CHAPTER 1

INTRODUCTION

1.1 FAST FOURIER TRANSFORM

A Fast Fourier Transform (FFT) is an efficient algorithm to compute the Discrete Fourier Transform (DFT) and its inverse. There are many distinct FFT algorithms involving a wide range of mathematics, from simple complex-number arithmetic to group theory and number theory. The fast Fourier Transform is a highly efficient procedure for computing the DFT of a finite series and requires less number of computations than that of direct evaluation of DFT. It reduces the computations by taking advantage of the fact that the calculation of the coefficients of the DFT can be carried out iteratively. Due to this, FFT computation technique is used in digital spectral analysis, filter simulation, autocorrelation and pattern recognition.

The FFT is based on decomposition and breaking the transform into smaller transforms and combining them to get the total transform. FFT reduces the computation time required to compute a discrete Fourier transform and improves the performance by a factor of 100 or more over direct evaluation of the DFT.

A DFT decomposes a sequence of values into components of different frequencies. This operation is useful in many fields but computing it directly from the definition is often too slow to be practical. An FFT is a way to compute the same result more quickly: computing a DFT of N points in the obvious way, using the definition, takes $O(N^2)$ arithmetical operations, while an FFT can compute the same result in only $O(N \log N)$ operations.

The difference in speed can be substantial, especially for long data sets where N may be in the thousands or millions—in practice, the computation time can be reduced by

several orders of magnitude in such cases, and the improvement is roughly proportional to N /log (N). This huge improvement made many DFT-based algorithms practical. FFT's are of great importance to a wide variety of applications, from digital signal processing and solving partial differential equations to algorithms for quick multiplication of large integers.

The most well known FFT algorithms depend upon the factorization of N, but there are FFT with O (N log N) complexity for all N, even for prime N. Many FFT algorithms only depend on the fact that is an N th primitive root of unity, and thus can be applied to analogous transforms over any finite field, such as number-theoretic transforms.

The Fast Fourier Transform algorithm exploit the two basic properties of the twiddle factor - the symmetry property and periodicity property which reduces the number of complex multiplications required to perform DFT.

FFT algorithms are based on the fundamental principle of decomposing the computation of discrete Fourier Transform of a sequence of length N into successively smaller discrete Fourier transforms. There are basically two classes of FFT algorithms.

A) Decimation In Time (DIT) algorithm

B) Decimation In Frequency (DIF) algorithm.

In decimation-in-time, the sequence for which we need the DFT is successively divided into smaller sequences and the DFTs of these subsequences are combined in a certain pattern to obtain the required DFT of the entire sequence. In the decimation-in-frequency approach, the frequency samples of the DFT are decomposed into smaller and smaller subsequences in a similar manner.

The number of complex multiplication and addition operations required by the simple forms both the Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT) is of order N^2 as there are N data points to calculate, each of which requires N complex arithmetic operations.

The discrete Fourier transform (DFT) is defined by the formula:

$$X(K) = \sum_{n=0}^{N-1} x(n) \bullet e^{\frac{-j2\Pi nK}{N}};$$

Where K is an integer ranging from 0 to $N - 1$.

The algorithmic complexity of DFT will O(N2) and hence is not a very efficient method. If we can't do any better than this then the DFT will not be very useful for the majority of practical DSP application. However, there are a number of different 'Fast Fourier Transform' (FFT) algorithms that enable the calculation the Fourier transform of a signal much faster than a DFT. As the name suggests, FFTs are algorithms for quick calculation of discrete Fourier transform of a data vector. The FFT is a DFT algorithm which reduces the number of computations needed for N points from O(N *2*) to O(N log N) where log is the base-2 logarithm. If the function to be transformed is not harmonically related to the sampling frequency, the response of an FFT looks like a 'sinc' function (sin x) / x.

The Radix-2 DIT algorithm rearranges the DFT of the function x_n into two parts: a sum over the even-numbered indices $n = 2m$ and a sum over the odd-numbered indices $n = 2m + 1$:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k}.$$

One can factor a common multiplier $e^{-\frac{2\pi i}{N}k}$ out of the second sum in the equation. It is the two sums are the DFT of the even-indexed part x_{2m} and the DFT of

$$X_k = \underbrace{\sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2}mk}}_{\text{DFT of even-indexed part of } x_m} + e^{-\frac{2\pi i}{N}k} \underbrace{\sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2}mk}}_{\text{DFT of odd-indexed part of } x_m} = E_k + e^{-\frac{2\pi i}{N}k} O_k.$$

odd-indexed part x_{2m+1} of the function x_n. Denote the DFT of the ***E***ven-indexed inputs x_{2m} by E_k and the DFT of the ***O***dd-indexed inputs x_{2m+1} by O_k and we obtain:

However, these smaller DFTs have a length of $N/2$, so we need compute only $N/2$ outputs: thanks to the periodicity properties of the DFT, the outputs for $N/2 \leq k < N$ from a DFT of length $N/2$ are identical to the outputs for $0 \leq k < N/2$. That is, $E_{k+N/2} = E_k$ and $O_{k+N/2} = O_k$. The phase factor $\exp[-2\pi ik/N]$ called a twiddle factor which obeys the relation: $\exp[-2\pi i(k+N/2)/N] = e^{-\pi i}\exp[-2\pi ik/N] = -\exp[-2\pi ik/N]$, flipping the sign of the $O_{k+N/2}$ terms. Thus, the whole DFT can be calculated as follows:

$$X_k = \begin{cases} E_k + e^{-\frac{2\pi i}{N}k} O_k & \text{if } k < N/2 \\ E_{k-N/2} - e^{-\frac{2\pi i}{N}(k-N/2)} O_{k-N/2} & \text{if } k \geq N/2. \end{cases}$$

This result, expressing the DFT of length N recursively in terms of two DFTs of size $N/2$, is the core of the radix-2 DIT fast Fourier transform. The algorithm gains its speed by re-using the results of intermediate computations to compute multiple DFT outputs. Note that final outputs are obtained by a +/− combination of E_k and $O_k\exp(-2\pi ik/N)$, which is simply a size-2 DFT; when this is generalized to larger radices below, the size-2 DFT is replaced by a larger DFT (which itself can be evaluated with an FFT).

This process is an example of the general technique of divide and conquers algorithms. In many traditional implementations, however, the explicit recursion is avoided, and instead one traverses the computational tree in breadth-first fashion.

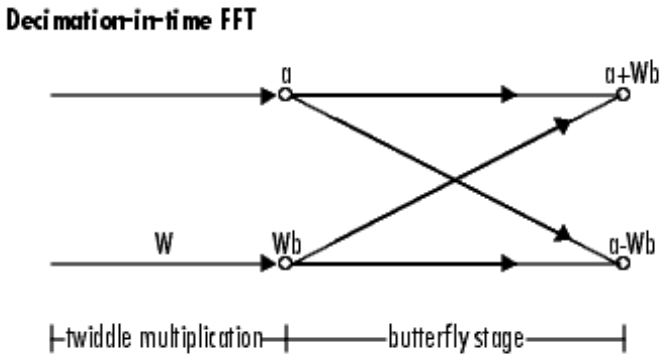


Fig 1.1 Decimation In Time FFT

In the DIT algorithm, the twiddle multiplication is performed before the butterfly stage whereas for the DIF algorithm, the twiddle multiplication comes after the Butterfly stage.

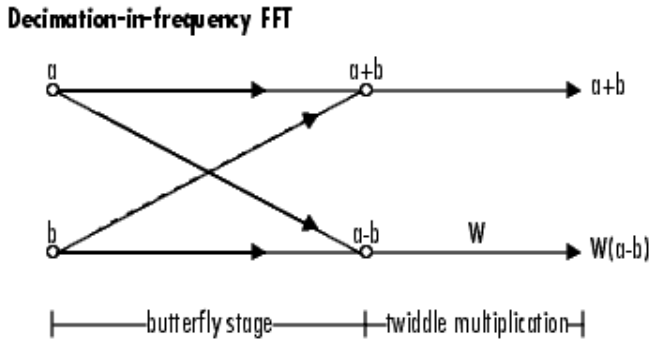


Fig 1.2 : Decimation In Frequency FFT

The 'Radix 2' algorithms are useful if *N* is a regular power of 2 (*N*=2*p*). If we assume that algorithmic complexity provides a direct measure of execution time and that the relevant logarithm base is 2 then as shown in table 1.1, ratio of execution times for the (DFT) vs. (Radix 2 FFT) increases tremendously with increase in N.

The term 'FFT' is actually slightly ambiguous, because there are several commonly used 'FFT' algorithms. There are two different Radix 2 algorithms, the so-called 'Decimation in Time' (DIT) and 'Decimation in Frequency' (DIF) algorithms. Both of these rely on the recursive decomposition of an *N* point transform into 2 (*N*/2) point transforms.

Number of Points, N	Complex Multiplications in Direct computations, N^2	Complex Multiplication in FFT Algorithm, (N/2) $\log_2 N$	Speed improvement Factor
4	16	4	4.0
8	64	12	5.3
16	256	32	8.0
32	1024	80	12.8
64	4096	192	21.3
128	16384	448	36.6

Table 1.1: Comparison of Execution Times, DFT & Radix – 2 FFT

1.2 BUTTERFLY STRUCTURES FOR FFT

Basically FFT algorithms are developed by means of divide and conquer method, the is depending on the decomposition of an N point DFT in to smaller DFT's. If N is factored as N = r_1,r_2,r_3 ..r_L where $r_1=r_2=\ldots=r_L=r$, then r_L =N. where r is called as Radix of FFFt algorithm.

