# Implementation of a Single FFT Processor

Grant Hampson

July 3, 2002

## Introduction

This document describes a FPGA implementation and simulation of the FFT component of the IIP Radiometer RFI processor described in [1]. A possible implementation of this FFT component has been previously described in [2]. Here a single FFT processor will be implemented and tested before proceeding to a larger design consisting of many such processors.

This document is broken into three separate sections. The first section presents simulations of the Altera FFT Megacore, with special emphasis on its floating point outputs. Secondly, synthesis results for an Altera EP20K100EQC208-1 FPGA will be presented. Finally, implementation results of the FFT processor will be illustrated.

## 1 Simulation of the FFT Core Outputs

The FFT core [3] has been simulated previously in [2] and it was noted that the FFT has a floating point output, i.e., a mantissa and exponent are output. (The data inputs are fixed point.) The exponent varies as data is being processed, because the FFT core is trying to maintain maximum dynamic range using block floating-point techniques. The exponent is identical for all outputs once the processing is finished. The major components of the Altera FFT example processor are shown in Figure 1.

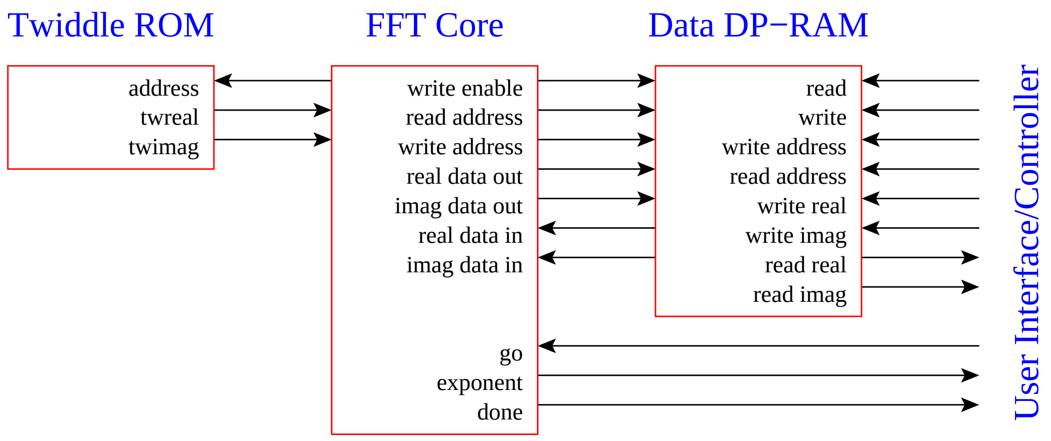


Figure 1: The Altera FFT example implementation contains the above ROM, processor and dual port RAM. The user is required to build a controller.

Given that floating point libraries are not freely available (for post-FFT processing), and given that a 16-bit output resolution will probably provide sufficient dynamic range, then a fixed point representation is desired. Consequently, additional logic is required to decode the exponent and scale the output accordingly.

Firstly, consider the dynamic range requirements of the FFT. Figure 2 illustrates the dynamic range requirements of an FFT using a sinusoid of varying amplitudes as a stimulus. The largest FFT bin requires over 24-bits to represent it in a fixed point format. This is obviously beyond the 16-bit requirement previously imposed and some truncation is required. The hardware used to perform this truncation (which results in normalization) is shown in Figure 3. The hardware is relatively simple, the main component being a bi-directional shifter. Depending on the sign and magnitude of the exponent the mantissa is shifted left or right. If the input is shifted right a truncation occurs. If the input is shifted left zeros are inserted. Appendix A lists the AHDL source code.

Figure 4 illustrates the results of the normalization hardware. Figure 4(a) illustrates some example exponents from the FFT core. The dynamic range of the exponent is not large. Figure 4(b) illustrates the results of the FFT output normalization. The fixed point output is simply a scaled version of the floating point output. Figure 4(c) shows the errors introduced due to the scaling. When the FFT output is large, the amount of error is large.

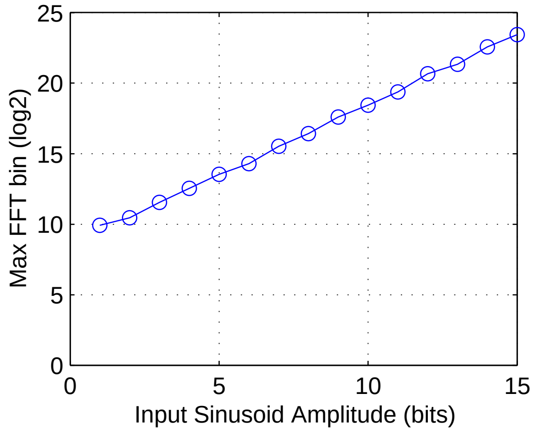


Figure 2: The magnitude of largest FFT bin for an input sinusoid of varying amplitude.

