

Automatic RTL Conversion of DSP Algorithms for a Channelized Wideband Receiver

Mike Groden

LNX Corporation
43 High St.

No. Andover, Ma. 01845
1 978 946 9200

mikeg@lncorp.com

ABSTRACT

Automatic RTL Conversion of DSP Algorithms for a Channelized Wideband Receiver

Categories and Subject Descriptors

EDA tools

General Terms

Algorithms, DSP, FPGA Receivers, Electronic Design Automation

Keywords

Algorithms, DSP, FPGA Receivers, Electronic Design Automation, Xilinx, MATLAB™, EDA, Electronic Warfare, Electronic Intelligence, Signal Search, wideband, DSP, signal processors, RF filters, MAC, FFT

Introduction

Critical military applications including Electronic Warfare and Electronic Intelligence require continuously searching for electronic signals over a wide range of frequencies. SETI, the project to search for signals from outer space, requires similar searching, on a grander scale. This paper describes a novel approach to optimizing the algorithms and architectures of FPGA-based channelized receivers used for these types of applications.

Recently developed EDA tools were used to convert DSP algorithms developed and simulated using MATLAB™ directly into RTL code. The automatic conversion capability significantly accelerated receiver development. Numerous architectures were simulated, and targeted to specific FPGAs, without having to manually convert individual algorithms from MATLAB syntax to RTL.

In the past, signal search problems have been solved using a bank of RF filters to reduce the sampling and processing requirements, or by constructing specialized multi-processor systems. These are typically constructed using arrays of Digital Signal Processors. Using multiple RF filters requires the output of each filter bank to be down converted to baseband for subsequent processing. This results in a bulky solution. DSP processing arrays can be large, and transferring large amounts of data between devices presents its own challenges.



Figure 1 LNX 's Digital Receiver module can sample at rates up to 2 Gsa/sec and accommodates up to three Virtex-II FPGAs for real-time DSP applications.

Mixed Radix FFT Algorithms

Today's very high speed A/D converters can perform sampling at rates in excess of 2 Gsa/sec. Field Programmable Gate Arrays (FPGAs) have likewise grown in capacity and speed. FPGAs are now available with large amounts of memory and features like multipliers and MACs. These embedded features enable these devices, with sufficient RTL coding, to implement the FFTs and other elements needed to construct a high performance channelized receiver. More importantly, software tools, such as AccelFPGA™ from AccelChip Incorporated have recently become available for rapidly simulating and tailoring DSP algorithms that target these feature-rich FPGAs.

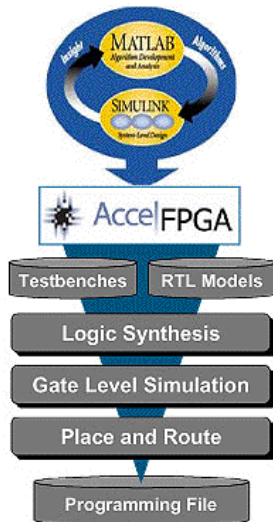


Diagram #1 - Algorithm flow from MATLAB to RTL with AccelFPGA

In our project, DSP algorithms were used for the design and implementation of very high speed fixed point FFTs, implemented in a Xilinx Virtex-II™, XC2V6000-5 FPGA. Although fast, well characterized, FFT IP cores already exist, our design approach allowed for greater flexibility in implementation and greater control over quantization and round-off noise effects. AccelFPGA enables a designer to create mixed-radix FFT implementations and thus effectively control resource use, throughput, and latency as well as to specify the width of data paths to best match an application's requirements.

A general approach to designing the FFT was selected to allow decomposing the FFT in an efficient manner. Any linear sequence can be decomposed into smaller transform sizes (the key to the FFT algorithm) by reshaping an N-point sequence into an L x M array, where $N = L \times M$. Figure 2 shows an example calculation of the DFT of a 15 point sequence, where the sequence is reshaped into a three by five matrix. First, the DFT is calculated down each column. The resulting matrix is multiplied by a set of twiddle factors, and then the DFT is calculated across each row.