If r= 2, then if is called as radix-2 FFT algorithm,. The basic DFT is of size of 2. The N point DFT is decimated into 2 point DFT by two ways such as Decimation In Time (DIT) and Decimation In Frequency (DIF) algorithm. Both the algorithm take the advantage of periodicity and symmetry property of the twiddle factor.

$$W_N^{nK} = e^{\frac{-j2\Pi nK}{N}}$$

The radix-2 decimation-in-frequency FFT is an important algorithm obtained by the divide and conquers approach. The Fig. 1.2 below shows the first stage of the 8-point

DIF algorithm.

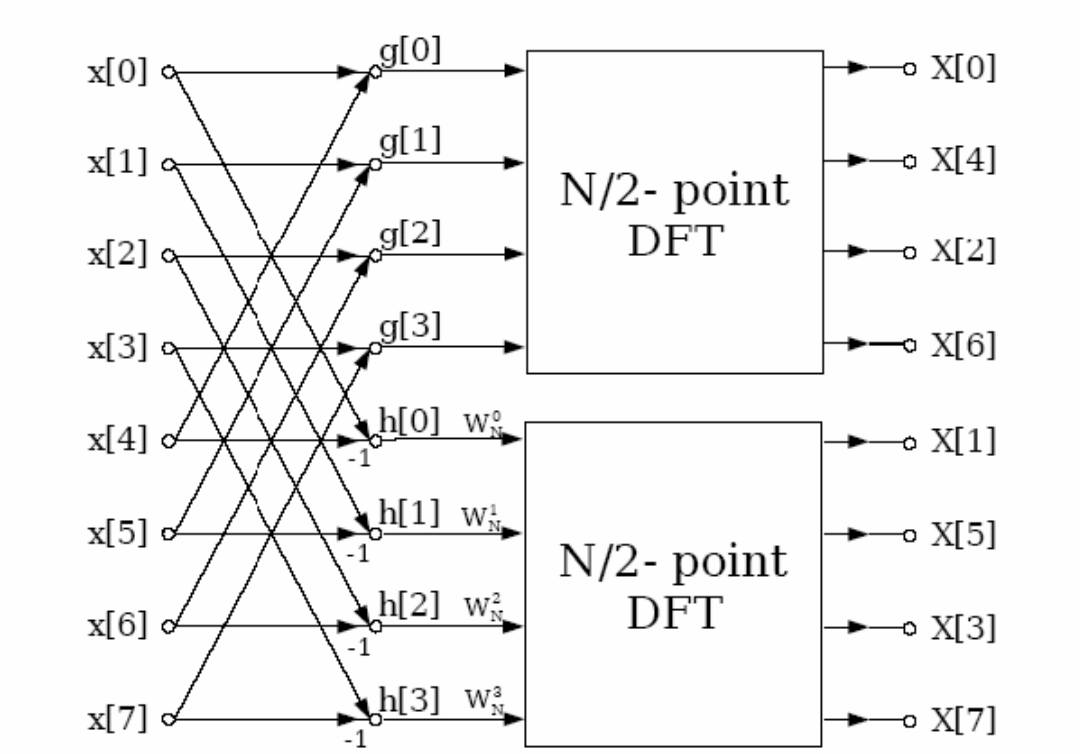


Fig. 1.1: First Stage of 8 point Decimation in Frequency Algorithm.

The decimation, however, causes shuffling in data. The entire process involves *v* = log2 *N* stages of decimation, where each stage involves *N*/2 butterflies of the type shown in the Fig. 1.3.

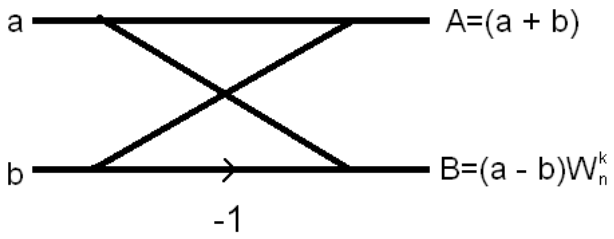


Fig. 1.4: Butterfly Scheme.

Here $W_n^k = e^{\frac{-j2\Pi nk}{N}}$ is the Twiddle factor.

Consequently, the computation of N-point DFT via this algorithm requires (*N*/2) log2 *N* complex multiplications. For illustrative purposes, the eight-point decimation-in frequency algorithm is shown in the Figure below. We observe, as previously stated, that the output sequence occurs in bit-reversed order with respect to the input. Furthermore, if we abandon the requirement that the computations occur in place, it is also possible to have both the input and output in normal order. The 8 point Decimation In frequency algorithm is shown in Fig 1.5.

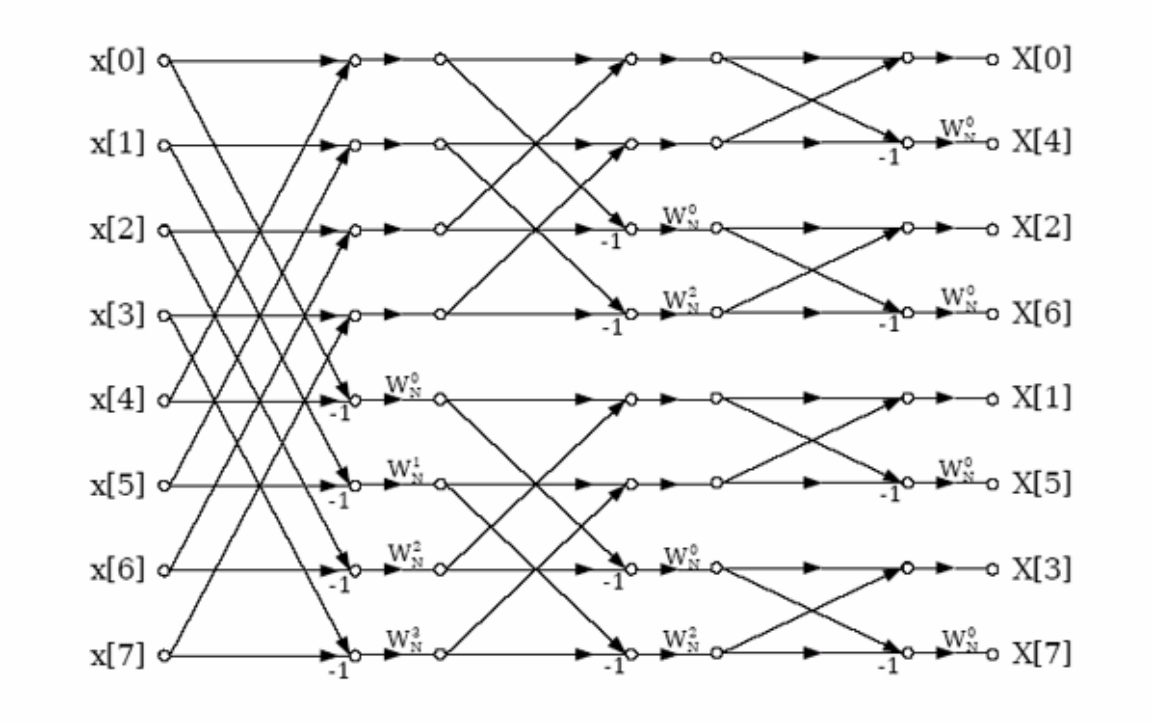


Fig. 1.5: 8 point Decimation in Frequency Algorithm

CHAPTER 2

CHAPTER 2

HARDWATE DESCRIPTION LANGUAGE

2.1 INTRODUCTION

Hardware Description Language (HDL) is a language that can describe the behavior and structure of electronic system, but it is particularly suited as a language to describe the structure and the behavior of the digital electronic hardware design, such as ASICs and FPGAs as well as conventional circuits. HDL can be used to describe electronic hardware at many different levels of abstraction such as Algorithm, Register transfer level (RTL) and Gate level. Algorithm is un synthesizable, RTL is the input to the synthesis, and Gate Level is the input from the synthesis. It is often reported that a large number of ASIC designs meet their specification first time, but fail to work when plunged into a system. HDL allows this issue to be addressed in two ways, a HDL specification can be executed in order to achieve a high level of confidence in its correctness before commencing design and may simulate one specification for a part in the wider system context(Eg:- Printed Circuited Board Simulation). This depends upon how accurately the specialization handles aspects such as timing and initialization.

2.2 ADVANTAGES OF HDL

A design methodology that uses HDLs has several fundamental advantages over traditional Gate Level Design Methodology. The following are some of the advantages:

- One can verify functionality early in the design process and immediately simulate the design written as a HDL description. Design simulation at this high level, before implementation at the Gate Level allows testing architectural and designing decisions.

- FPGA synthesis provides logic synthesis and optimization, so one can automatically convert a VHDL description to gate level implementation in a given technology.
- HDL descriptions provide technology independent documentation of a design and its functionality. A HDL description is more easily read and understood than a net-list or schematic description.
- HDLs typically support a mixed level description where structural or net-list constructs can be mixed with behavioral or algorithmic descriptions. With this mixed level capabilities one can describe system architectures at a high level or gate level implementation.

2.3 VHDL

VHDL is a hardware description language. It describes the behavior of an electronic circuit or system, from which the physical circuit or system can then be attained.

VHDL stands for VHSIC Hardware Description Language. VHSIC is itself an abbreviation for Very High Speed Integrated Circuits, an initiative funded by United States Department of Defense in the 1980s that led to creation of VHDL. Its first version was VHDL 87, later upgraded to the VHDL 93. VHDL was the original and first hardware description language to be standardized by Institute of Electrical and Electronics Engineers, through the IEEE 1076 standards. An additional standard, the IEEE 1164, was later added to introduce a multi-valued logic system.

