**Abstract**

In digital signal processing (DSP), the Fast Fourier Transform (FFT) is one of the most fundamental and useful system building block and algorithm for designing a system. Beside the application of FFT in software developing for processing image and sound data, the FFT algorithm also can be implemented in hardware design (i.e. in digital logic design, field programmable gate arrays (FPGA), etc.) for achieving high-speed in calculation as well as managing the allocating the consumption of the program so that it suitable for available resources.

[6] W. H. Press, S. A. Teukolski, W. T. Vetterling and B. P. Flannery, Numerical Recipes in C, The Art of Scientific Computing, Cambridge Univ. Press, 2002

In this project, we focus on the Cooley-Tukey Radix-2 FFT algorithm [6], which is highly efficient and easy to implement and widely used in practice. We review briefly the mathematical basis of the algorithm before implement and simulate the algorithm by programming in VHDL by using Quartus as well as generating test bench for verification of the codes generated in ModelSim. Finally, we implement the said algorithm by generating in Nios II Software Build Tools and Eclipse by using C programming language.

**Introduction**

In the recent decade, Discrete Fourier Transform (DFT) has been playing several important roles in advanced application in the field of electrical engineering and computer science, namely image compression, biomedical images construction, audiology research, sound filtering, data research, etc.

However, the number of multiplication and addition operations required for complex number for the DFT is of order *N*2 or we can say algorithm complexity of O(*N*2) for *N* data points of complex number to calculate. Hence it is not a very efficient method for the practical application. However, there are a number of different Fast Fourier Transform (FFT) algorithms that can reduce the complexity. The computations needed for *N* points from O(*N*2) to O(*NlogN*) where log is the base-2 logarithm. Implementation of different schemes of FFT algorithms such as Cooley-Tukey, Good-Thomas, Radix-2 and Rader FFT algorithm. And in this project, we implement the Radix-2 Fast Fourier Transform algorithm for calculating the complex data numbers using VHDL and C programming.

**Theory**

**1/ The Cooley-Tukey FFT Algorithm**

This methods was proposed by Cooley and Tukey [11] for computing FFT by approaching the problem by dividing the total numbers of FFT points into two factors [12], N1 and N2 as follow:

[11] Cooley, James W., and John W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Math. Comput., 19, 297–301, (1965). [12] Duhamel, P., and M. Vetterli, ―Fast Fourier transforms: A tutorial review and a state of the art‖, Signal Processing, 19, 259–299, (1990).

For example, when N is 15 then N1 and N2 are chosen to be 3 and 5 respectively. According to Cooley and Tukey, the input and output indexes can be described by expressions:

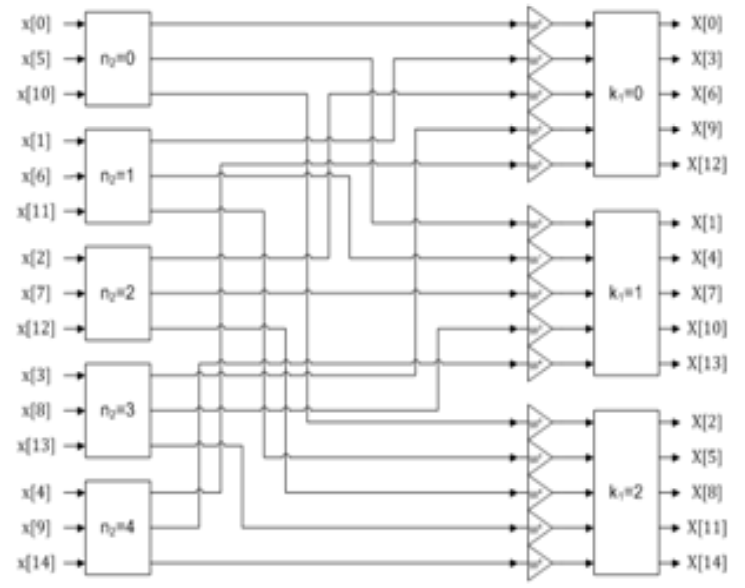
The block diagram of this method for N = 15 can be interpreted as below:

Figure 1. The block diagram of 15 points Cooley-Tukey FFT algorithm

**2/ The Radix-2 FFT Algorithm**

[15] P. Duhamel and H. Hollmann, ―Split-radix FFT Algorithm‖, Electron. Lett., 20(1), 14–16, (1984).

