	s_2	s_1	s_0				
0	0	0	0	0	1.75	0 – 3.50	0 – 3.542	0
1	0	0	0	1	5.25	3.50 – 7.00	3.458 – 7.042	1
2	0	0	1	0	8.75	7.00 – 10.50	6.958 – 10.542	2
3	0	0	1	1	12.25	10.50 – 14.00	10.458 – 14.042	3
4	0	1	0	0	15.75	14.00 – 17.50	13.958 – 17.542	4
5	0	1	0	1	19.25	17.50 – 21.00	17.458 – 21.042	5
6	0	1	1	0	22.75	21.00 – 24.50	20.958 – 24.542	6
7	0	1	1	1	26.25	24.50 – 28.00	24.458 – 28.042	7
8	1	0	0	0	29.75	28.00 – 31.50	27.958 – 31.542	8
9	1	0	0	1	33.25	31.50 – 35.00	31.458 – 35.042	9
10	1	0	1	0	36.75	35.00 – 38.50	34.958 – 38.542	A
11	1	0	1	1	40.25	38.50 – 42.00	38.458 – 42.042	b
12	1	1	0	0	43.75	42.00 – 45.50	41.958 – 45.542	c
13	1	1	0	1	47.25	45.50 – 49.00	45.458 – 49.042	d
14	1	1	1	0	50.75	49.00 – 52.50	48.958 – 52.542	E
15	1	1	1	1	54.25	52.50 – 56.00	52.458 – 56.042	F

Table 3: FPGA resource utilization with different filter structures. Percentages indicate fraction of total available resources used.

		Multirate FIR Filter	CIC and Comp. Filters
Logic Cells	Combination ALUTS	29784/48352 (63%)	17835/48352 (37%)
	Dedicated Logic Registers	35391/48352 (73%)	18860/48352 (39%)
	Total	98%	50%
Total Block Memory Bits		1973728/2544192 (78%)	1973952/2544192 (78%)



(a) Ideal response of the CIC filter scheme.

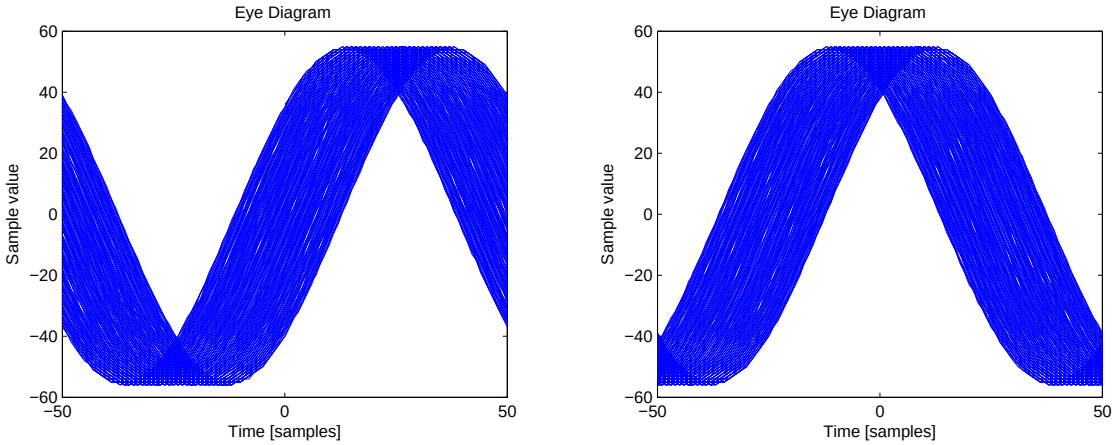


(b) Ideal response of the multirate FIR filter scheme.

Figure 4: Comparison of ideal filter frequency responses.

+1.75 MHz, and drops off sharply thereafter.

The results of the third experiment are shown in Figure 7. In this test, we tune the receiver to 15.75 MHz center frequency and average 2867 2048-point FFTs to compute the PSD. The tone in Figure 7(a) is 14.00 MHz with amplitude of 676 mV_{pp} and thus is downconverted to -1.75 MHz at baseband; the tone in Figure 7(b) is 15.25 MHz with amplitude of 709 mV_{pp} and thus is downconverted to zero at baseband; the tone in Figure 7(c) is 17.50 MHz with the amplitude of 734 mV_{pp} and thus is downconverted to +1.75 MHz at baseband.



(a) Eye diagram for baseband "I". (b) Eye diagram for baseband "Q".