VHDL is intended for circuit synthesis as well as circuit simulation. However, though VHDL is fully simulatable, not all constructs are synthesizable. The two main immediate applications of VHDL are in the field of Programmable Logic Devices and in the field of ASICs (Application Specific Integrated Circuits). Once the VHDL code has been written, it can be used either to implement the circuit in a programmable device or

can be submitted to a foundry for fabrication of an ASIC chip.

VHDL is a fairly general-purpose language, and it doesn't require a simulator on which to run the code. There are many VHDL compilers, which build executable binaries. It can read and write files on the host computer, so a VHDL program can be written that generates another VHDL program to be incorporated in the design being developed. Because of this general-purpose nature, it is possible to use VHDL to write a *test bench* that verifies the functionality of the design using files on the host computer to define stimuli, interacts with the user, and compares results with those expected.

The key advantage of VHDL when used for systems design is that it allows the behavior of the required system to be described (modeled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires). The VHDL statements are inherently concurrent and the statements placed in a PROCESS, FUNCTION or PROCEDURE are executed sequentially.

2.4 EDA Tools

There are several EDA (Electronic Design Automation) tool available for circuit synthesis, implementation and simulation using VHDL. Some tools are offered as part of a vendor’s design suite such as Altera’s Quatus II which allows the synthesis of VHDL code onto Altera’s CPLD/FPGA chips, or Xilinx’s ISE suite, for Xilinx’s CPLD/FPGA chips.

ISE® *WebPACK*™ design software is the industry´s only FREE, fully featured front-to-back FPGA design solution for Linux, Windows XP, and Windows Vista. ISE *WebPACK* is the ideal downloadable solution for FPGA and CPLD design offering HDL synthesis and simulation, implementation, device fitting, and JTAG programming. ISE *WebPACK* delivers a complete, front-to-back design flow providing instant access to the ISE features and functionality at no cost. Xilinx has created a solution that allows convenient productivity by providing a design solution that is always up to date with error-free downloading and single file installation.

CHAPTER 3

DESIGN OF FFT

3.1 IMPLEMENTATION OF 16-POINT FFT BLOCKS

The FFT computation is accomplished in three stages. The x(0) until x(15) variables are denoted as the input values for FFT computation and X(0) until X(15) are denoted as the outputs. The pipeline architecture of the 16 point FFT is shown in Fig 4.1 consisting of butterfly schemes in it. There are two operations to complete the computation in each stage.

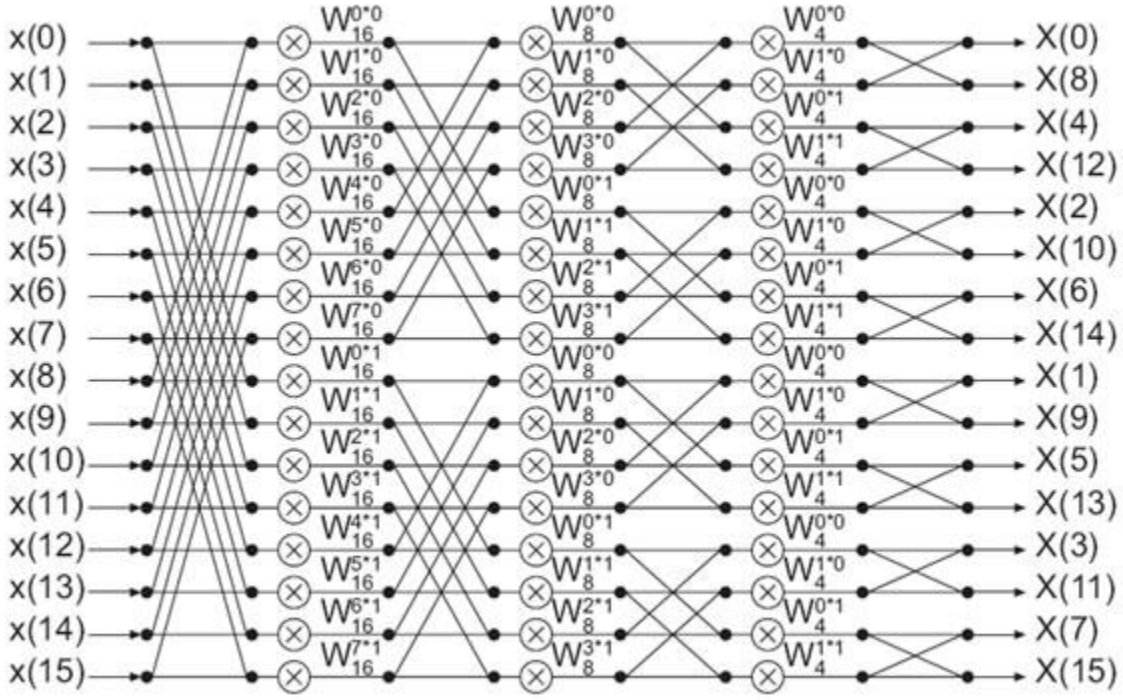


Fig 3.1: Architecture of 16 point FFT.

The upward arrow will execute addition operation while downward arrow will execute subtraction operation. The subtracted value is multiplied with twiddle factor

value before being processed into the next stage. This operation is done concurrently and is known as butterfly process.

The implementation of FFT flow graph in the VHDL requires three stages, final computation is done and the result is sent to the variable Y (0) to Y (15). Equation in each stage is used to construct scheduling diagram.

For stage one, computation is accomplished in three clock cycles denoted as S0 to S2.The operation is much simpler compared with FFT. This is because FFT processed both real and imaginary value. The result from FFT is represented in real and imaginary value because of the multiplication of twiddle factor. Twiddle factor is a constant defined by the number of point used in this transform. This scheduling diagram is derived from the equations obtain in FFT signal flow graph. The rest of the scheduling diagrams can be sketched in the same way as shown in figure 4.2. Thus each stage requires a clock cycle and totally three clock cycles are needed. Scheduling diagrams are a part of behavioral modeling and Synthesis steps to translate the algorithmic description into RTL (register transfer level) in VHDL design.

3.2 DESIGN OF A GENERAL RADIX-2 FFT USING VHDL

As we move to higher-point FFTs, the structure for computing the FFT becomes more complex and the need for an efficient complex multiplier to be incorporated within the butterfly structure arises. Hence we propose an algorithm for an efficient complex multiplier that overcomes the complication of using complex numbers throughout the process.

A radix-2 FFT can be efficiently implemented using a butterfly processor which includes, besides the butterfly itself, an additional complex multiplier for the twiddle factors.

A radix-2 butterfly processor consists of a complex adder, a complex subtraction, and a complex multiplier for the twiddle factors. The complex multiplication with the

twiddle factor is often implemented with four real multiplications and 2 add / subtract operations.

Normal Complex Operation:

$$(X+jY)\,(C+jS) = CX + jSX + jCY - YS$$

$$= CX - YS + j\,(SX + CY)$$

Real Part $R = CX - YS$

Imaginary Part $I = SX + CY$

Using the twiddle factor multiplier that has been developed, it is possible to design a butterfly processor for a radix-2 Cooley-Tukey FFT. Hence this basic structure of radix-2 FFT can be used as a building block to construct higher N-point FFTs. This structure has been developed as an extension to provide for the computation of higher value index FFTs.

CHAPTER 4

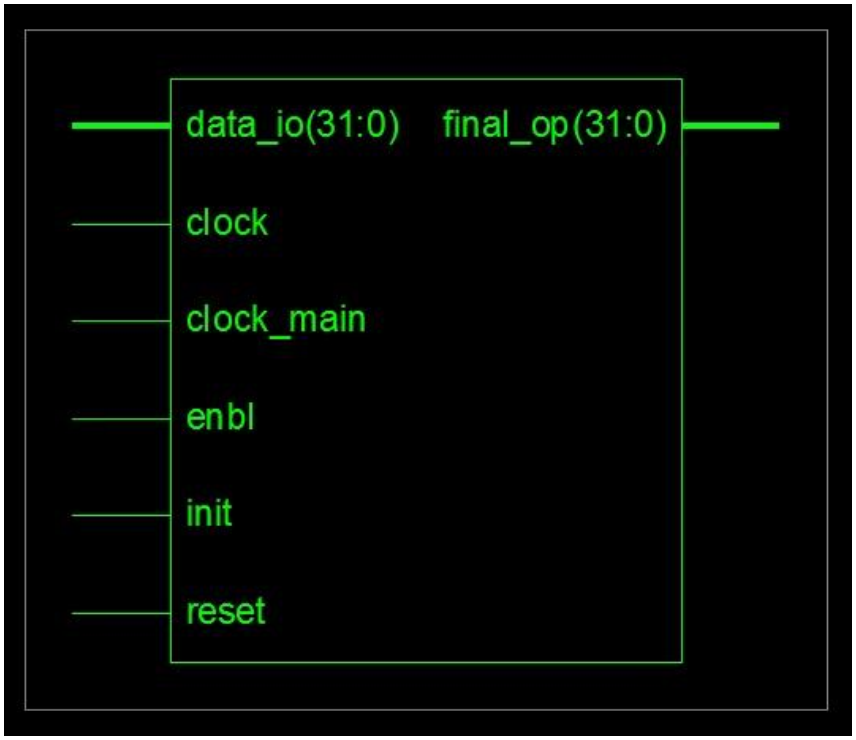
VHDL IMPLEMENTATION

4.1 DESIGN SOFTWARE

The implementations have been carried out using the software, Xilinx ISE 9.2i. The hardware language used is the Very High Speed Integrated Circuit Hardware Description Language (VHDL). VHDL is a widely used language for register transfer level description of hardware. It is used for design entry, compile and simulation of digital systems.