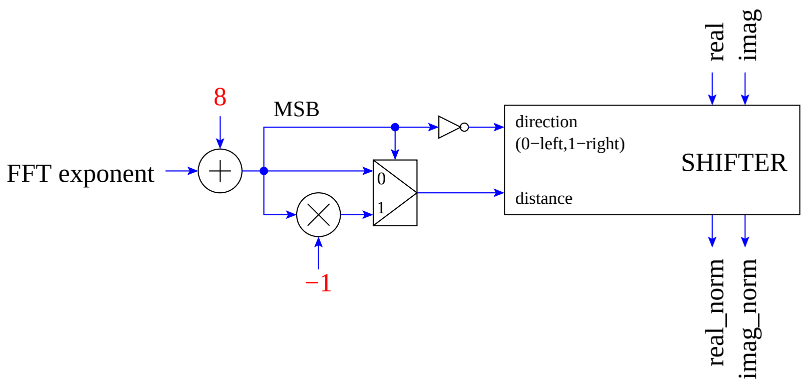
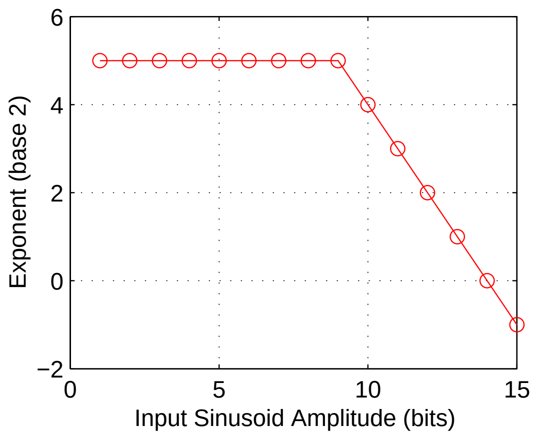
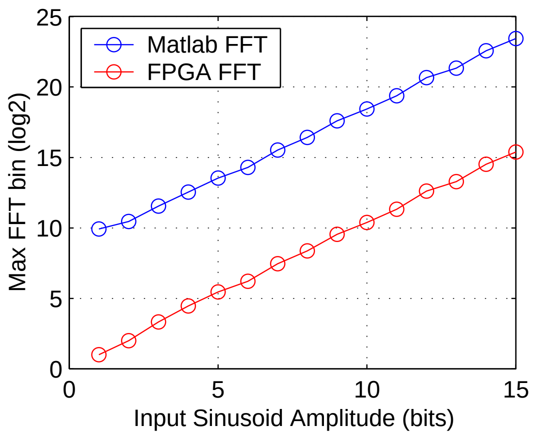


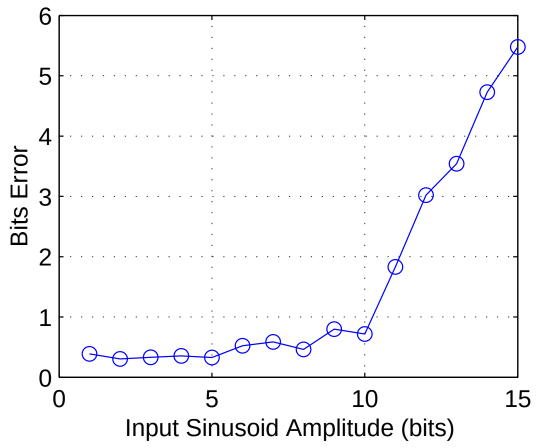
Figure 3: Extra hardware required to convert a floating point format into fixed point.



(a) FFT Core Exponent



(b) FFT Amplitude



(c) Truncation Errors

Figure 4: Results from the comparison of a floating point FFT output to a fixed point FFT output. (a) The Altera FFT core exponent output for various input sinusoids. When the input sinusoid is large, the result is scaled down. When the result fits in data width no scaling occurs. (b) Largest FFT output for the Matlab FFT (Figure 2) and the Altera FFT core result with the added floating point to fixed point conversion. (c) The resulting bits of error between a scaled version of the Matlab FFT and the Altera FFT core. When the FFT output is large the exponent shift truncates bits off the result increasing the error. A majority of the error is within $\pm 1$ when the FFT output fits in the data width.