Figure 5: Eye diagram for baseband signals. The "smearing" is due to the slight difference in clock rate between the signal generator and the development board, for they are not phase-locked.

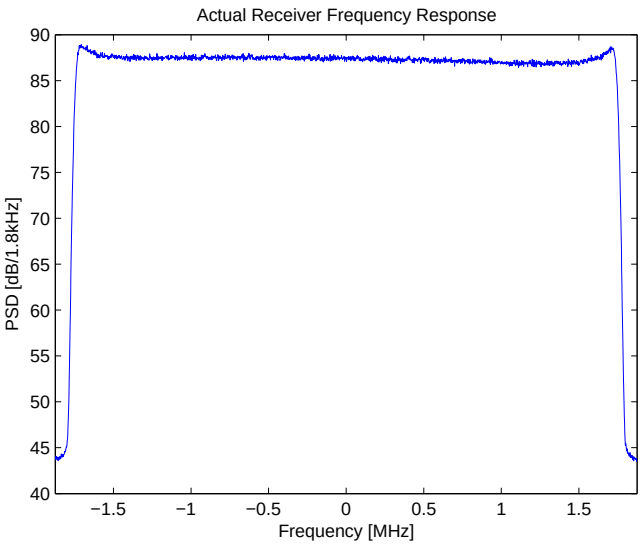
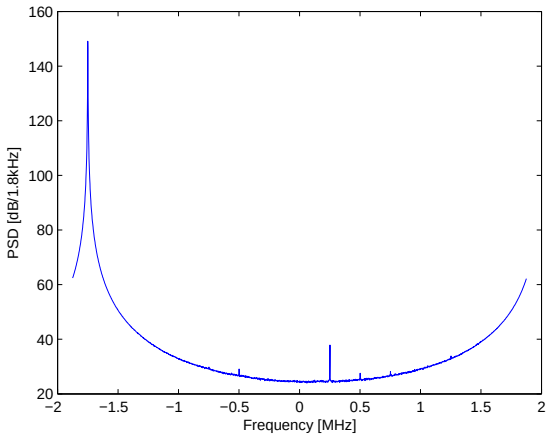
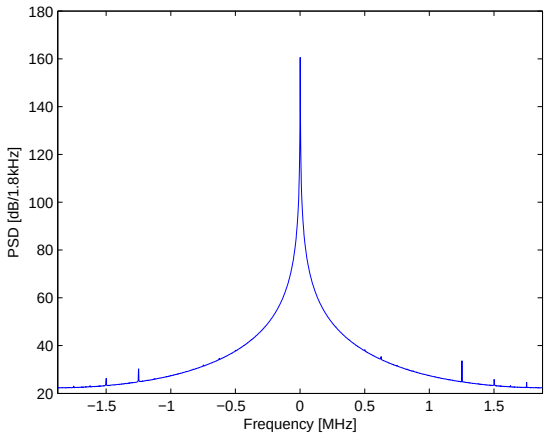


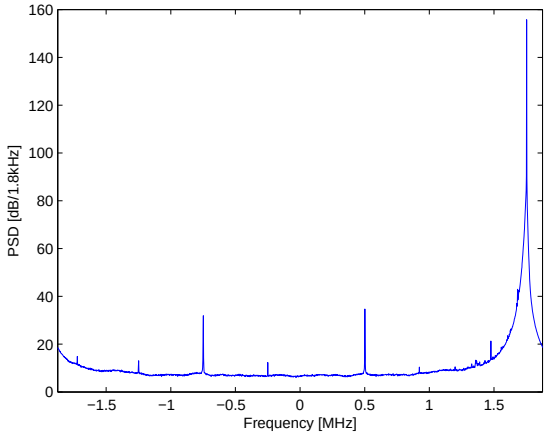
Figure 6: Computed PSD of receiver output when input is white noise. The slight slope across the passband is due to the input noise. The reason for the peaks on the edges of the passband is due to the finite length of the filter that performs CIC Compensation.



(a) 14.00 MHz tone.



(b) 15.75 MHz tone.



(c) 17.50 MHz tone.

Figure 7: Computed PSD of receiver with different tones.

References

[1] Altera Corporation, *Stratix II DSP Development Board Reference Manual*, ver. 6.0.1, August 2006.

[2] Q. Liu and S. W. Ellingson, "Method of High-Speed Data Acquisition and Continuous Data Transfer using Altera® Stratix® II EP2S60 DSP Development Board," Internal Tech. Rep., Virginia Polytechnic Institute and State University, VA, June 14 2009. [online] http://www.ece.vt.edu/swe/eta.