[16] M. Vetterli and H. J. Nussbaumer, ―Simple FFT and DCT algorithms with reduced number of operations,‖ Signal Processing, 6( 4),. 267–278, (1984).

[17] J. B. Martens, ―Recursive cyclotomic factorization—A new algorithm for calculating the discrete Fourier transform, ‖IEEE Trans. Acoust., Speech, Signal Processing, 32(4), 750–761, (1984).

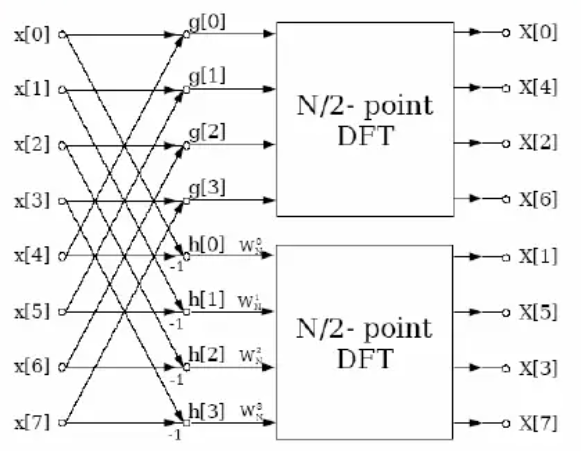
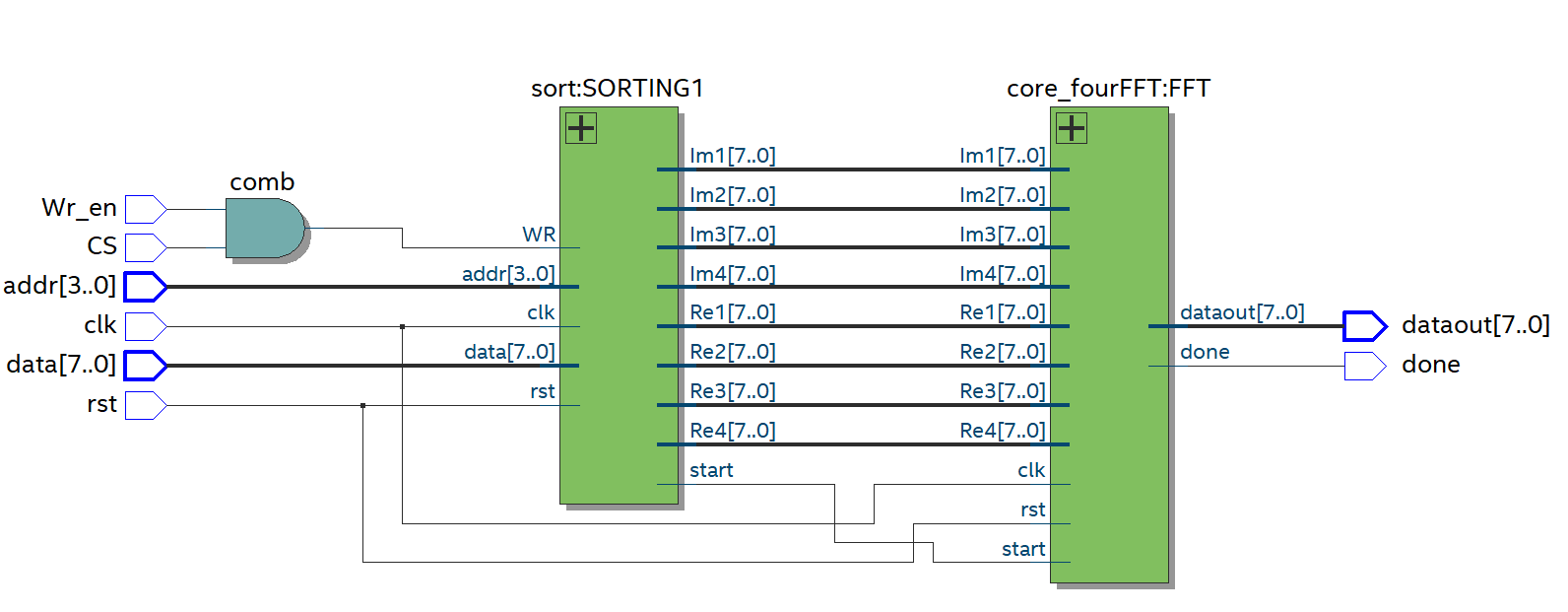
This method is the subset of the Cooley-Tukey method. In this method, N1 and N2 are chosen to be 2 and N/2 and it is assumed that N is a power of 2 [15][16][17]. As an example, for N = 8, N1 = 2 and N2 = 4 we can get the 8 points divide-and-conquer approach for calculating DFT as below:

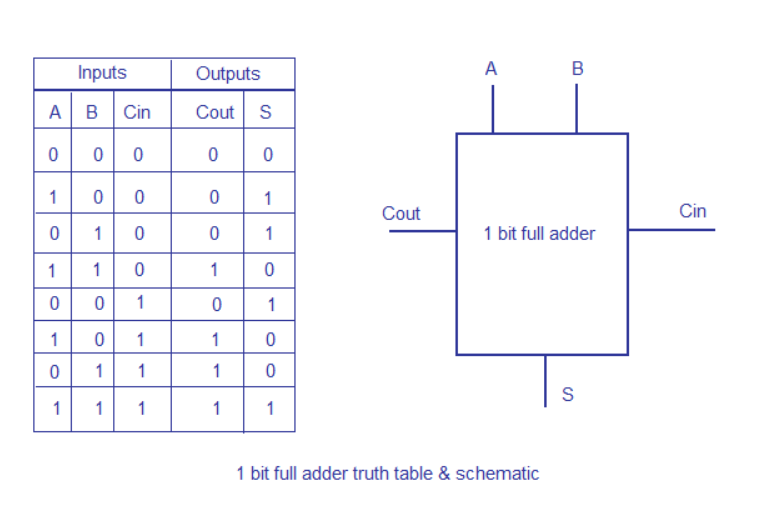
Figure 2. The block diagram of 8 points Radix-2 FFT algorithm

**Methodology**

In this section, we implemented the FFT algorithm to calculate addition / substraction and multiplication for 4 inputs of complex number data in VHDL on Quartus with the overall building methodology block diagram as below:

Figure 3. Overall block diagram for the implemented FFT algorithm to calculate 4 complex input datas.

**I/ Complex Adder-Substractor**

***a) Ripple carry adder***

Base on the 1 bit Full Adder truth table:

Figure 4. Block diagram of a 1-bit Full Adder

***b) Adder-Subtractor***

Consider two 4-bit binary numbers A and B as inputs to the Digital Circuit for the operation with digits

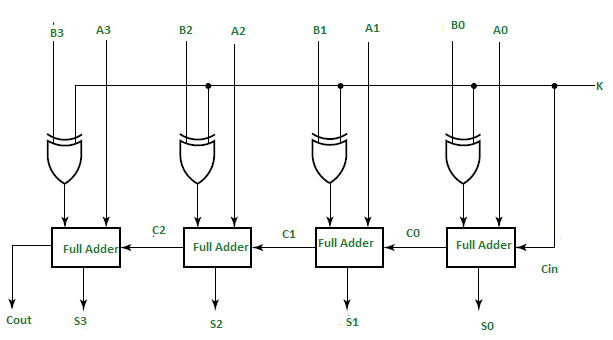


Figure 5. Block diagram of 4-bit binary Adder-Subtractor

In this figure, we have 4-bit input A and B with 1 control line K that holds a binary value of either 0 or 1 which determines that the operation being carried out is addition or subtraction.