4.2 INTERFACE

The architectural design consist of data inputs, control unit, clocks and the data output. The register may be of the array of four or eight variable in the type of real. The FFT implementation in VHDL consists of three states such as start, load and run.



4.3 Design Summary

PROJ_FFT_II Project Status			
Project File:	proj_fft_II.ise	**Current State:**	Synthesized
Module Name:	synth_main	• **Errors:**	No Errors
Target Device:	xc3s200-4ft256	• **Warnings:**	1799 Warnings
Product Version:	ISE 9.2i	• **Updated:**	Sun Nov 27 20:39:39 2011

PROJ_FFT_II Partition Summary
No partition information was found.

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	0	1920	0%
Number of bonded IOBs	32	173	18%

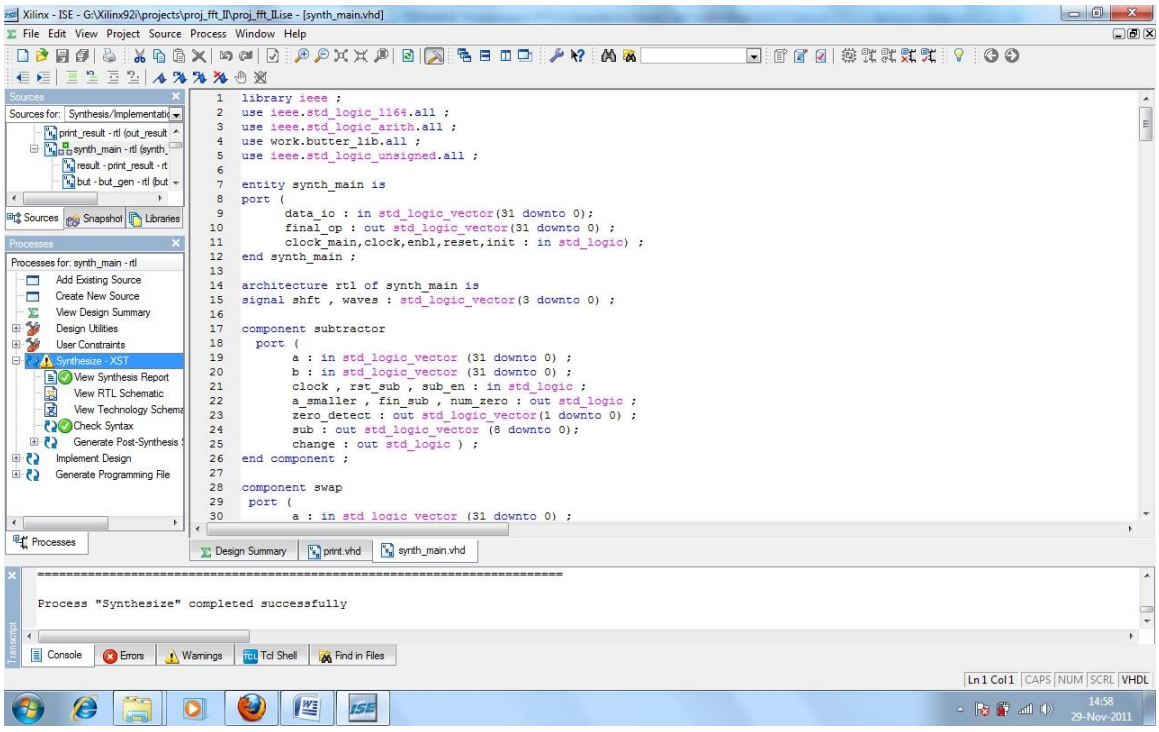
Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Sun Nov 27 17:16:18 2011	0	1799 Warnings	35 Infos
Translation Report					
Map Report					
Place and Route Report					
Static Timing Report					
Bitgen Report					

CHAPTER 5

RESULTS

The simulation of this whole project has been done using the Xilinx ISE of version 9.2i. Xilinx ISE is a simulation tool for programming {VLSI} {ASIC}s, {FPGA}s, {CPLD}s, and {SoC}s. It provides a comprehensive simulation and debug environment for complex ASIC and FPGA designs. Support is provided for multiple languages including Verilog, SystemVerilog, VHDL and SystemC.

5.1 SIMULATION RESULT OBTAINED



CHAPTER 6

CONCLUSION AND FUTURE SCOPE

6.1. CONCLUSION

This project describes the efficient use of VHDL code for the implementation of radix 2 based FFT architecture and the wave form result of the various stages has been obtained successfully. The accuracy in obtained results has been increased with the help of efficient coding in VHDL. The accuracy in results depends upon the equations obtained from the butterfly diagram and then on the correct drawing of scheduling diagrams based on these equations.

6.2. FUTURE SCOPE

The future scopes of this project are to implement the proposed FFT architecture using Field-Programmable Gate Arrays (FPGAs).

The FFT (Fast Fourier Transform) processor plays a critical part in speed and power consumption of the Orthogonal Frequency Division Multiplexing (OFDM) communication system. Thus the FFT block can be implemented in OFDM.

CHAPTER 7

VHDL CODE

Top rtl – synth_main.vhd

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity synth_main is
port (
      data_io : in std_logic_vector(31 downto 0);
      final_op : out std_logic_vector(31 downto 0) ;
      clock_main,clock,enbl,reset,init : in std_logic) ;
end synth_main ;

architecture rtl of synth_main is
signal shft , waves : std_logic_vector(3 downto 0) ;

component subtractor
  port (
       a : in std_logic_vector (31 downto 0) ;
       b : in std_logic_vector (31 downto 0) ;
       clock , rst_sub , sub_en : in std_logic ;
       a_smaller , fin_sub , num_zero : out std_logic ;
       zero_detect : out std_logic_vector(1 downto 0) ;
       sub : out std_logic_vector (8 downto 0);
       change : out std_logic ) ;
end component ;

component swap
 port (
       a : in std_logic_vector (31 downto 0) ;
       b : in std_logic_vector (31 downto 0) ;
       clock : in std_logic ;
       rst_swap , en_swap : in std_logic ;
       finish_swap : out std_logic ;
       d : out std_logic_vector (31 downto 0) ;
       large_exp : out std_logic_vector (7 downto 0) ;
       c  : out std_logic_vector (32 downto 0 ) ) ;
end component ;


component shift2
 port (
       sub_control : in std_logic_vector (8 downto 0) ;
       c_in  : in std_logic_vector (32 downto 0) ;
```

```
       shift_out : out std_logic_vector (31 downto 0) ;
       clock , shift_en , rst_shift : in std_logic ;
       finish_out : out std_logic ) ;
end component ;

component control_main
 port (
       a_small , sign_a , sign_b : in std_logic ;
       sign_out , add_sub , reset_all : out std_logic ;
       en_sub , en_swap , en_shift , addpulse , normalise : out
std_logic ;
       fin_sub , fin_swap , finish_shift , add_finish , end_all :
in std_logic ;
       clock_main , clock , reset , enbl , zero_num , change : in
std_logic ) ;
end component ;

component summer
 port (
       num1 , num2 : in std_logic_vector (31 downto 0) ;
       exp : in std_logic_vector (7 downto 0) ;
       addpulse_in , addsub , rst_sum : in std_logic ;
       add_finish : out std_logic ;
       sumout : out std_logic_vector ( 32 downto 0) ) ;
end component ;

component normalize
 port (
      a , b : in std_logic_vector (31 downto 0) ;
      numb : in std_logic_vector (32 downto 0) ;
      exp : in std_logic_vector (7 downto 0) ;
      signbit , addsub , clock , en_norm , rst_norm  : in
std_logic  ;
      zero_detect : in std_logic_vector(1 downto 0) ;
      exit_n : out std_logic ;
      normal_sum : out std_logic_vector (31 downto 0) ) ;
end component ;

component but_gen
port (
      add_incr , add_clear , stagedone : in std_logic ;
      but_butterfly : out std_logic_vector(3 downto 0) ) ;
end component ;

component stage_gen
port (
      add_staged , add_clear : in std_logic ;
      st_stage : out std_logic_vector(1 downto 0) ) ;
end component ;

component iod_staged is
port (
```

```
      but_fly : in std_logic_vector(3 downto 0) ;
      stage_no : in std_logic_vector(1 downto 0) ;
      add_incr , io_mode  : in std_logic ;
      add_iod , add_staged , add_fftd : out std_logic ;
      butterfly_iod : out std_logic_vector(3 downto 0) ) ;
end component ;

component baseindex
port (
      ind_butterfly : in std_logic_vector(3 downto 0) ;
      ind_stage : in std_logic_vector(1 downto 0) ;
      add_fft : in std_logic ;
      fftadd_rd : out std_logic_vector(3 downto 0) ;
      c0 , c1 , c2 , c3 : in std_logic ) ;
end component ;

component ioadd_gen
port (
      io_butterfly : in std_logic_vector(3 downto 0) ;
      add_iomode , add_ip , add_op : in std_logic ;
      base_ioadd : out std_logic_vector(3 downto 0) ) ;
end component ;

component mux_add
port (
      a , b : in std_logic_vector(3 downto 0) ;
      sel : in std_logic ;
      q : out std_logic_vector(3 downto 0) ) ;
end component ;

component ram_shift
port (
      data_in : in std_logic_vector(3 downto 0) ;
      clock_main : in std_logic ;
      data_out : out std_logic_vector(3 downto 0) ) ;
end component ;

component cycles
port (
      clock_main , preset , c0_en , cycles_clear : in std_logic ;
      waves : out std_logic_vector(3 downto 0) ) ;
end component ;

component counter
port (
      c : out std_logic_vector(2 downto 0) ;
      disable , clock_main , reset : in std_logic) ;
end component ;