## 2 Synthesis Results

The FFT circuitry shown in Figure 1 requires a controlling state machine. Figure 5 illustrates a state machine which writes data points to the FFT memory, starts the FFT core and waits for completion, and then reads the result (complicated by a 2 clock read latency, FIFO write enables and marker data.) Each FFT block contains 1024 complex samples as well as a marker equaling -32768. Hence for each FFT computed 1025 samples are written to the capture FIFO [4]. Appendix B contains the source to this state machine.

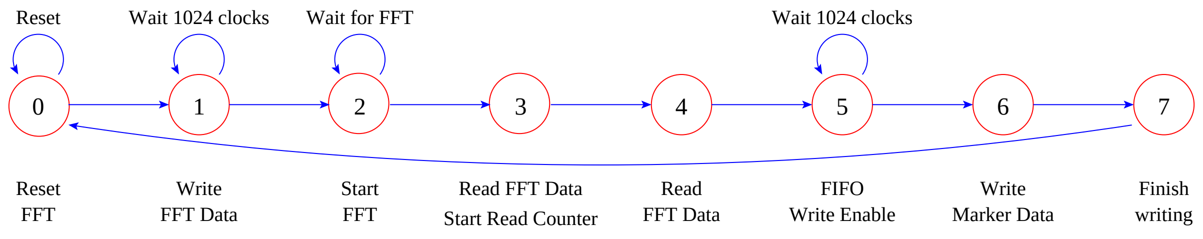


Figure 5: State machine which controls a single FFT processor.

The single FFT processor was synthesized for various input data widths (10 to 16-bits) and a twiddle factor width of 8-bits. The results of the synthesis are summarized in Table 1. The FFT processor is capable of running at input resolutions up to 14-bits at the clock rate of 100MHz (beyond this processing errors occur.)

Table 2 lists the *uniqueness* of the twiddle factors for a 1024-point FFT. As the number of bits increases, each twiddle factor becomes more unique and the number of duplicates decreases. For length 1024 FFTs twiddle widths beyond 10-bits achieve no gain in FFT accuracy. The results shown here use 8-bit twiddles, which provides adequate resolution.

The target FPGA for the 8 FFT processor system [2] would use an FPGA 2.7 times (270%) the size of this FPGA. Each FPGA in the 8-FFT system would contain 4-FFT processors. So each FFT processor could consume 270%/4 = 67% of the current FPGA. Given that there are other overheads around the FFT processors (multiplexers, controllers, etc.) and a FPGA fill ratio of 80% is sensible, then the size of each processor should be approximately 80% of 67%=54%. From Table 1 each FFT processor could have up to 14-bit data inputs and 8-bit twiddle factors.

Table 1: Synthesis results for a length 1024 FFT using 8-bit twiddle factors. The target FPGA is a EP20K100EQC208-1.

| Data Width | Logic Elements (% of max) | Memory Bits (% of max) | Speed |
|---|---|---|---|
| 10 | 1779 (42%) | 24576 (46%) | 112 MHz |
| 11 | 1905 (45%) | 26624 (50%) | 108 MHz |
| 12 | 2026 (48%) | 28672 (53%) | 111 MHz |
| 13 | 2149 (51%) | 30720 (57%) | 110 MHz |
| 14 | 2272 (54%) | 32768 (61%) | 109 MHz |
| 15 | 2392 (57%) | 34816 (65%) | 101 MHz |
| 16 | 2513 (60%) | 36864 (69%) | 96 MHz |

Table 2: Twiddle factor characteristics for a length 1024 FFT. A sinusoid is quantised to the twiddle width and a histogram calculates the unique number of integers and the mean number of duplicates.

| Twiddle Width | % of unique integers | Mean number of duplicate integers |
|---|---|---|
| 1 | 0.292969 | 341.333333 |
| 2 | 0.488281 | 204.800000 |
| 3 | 0.878906 | 113.777778 |
| 4 | 1.660156 | 60.235294 |
| 5 | 3.222656 | 31.030303 |
| 6 | 6.347656 | 15.753846 |
| 7 | 12.597656 | 7.937984 |
| 8 | 25.097656 | 3.984436 |
| 9 | 39.550781 | 2.528395 |
| 10 | 44.921875 | 2.226087 |
| 11 | 44.921875 | 2.226087 |
| 12 | 44.824219 | 2.230937 |
| 13 | 44.824219 | 2.230937 |
| 14 | 44.824219 | 2.230937 |
| 15 | 44.824219 | 2.230937 |
| 16 | 44.824219 | 2.230937 |

## 3 FFT Implementation Results

The FFT processor can be implemented on the same hardware as the preliminary Asynchronous Pulse Blanker [5], as shown in Figure 6. This board is also equipped with a Rabbit processor [6] which will be later useful for controlling FFT core (possible scaling) and settings for the window function and integration.

