Carlos Georges
Nicholas VanDeventer

16 Point Fast Fourier Transform

The Fourier transform is a useful operation in the signal processing space. The transform allows for a signal to be operated on in the frequency domain which can make certain filtering and processing operations much more efficient. A time domain signal arrives in time and the amplitudes represent intensity, however a frequency domain signal arrives in frequency and the amplitudes represent intensity of the frequencies. The discrete Fourier transform (DFT), is the comprehensive transform for a number of sample points. The operation is often times redundant for many values, so researchers developed the fast Fourier transform (FFT) algorithm which is often regarded as the most important algorithm developed in the last century. Modern day digital signal processors contain a multitude of large FFT cores capable of processing dense signals reliably. The FFT is easily implemented in software for a large range of sample points, but is much slower than a hardware implementation. For the embedded DSP course, we are expected to implement the algorithm using computer hardware.

Our approach was a very structural one, using principles of hierarchy, abstraction, and optimization. The 16 point FFT can be broken down into 4 stages of a butterfly operation. The butterfly operation is the algorithmic method of computing the FFT. The 4 stages consist of 8 2-point butterfly operations, 4 4-point butterfly operations, 2 8-point butterfly operations, and 1 16-point butterfly operation.

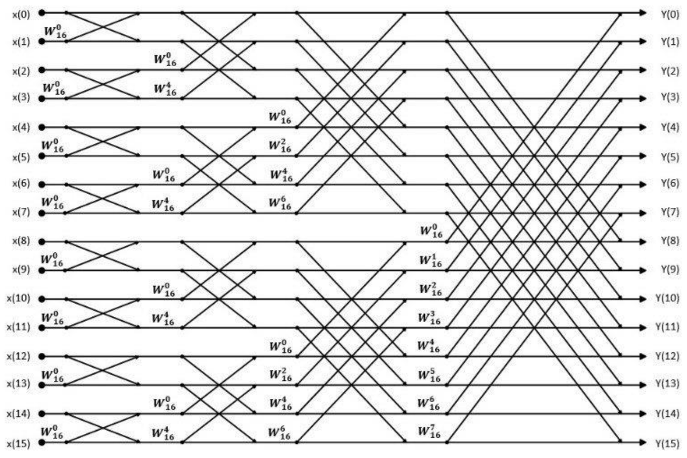


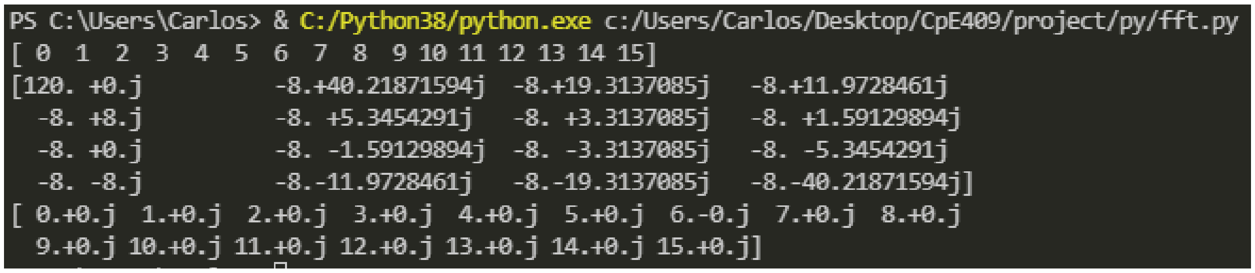
Figure 1: A diagram of the 16 point FFT butterfly operations

The butterfly operations require arithmetic with complex numbers, as the signals in the frequency domain present themselves in the complex plane conveniently. We can simply represent complex numbers using two registers and perform the arithmetic with those. Our approach is a decimation-in-time FFT, which requires a bit reversal of the input order to receive the frequency domain signals in the proper order. The inverse transform is of interest as well. The inverse FFT can be computed by using another forward FFT and passing in the complex conjugate of the output of the first FFT. The complex conjugate is obtained by simply negating the imaginary part of the complex number. After the signal passes through a second FFT module, the output is divided by N (16) to return the original signal. As for the HDL implementation, we followed a “control unit, datapath” architecture, effectively making a single FFT core. The circuit contains the forward and inverse modules, input and output registers, and the control unit. The modules require 16 parallel inputs, and will have 16 parallel outputs, but each I/O is 64 bits and this will require far too many I/O pins on a physical implementation, so a control unit is used to facilitate serial I/O, significantly reducing the number of I/O pins. The control unit is a simple FSM that transitions between filling the SIPO and reading the PISO.