If the value of K is 1, input B will become B’ and the operation will be A+B’+1. Thus, we performed 2’s complement subtraction for two numbers A and B. If the value of K is 0, input B will be kept and the operation will be A+B. Thus, we performed addition for two numbers A and B.

In this project, we designed 8-bit binary Adder-Subtractor that is capable of both addition and subtraction of binary numbers in one circuit itself. Our 8-bit inputs are in1 and in2. The control line called mode will determine the calculation whether it is addition with 0 or subtraction with 1.

AddSub8

8

8

8

in1

in2

mode

out

Figure 6. Block diagram of 8-bit binary Adder-Subtractor block diagram

***c) Implementation***

Consider the general case when we take addition and subtraction with 2 complex numbers (a+bj) and (c+dj).

AddSub8

8

8

8

a

c

Real (Re)

AddSub8

8

8

8

b

d

mode

Imaginary (Im)

Figure 7. Block diagram of complex adder-subtractor block diagram

**II/ Complex Multiplier**

1. ***8-bit signed binary Multiplier:***

We designed the 8-bit multiplication using weighted 2 state Booth algorithm. It is a method that help us speed up the signed multiplication by reducing the number of partial products. In this report, we will only mention the basic theory of this method. The table shows the conversion

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Yg+3 Yg+2 Yg+1 Yg Yg-1 | PP | Yg+1 Yg Yg-1 | Base PP | Yg+3 Yg+2 Yg+1 | Weighted PP | 2-stage PP (Base PP + Weighted PP) |
| 00000 | 0 | 000 | 0F | 000 | 0F | 0F |
| 00001 | +1 \* X | 001 | 1F | 1F |
| 00010 | +1 \* X | 010 | 1F | 1F |
| 00011 | +2 \* X | 011 | 2F | 2F |
| 00100 | +2 \* X | 100 | -2F | 001 | 4F | 2F |
| 00101 | +3 \* X | 101 | -1F | 3F |
| 00110 | +3 \* X | 110 | -1F | 3F |
| 00111 | +4 \* X | 111 | 0F | 4F |
| 01000 | +4 \* X | 000 | 0F | 010 | 4F | 4F |
| 01001 | +5 \* X | 001 | 1F | 5F |
| 01010 | +5 \* X | 010 | 1F | 5F |
| 01011 | +6 \* X | 011 | 2F | 6F |
| 01100 | +6 \* X | 100 | -2F | 011 | 8F | 6F |
| 01101 | +7 \* X | 101 | -1F | 7F |
| 01110 | +7 \* X | 110 | -1F | 7F |
| 01111 | +8 \* X | 111 | 0F | 8F |
| 10000 | -8 \* X | 000 | 0F | 100 | -8F | -8F |
| 10001 | -7 \* X | 001 | 1F | -7F |
| 10010 | -7 \* X | 010 | 1F | -7F |
| 10011 | -6 \* X | 011 | 2F | -6F |
| 10100 | -6 \* X | 100 | -2F | 101 | -4F | -6F |
| 10101 | -5 \* X | 101 | -1F | -5F |
| 10110 | -5 \* X | 110 | -1F | -5F |
| 10111 | -4 \* X | 111 | 0F | -4F |
| 11000 | -4 \* X | 000 | 0F | 110 | -4F | -4F |
| 11001 | -3 \* X | 001 | 1F | -3F |
| 11010 | -3 \* X | 010 | 1F | -3F |
| 11011 | -2 \* X | 011 | 2F | -2F |
| 11100 | -2 \* X | 100 | -2F | 111 | 0F | -2F |
| 11101 | -1 \* X | 101 | -1F | -F |
| 11110 | -1 \* X | 110 | -1F | -F |
| 11111 | 0 | 111 | 0F | 0F |

Table 1. Weighted 2 state Booth algorithm conversion table

Instead of having 8 partial products in traditional mutiplication (array multiplication), we only create 2 partial products if applying this method.

x7x6x5x4x3x2x1x00

When taking sum of 2 partial products, we use 16-bit Ripple carry adder

The design of Weighted 2 stage Booth algorithm:

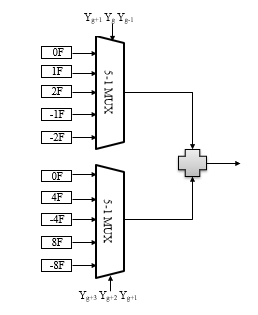


Figure 8. Weighted 2 stage Booth block diagram

***b) Implementation***

Consider the general case when we take multiplication with 2 complex numbers (a+bj) and (c+dj).

a

c

b

d

8-bit Multiplier

8

8

a

c

16

ac

8-bit Multiplier

8

8

b

d

16

bd

8-bit Multiplier

8

8

a

d

16

ad

8-bit Multiplier

8

8

b

c

16

bc

AddSub16

AddSub16

mode =1 (subtraction)

mode =0 (addition)

16

ac-bd

ad+bc

16

Figure 9. Complex Multiplier block diagram

**III/ FFT Butterfly**

***a) Fast Fourier Transform***

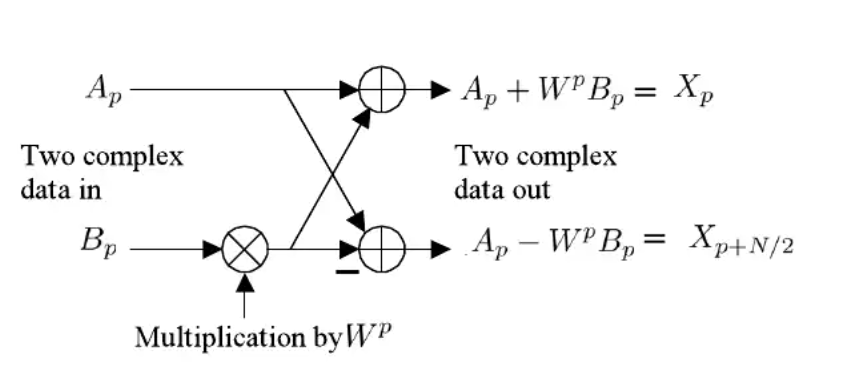


Figure 10. FFT Butterfly structure

***b) Implementation***

We will create FFT butterfly structure base on general complex data inputs which are Ap and Bp and coefficient Wp

Re (Ap+ Bp.Wp)

8

8

Im (Ap)

Complex Mutliplier

Re (Bp)

Re (Wp)

Im (Wp)

Im (Bp)

8

8

8

8

ComplexAddSub

ComplexAddSub

8

8

Re (Ap)

Re (Bp.Wp)

Im (Bp.Wp)

8

8

8

mode = 0 (addition)

8

mode = 1 (subtraction)

8

8

8

8

Im (Ap+ Bp.Wp)

Re (Ap - Bp.Wp)

Im (Ap - Bp.Wp)

Figure 11. FFT Butterfly block diagram

**IV/ 4-FFT**

Consider general case that we have 4 inputs of complex number.

a

c

a+c

a-c

b+d

b-d

b

d

2-DFT

2-DFT

1

-j

=

=

a+c

a-c

b+d

-(b-d)j

-

a+c+b+d

a-c-(b-d)j

a+c-b-d

a-c+(b-d)j

We will take 4 blocks of FFT Butterfly to do the calculation

FFT

Butterfly

Re (a)

Im (a)

Im (c)

Re (Wp)= 1

Im (Wp)= 0

Re (c)

8

8

8

8

8

8

FFT

Butterfly

Re (b)

Im (b)

Im (d)

Re (Wp)= 1

Im (Wp)= 0

Re (d)

8

8

8

8

8

8

FFT

Butterfly

Re (a-c)

Im (a-c)

Im (b-d)

Re (Wp)= 0

Im (Wp)= -1

Re (b-d)

8

8

8

8

8

8

FFT

Butterfly

Re (a+c)

Im (a+c)

Im (b+d)

Re (Wp)= 1

Im (Wp)= 0

Re (b+d)

8

8

8

8

8

8

Re (a+c+b+d)

Im (a+c+b+d)

8

8

Re (a+c-b-d)