Figure 7 illustrates the first test conducted with the 14-bit FFT. Two data sets were collected here. The first was to integrate the FFT output 25600 times (shown in blue.) The second was to collect enough raw data (16-bit) from the digital IF processor for 25600 1024-point FFTs. Matlab was used to perform these integrations (with and without a Bartlett window.) The pass band of the spectrums have almost identical results. The use of a Bartlett window removes the FFT edge effects and makes it possible to see deeper into the spectrum. The origin of the pass-band ripple has yet to be determined.

Figure 8(a) illustrates the raw output of the FFT processor. The FFT end marker value of -32768 can be clearly seen. For a single FFT processor it is possible to process approximately 14% of the input data. (Using 8 FFT processors it will be possible to process 112%, or all of the input data.)

The FFT data requires a `fftshift()` to obtain the correct spectrum frequencies. Figure 8(b&c) illustrates a single FFT where a radar pulse is absent and present [7]. The radar pulse is clearly 30dB above the normal pass band level, as indicated also in [7]. This test indicates that the floating point to fixed point conversion hardware shown in Figure 3 is functioning correctly.

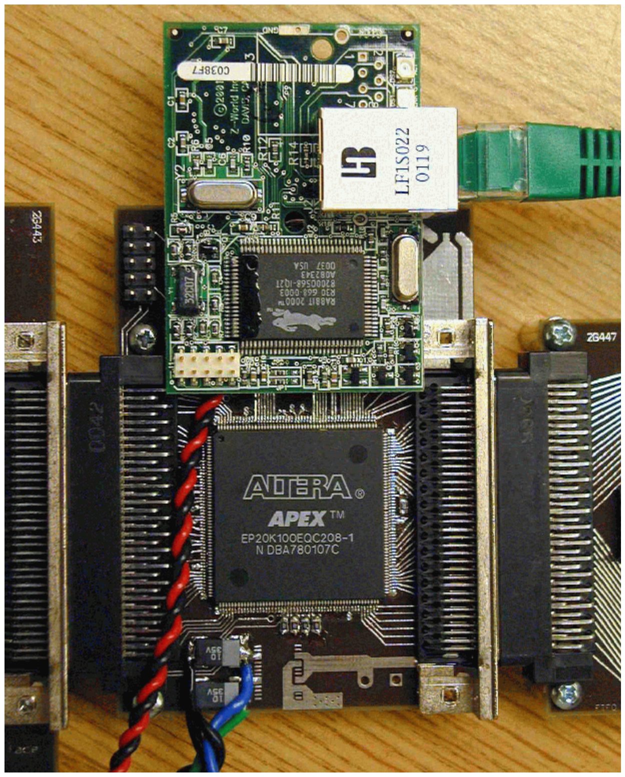


Figure 6: Photograph of the Rabbit processor and the FPGA on which the single FFT processor is implemented. (Digital IF to the left, FIFO capture card to right.)