component mult_clock
port (
```

```
      clock_main , mult1_c0 , mult1_iomode , mult_clear : in
std_logic ;
      mult1_addincr : out std_logic ) ;
end component ;

component cont_gen
port (
      con_staged , con_iod , con_fftd , con_init : in std_logic ;
      con_ip , con_op , con_iomode , con_fft : out std_logic ;
      con_enbw , con_enbor , c0_enable , con_preset : out
std_logic ;
      con_clear , disable : out std_logic ;
      c0 , clock_main : in std_logic ;
      en_rom , en_romgen , reset_counter : out std_logic ;
      con_clkcount : in std_logic_vector(2 downto 0) ) ;
end component ;

component and_gates
port (
      waves_and : in std_logic_vector(3 downto 0) ;
      clock_main , c0_en : in std_logic ;
      c0,c1,c2,c3 : out std_logic ;
      c0_c1,c2_c3,c0_c2,c1_c3 : out std_logic ) ;
end component ;

component r_block
port (
       data : in std_logic_vector(31 downto 0) ;
       trigger : in std_logic ;
       r_out : out std_logic_vector(31 downto 0) ) ;
end component ;

component l_block
port (
       data_l : in std_logic_vector(31 downto 0) ;
       trigger_l : in std_logic ;
       l_out : out std_logic_vector(31 downto 0) ) ;
end component ;

component level_edge
 port (
       data_edge : in std_logic_vector(31 downto 0) ;
       trigger_edge : in std_logic ;
       edge_out : out std_logic_vector(31 downto 0) ) ;
end component ;

component mux
port (
       d0 , d1 : in std_logic_vector(31 downto 0) ;
       mux_out : out std_logic_vector(31 downto 0) ;
       choose : in std_logic ) ;
end component ;
```

```
component negate
port (
       neg_in : in std_logic_vector(31 downto 0) ;
       neg_en , clock_main : in std_logic ;
       neg_out : out std_logic_vector(31 downto 0) ) ;
end component ;

component multiply
port(
      num_mux , num_rom : in std_logic_vector(31 downto 0) ;
      clock  : in std_logic ;
      mult_out : out std_logic_vector(31 downto 0) ) ;
end component ;

component divide
port (
       data_in : in std_logic_vector(31 downto 0) ;
       data_out : out std_logic_vector(31 downto 0) ) ;
end component ;

component romadd_gen is
port (
      io_rom,c0,c1,c2,c3 : in std_logic ;
      stage_rom : in std_logic_vector(1 downto 0) ;
      butterfly_rom : in std_logic_vector(3 downto 0) ;
      romadd : out std_logic_vector(2 downto 0) ;
      romgen_en : in std_logic );
end component ;

component reg_dpram is
port (
      data_fft , data_io : in std_logic_vector (31 downto 0);
      q : out std_logic_vector (31 downto 0);
      clock , io_mode : in std_logic;
      we , re : in std_logic;
      waddress: in std_logic_vector (3 downto 0);
      raddress: in std_logic_vector (3 downto 0));
end component ;

component rom is
port (
      clock , en_rom : in std_logic ;
      romadd : in std_logic_vector(2 downto 0) ;
      rom_data : out std_logic_vector(31 downto 0) ) ;
end component ;

COMPONENT print_result is
     PORT(
          clock : IN std_logic;
          op : IN std_logic;
          fin_res : OUT std_logic_vector(31 downto 0);
```

```
          result : IN std_logic_vector(31 downto 0));
end component ;

begin

result: print_result port map (clock_main, op,final_op,ram_data
);
but : but_gen port map (incr , clear , staged ,butterfly_iod) ;
stg : stage_gen port map (staged , clear , stage) ;
iod_stgd : iod_staged port
map(butterfly_iod,stage,incr,io_mode,iod,staged,fftd,butterfly) ;
base : baseindex port map (butterfly , stage , fft_en , fftadd_rd
, c0 , c1 , c2 , c3) ;
ioadd : ioadd_gen port map (butterfly , io_mode , ip , op ,
io_add) ;
ram_shift1 : ram_shift port map (fftadd_rd , clock_main , shift1)
;
ram_shift2 : ram_shift port map (shift1 , clock_main , shft) ;
ram_shift3 : ram_shift port map (shft , clock_main , shift3) ;
ram_shift4 : ram_shift port map (shift3 , clock_main ,shift4) ;
ram_shift5 : ram_shift port map (shift4 , clock_main , shift5) ;
--ram_shift6 : ram_shift port map (shift5 , clock_main , shift6)
;
multx1 : mux_add port map (shift5 , io_add , io_mode , ram_wr) ;
multx2 : mux_add port map (fftadd_rd , io_add , io_mode , ram_rd)
;
cyc : cycles port map (clock_main , preset , c0_en , cyc_clear ,
waves) ;
gates : and_gates port
map(waves,clock_main,c0_en,c0,c1,c2,c3,c0_c1,c2_c3,c0_c2,c1_c3) ;
cnt : counter port map (clk_count , disable , clock_main ,
reset_count) ;
mux_clock : mult_clock port map (clock_main , c0 , io_mode ,
clear , incr) ;
control : cont_gen port map (staged , iod , fftd , init , ip , op
, io_mode , fft_en ,
enbw , enbor , c0_en , preset , clear , disable , c0 , clock_main
,rom_en,romgen_en,reset_count,clk_count) ;

reg_ram : reg_dpram port map
(out_data,data_io,ram_data,clock_main,io_mode,enbw,enbor,ram_wr,r
am_rd) ;

f1 : r_block port map (ram_data , c0 , d2) ;
f2 : l_block port map (ram_data , c1 , d3) ;
f3 : r_block port map (ram_data , c2 , d4) ;
f4 : r_block port map (ram_data , c3 , d5) ;
f5 : r_block port map (d8 , c1_c3 , d9) ;
f6 : l_block port map (d8 , c0_c2 , d10) ;
f7 : l_block port map (d12 , c3 , d13) ;
f8 : l_block port map (d12 , c1 , d14) ;
f9 : r_block port map (d17 , clock_main , d18) ;
```

```
f10 : r_block port map (data_rom , clock_main , rom_ff) ;
mux1 : mux port map (d2 , d3 , d6 , c2_c3) ;
mux2 : mux port map (d4 , d5 , d7 , c1_c3) ;
mux3 : mux port map (d13 , d14 , d15 , c1_c3) ;
neg1 : negate port map (d10 , c0_c1 ,clock_main , d11) ;
neg2 : negate port map (d15 , c0_c1 ,clock_main , d16) ;
mult1 : multiply port map (d6 , rom_ff , clock_main , d8) ;
div : divide port map (d18 , d19) ;
f11 : level_edge port map (d19,clock_main,out_data) ;

rom_add1 : romadd_gen port map
(io_mode,c0,c1,c2,c3,stage,butterfly,rom_add,romgen_en) ;
rom1 : rom port map (clock ,rom_en,rom_add,data_rom) ;

b11 : subtractor port map ( d16 , d7 , clock , rstb , ensubb ,
a_smallb , finsubb , numzerob , zerodetectb , subb ,  changeb) ;
b2 : swap port map ( a=>d16 , b=>d7 , clock=>clock ,
rst_swap=>rstb , en_swap=>enswapb , finish_swap=>finswapb ,
d=>swap_num2b , large_exp=>expb , c=>swap_num1b ) ;
b4 : shift2 port map (sub_control=>subb , c_in=>swap_num1b ,
shift_out=>shift_outb , clock=>clock , shift_en=>enshiftb,
rst_shift=>rstb , finish_out=>finshiftb ) ;
b5 : control_main port map ( a_smallb , d16(31) , d7(31) ,
signbitb , addsubb , rstb , ensubb ,
enswapb , enshiftb , addpulseb , normaliseb , finsubb , finswapb
, finshiftb ,finish_sumb , end_allb ,
clock_main , clock , reset , enbl , numzerob , changeb ) ;
b6 : summer port map ( shift_outb , swap_num2b , expb , addpulseb
, addsubb , rstb , finish_sumb , sum_outb ) ;
b7 : normalize port map (d16 , d7 , sum_outb , expb , signbitb ,
addsubb , clock , normaliseb , rstb , zerodetectb , end_allb ,
d17) ;

a1 : subtractor port map ( d9 ,  d11 , clock , rst , ensub ,
a_small , finsub , numzero , zerodetect , suba , changea) ;
a2 : swap port map (d9 ,d11 ,clock ,rst ,enswap , finswap
,swap_num2 , exp , swap_num1 ) ;
a4 : shift2 port map (suba ,swap_num1 ,shift_outa ,clock ,
enshift , rst , finshift ) ;
a5 : control_main port map ( a_small , d9(31) , d11(31) , signbit
, addsub , rst , ensub ,
enswap , enshift , addpulse , normalise , finsub , finswap ,
finshift ,finish_sum , end_all ,
clock_main , clock , reset , enbl , numzero , changea ) ;
a6 : summer port map ( shift_outa , swap_num2 , exp , addpulse ,
addsub , rst , finish_sum , sum_out ) ;
a7 : normalize port map (d9 , d11 , sum_out , exp , signbit ,
addsub , clock , normalise , rst , zerodetect , end_all , d12) ;