Carlos Georges
Nicholas VanDeventer

Testing

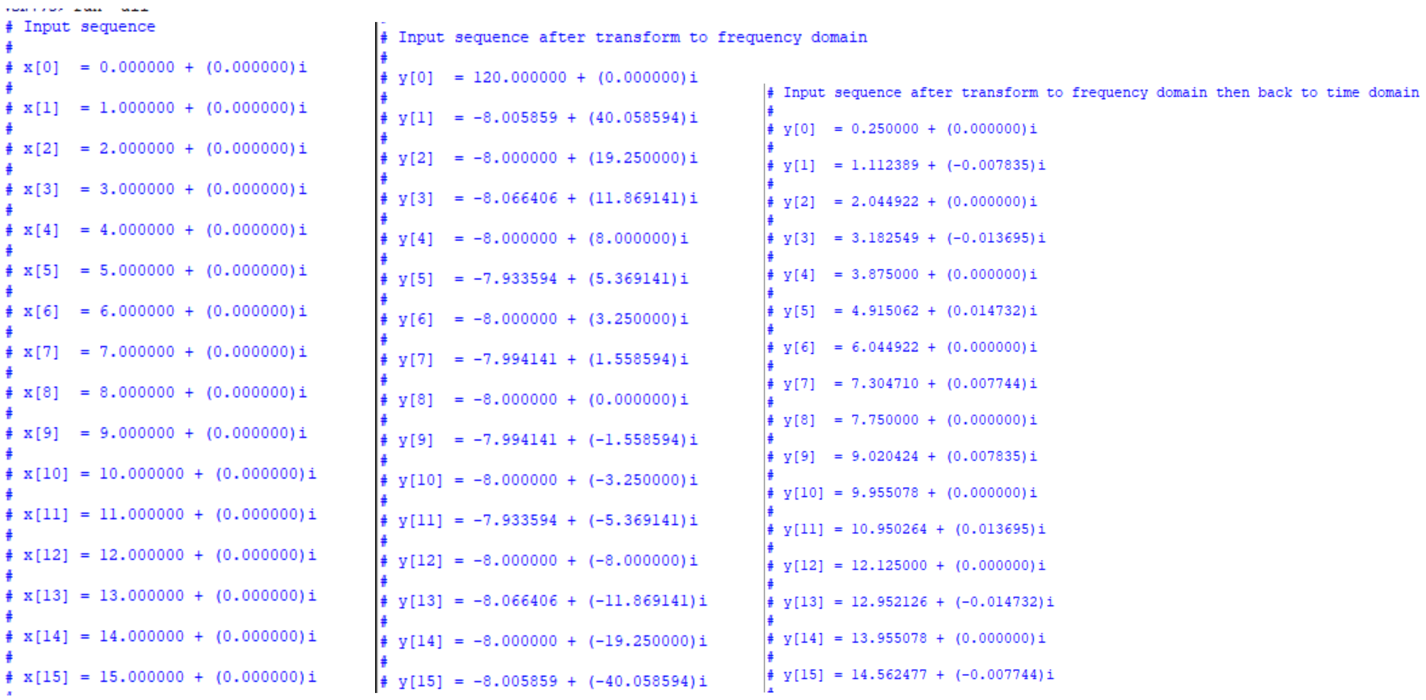
Of course, a testbench was developed to properly stimulate and verify the RTL design. The testbench has 16 real inputs which can be easily modified to test different input sequences. The simulation then proceeds to display the original input sequence, the frequency domain signal, and then the output of the inverse transform, which should be the original time domain signal. The circuit's functional operation was confirmed with a software implementation (python's numpy library). For demonstration purposes, the input sequence is simply the numbers 0-15. The results were on target but had lost some precision because of the scaling factor. With 64 bits and 4 stages of scaling the most reasonable scaling factor was 2^7 =128. A smaller scaling factor does result in a loss of precision, but the testbench does demonstrate the proof of concept.



```
PS C:\Users\Carlos> & C:/Python38/python.exe c:/Users/Carlos/Desktop/CpE409/project/py/fft.py
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[120. +0.j          -8.+40.21871594j  -8.+19.3137085j   -8.+11.9728461j
  -8. +8.j          -8. +5.3454291j   -8. +3.3137085j   -8. +1.59129894j
  -8. +0.j          -8. -1.59129894j  -8. -3.3137085j   -8. -5.3454291j
  -8. -8.j          -8.-11.9728461j   -8.-19.3137085j   -8.-40.21871594j]
[ 0.+0.j  1.+0.j  2.+0.j  3.+0.j  4.+0.j  5.+0.j  6.-0.j  7.+0.j  8.+0.j
  9.+0.j 10.+0.j 11.+0.j 12.+0.j 13.+0.j 14.+0.j 15.+0.j]
```