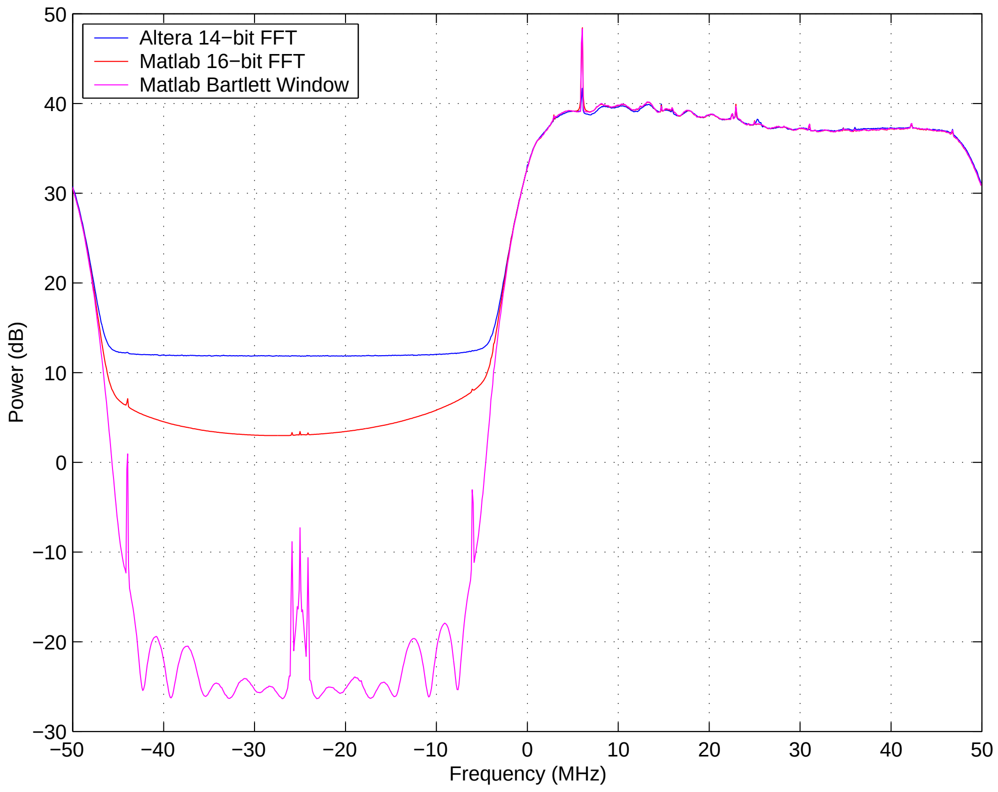
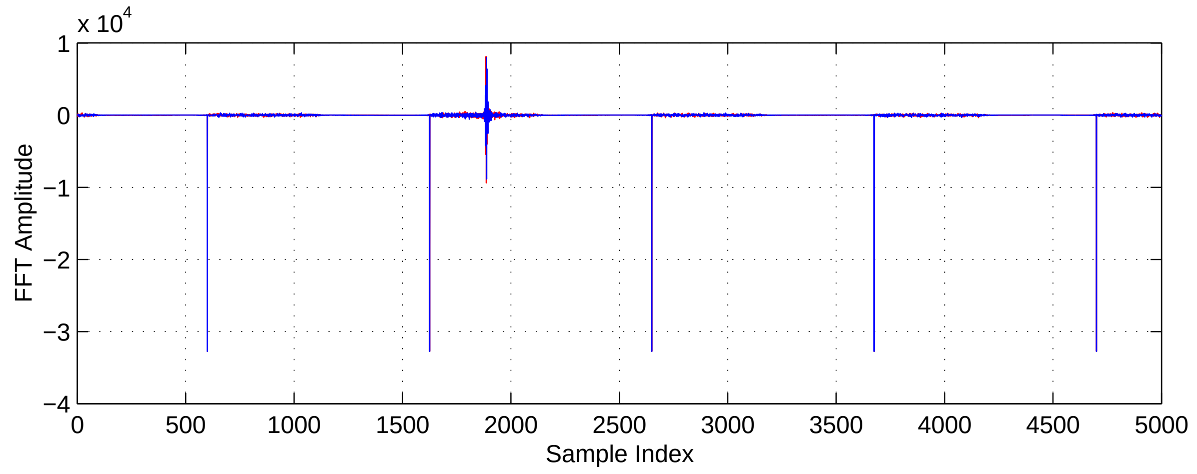
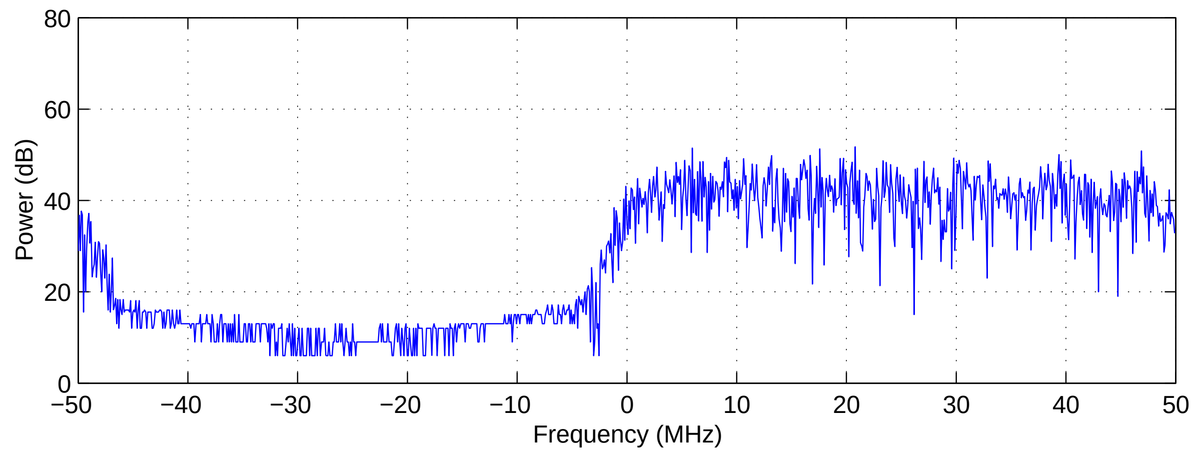