end rtl ;
```

Testbench file – synth_test.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
library work;
use work.butter_lib.all;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE STD.TEXTIO.ALL;

ENTITY tb IS
END tb;

ARCHITECTURE testbench_arch OF tb IS
    FILE RESULTS: TEXT OPEN WRITE_MODE IS "results.txt";

    COMPONENT synth_main
        PORT (
            data_io : In std_logic_vector (31 DownTo 0);
            final_op : Out std_logic_vector (31 DownTo 0);
            clock_main : In std_logic;
            clock : In std_logic;
            enbl : In std_logic;
            reset : In std_logic;
            init : In std_logic
        );
    END COMPONENT;

    SIGNAL data_io : std_logic_vector (31 DownTo 0) :=
"00000000000000000000000000000000";
    SIGNAL final_op : std_logic_vector (31 DownTo 0) :=
"00000000000000000000000000000000";
    SIGNAL clock_main : std_logic := '0';
    SIGNAL clock : std_logic := '0';
    SIGNAL enbl : std_logic := '0';
    SIGNAL reset : std_logic := '0';
    SIGNAL init : std_logic := '0';

    constant PERIOD_clock : time := 200 ns;
    constant DUTY_CYCLE_clock : real := 0.5;
    constant OFFSET_clock : time := 100 ns;
    constant PERIOD_clock_main : time := 200 ns;
    constant DUTY_CYCLE_clock_main : real := 0.5;
    constant OFFSET_clock_main : time := 0 ns;

    BEGIN
        UUT : synth_main
        PORT MAP (
            data_io => data_io,
            final_op => final_op,
            clock_main => clock_main,
```

```
            clock => clock,
            enbl => enbl,
            reset => reset,
            init => init
        );

        process
variable i : integer := 0 ;
begin
for i in 1 to 1000 loop
clock <= '1' ;
wait for 5 ns ;
clock <= '0' ;
wait for 5 ns ;
end loop ;
end process ;

process
variable j : integer := 0 ;
begin
for j in 1 to 1000 loop
clock_main <= '1' ;
wait for 200 ns ;
clock_main <= '0' ;
wait for 200 ns ;
end loop ;
end process ;

process
file vector_file : text open read_mode is
"G:\Xilinx92i\projects\proj_fft_II\rom_ram.vhd" ;
variable l , l2 : line ;
variable q , p : integer := 0 ;
variable count : integer ;
variable t_a , t_b : std_logic_vector (31 downto 0) ;
variable space : character ;
begin

while not endfile(vector_file) loop
--for count in 1 to 16 loop
q := 31 ;
readline(vector_file , l2) ;

for p in 1 to 32 loop -- data from RAM
read(l2 , t_b(q)) ;
q := q - 1 ;
end loop ;
q := 31 ;
data_io <= t_b(31 downto 0) ;

wait for 400 ns ;
end loop ;
```

```
wait for 8 ms ;
--wait for 650 ns ;
end process;

-- process to reset
process
begin
reset <= '1' ;
enbl <= '1' ;
wait for 10 ns ;
reset <= '0' ;
wait ;
end process ;

process
begin
init <= '1' ;
wait for 15 ns ;
init <= '0' ;
wait ;
end process ;


END testbench_arch;
```

swap.vhd

```
-- SWAP UNIT
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity swap is
port (
       a : in std_logic_vector (31 downto 0) ;
       b : in std_logic_vector (31 downto 0) ;
       clock : in std_logic ;
       rst_swap , en_swap : in std_logic ;
       finish_swap : out std_logic ;
       d : out std_logic_vector (31 downto 0) ;
       large_exp : out std_logic_vector (7 downto 0) ;
       c  : out std_logic_vector (32 downto 0 ) ) ;
end swap ;

architecture rtl of swap is
begin
process (a , b , clock , rst_swap , en_swap)
variable x , y : std_logic_vector (7 downto 0) ;
variable p , q : std_logic_vector (22 downto 0) ;
begin
if(rst_swap = '1' ) then
c <= '0' & a(22 downto 0) & "000000000" ;
finish_swap <= '0' ;
elsif(rst_swap = '0') then
if(en_swap = '1') then
x := a (30 downto 23) ;
y := b (30 downto 23) ;
p := a (22 downto 0) ;
q := b (22 downto  0) ;
if (clock = '1') then
if (x < y) then
c <= '1' & a (22 downto 0) & "000000000" ; -- '1' for checking
d <= '1' & b (22 downto 0) & "00000000" ; -- '1' for implicit one
large_exp <= b (30 downto 23) ;
finish_swap <= '1' ;

elsif (y < x) then
c <= '1' & b (22 downto 0) & "000000000" ;
d <= '1' & a (22 downto 0) & "00000000" ; -- '1' for implicit 1.
large_exp <= a (30 downto 23) ;
finish_swap <= '1' ;

elsif ( (x=y) and (p < q)) then
c <= '1' & a (22 downto 0) & "000000000" ; -- '1' for checking
d <= '1' & b (22 downto 0) & "00000000" ; -- '1' for implicit one
```

```
large_exp <= b (30 downto 23) ;
finish_swap <= '1' ;

else
c <= '1' & b (22 downto 0) & "000000000" ;
d <= '1' & a (22 downto 0) & "00000000" ; -- '1' for implicit 1.
large_exp <= a (30 downto 23) ;
finish_swap <= '1' ;

end if ;
end if ;
end if ;
end if ;
end process;
end rtl;
```

summer.vhd

```
-- SUMMER
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity summer is
port (
       num1 , num2 : in std_logic_vector (31 downto 0) ;
       exp : in std_logic_vector (7 downto 0) ;
       addpulse_in , addsub , rst_sum : in std_logic ;
       add_finish : out std_logic ;
       sumout : out std_logic_vector ( 32 downto 0) ) ;
end summer ;
architecture rtl of summer is
begin
process (num1 , num2 , addpulse_in , rst_sum)
variable temp_num1 , temp_sum , temp_num2 , temp_sum2 , res :
std_logic_vector (32 downto 0);
variable temp_exp : std_logic_vector (7 downto 0) ;
begin
if (rst_sum = '0') then
if (addpulse_in = '1') then
temp_num1 := '0' & num1 (31 downto 0) ; --0 to find whether
normalisation is required.
temp_num2 := '0' & num2 (31 downto 0) ; --if required MSB will be
1 after addition

if (addsub = '1') then
temp_sum := temp_num1 + temp_num2 ;
sumout <= temp_sum ;
add_finish <= '1' ;

else
temp_sum := temp_num2 - temp_num1 ;
--res := temp_sum + temp_num1 ;
sumout <= temp_sum ;
add_finish <= '1' ;
end if ;
end if ;

elsif (rst_sum = '1') then
add_finish <= '0';
end if ;
end process ;

end rtl ;
```

subtractor.vhd

```
-- SUBTRACTOR UNIT
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity subtractor is
 port (
       a : in std_logic_vector (31 downto 0) ;
       b : in std_logic_vector (31 downto 0) ;
       clock , rst_sub , sub_en : in std_logic ;
       a_smaller , fin_sub , num_zero : out std_logic ;
       zero_detect : out std_logic_vector(1 downto 0) ;
       sub : out std_logic_vector (8 downto 0) ;
       change : out std_logic ) ;
end subtractor ;

architecture rtl of subtractor is
begin
process (a , b , clock , rst_sub , sub_en)
variable temp ,c , d : std_logic_vector (7 downto 0) ;
variable e , f : std_logic_vector (22 downto 0) ;
begin
if (rst_sub = '0') then
c := a (30 downto 23) ;
d := b (30 downto 23) ;
e := a (22 downto 0) ;
f := b (22 downto 0) ;

if(sub_en = '1') then
if (clock = '1') then
if ((c=0)) then
zero_detect <= "01" ;
num_zero <= '1' ;

elsif ((d=0)) then
zero_detect <= "10" ;
num_zero <= '1' ;

elsif (c < d ) then
temp := d - c ;
a_smaller <= '1' ;
sub <= '1' & temp (7 downto 0) ;
fin_sub <= '1' ;
zero_detect <= "00" ;
num_zero <= '0' ;

elsif (d < c) then
temp := c - d ;
```

```
a_smaller <= '0' ;
sub <= '1' & temp (7 downto 0) ;
fin_sub <= '1' ;
zero_detect <= "00" ;
num_zero <= '0' ;

elsif((c=d) and e < f) then
a_smaller <= '1' ;
temp:= c-d ;
sub <= '1' & temp (7 downto 0) ;
fin_sub <= '1' ;
zero_detect <= "00" ;
num_zero <= '0' ;

elsif ((c=d) and e > f) then
a_smaller <= '0' ;
temp := c-d ;
sub <= '1' & temp (7 downto 0) ;
zero_detect <= "00" ;
num_zero <= '0' ;
fin_sub <= '1' ;

elsif ((c=d) and (e = f)) then
temp := c-d ;
a_smaller <= '0' ;
sub <= '1' & "00000000" ;
fin_sub <= '1' ;
zero_detect <= "00" ;
num_zero <= '0' ;


end if ;
end if ;
end if ;

elsif(rst_sub = '1') then
fin_sub <= '0' ;
sub <= "000000000" ;
num_zero <= '0' ;
zero_detect <= "00" ;

end if ;

end process ;

process(a , b) -- process to identify when a new number comes
begin
change <= transport '1' after 1 ns ;
change <= transport '0' after 5 ns ;
end process ;

end rtl ;
```

stage.vhd

```
-- STAGE NUMBER GENERATOR.
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity stage_gen is
port (
      add_staged , add_clear : in std_logic ;
      st_stage : out std_logic_vector(1 downto 0) ) ;
end stage_gen ;

architecture rtl of stage_gen is

begin
process(add_staged , add_clear)
variable s_count : std_logic_vector(1 downto 0) ;
begin
if (add_clear = '1') then
st_stage <= "00" ;
s_count := "00" ;
elsif(add_staged'event and add_staged= '1' ) then
st_stage <= s_count + 1 ;
s_count := s_count + 1 ;
end if ;
end process ;
end rtl ;
```