This technique can be extended, decomposing the FFT into a mixed-radix solution that maps implementations to the available FPGA resources. For example, a 256-point sequence can be reshaped into an 8 row by 32 column matrix. The first step is to calculate an 8-point DFT down each column and then multiply the resulting matrix by the appropriate twiddle factors. The next step requires the calculation of a 32-point DFT across each row, however this calculation can be simplified by reshaping the 32 element row into an 8 row by 4 column matrix and calculating the DFT as described above.

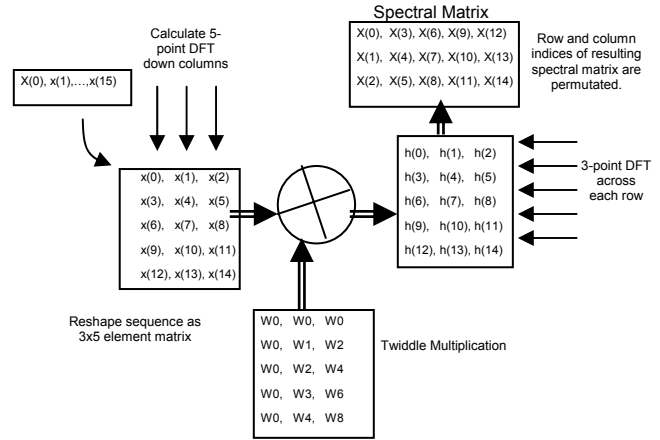


Figure 2 DFT for a 15-point sequence by reshaping the sequence into a 3x5 matrix.

The twiddle factors shown in figure 1 are calculated as $e^{-j(\text{row}-1)(\text{col}-1)/(N-1)}$ where N is the number of elements (15 in this case). The row and column indices of the spectral matrix are permuted with respect to the input matrix; this is similar to the bit reversed outputs of a radix-2, decimated in frequency FFT algorithm. In fact, the technique is simply a generalization of many FFT algorithms as described in Reference 1.

Appendix A shows the MATLAB code for a 256-point FFT decomposed as described above. That is, the 256 point sequence was first reshaped into an 8 row by 32 column matrix and then an 8-point DFT was calculated down each column. This particular decomposition was chosen because an 8-point DFT routine was written that does not use any loops and takes advantage of the constant coefficient multipliers available as Xilinx IP cores. This allows better utilization of the FPGA resources because the constant coefficient multipliers are implemented using the look up tables in the CLBs instead of the embedded multipliers or memory blocks. The available constant coefficient multipliers, hence the number of 8-point DFTs used, is only limited by the available logic. For example, the 8-point DFT requires 8 constant coefficient multipliers; a 16-bit by 16-bit, signed constant coefficient multiplier requires 28 CLBs or 100 slices; there are 33,792 slices in an XC2V6000.

Design Tools

MATLAB is a popular tool used for implementing and testing DSP algorithms. Its rich set of built-in functions, its ability to operate on matrices, and its built-in plotting utilities makes it an efficient tool for writing and testing algorithms. However converting an algorithm written and tested in MATLAB to a hardware description language, such as VHDL, can be difficult and time consuming.

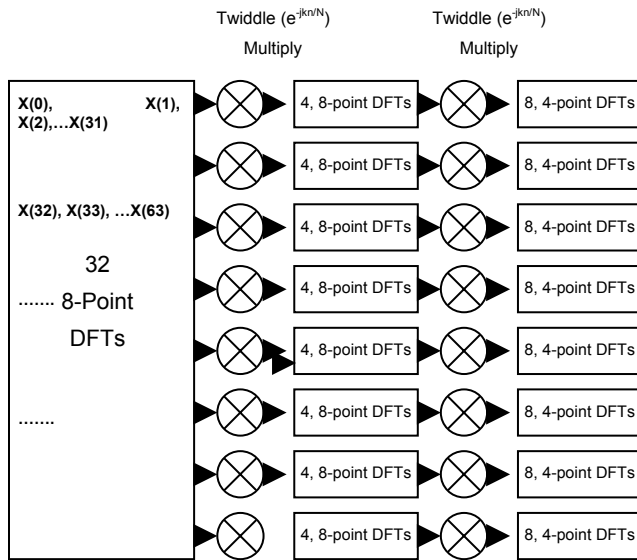


Figure 3 Decomposition of 256-point FFT using an 8 by 32 matrix.