[3] Altera Corporation, *CIC MegaCore Function User Guide*, ver. 8.0, May 2008.

[4] Altera Corporation, *Understanding CIC Compensation Filters*, ver. 1.0, April 2007.

A FPGA Firmware

The FPGA firmware is written in Verilog HDL. The Altera® NCO MegaCore® function is used to generate the local oscillator, the Altera® LMP_MULT MegaCore® function is used for mixers, the Altera® CIC Compiler MegaCore® function is used to implement the CIC filters, and the Altera® FIR Compiler MegaCore® function is used to implement the CIC Compensation filters.

The following firmware is for ADC-A. The ADC-B firmware is identical, except that the ADC pin assignments are changed, and the line "assign ssd = {ssdreg, 8'b1110_1000};" should be replaced with "assign ssd = {ssdreg, 8'b1110_0000};" to make the seven-segment display U13 show "b" instead of "A".

Listing 1: receiver.v

```
// ============================================================
// Developed by Q. Liu <qianliu@vt.edu> in 2009
// ============================================================
module receiver(clk120M, clk3750K, a2dc, fctrl, datu, ssd);

    // ------------- Port Definition ---------------
    input          clk120M;
    input          clk3750K;
    input  [11:0] a2dc;
    input  [ 3:0] fctrl;
    output [15:0] datu;
    output [15:0] ssd;

    // ------------ Intermediate Variable -----------
    reg    [31:0] phi_inc;
    reg    [11:0] fcos, fsin; // NCO output
    wire   [23:0] xi1, xq1;   // I24+Q24 @ 120MSPS
    wire   [33:0] xi2, xq2;   // I34+Q34 @ 3.75MSPS
    wire   [ 6:0] xi3, xq3;   // I7+Q7   @ 3.75MSPS

    reg    [ 3:0] swfreq;
    reg    [ 7:0] ssdreg;

    // ------------------------------------------------
    // Frequency Control
    // ------------------------------------------------
    parameter FREQ0 = 4'h0,
              FREQ1 = 4'h1,
              FREQ2 = 4'h2,
              FREQ3 = 4'h3,
              FREQ4 = 4'h4,
              FREQ5 = 4'h5,
              FREQ6 = 4'h6,
              FREQ7 = 4'h7,
              FREQ8 = 4'h8,
              FREQ9 = 4'h9,
              FREQa = 4'hA,
              FREQb = 4'hB,
              FREQc = 4'hC,
              FREQd = 4'hD,
              FREQe = 4'hE,
              FREQf = 4'hF;

    always @(negedge fctrl[0]) begin
        swfreq[0] <= ~swfreq[0];
    end
```

```
always @(negedge fctrl[1]) begin
    swfreq[1] <= ~swfreq[1];
end

always @(negedge fctrl[2]) begin
    swfreq[2] <= ~swfreq[2];
end

always @(negedge fctrl[3]) begin
    swfreq[3] <= ~swfreq[3];
end

always @(posedge clk120M) begin
    case(fctrl)
        FREQ0: begin
            phi_inc <= 32'd62634940;        // 1.75MHz
            ssdreg  <= 8'b1000_0001;        // "0"
        end
        FREQ1: begin
            phi_inc <= 32'd187904819;       // 5.25MHz
            ssdreg  <= 8'b1100_1111;        // "1"
        end
        FREQ2: begin
            phi_inc <= 32'd313174699;       // 8.75MHz
            ssdreg  <= 8'b1001_0010;        // "2"
        end
        FREQ3: begin
            phi_inc <= 32'd438444578;       // 12.25MHz
            ssdreg  <= 8'b1000_0110;        // "3"
        end
        FREQ4: begin
            phi_inc <= 32'd563714458;       // 15.75MHz
            ssdreg  <= 8'b1100_1100;        // "4"
        end
        FREQ5: begin
            phi_inc <= 32'd688984337;       // 19.25MHz
            ssdreg  <= 8'b1010_0100;        // "5"
        end
        FREQ6: begin
            phi_inc <= 32'd814254217;       // 22.75MHz
            ssdreg  <= 8'b1010_0000;        // "6"
        end
        FREQ7: begin
            phi_inc <= 32'd939524096;       // 26.25MHz
            ssdreg  <= 8'b1000_1111;        // "7"
        end
```

```
        FREQ8: begin
            phi_inc <= 32'd1064793975;      // 29.75MHz
            ssdreg  <= 8'b1000_0000;        // "8"
        end
        FREQ9: begin
            phi_inc <= 32'd1190063855;      // 33.25MHz
            ssdreg  <= 8'b1000_1100;        // "9"
        end
        FREQa: begin
            phi_inc <= 32'd1315333734;      // 36.75MHz
            ssdreg  <= 8'b1000_1000;        // "A"
        end
        FREQb: begin
            phi_inc <= 32'd1440603614;      // 40.25MHz