shift2.vhd

```
-- SHIFT UNIT
library ieee ;
use ieee.std_logic_1164.all ;
use work.butter_lib.all ;
use ieee.std_logic_arith.all ;
use ieee.std_logic_unsigned.all ;

entity shift2 is
port (
       sub_control : in std_logic_vector (8 downto 0) ;
       c_in  : in std_logic_vector (32 downto 0) ;
       shift_out : out std_logic_vector (31 downto 0) ;
       clock , shift_en , rst_shift : in std_logic ;
       finish_out : out std_logic ) ;
end shift2 ;
architecture rtl of shift2 is
begin
process(clock)
variable sub_temp : std_logic_vector(7 downto 0) ;
variable  temp2 , temp4 : std_logic_vector(31 downto 0) ;
variable temp3 , t : std_logic ;
begin
if(rst_shift='0') then
if(shift_en = '1') then
if(temp3 = '1') then
if(sub_control(8) = '1') then
sub_temp := sub_control (7 downto 0) ;
temp2 := '1' & c_in (31 downto 1) ; --'1' for implicit one
temp3 := '0' ;
end if ;
end if ;
end if ;
end if ;

if(rst_shift='0') then
if(shift_en = '1') then
if(t = '1') then
if (sub_control(8) = '1') then
if (conv_integer(sub_temp(7 downto 0)) = 0) then
shift_out <= temp2 ;
finish_out <= '1' ;
t := '0' ;
elsif ( clock = '1') then
temp2 := '0' & temp2 (31 downto 1) ;
sub_temp := sub_temp - "00000001" ;
end if ;
end if ;
end if ;
end if ;
elsif(rst_shift='1') then
```

```
temp3 := '1' ;
finish_out <= '0' ;
t := '1' ;
end if ;

end process ;
end rtl ;
```

romadd_gen.vhd

```
-- ADDRESS GENERATOR FOR ROM
library ieee ;
use ieee.std_logic_1164.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity romadd_gen is
port (
      io_rom,c0,c1,c2,c3 : in std_logic ;
      stage_rom : in std_logic_vector(1 downto 0) ;
      butterfly_rom : in std_logic_vector(3 downto 0) ;
      romadd : out std_logic_vector(2 downto 0) ;
      romgen_en : in std_logic );
end romadd_gen ;

architecture rtl of romadd_gen is
begin
process(io_rom,c0,c1,c2,c3,stage_rom,butterfly_rom)
begin
if(romgen_en = '1') then
if(io_rom = '0') then
 case stage_rom is

 when "00" =>
 if(c0='1' or c2='1') then
 romadd <= "000" ;
 elsif(c1='1' or c3='1') then
 romadd <= "001" ;
 end if ;

 when "01" =>
 if(butterfly_rom=0 or butterfly_rom=1) then
  if(c0='1' or c2='1') then
  romadd <= "000" ;
  elsif(c1='1' or c3='1') then
  romadd <= "001" ;
  end if ;
 elsif(butterfly_rom=2 or butterfly_rom=3) then
  if(c0='1' or c2='1') then
  romadd <= "100" ;
  elsif(c1='1' or c3='1') then
  romadd <= "101" ;
  end if ;
 end if ;

 when "10" =>
  if(butterfly_rom=0) then
   if(c0='1' or c2='1') then
   romadd <= "000" ;
   elsif(c1='1' or c3='1') then
```

```
   romadd <= "001" ;
   end if ;
  elsif(butterfly_rom=1) then
   if(c0='1' or c2='1') then
   romadd <= "100" ;
   elsif(c1='1' or c3='1') then
   romadd <= "101" ;
   end if ;
  elsif(butterfly_rom=2) then
   if(c0='1' or c2='1') then
   romadd <= "010" ;
   elsif(c1='1' or c3='1') then
   romadd <= "011" ;
   end if ;
  elsif (butterfly_rom=3) then
   if(c0='1' or c2='1') then
   romadd <= "110" ;
   elsif(c1='1' or c3='1') then
   romadd <= "111" ;
   end if ;
  end if ;

 when others =>
  romadd <= "000" ;

 end case ;
end if ;
end if ;
end process ;
end rtl ;
```

rom.vhd

```
-- ROM TO STORE SINE AND COSINE VALUES
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity rom is
port (
      clock , en_rom : in std_logic ;
      romadd : in std_logic_vector(2 downto 0) ;
      rom_data : out std_logic_vector(31 downto 0) ) ;
end rom ;

architecture rtl of rom is
begin
process(clock,en_rom)
begin
if(en_rom = '1') then
if(clock = '1') then
case romadd is
 when "000" =>
 rom_data <= "00111111100000000000000000000000" ;
 when "001" =>
 rom_data <= "00000000000000000000000000000000" ;
 when "010" =>
 rom_data <= "00111111001101010000010010000001" ;
 when "011" =>
 rom_data <= "00111111001101010000010010000001" ;
 when "100" =>
 rom_data <= "00000000000000000000000000000000" ;
 when "101" =>
 rom_data <=  "00111111100000000000000000000000" ;
 when "110" =>
 rom_data <= "10111111001101010000010010000001"  ;
 when "111" =>
 rom_data <= "00111111001101010000010010000001" ;
 when others =>
 rom_data <= "01000000000000000000000000000000" ;
end case ;
end if ;
end if ;
end process ;
end rtl ;
```

rblock.vhd

```
-- NEGATIVE EDGE TRIGGERED FLIP FLOPS
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity r_block is
 port (
       data : in std_logic_vector(31 downto 0) ;
       trigger : in std_logic ;
       r_out : out std_logic_vector(31 downto 0) ) ;
end r_block ;

architecture rtl of r_block is
begin
process(data , trigger)
begin
if (trigger='0' and trigger'event) then
r_out <= data(31 downto 0) ;
end if ;
end process ;
end rtl ;
```

ram_shift.vhd

```
-- PARALLE IN PARALLEL OUT SHIFTER IN THE ADDRESS GENERATION
UNIT.
-- REQUIRED BECAUSE FFT IS COMPUTED ON DATA AND WRITTEN BACK INTO
THE SAME
-- LOCATION AFTER 5 CYCLES. SO THE READ ADDRESS IS SHIFTED
THROUGH 5 CYCLES
-- AND GIVEN AS WRITE ADDRESS.
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity ram_shift is
port (
      data_in : in std_logic_vector(3 downto 0) ;
      clock_main : in std_logic ;
      data_out : out std_logic_vector(3 downto 0) ) ;
end ram_shift ;

architecture rtl of ram_shift is
begin
process(clock_main , data_in)
begin
if (clock_main'event and clock_main = '0') then
data_out <= data_in(3 downto 0) ;
end if ;
end process ;
end rtl ;
```

ram.vhd

```
-- Behavioral description of dual-port SRAM with :
-- Active High write enable (WE)
-- Active High read enable (RE)
-- Rising clock edge (Clock)
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.butter_lib.all ;
entity reg_dpram is
port (
      data_fft , data_io : in std_logic_vector (31 downto 0);
      q : out std_logic_vector (31 downto 0);
      clock , io_mode : in std_logic;
      we , re : in std_logic;
      waddress: in std_logic_vector (3 downto 0);
      raddress: in std_logic_vector (3 downto 0));
end reg_dpram;
architecture behav of reg_dpram is
type MEM is array (0 to 15) of std_logic_vector(31 downto 0);
signal ramTmp : MEM;

begin

-- Write Functional Section
process (clock,waddress,we)
begin
if (clock='0') then
if (we = '1') then
if (io_mode = '0') then
ramTmp (conv_integer (waddress)) <= data_fft ;
elsif (io_mode = '1') then
ramTmp (conv_integer (waddress)) <= data_io ;
end if ;
end if ;
end if ;
end process ;

-- Read Functional Section
process (clock,raddress,re)
begin
if (clock='1') then
if (re = '1') then
q <= ramTmp(conv_integer (raddress)) ;
end if;
end if;
end process;
end behav;
```

print.vhd

```
-- USED TO PRINT THE RESULTS IN A NEAT FORMAT. NOT SYNTHESISABLE.
-- USED ONLY FOR SIMULATION PURPOSE.
library ieee ;
use ieee.std_logic_1164.all ;
use std.textio.all ;
use work.butter_lib.all ;
use ieee.std_logic_textio.all ;
use ieee.std_logic_unsigned.all ;
use IEEE.math_real.all;
use IEEE.std_logic_arith.all;
use work.txt_util.all;


entity print_result is
port (clock,op : in std_logic ;
      fin_res : OUT std_logic_vector(31 downto 0);
           result : in std_logic_vector(31 downto 0));
end print_result ;

architecture rtl of print_result is
file vectorw_file : text open write_mode is
"G:\Xilinx92i\projects\proj_fft_II\result.txt" ;
begin
process(op,clock)
variable l , l2 : line ;
variable q , p : integer := 0 ;
variable count : integer := 1  ;
begin
if (op = '1') then
if (count < 17) then
if(clock='0' and clock'event) then
q := 31 ;
count := count + 1 ;
for p in 1 to 32 loop -- data from RAM
--write(l2 , result(p)) ;
q := q - 1 ;
end loop ;
q := 31 ;
--writeline(vectorw_file , l2) ;
end if ;
end if ;
end if ;
end process ;
end rtl ;
```

out_result.vhd

```
-- OUTPUT RESULTS. SYNTHESISABLE
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity print_result is
PORT(
           clock : IN std_logic;
           op : IN std_logic;
           fin_res : OUT std_logic_vector(31 downto 0);
           result : IN std_logic_vector(31 downto 0));
end print_result ;

architecture rtl of print_result is
begin
process(op,clock)
variable count : integer := 1  ;
begin
if (op = '1') then
if (count < 17) then
if(clock='0' and clock'event) then
fin_res <= result ;
count := count + 1 ;
end if ;
end if ;
end if ;
end process ;
end rtl ;
```