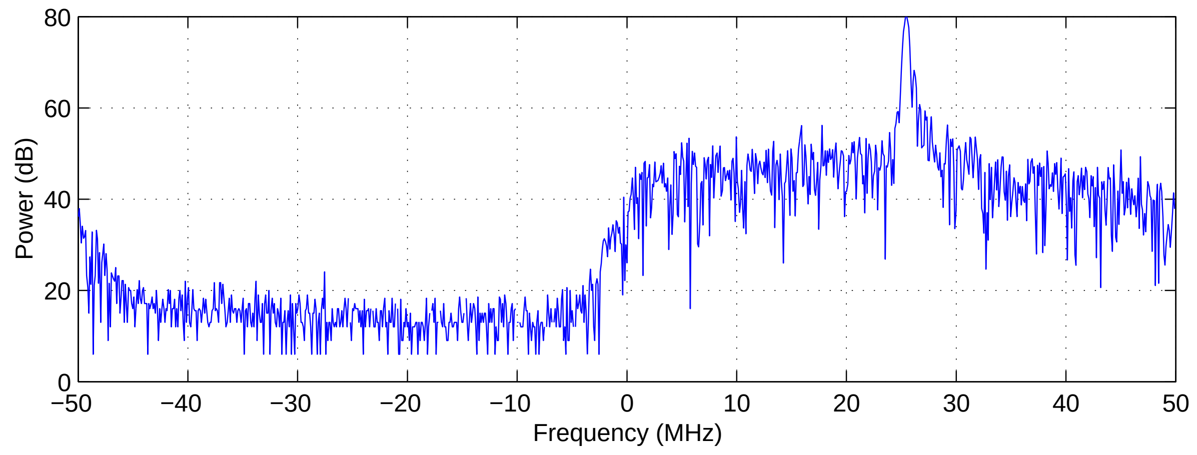
Figure 7: Results from the 14-bit FFT processor (no window), and also from capturing 16-bit data and calculating the FFT in Matlab. 25600 integrations are calculated.



(a) Data from the Single FFT Processor



(b) Single FFT containing no Radar Pulses



(c) Single FFT containing a Radar Pulse

Figure 8: (a) Raw output from the FFT processor with FFT marker data. (b) Power spectrum of a FFT containing no radar pulses. (c) A spectrum of a FFT containing a radar pulse. No spectral windows precede the FFT.

## Summary and Conclusions

This report has shown an example implementation of a single FFT processor. Simulations of FFTs in Matlab reveal that the dynamic range of the FFT output is larger than the input dynamic range. Consequently, the FFT core provided by Altera has a floating point output consisting of a mantissa and exponent. Given that we don't have any floating point libraries for post-processing the FFT data, additional hardware is constructed to convert the result to fixed point.

The Altera FFT core also requires a controller to make the FFT core functional. A state machine which can control a single FFT core was designed, implemented and tested. The single FFT design was synthesized for a $150 Altera FPGA (EP20K100EQC208-1). It was found that the maximum size FFT possible has a 14-bit data width and 8-bit twiddle factors. The single FFT design is capable of processing 14% of the data.

Implementation results from the single FFT processor were also presented. Integration tests indicate that the FFT core is producing similar results to that of Matlab. Additionally, the processor was tested on real data containing radar pulses. The design also passed these tests.

It was also calculated that it should be possible to implement 8 FFT processors on two EP20K300EQC240-1 Altera FPGAs ($512 each) with the same resolution as the single FFT processor. This initial implementation indicates that it should be possible to process the full 100MSPS.

## References

[1] S. W. Ellingson, "Design Concept for the IIP Radiometer RFI Processor," January 23 2002. http://esl.eng.ohio-state.edu/rfse/iip/rfiproc1.pdf.

[2] G. A. Hampson, "A Possible 100MSPS Altera FPGA FFT Processor," March 12 2002. http://esl.eng.ohio-state.edu/rfse/iip/fftproc.pdf.

[3] *FFT MegaCore Function User Guide*, Altera Corporation, March 2001. http://www.altera.com/literature/ug/fft_ug.pdf.

[4] G. A. Hampson, "A 256k@32-bit Capture Card for the IIP Radiometer," May 10 2002. http://esl.eng.ohio-state.edu/rfse/iip/fifocapture.pdf.