Structures that are efficient in MATLAB may not be efficient when implemented in embedded hardware. In MATLAB no consideration is given to memory resources, data paths, or hardware resources. MATLAB uses double precision floating point arithmetic whereas embedded applications typically use fixed point calculations.

As noted above, FPGAs are growing rapidly in capacity and speed and are beginning to embed specialized functions like memory banks, embedded multipliers, MACs, and even embedded processors like the PowerPC. It's typical for logic designs targeting FPGAs to first be implemented as a hardware description language like VHDL or Verilog. Powerful synthesis tools can optimize and implement even the largest designs.

The AccelFPGA compiler is a key enabler tool for high level synthesis of DSP algorithms for implementation in FPGAs. It provides automated conversion of programs developed in MATLAB into synthesizable RTL, VHDL, or Verilog. This automated conversion results in tremendous time savings. Because AccelFPGA compiles fixed-point arithmetic MATLAB models, it supports quantization noise analysis at the highest level of abstraction with all of MATLAB capabilities. AccelFPGA then generates bit-accurate RTL that guarantees an exact match between the behavioral MATLAB algorithm description and the synthesized hardware. In addition, AccelFPGA provides effective and intuitive mechanisms to control the characteristics of the synthesized hardware architecture with compiler directives. With these tool capabilities, system engineers and algorithm developers can test concepts quickly and accurately to arrive at the hardware architecture that best meets their area and speed requirements.

Design Flow

The design was first modeled in MATLAB using implicit double precision, floating-point number representation. This allowed the mixed-radix FFT receiver design to be quickly and efficiently verified.

The floating-point model was then enhanced to include fixed-point arithmetic with quantizing mechanisms from the MATLAB Filter Design Toolbox. This allowed the analysis of round-off errors associated with (FPGA) fixed-point precision arithmetic, and it allowed determining the appropriate quantization parameters to achieve the desired performance in the algorithm and the hardware to be realized. Figure 4 is a plot of the spectrum of a 250 MHz signal, calculated using the built-in FFT function of Matlab. Figure 5 is a plot of the same spectrum, calculated using the quantized and mixed radix algorithm.

The fixed-point MATLAB model was then augmented with hardware architecture information to effectively direct the generation of the RTL model. The added information was:

- Iterative processing structure to model the 'streaming' of frames of data to be processed by the hardware in the real-time implementation.
- Specification of an FPGA target device intended for the hardware implementation.
- Specification of mapping of variables and arrays in the MATLAB program to hardware storage types (either registers or memory structures) to control the allocation of resources. Because of the 256-element arrays involved in the FFT algorithm, mapping to memory structures was preferred to achieve area efficiency.
- Specification of processing concurrency with the encapsulation of the FFT processing into functions to form a 2-level hierarchy. Functions to perform the 8-point and 4-point DFTs as well as multiplications by the appropriate twiddle factors form the lower level of the hierarchy. These functions are compiled into concurrent RTL entities to yield a 'block-pipelined architecture for high throughput.
- Calls to AccelFPGA MATLAB utility functions to generate bit-accurate reference test vectors to be used in RTL functional verification.

The fixed-point model augmented with hardware architecture information was then compiled with AccelFPGA to generate synthesizable VHDL and automatically generate a functional test bench model. This test bench was then used to perform an RTL simulation of the algorithm model using the MATLAB generated bit-accurate reference test vectors for functional verification. A standard FPGA design flow was then used to implement and simulate the design.

Implementation Results

Table 1 is a summary of the synthesis results for an area efficient implementation that makes use of RAM blocks and embedded multipliers available in the XILINX Virtex II XC2V6000 part. This implementation used less than 10% of the available resources. Other implementations are possible that could be tailored to meet specific area/speed requirements.