Figure 2: Software verification of the input sequence

The above figure shows the same data that will be shown in the testbench, that is, the original input sequence, the transformed signal, and the signal after passing through the inverse transform.



```
# Input sequence
#
# x[0]  = 0.000000 + (0.000000)i
#
# x[1]  = 1.000000 + (0.000000)i
#
# x[2]  = 2.000000 + (0.000000)i
#
# x[3]  = 3.000000 + (0.000000)i
#
# x[4]  = 4.000000 + (0.000000)i
#
# x[5]  = 5.000000 + (0.000000)i
#
# x[6]  = 6.000000 + (0.000000)i
#
# x[7]  = 7.000000 + (0.000000)i
#
# x[8]  = 8.000000 + (0.000000)i
#
# x[9]  = 9.000000 + (0.000000)i
#
# x[10] = 10.000000 + (0.000000)i
#
# x[11] = 11.000000 + (0.000000)i
#
# x[12] = 12.000000 + (0.000000)i
#
# x[13] = 13.000000 + (0.000000)i
#
# x[14] = 14.000000 + (0.000000)i
#
# x[15] = 15.000000 + (0.000000)i
```

```
# Input sequence after transform to frequency domain
#
# y[0]  = 120.000000 + (0.000000)i
#
# y[1]  = -8.005859 + (40.058594)i
#
# y[2]  = -8.000000 + (19.250000)i
#
# y[3]  = -8.066406 + (11.869141)i
#
# y[4]  = -8.000000 + (8.000000)i
#
# y[5]  = -7.933594 + (5.369141)i
#
# y[6]  = -8.000000 + (3.250000)i
#
# y[7]  = -7.994141 + (1.558594)i
#
# y[8]  = -8.000000 + (0.000000)i
#
# y[9]  = -7.994141 + (-1.558594)i
#
# y[10] = -8.000000 + (-3.250000)i
#
# y[11] = -7.933594 + (-5.369141)i
#
# y[12] = -8.000000 + (-8.000000)i
#
# y[13] = -8.066406 + (-11.869141)i
#
# y[14] = -8.000000 + (-19.250000)i
#
# y[15] = -8.005859 + (-40.058594)i
```