[5] G. A. Hampson, "Implementation of the Asynchronous Pulse Blanker," March 12 2002. http://esl.eng.ohio-state.edu/rfse/iip/apbproc.pdf.

[6] *RCM2200 RabbitCore*, Rabbit Semiconductor. http://www.rabbitsemiconductor.com/products/rcm2200/index.html.

[7] S. W. Ellingson and G. A. Hampson, "On-Air Test of the IIP Receiver Using Observations of an ATC Radar," June 29 2002. http://esl.eng.ohio-state.edu/rfse/iip/test020618.pdf.

# Appendix A: Single FFT AHDL Source Code

```
-- Single FFT Processor Implementation
-- Grant Hampson May 1 2002

INCLUDE "aukfft_fftchipa.inc"; -- include file for Altera FFT processor
INCLUDE "lpm_add_sub.inc";
INCLUDE "lpm_clshift.inc";

PARAMETERS(floatwidth = 4,
           expwidth = floatwidth + 1,
           datawidth = 16,
           twiddlewidth = 16,
           points = 1024,
           addresswidth = log2(points));

SUBDESIGN SingleFFT
(
   sysclk,
   reset,
   go,
   writeaddress[addresswidth..1],
   readaddress[addresswidth..1],
   read,
   write,
   writereal[datawidth..1],
   writeimag[datawidth..1]
      : INPUT;

   readreal[datawidth..1],
   readimag[datawidth..1],
   exponent[expwidth..1],
   done
      : OUTPUT;
)

VARIABLE
   FFTproc : aukfft_fftchipa WITH (floatwidth = 4,
                                   datawidth = 16,
                                   twiddlewidth = 8,
                                   points = 1024);

   exp_add : lpm_add_sub WITH (LPM_WIDTH = expwidth,
                               LPM_REPRESENTATION = "SIGNED",
                               LPM_DIRECTION = "ADD",
                               LPM_ONE_INPUT_IS_CONSTANT = "YES");

   exp_negate : lpm_add_sub WITH (LPM_WIDTH = expwidth,
                                  LPM_REPRESENTATION = "SIGNED",
                                  LPM_DIRECTION = "SUB",
                                  LPM_ONE_INPUT_IS_CONSTANT = "YES");

   out_shift_real,
   out_shift_imag : lpm_clshift WITH (LPM_WIDTH = datawidth,
                                      LPM_WIDTHDIST = expwidth-1,  -- positive number
                                      LPM_SHIFTTYPE = "ARITHMETIC");
   shift_direction,
   shift_distance[(expwidth-1)..0] : NODE;

BEGIN
   FFTproc.sysclk = sysclk; -- connect FFT processor
   FFTproc.reset = reset;
   FFTproc.go = go;
   FFTproc.writeaddress[addresswidth..1] = writeaddress[addresswidth..1];
   FFTproc.readaddress[addresswidth..1] = readaddress[addresswidth..1];
   FFTproc.read = read;
   FFTproc.write = write;
   FFTproc.writereal[datawidth..1] = writereal[datawidth..1];
   FFTproc.writeimag[datawidth..1] = writeimag[datawidth..1];
   done = FFTproc.done;

   exp_add.dataa[4..0] = B"01000";     -- add 8 to exponent make FFT 16-bit
   exp_add.datab[4..0] = FFTproc.exponent[expwidth..1];

   exp_negate.dataa[4..0] = GND;
   exp_negate.datab[4..0] = exp_add.result[4..0];  -- negate exponent

   if exp_add.result[4] == B"0" then               -- exponent is positive
      shift_direction = B"1";                      -- shift right (make smaller)
      shift_distance[] = exp_add.result[];
   else                                            -- exponent is negative
      shift_direction = B"0";                      -- shift left (make larger)
      shift_distance[] = exp_negate.result[];
   end if;

   out_shift_real.data[(datawidth-1)..0] = FFTproc.readreal[datawidth..1];
   out_shift_imag.data[(datawidth-1)..0] = FFTproc.readimag[datawidth..1];
   out_shift_real.distance[3..0] = shift_distance[(expwidth-2)..0];  -- lower bits
   out_shift_imag.distance[3..0] = shift_distance[(expwidth-2)..0];
   out_shift_real.direction = shift_direction;     -- shift right (make smaller)
   out_shift_imag.direction = shift_direction;

   exponent[expwidth..1] = shift_distance[(expwidth-1)..0];
   readreal[datawidth..1] = out_shift_real.result[(datawidth-1)..0];
   readimag[datawidth..1] = out_shift_imag.result[(datawidth-1)..0];
END;
```