Table 1

LUTs	5231 (7%)
RAM Blocks	5 of 144 (10%)
Embedded Multipliers	8 of 144 (5%)
Clock Speed	97.1 MHz
Cycles per Iteration	331
Transform Processing Time	3.4 usec

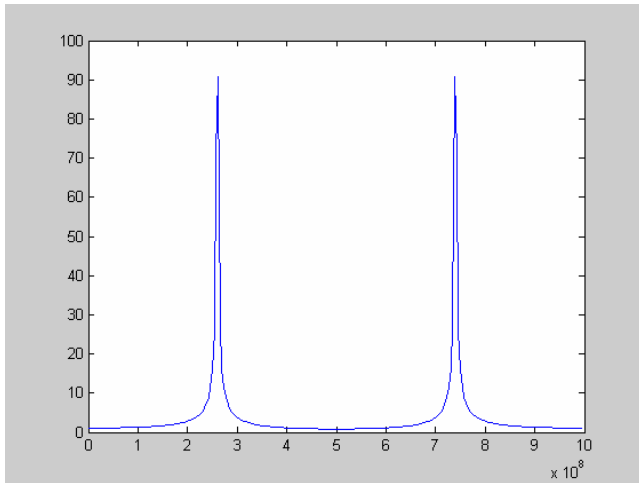


Figure 4 256-point FFT of a 250 MHz sin wave generated MATLAB. The built-in FFT and double precision floating point arithmetic was used.

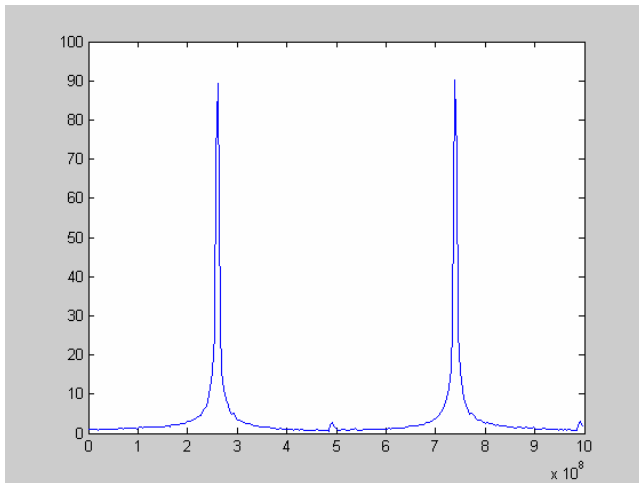


Figure 5 Quantized FFT calculated using mixed-radix FFT.

Conclusion

This paper described the design and implementation of a mixed radix FFT algorithm used in a wideband channelized receiver application. A generalized approach to implementing the FFT was described and an example decomposition of a 256-point FFT was presented. This design was first implemented and tested in MATLAB and then converted to VHDL using the AccelChip design tools.

This design was then synthesized and implemented on a Xilinx XC2V6000 FPGA. The generalized implementation of the FFT described allowed the design to be tailored to the available resources in the target hardware. Designing the algorithms in MATLAB and using AccelFPGA to convert the design to RTL significantly reduces the time required to implement and test different approaches using a "bit true" simulation.

References

- [1] L. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1975.
- [2] A. Oppenheim and R. Schaffer, *Digital Signal Processing*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1975.