            ssdreg  <= 8'b1110_0000;        // "b"
        end
        FREQc: begin
            phi_inc <= 32'd1565873493;      // 43.75MHz
            ssdreg  <= 8'b1111_0010;        // "c"
        end
        FREQd: begin
            phi_inc <= 32'd1691143373;      // 47.25MHz
            ssdreg  <= 8'b1100_0010;        // "d"
        end
        FREQe: begin
            phi_inc <= 32'd1816413252;      // 50.75MHz
            ssdreg  <= 8'b1011_0000;        // "E"
        end
        FREQf: begin
            phi_inc <= 32'd1941683132;      // 54.25MHz
            ssdreg  <= 8'b1011_1000;        // "F"
        end
    endcase
end

// ------------------------------------------------------------
// Tuner (12 bits 2's complement output)
// ------------------------------------------------------------
nco0 tuner0 (
              .phi_inc_i (phi_inc),
              .clk       (clk120M),
              .reset_n   (1'b1),
              .clken     (1'b1),
              .fsin_o    (fsin),
              .fcos_o    (fcos),
              .out_valid ()
             );
```

```
// ------------------------------------------------------
// Complex Mixer
// ------------------------------------------------------
mixer baseI (
             .clock  (clk120M),
             .dataa  (a2dc),
             .datab  (fsin),
             .result (xi1)
            );
mixer baseQ (
             .clock  (clk120M),
             .dataa  (a2dc),
             .datab  (fcos),
             .result (xq1)
            );

// ------------------------------------------------------
// Decimation CIC Filter @ 120MHz
// ------------------------------------------------------
cicfir dnfirI (
                .clk       (clk120M),
                .clken     (1'b1),
                .reset_n   (1'b1),
                .in_data   (xi1[22:9]),
                .in_valid  (1'b1),
                .out_ready (1'b1),
                .in_error  (2'b00),
                .out_data  (xi2),
                .in_ready  (),
                .out_valid (),
                .out_error ()
               );
cicfir dnfirQ (
                .clk       (clk120M),
                .clken     (1'b1),
                .reset_n   (1'b1),
                .in_data   (xq1[22:9]),
                .in_valid  (1'b1),
                .out_ready (1'b1),
                .in_error  (2'b00),
                .out_data  (xq2),
                .in_ready  (),
                .out_valid (),
                .out_error ()
               );
```

```
    // ------------------------------------------------------
    // Compensation FIR Filter @ 3.75MHz
    // ------------------------------------------------------
    compfir firI (
                    .clk              (clk3750K),
                    .reset_n          (1'b1),
                    .ast_sink_data    (xi2[32:19]),
                    .ast_sink_valid   (1'b1),
                    .ast_source_ready (1'b1),
                    .ast_sink_error   (2'b00),
                    .ast_source_data  (xi3),
                    .ast_sink_ready   (),
                    .ast_source_valid (),
                    .ast_source_error ()
                );
    compfir firQ (
                    .clk              (clk3750K),
                    .reset_n          (1'b1),
                    .ast_sink_data    (xq2[32:19]),
                    .ast_sink_valid   (1'b1),
                    .ast_source_ready (1'b1),
                    .ast_sink_error   (2'b00),
                    .ast_source_data  (xq3),
                    .ast_sink_ready   (),
                    .ast_source_valid (),
                    .ast_source_error ()
                );


    // Data Format Conversion
    assign datu[15]   =  1'b0;
    assign datu[14]   = ~xq3[6];
    assign datu[13:8] =  xq3[5:0];
    assign datu[7]    =  1'b0;
    assign datu[6]    = ~xi3[6];
    assign datu[5:0]  =  xi3[5:0];

    // Seven-Segment Display
    assign ssd = {ssdreg, 8'b1110_1000};

endmodule
```

B Data Acquisition Script

Listing 2 is Python code that runs on PC for the data capture. Users can also use other languages (i.e., C/C++, etc.) to write their own code to receive the UDP packets.