Im (a+c-b-d)

8

8

Re (a-c-(b-d)j)

Im (a-c-(b-d)j)

8

8

Re (a-c+(b-d)j)

Im (a-c+(b-d)j)

8

8

Figure 12. 4-FFT Block diagram

**V/ Core block**

***a) First part***

We will use 1 multiplexer 8 to 1 to take out each calculated result at a time (select pin will run from 0 to 7)

4 – FFT Block diagram

Re (a)

8

Im (a)

8

Re (c)

8

Im (c)

8

Re (b)

8

Im (b)

8

Re (d)

8

Im (d)

8

Re (1st result)

8

Im (1st result)

8

Re (2nd result)

8

Im (2nd result)

8

Re (3rd result)

8

Re (4th result)

8

Im (3rd result)

8

Im (4th result)

8

dataout

8

select

Mux 8-1

Figure 13. First part of the core

***b) Second part***

First part

Re (a)

8

Im (a)

8

Re (b)

8

Im (b)

8

Re (d)

8

Im (d)

8

Im (c)

8

Re (c)

8

dataout

8

select

Second part

done

clk

rst

start

Figure 14. Core block diagram

In general, we will have 8 inputs which are for real parts and imaginary parts of 4 complex numbers when calculating 4-FFT. Besides, 3 inputs which are clk, rst, start will be integrated in the second part which will control the block to get the results. Output done will be used for checking whether we have results.

The table below will show you the purpose of each pin

|  |  |  |  |
| --- | --- | --- | --- |
| Logic\Pin | rst | start | done |
| 0 | Normal operation | Not calculating | Not having results |
| 1 | All input values stored in the core and Start pin become 0 again | Begin calculating | Having results |

Table 2. Purposes and asignment of pins

**C Code**

Firstly, the overall block diagram is shown. In this figure, we can see that CPU send 5 signals to wrapped IP: address, data to send, write enable to write to register files, chip select to choose the wrapped IP and reset.

In the wrapped IP, register file interacts with 4-FFT calculation block to get the final result

DATAA

WR\_EN

CPU

ADDR

CS

RESET

REGISTER FILE

CALCULATION

DATAOUT

DONE

Figure 15. Block diagram of connection between CPU and IP

Next, we will explain deeply the structure of the wrapped IP and how we apply C code to check the result.

CALCULATION

dataout register

input registers

R: reset bit

W: write enable bit

CS: chip select bit

D: done bit

S: start bit

|  |  |  |
| --- | --- | --- |
| Register Files | | |
| Address | Data | |
| 0x12 |  | R |
| 0x11 |  | W |
| 0x10 |  | CS |
| . |  | |
| . |  | |
| . |  | |
| 0x0A |  | |
| 0x09 |  | D |
| 0x08 |  | S |
| 0x07 |  | |
| 0x06 |  | |
| 0x05 |  | |
| 0x04 |  | |
| 0x03 |  | |
| 0x02 |  | |
| 0x01 |  | |
| 0x00 |  | |

Figure 16. IP structure

Yes

While done bit is 1

Set start bit to 1

Set reset bit to 1

Set write enable bit to 0

Take the result from dataout register

Set write enable bit to 1

Set reset bit to 0

Set chip select bit to 1

No

While not at the end of input file

Take 4 inputs

No

Figure 17. C code block diagram

**Simulation**

Due to the limit in the range of time in Simulation Waveform Editor, we can only apply limited cases for simulation (in this case is 2)