Appendix A

```

% write input test vector
accel_write_quantized(qin,tv_in,'tv_in');
% introduce iterative processing loop
%!ACCEL STREAM iter
for iter = 256:256:256

%!ACCEL BEGIN_HARDWARE tv_in

% stream new data in
%!ACCEL MEM_MAP array1 TO ram_s9_s9(0) AT 0
array1 = reshape( tv_in(iter-255:iter), 8, 32);

% ***** FPGA Implementation begins here *****
% Now DFT down columns using 8-point dft function;
%!ACCEL MEM_MAP fft_array1_real TO ram_s18_s18(1)
AT 0
%!ACCEL MEM_MAP fft_array1_imag TO ram_s18_s18(2)
AT 0
% do the 8-point DFTs on entire 2-d array.
[fft_array1_real, fft_array1_imag] =
dft8(array1,qin,qinp3);

% Multiply times twiddles
%!ACCEL MEM_MAP fft_array5_real TO ram_s18_s18(3)
AT 0
%!ACCEL MEM_MAP fft_array5_imag TO ram_s18_s18(4)
AT 0
[fft_array5_real, fft_array5_imag] = ...
    twid_mult 1(fft_array1_real, fft_array1_imag,
    twid1_real, twid1_imag, qinp3);

%Calculate FFT across rows by decomposition into
an 8 by 8 matrix
%!ACCEL MEM_MAP fft_array2_real TO ram_s18_s18(7)
AT 0
%!ACCEL MEM_MAP fft_array2_imag TO ram_s18_s18(8)
AT 0
[fft_array2_real, fft_array2_imag] = ...
    dft8a(fft_array5_real,
    fft_array5_imag, qinp3, qinp6);

%!ACCEL MEM_MAP fft_array3_real TO ram_s18_s18(11)
AT 0
%!ACCEL MEM_MAP fft_array3_imag TO ram_s18_s18(12)
AT 0
[fft_array3_real, fft_array3_imag] = ...
    twid_mult 2(fft_array2_real,
    fft_array2_imag, twid2_real, twid2_imag, qinp6);

%!ACCEL MEM_MAP fft_array4_real TO ram_s18_s18(15)
AT 0
%!ACCEL MEM_MAP fft_array4_imag TO ram_s18_s18(16)
AT 0
[fft_array4_real, fft_array4_imag] = ...
dft4(fft_array3_real,fft_array3_imag,qinp6,qout);

out_real(iter-255:iter) =
reshape(fft_array4_real,1,256);
out_imag(iter-255:iter) =
reshape(fft_array4_imag,1,256);
%!ACCEL END_HARDWARE out_real, out_imag
% ***** FPGA Implementation Ends *****

```

Figure A1 Top level Matlab code for calculation of a mixed radix 256-point FFT.

```

function [Xoutr_2d,Xouti_2d] =
dft8(xinr_2d,qin,qout)
% 8-point DFT function
pt7 = quantize(qin,0.70710678118655);

% operate on entire 2-d array one column at a time
for col=1:32

%xinr = quantize(qin,xinr_2d(:,col));
%!ACCEL SHAPE xinr(8)
xinr1 = quantize(qin,xinr_2d(1,col));