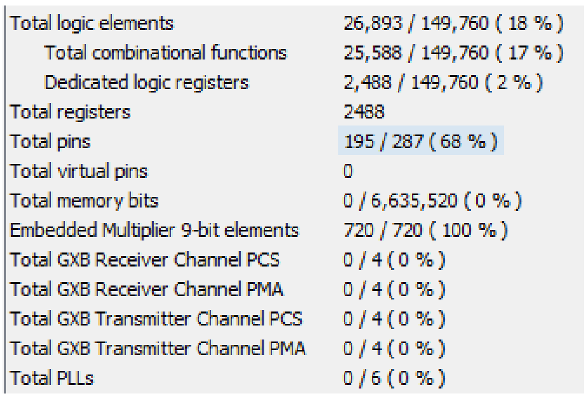
```
# Input sequence after transform to frequency domain then back to time domain
#
# y[0]  = 0.250000 + (0.000000)i
#
# y[1]  = 1.112389 + (-0.007835)i
#
# y[2]  = 2.044922 + (0.000000)i
#
# y[3]  = 3.182549 + (-0.013695)i
#
# y[4]  = 3.875000 + (0.000000)i
#
# y[5]  = 4.915062 + (0.014732)i
#
# y[6]  = 6.044922 + (0.000000)i
#
# y[7]  = 7.304710 + (0.007744)i
#
# y[8]  = 7.750000 + (0.000000)i
#
# y[9]  = 9.020424 + (0.007835)i
#
# y[10] = 9.955078 + (0.000000)i
#
# y[11] = 10.950264 + (0.013695)i
#
# y[12] = 12.125000 + (0.000000)i
#
# y[13] = 12.952126 + (-0.014732)i
#
# y[14] = 13.955078 + (0.000000)i
#
# y[15] = 14.562477 + (-0.007744)i
```

Testbench results. As we can see, the results are accurate but not extremely precise.

Carlos Georges
Nicholas VanDeventer

Hardware Implementation

The top module was synthesized in the Quartus software. The circuitry is very resource intensive and the combinational logic has a considerable propagation delay. However, the circuit can complete the operation in only 16 clock cycles and the data can be read out serially in another 16 cycles. The resources can be optimized by implementing a pipeline structure to reduce the number of multiplier elements. Although, modern DSPs contain many MAC units, which are extensively used in this operation, so with more specialized hardware, the design would be less resource intensive.



Total logic elements	26,893 / 149,760 (18 %)
Total combinational functions	25,588 / 149,760 (17 %)
Dedicated logic registers	2,488 / 149,760 (2 %)
Total registers	2488
Total pins	195 / 287 (68 %)
Total virtual pins	0
Total memory bits	0 / 6,635,520 (0 %)
Embedded Multiplier 9-bit elements	720 / 720 (100 %)
Total GXB Receiver Channel PCS	0 / 4 (0 %)
Total GXB Receiver Channel PMA	0 / 4 (0 %)
Total GXB Transmitter Channel PCS	0 / 4 (0 %)
Total GXB Transmitter Channel PMA	0 / 4 (0 %)
Total PLLs	0 / 6 (0 %)

Figure 4: Resource utilization

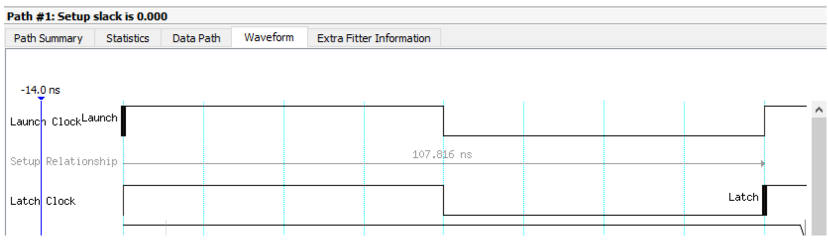
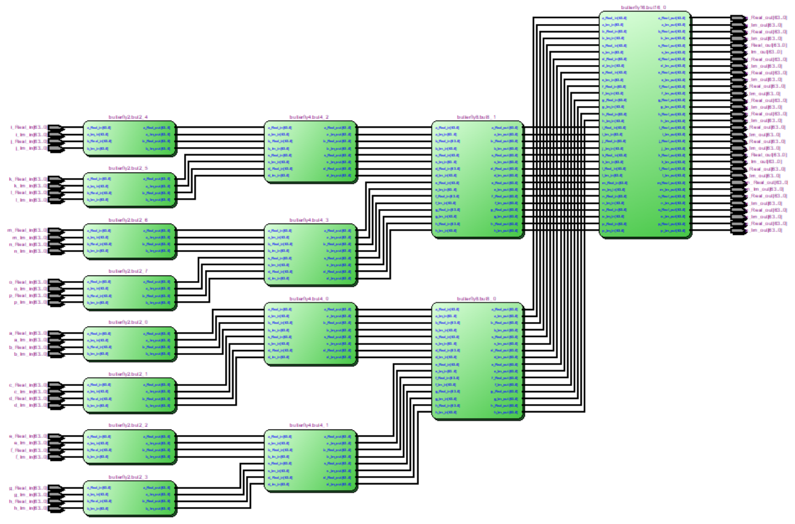