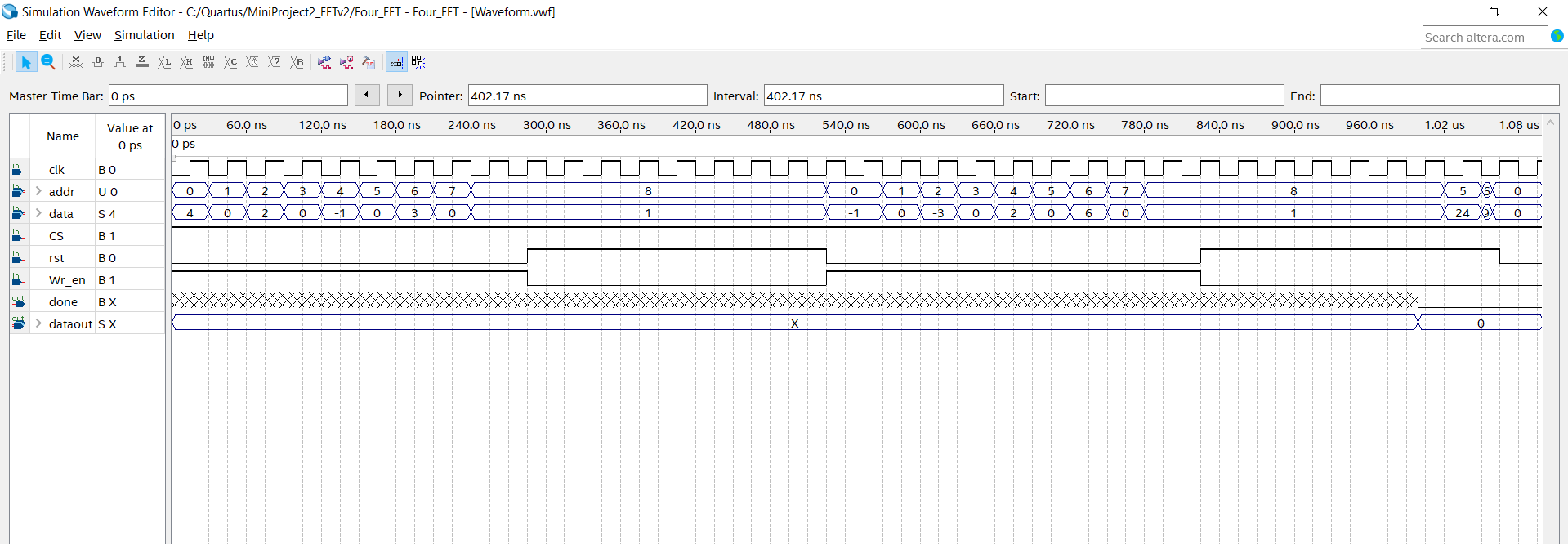


Figure 18. Simulation set up

**Waveform explaination:**

- Addr: We will have 9 addresses from 0 to 8 to write data in. Addresses 0,2,4,6 will contain values for real parts of 4 inputs and addresses 1,3,5,7 will contain values for imaginary parts of 4 inputs. Address 8 will become start register.

- Data: We will write each value into each address from 0 to 7. After that we will write value 1 into address 8, which means turning on start signal to begin calculating result.

- Rst: After setting address 8 to value 1 for one cycle which is amount of time for calculating result, we assigned reset signal to 1 to set all the values in addresses to 0.

-Wr\_en: Signal will become 0 when reset signal is 1 for making sure that we do not write values into addresses when taking out the result. Signal will become 1 when we want to write values into addresses.