# Appendix B: FFT Processor (Top level) Source Code

```
-- FFT Processor (Single)
-- Grant Hampson June 26 2002

FUNCTION SingleFFT (sysclk, reset, go, writeaddress[10..1], readaddress[10..1],
                    read, write, writereal[14..1], writeimag[14..1])
   RETURNS (readreal[14..1], readimag[14..1], exponent[5..1], done);

SUBDESIGN FFTProcessor
(
   control1in,                 -- clocks from general connector
   control2in,
   realin[15..2],              -- data inputs from general connector
   imagin[15..2]
      :INPUT;

   control1out,                -- control lines to general connector
   control2out,
   realout[15..0],             -- data outputs for general connector
   imagout[15..0]
      :OUTPUT;
)

VARIABLE
   FFT : SingleFFT;            -- instance of a FFT processor

   cntr_write[9..0],           -- counter for writing memory address
   cntr_read[9..0],            -- counter for reading memory address
   FFT_cont_in[2..0],
   FFT_cont_out[7..0],
   sm_FFTcont[2..0],
   reg_outreal[15..0],
   reg_outimag[15..0]: DFF;    -- FFT state machine

   write_done, read_done,      -- Signals indicate end of write/read
   output_select,
   outclock_enable: NODE;      -- Signals indicate end of write/read

BEGIN
   FFT.sysclk = control2in;                      -- connection of FFT processor inputs
   FFT.writeaddress[10..1] = cntr_write[9..0].q;
   FFT.readaddress[10..1] = cntr_read[9..0].q;
   FFT.writereal[14..1] = realin[15..2];
   FFT.writeimag[14..1] = imagin[15..2];

   reg_outreal[].clk = control2in;               -- Control output data
   reg_outimag[].clk = control2in;
   realout[] = reg_outreal[];
   imagout[] = reg_outimag[];
   if output_select == B"1" then
      reg_outreal[15..2].d = FFT.readreal[14..1];
      reg_outimag[15..2].d = FFT.readimag[14..1];
      reg_outreal[1..0].d = GND;
      reg_outimag[1..0].d = GND;
   else
```

```
      reg_outreal[15..0].d = B"1000000000000000";
      reg_outimag[15..0].d = B"1000000000000000";-- FFT start/end marker
   end if;
   control2out = outclock_enable;                -- clock enable for capture FIFO
   control1out = control2in;                     -- 100MHz output clock
   cntr_write[].clk = control2in;                -- counter for write address
   cntr_write[].d = cntr_write[].q + 1;
   write_done = cntr_write[].q == B"1111111111"; -- = 1023

   cntr_read[].clk = control2in;                 -- counter for read address
   cntr_read[].d = cntr_read[].q + 1;
   read_done = cntr_read[].q == B"1111111111";   -- = 1023
   FFT_cont_in[2].d = write_done;                -- Inputs to state machine
   FFT_cont_in[1].d = FFT.done;
   FFT_cont_in[0].d = read_done;
   outclock_enable = FFT_cont_out[0];            -- Outputs of state machine
   output_select = FFT_cont_out[1];
   cntr_read[].clrn = FFT_cont_out[2];
   FFT.read = FFT_cont_out[3];
   FFT.go = FFT_cont_out[4];
   FFT.write = FFT_cont_out[5];
   cntr_write[].clrn = FFT_cont_out[6];
   FFT.reset = FFT_cont_out[7];

   TABLE
      sm_FFTcont[].q, FFT_cont_in[2..0] => sm_FFTcont[].d, FFT_cont_out[7..0].d;
              B"000",           B"XXX" => B"001",        B"10000011"; -- reset FFT
              B"001",           B"0XX" => B"001",        B"01100011"; -- write raw data
              B"001",           B"1XX" => B"010",        B"01100011"; -- done writing
              B"010",           B"X0X" => B"010",        B"00010011"; -- start FFT
              B"010",           B"X1X" => B"011",        B"00011111"; -- FFT finished
              B"011",           B"XXX" => B"100",        B"00011111"; -- read FFT data
              B"100",           B"XXX" => B"101",        B"00011111"; -- read FFT data
              B"101",           B"XX0" => B"101",        B"00011110"; -- enable FIFO writes
              B"101",           B"XX1" => B"110",        B"00011110"; -- finished reading
              B"110",           B"XXX" => B"111",        B"00011100"; -- select marker data
              B"111",           B"XXX" => B"000",        B"00011110"; -- write marker data
   END TABLE;
   sm_FFTcont[].clk = control2in;
   FFT_cont_in[].clk = control2in;
   FFT_cont_out[].clk = control2in;
END;
```