xinr2 = quantize(qin,xinr_2d(2,col));
xinr3 = quantize(qin,xinr_2d(3,col));
xinr4 = quantize(qin,xinr_2d(4,col));
xinr5 = quantize(qin,xinr_2d(5,col));
xinr6 = quantize(qin,xinr_2d(6,col));
xinr7 = quantize(qin,xinr_2d(7,col));
xinr8 = quantize(qin,xinr_2d(8,col));
xin2rpt7 = xinr2 * pt7;xin4rpt7 = xinr4 * pt7;
xin6rpt7 = xinr6 * pt7;xin8rpt7 = xinr8 * pt7;
%%!ACCEL SHAPE Xoutr(8,1)
%%!ACCEL SHAPE Xouti(8,1)
temp11r = quantize(qout,(xinr1 + xinr3) +
(xinr2 + xinr4));
temp12r = quantize(qout,(xinr5 + xinr7) +
(xinr6 + xinr8));

Xoutr1 = quantize(qout,temp11r + temp12r);
Xouti1 = quantize(qout,0);
temp21r = quantize(qout,(xinr1 - xinr5) +
(xin2rpt7 - xin4rpt7) - (xin6rpt7 - xin8rpt7));
Xoutr2 = quantize(qout,temp21r);
temp21i = quantize(qout,(-xin2rpt7 - xin4rpt7) +
(xin6rpt7 + xin8rpt7) - (xinr3 - xinr7));
Xouti2 = quantize(qout,temp21i);
temp31r = quantize(qout,xinr1 - xinr3 +
xinr5 - xinr7);
Xoutr3 = quantize(qout,temp31r);
temp31i = quantize(qout,(-xinr2 + xinr4) -
xinr6 - xinr8);
Xouti3 = quantize(qout,temp31i);
temp41r = quantize(qout,(xinr1 - xinr5) -
(xin2rpt7 - xin4rpt7) + (xin6rpt7 - xin8rpt7));
Xoutr4 = quantize(qout,temp41r);
temp41i = quantize(qout,(-xin2rpt7 - xin4rpt7) +
(xin6rpt7 + xin8rpt7) + (xinr3 - xinr7));
Xouti4 = quantize(qout,temp41i);
temp51r = quantize(qout,(xinr1 + xinr3) -
(xinr2 + xinr4));
temp52r = quantize(qout,(xinr5 + xinr7) -
(xinr6 + xinr8));
Xoutr5 = quantize(qout,temp51r + temp52r);
Xouti5 = quantize(qout,0);
Xoutr6 = quantize(qout,temp41r);
Xouti6 = quantize(qout,(-temp41i));
Xoutr7 = quantize(qout,temp31r);
Xouti7 = quantize(qout,(-temp31i));
Xoutr8 = quantize(qout,temp21r);
Xouti8 = quantize(qout,(-temp21i));
%!ACCEL SHAPE Xoutr_2d(8,32)
Xoutr_2d(1,col) = Xoutr1;Xoutr_2d(2,col) = Xoutr2;
Xoutr_2d(3,col) = Xoutr3;Xoutr_2d(4,col) = Xoutr4;
Xoutr_2d(5,col) = Xoutr5;Xoutr_2d(6,col) = Xoutr6;
Xoutr_2d(7,col) = Xoutr7;Xoutr_2d(8,col) = Xoutr8;
%!ACCEL SHAPE Xouti_2d(8,32)
Xouti_2d(1,col) = Xouti1;Xouti_2d(2,col) = Xouti2;
Xouti_2d(3,col) = Xouti3;Xouti_2d(4,col) = Xouti4;
Xouti_2d(5,col) = Xouti5;Xouti_2d(6,col) = Xouti6;
Xouti_2d(7,col) = Xouti7;Xouti_2d(8,col) = Xouti8;

end;