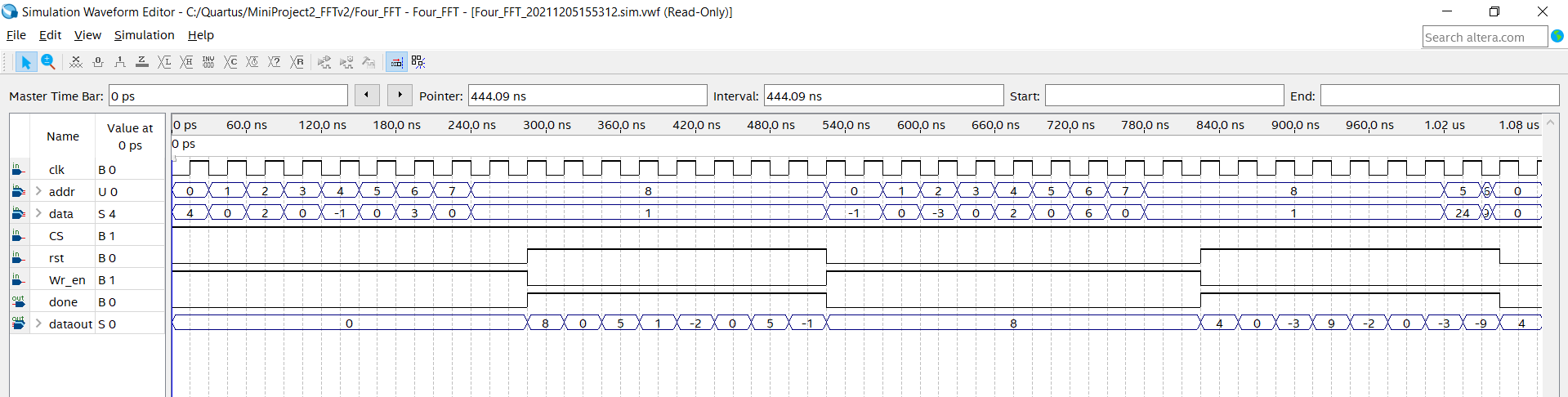
****

Figure 19. Simulation Result

**Waveform explanation:**

-Done: Signal will be 1 when we taking out the calculated results and 0 when we prepare and do calculation.

-Dataout: We will taking out each calculated result in order from real part and imaginary part of first result to the real part and imaginary part of the last results.

**Appendix**

C program for controlling input registers

**#include** <iostream>

**#include** <string>

**#include** <fstream>

**#include** <sstream>

**#include** <math.h>

**#define** re1 (\*(**volatile** **int**\*)0x0000000) //register of first number's real

**#define** im1 (\*(**volatile** **int**\*)0x0000001) //register of first number's imaginary

**#define** re2 (\*(**volatile** **int**\*)0x0000002) //register of second number's real

**#define** im2 (\*(**volatile** **int**\*)0x0000003) //register of second number's imaginary

**#define** re3 (\*(**volatile** **int**\*)0x0000004) //register of third number's real

**#define** im3 (\*(**volatile** **int**\*)0x0000005) //register of third number's imaginary

**#define** re4 (\*(**volatile** **int**\*)0x0000006) //register of fourth number's real

**#define** im4 (\*(**volatile** **int**\*)0x0000007) //register of fourth number's imaginary

**#define** start (\*(**volatile** **int**\*)0x0000008) //start register

**#define** done (\*(**volatile** **int**\*)0x0000009) //done register

**#define** dataout (\*(**volatile** **int**\*)0x000000A) //dataout register

**#define** cs (\*(**volatile** **int**\*)0x0000010) //chip select register

**#define** wr\_e (\*(**volatile** **int**\*)0x0000011) //write enable register

**#define** rst (\*(**volatile** **int**\*)0x0000012) //reset register

**int** **main**(){

**int** \*t=&cs;

\*t=1; //Turn on chip select

**int** \*p=&re1;

**int** samples[4] = { 0,0,0,0 };

**int** x;

ifstream input("C:\Users\Admin\Desktop\Test\input1.txt");// read data

**while** (input >> samples[0] >> samples[1] >> samples[2] >> samples[3])

{ // read data until the end of input file

\*(t+1)=1; //turn on write enable

\*(t+2)=0; //Turn off reset

\*p=samples[0]; //Write first number's real to register

\*(p+1)=0; //Write first number's imaginary to register

\*(p+2)=samples[1];//Write second number's real to register

\*(p+3)=0; //Write second number's imaginary to register

\*(p+4)=samples[2];//Write third number's real to register

\*(p+5)=0; //Write third number's imaginary to register

\*(p+6)=samples[3];//Write fourth number's real to register

\*(p+7)=0; //Write fourth number's imaginary to register

\*(p+8)=1; //Set start register to 1 to begin calculating after getting 8 inputs

delay\_ms(0,00003);

**while**(\*(p+9)){

\*(t+2)=1; //Turn on reset

\*(t+1)=0; //turn off write enable

x=\*(p+10);

**printf**("%d\n",x); //print results

delay\_ms(0,00003);

}

}

}

C program for generate input complex datas for simulating