Listing 2: server.py

```
# ============================================================
# Developed by Q. Liu <qianliu@vt.edu> in 2009
# ============================================================
from socket import *
from ctypes import *
import binascii

# Set the socket parameters
host = '192.168.1.213'
port = 1739
addr = (host, port)
reclen = 1468

# Set the packet parameter
PacketNum = 8000

# Set the socket parameter
BufferSize = reclen*PacketNum

# Create the socket
UDPSock = socket(AF_INET, SOCK_DGRAM)

# Set options
UDPSock.setsockopt(SOL_SOCKET, SO_RCVBUF, BufferSize)
UDPSock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

# Create a binary file
filename = 'test.dat'
FILE0 = open(filename, 'wb', 0)

# ================ Receive messages ================= #
for j in range(0, looptime*PacketNum):
    data,address = UDPSock.recvfrom(reclen, 0)
    FILE0.write(data)

UDPSock.close()
FILE0.close()

# ================= Post-processing ================= #
FILE1 = open('test.dat', 'rb');
data_bytes = FILE1.read(reclen*PacketNum*looptime)
FILE1.close()

data_hex = binascii.hexlify(data_bytes)
FILE2 = open('test.txt','w')
FILE2.write(data_string)
```

```
FILE2.close()

print('Complete!')
```

C Data Analysis Scripts

These scripts read the file generated by the Python capture code (see Appendix B), and then process the data according to the application objectives. Both can run under either MATLAB 7.1 or Octave 3.0.1. The first script (listing 3) generates two eye diagrams. The second script (listing 4) computes the averaged PSD using a 2048-point FFT.

Listing 3: glitch.m

```
% ========================================================
% Developed by Q. Liu <qianliu@vt.edu> in 2009
% ========================================================
clear all;
close all;

disp(‘reading...’);
fid = fopen(‘test.txt’, ‘r’); % open the file
data = fscanf(fid, ‘%2X’); % array of 4-byte words (2 samples each)
fclose(fid);

disp(‘analyzing...’);

L = max(size(data));
index = 1 : L;
i0 = data(find(mod(index,4)==1)); % demultiplex into 1-byte values
q0 = data(find(mod(index,4)==2));
i1 = data(find(mod(index,4)==3));
q1 = data(find(mod(index,4)==0));

num = 1 : L/2;
i = zeros(L/2, 1);
i(find(mod(num,2)==1)) = i0; % remultiplex into a single pair of
i(find(mod(num,2)==0)) = i1; % arrays, representing ‘‘I’’ and ‘‘Q’’
q = zeros(L/2, 1);           % respectively
q(find(mod(num,2)==1)) = q0;
q(find(mod(num,2)==0)) = q1;

disp('plot eye diagram...');
lp = 100;
times = 200;
eyediagram(i(1:lp*times), lp, lp);
xlabel('Time [samples]');
ylabel('Sample value');
eyediagram(q(1:lp*times), lp, lp);
xlabel('Time [samples]');
ylabel('Sample value');

return;
```

Listing 4: spectrum.m

```
% ============================================================
% Developed by S. W. Ellingson <ellingson@vt.edu> in 2009
% Modified by Q. Liu
% ============================================================
clear all;
close all;

disp('reading...');
fid = fopen('test.txt', 'r'); % open the file
data = fscanf(fid, '%2X');
fclose(fid);

disp('analyzing...');

LFFT = 2048;

L = max(size(data));
index = 1 : L;
i0 = data(find(mod(index,4)==1));
q0 = data(find(mod(index,4)==2));
i1 = data(find(mod(index,4)==3));
q1 = data(find(mod(index,4)==0));

num = 1 : L/2;
i = zeros(L/2, 1);
i(find(mod(num,2)==1)) = i0;
i(find(mod(num,2)==0)) = i1;
q = zeros(L/2, 1);
q(find(mod(num,2)==1)) = q0;
q(find(mod(num,2)==0)) = q1;

block = floor((L/2)/LFFT);

r = i(1:LFFT*block) + sqrt(-1)*q(1:LFFT*block);
r = r - mean(r); % remove DC
Lr = max(size(r));

SY = zeros(LFFT,1);
QE = 1.761 + 6.0206*7;
w = -1.875 : 3.75/(LFFT-1) : 1.875;
m=0;
for l = 1:LFFT:Lr,
  m=m+1;
  y = r(l:l+LFFT-1);
  Y = fftshift(fft(y));
  PY = abs(Y).^2;
```

```
  SY = SY+PY;
  plot(w, 20*log10(SY/m)-QE);
  drawnow;
end
xlim([-1.875 1.875]);
xlabel('Frequency [MHz]');
ylabel('PSD [dB/1.8kHz]');

return;
```