Path #1: Setup slack is 0.000

Figure 3: Timing report

The clock period needed to avoid data hazards came out to 107.8 ns. With approximately 32 clock cycles to complete a transform, inverse transform, and read, the whole sequence will take about 3.45 us.



In this view we see the connections of the four stages of butterfly operations as previously described in the paper. This circuitry serves as the combinational logic for the datapath and contains the majority of the resources.

Figure 5: RTL view of 16 point butterfly operation

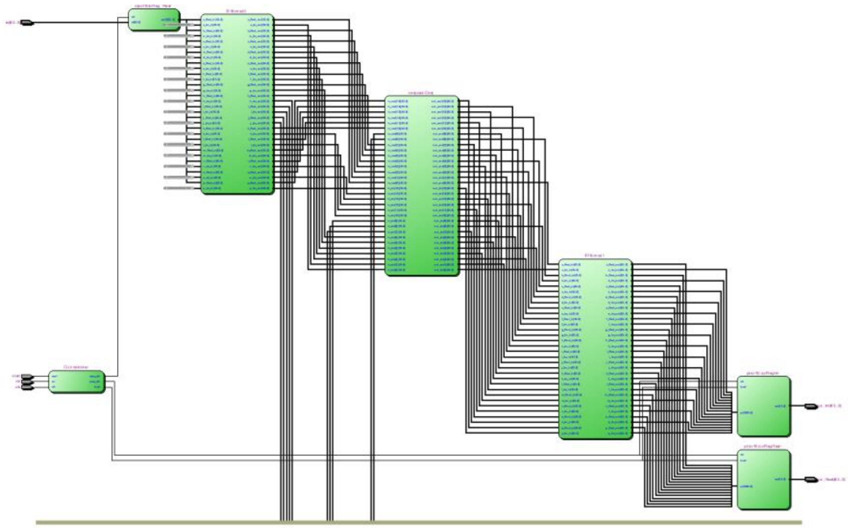
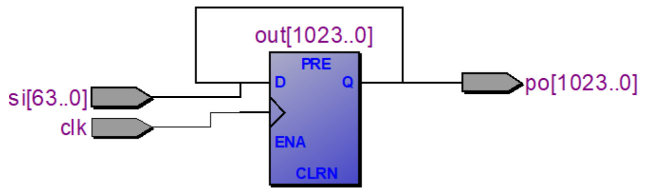
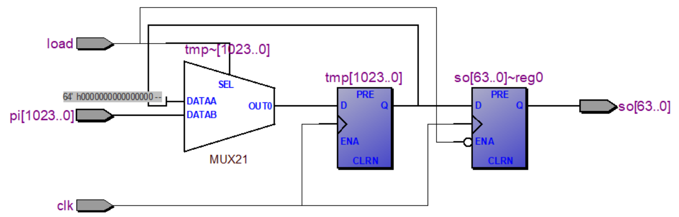