```

Figure A2 Matlab code for calculation of the first 8-point DFT.

```

function [Xoutr_2d,Xouti_2d] =
dft8a(xinr_2d,xini_2d,qin,qout)
% 8-point DFT function

for col = 1:4
    for row = 1:8

        %xinr = xinr_2d(row,col:4:end);
        xinr1 = quantize(qin,xinr_2d(row,col));
        xinr2 = quantize(qin,xinr_2d(row,col+4));
        xinr3 = quantize(qin,xinr_2d(row,col+8));
        xinr4 = quantize(qin,xinr_2d(row,col+12));
        xinr5 = quantize(qin,xinr_2d(row,col+16));
        xinr6 = quantize(qin,xinr_2d(row,col+20));
        xinr7 = quantize(qin,xinr_2d(row,col+24));
        xinr8 = quantize(qin,xinr_2d(row,col+28));
        %xini = xini_2d(row,col:4:end);
        xini1 = quantize(qin,xini_2d(row,col));
        xini2 = quantize(qin,xini_2d(row,col+4));
        xini3 = quantize(qin,xini_2d(row,col+8));
        xini4 = quantize(qin,xini_2d(row,col+12));
        xini5 = quantize(qin,xini_2d(row,col+16));
        xini6 = quantize(qin,xini_2d(row,col+20));
        xini7 = quantize(qin,xini_2d(row,col+24));
        xini8 = quantize(qin,xini_2d(row,col+28));

        xin2rpt7 = xinr2 * pt7;
        xin4rpt7 = xinr4 * pt7;
        xin6rpt7 = xinr6 * pt7;
        xin8rpt7 = xinr8 * pt7;
        xin2ipt7 = xini2 * pt7;
        xin4ipt7 = xini4 * pt7;
        xin6ipt7 = xini6 * pt7;
        xin8ipt7 = xini8 * pt7;

        temp11r = quantize(qout,(xinr1 + xinr3) +
(xinr2 + xinr4));
        temp12r = quantize(qout,(xinr5 + xinr7) +
(xinr6 + xinr8));
        Xoutr1 = quantize(qout,temp11r + temp12r);
        temp11i = quantize(qout,(xini1 + xini3) +
(xini2 + xini4));
        temp12i = quantize(qout,(xini5 + xini7) +
(xini6 + xini8));
        Xouti1 = quantize(qout,temp11i + temp12i);
        temp21r = quantize(qout,(xinr1 - xinr5) +
(xin2rpt7 - xin4rpt7) -
(xin6rpt7 - xin8rpt7));
        temp22r = quantize(qout,(xin2ipt7 +
xin4ipt7) - (xin6ipt7 + xin8ipt7) +
(xini3 - xini7));
        Xoutr2 = quantize(qout,temp21r + temp22r);
        temp21i = quantize(qout,
(-xin2rpt7 - xin4rpt7) +
(xin6rpt7 + xin8rpt7) - (xinr3 - xinr7));
        temp22i = quantize(qout,(xini1 - xini5) +
(xin2ipt7 - xin4ipt7) -
(xin6ipt7 - xin8ipt7));
        Xouti2 = quantize(qout,temp21i + temp22i);
        temp31r = quantize(qout,(xinr1 - xinr3) +
(xinr5 - xinr7));
        temp32r = quantize(qout,(xini2 - xini4) +
(xini6 - xini8));
        Xoutr3 = quantize(qout,temp31r + temp32r);

        temp31i = quantize(qout,(-xinr2 + xinr4) -
(xinr6 - xinr8));
        temp32i = quantize(qout,(xini1 - xini3) +
(xini5 - xini7));
        Xouti3 = quantize(qout,temp31i + temp32i);
        temp41r = quantize(qout,(xinr1 - xinr5) -
(xin2rpt7 - xin4rpt7) +
(xin6rpt7 - xin8rpt7));
        temp42r = quantize(qout,
(xin2ipt7 + xin4ipt7) -
(xin6ipt7 + xin8ipt7) - (xini3 - xini7));
        Xoutr4 = quantize(qout,temp41r + temp42r);
        temp41i = quantize(qout,
(-xin2rpt7 - xin4rpt7) +
(xin6rpt7 + xin8rpt7) + (xinr3 - xinr7));
        temp42i = quantize(qout,
(xini1 - xini5) - (xin2ipt7 - xin4ipt7) +
(xin6ipt7 - xin8ipt7));
        Xouti4 = quantize(qout,temp41i + temp42i);

        temp51r = quantize(qout,(xinr1 + xinr3) -
(xinr2 + xinr4));
        temp52r = quantize(qout,(xinr5 + xinr7) -
(xinr6 + xinr8));
        Xoutr5 = quantize(qout,temp51r + temp52r);
        temp51i = quantize(qout,(xini1 + xini3) -
(xini2 + xini4));
        temp52i = quantize(qout,(xini5 + xini7) -
(xini6 + xini8));
        Xouti5 = quantize(qout,temp51i + temp52i);
        Xoutr6 = quantize(qout,temp41r - temp42r);
        Xouti6 = quantize(qout,
-(temp41i -temp42i));
        Xoutr7 = quantize(qout,temp31r - temp32r);
        Xouti7 = quantize(qout,
-(temp31i - temp32i));
        Xoutr8 = quantize(qout,temp21r - temp22r);
        Xouti8 = quantize(qout,
-(temp21i - temp22i));

        %Xoutr_2d(row,col:4:32) = Xoutr;
        Xoutr_2d(row,col) = Xoutr1;
        Xoutr_2d(row,col+4) = Xoutr2;
        Xoutr_2d(row,col+8) = Xoutr3;
        Xoutr_2d(row,col+12) = Xoutr4;
        Xoutr_2d(row,col+16) = Xoutr5;
        Xoutr_2d(row,col+20) = Xoutr6;
        Xoutr_2d(row,col+24) = Xoutr7;
        Xoutr_2d(row,col+28) = Xoutr8;
        %Xouti_2d(row,col:4:32) = Xouti;
        Xouti_2d(row,col) = Xouti1;
        Xouti_2d(row,col+4) = Xouti2;
        Xouti_2d(row,col+8) = Xouti3;
        Xouti_2d(row,col+12) = Xouti4;
        Xouti_2d(row,col+16) = Xouti5;
        Xouti_2d(row,col+20) = Xouti6;
        Xouti_2d(row,col+24) = Xouti7;
        Xouti_2d(row,col+28) = Xouti8;

    end;
end;

```

Figure A3 Matlab code for calculation of second 8-point DFT.