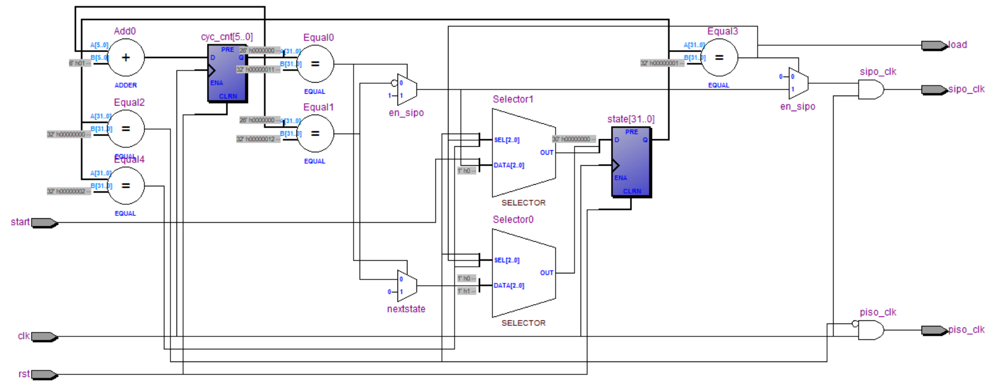
Figure 6: RTL view of the top module

The RTL view of the top module shows all the components of the core. The view is a very close realization of our initial drawup and works as expected.



Left we see the RTL views of the input and output registers.



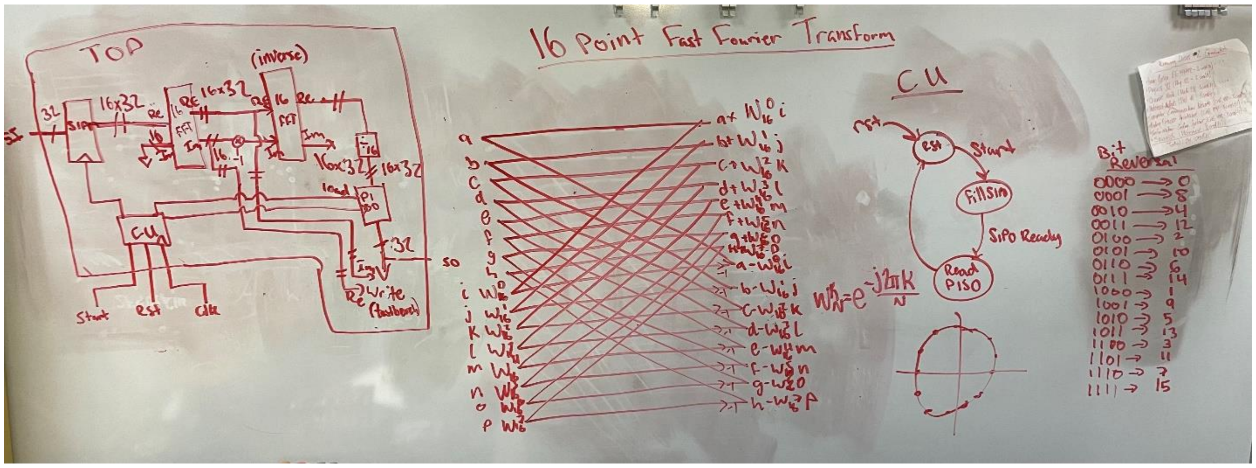


Finally, we see the internals of the control unit state machine.

Carlos Georges
Nicholas VanDeventer

<u>Conclusion</u>

The project was rigorous and thorough. Our understanding of signals and systems was enhanced greatly through the research and development of the unit. The project involved many aspects of computer engineering including the principles of abstraction, signal processing, frequency transforms, RTL design, processor architecture, and simulation/verification. The process was comprehensive and informative and we are glad we chose the 16 point FFT for the final project. It was a strong project to demonstrate the skills we've learned in this course and the prerequisites.


16 Point Fast Fourier Transform
TOP
SIPO
16x32
FFT
(inverse)
Re
Im
PISO
CU
Start
rst
clk
SI
SO
load
write (testbench)
CU
rst
Start
Fill SIPO
SIPO Ready
Read PISO
$W_N^k = e^{-j2\pi k/N}$
Bit Reversal
0000 → 0
0001 → 8
0010 → 4
0011 → 12
0100 → 2
0101 → 10
0110 → 6
0111 → 14
1000 → 1
1001 → 9
1010 → 5
1011 → 13
1100 → 3
1101 → 11
1110 → 7
1111 → 15