normalize.vhd

```
library ieee ;
use ieee.std_logic_1164.all ;
use work.butter_lib.all ;
use ieee.std_logic_arith.all ;
use std.textio.all ;
use ieee.std_logic_textio.all ;
use ieee.std_logic_unsigned.all ;

entity normalize is
port (
      a , b : in std_logic_vector (31 downto 0) ;
      numb : in std_logic_vector (32 downto 0) ;
      exp : in std_logic_vector (7 downto 0) ;
      signbit , addsub , clock , en_norm , rst_norm  : in
std_logic  ;
      zero_detect : in std_logic_vector(1 downto 0) ;
      exit_n : out std_logic ;
      normal_sum : out std_logic_vector (31 downto 0) ) ;
end normalize ;

architecture rtl of normalize is
begin
process (clock)
variable numb_temp : std_logic_vector (31 downto 0) ;
variable temp_exp : std_logic_vector (7 downto 0) ;
variable t , t2 : std_logic := '1' ;

begin
if (rst_norm = '0') then
if (en_norm = '1') then
if (t = '1') then
numb_temp := numb(31 downto 0) ;
temp_exp := exp (7 downto 0) ;
t := '0';
end if ;
if (t2 = '1') then
if (zero_detect = 0) then
 if (addsub = '0') then
  if (numb_temp = 0) then
  normal_sum <= numb_temp(31 downto 0) ;
  exit_n <= '1' ;
  t2 := '0' ;
  elsif (numb_temp(31) = '1' and clock = '1') then
   normal_sum <= signbit & temp_exp(7 downto 0) & numb_temp(30
downto 8) ;--check!!
   exit_n <= '1' ;
   t2 := '0' ;
  elsif (clock = '1') then
   numb_temp := numb_temp(30 downto 0) & '0' ;
   temp_exp := temp_exp - "00000001" ;
```

```
  end if ;
 elsif (addsub = '1' and numb(32) = '1' and clock = '1') then
 temp_exp := temp_exp + "00000001" ;
 normal_sum <= signbit & temp_exp(7 downto 0) & numb_temp(31
downto 9) ;
 exit_n <= '1' ;
 t2 := '0' ;
 elsif (clock = '1') then
 normal_sum <= signbit & temp_exp(7 downto 0) & numb_temp(30
downto 8) ;
 exit_n <= '1' ;
 t2 := '0' ;
 end if;
elsif (zero_detect = 1) then
normal_sum <= b;
exit_n <= '1' ;
t2 := '0' ;
elsif (zero_detect = 2) then
normal_sum <= a ;
exit_n <= '1' ;
t2 := '0' ;
end if ;
end if ;
end if ;
elsif (rst_norm = '1') then
exit_n <= '0' ;
t := '1' ;
t2 := '1' ;
end if ;
end process ;
end rtl ;
```

negate.vhd

```
--NEGATION UNIT
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity negate is
 port (
       neg_in : in std_logic_vector(31 downto 0) ;
       neg_en , clock_main : in std_logic ;
       neg_out : out std_logic_vector(31 downto 0) ) ;
end negate ;

architecture rtl of negate is
begin
process(neg_in , neg_en , clock_main)
variable neg_temp : std_logic_vector(31 downto 0) ;
begin
neg_temp := neg_in(31 downto 0) ;
if (clock_main = '1') then
if (neg_en = '1') then
if(neg_in(31) = '0') then
neg_temp := '1' & neg_temp (30 downto 0) ;
else
neg_temp := '0' & neg_temp (30 downto 0) ;
end if ;
neg_out <= neg_temp ;
else
neg_out <= neg_in(31 downto 0) ;
end if ;
end if ;
end process ;
end rtl ;
```

mux_but.vhd

```
-- MULTIPLEXER IN THE BUTTERFLY PROCESSING UNIT
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity mux is
 port (
       d0 , d1 : in std_logic_vector(31 downto 0) ;
       mux_out : out std_logic_vector(31 downto 0) ;
       choose : in std_logic ) ;
end mux ;

architecture rtl of mux is
begin
process(d0 , d1 , choose)
begin
if (choose = '0') then
mux_out <= d0(31 downto 0) ;
elsif (choose = '1') then
mux_out <= d1(31 downto 0) ;
end if ;
end process ;
end rtl ;
```

mux_add.vhd

```
-- multiplexer in the address generation unit
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity mux_add is
port (
      a , b : in std_logic_vector(3 downto 0) ;
      sel : in std_logic ;
      q : out std_logic_vector(3 downto 0) ) ;
end mux_add ;

architecture rtl of mux_add is
begin
process (a , b , sel)
begin
if(sel = '0') then
q <= a(3 downto 0) after 2 ns ;
elsif(sel = '1') then
q <= b(3 downto 0) after 2 ns ;
end if ;
end process ;
end rtl ;
```

multiply.vhd

```
-- MULTIPLY UNIT
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity multiply is
 port(
      num_mux , num_rom : in std_logic_vector(31 downto 0) ;
      clock : in std_logic ;
      mult_out : out std_logic_vector(31 downto 0) ) ;
end multiply ;

architecture rtl of multiply is
begin
process(num_mux , num_rom , clock)
variable sign_mult , t : std_logic := '0' ;
variable temp1 , temp2 : std_logic_vector(22 downto 0) ;
variable exp_mux , exp_rom : std_logic_vector(7 downto 0) ;
variable mant_temp : std_logic_vector(45 downto 0) ;
variable exp_mult , mux_temp , rom_temp : std_logic_vector(8
downto 0) ;
variable res_temp : std_logic_vector(31 downto 0) ;
begin

temp1 := '1' & num_mux(22 downto 1) ; -- '1' for implicit '1'.
temp2 := '1' & num_rom(22 downto 1) ;
if (num_mux(31) = '1' and num_rom(31) = '1' and clock = '1') then
-- sign of results
sign_mult := '0' ;
elsif (num_mux(31) = '0' and num_rom(31) = '0' and clock = '1')
then
sign_mult := '0' ;
elsif(clock = '1') then
sign_mult := '1' ;
end if ;

if (num_mux = 0 and clock = '1') then -- ie, the number is zero.
t := '1' ;
elsif (num_rom = 0 and clock = '1') then
t := '1' ;
elsif (clock = '1') then
t := '0' ;
end if ;

if (t = '0' and clock = '1') then -- separation of mantissa and
exponent
exp_mux := num_mux (30 downto 23) ;
exp_rom := num_rom (30 downto 23) ;
```

```
mux_temp := '0' & exp_mux(7 downto 0) ;
rom_temp := '0' & exp_rom(7 downto 0) ;
exp_mult := mux_temp + rom_temp ;
exp_mult := exp_mult - 127 ;

mant_temp := temp1 * temp2 ;

if(mant_temp(45) = '1') then -- normalisation.
exp_mult := exp_mult + 1 ;
res_temp := sign_mult & exp_mult(7 downto 0) & mant_temp(44
downto 22) ;
mult_out <= res_temp(31 downto 0) ;
elsif(mant_temp(45) = '0') then
res_temp := sign_mult & exp_mult(7 downto 0) & mant_temp(43
downto 21) ;
mult_out <= res_temp(31 downto 0) ;
end if ;
elsif (t = '1' and clock = '1') then -- number zero
mult_out <= "00000000000000000000000000000000" ;
t := '0' ;
end if ;
end process ;
end rtl ;
```

mult.vhd

```
-- MULTIPLEXER TO CHOOSE BETWEEN CLOCK AND C0
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity mult_clock is
port (
      clock_main , mult1_c0 , mult1_iomode , mult_clear : in
std_logic ;
      mult1_addincr : out std_logic ) ;
end mult_clock ;

architecture rtl of mult_clock is
begin
process(clock_main , mult1_c0 , mult1_iomode , mult_clear)
variable temp1 : std_logic ;
variable temp2 : std_logic ;
begin
if(mult1_iomode = '0') then -- ie, fft computation mode
temp2 := mult1_c0 ;
elsif(mult1_iomode = '1') then -- ie, io mode
temp1 := clock_main ;
end if ;
if (mult1_iomode = '1') then
mult1_addincr <= temp1 ;
elsif(mult1_iomode = '0') then
mult1_addincr <= temp2 ;
end if ;
end process ;
end rtl ;
```

lblock.vhd

```
-- POSITIVE LEVEL TRIGGERED FLIP FLOPS
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity l_block is
 port (
       data_l : in std_logic_vector(31 downto 0) ;
       trigger_l : in std_logic ;
       l_out : out std_logic_vector(31 downto 0) ) ;
end l_block ;

architecture rtl of l_block is
begin
process(data_l , trigger_l)
begin
if (trigger_l='1') then
l_out <= data_l ;
end if ;
end process ;
end rtl ;
```

iod_staged.vhd

```
-- THIS FILE OUTPUTS THE "IO DONE" AND "STAGE DONE" AND "FFT
DONE" SIGNALS AT THE
-- CORRECT TIME. IT ALSO RECEIVES THE OUTPUT OF THE BUTTERFLY
GENERATOR
-- AND OUTPUTS IT UNCHANGED.
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity iod_staged is
port (
      but_fly : in std_logic_vector(3 downto 0) ;
      stage_no : in std_logic_vector(1 downto 0) ;
      add_incr , io_mode  : in std_logic ;
      add_iod , add_staged , add_fftd : out std_logic ;
      butterfly_iod : out std_logic_vector(3 downto 0) ) ;
end iod_staged ;

architecture rtl of iod_staged is
begin
process(but_fly,add_incr,io_mode)
begin
if(but_fly = 15 and io_mode = '1' and add_incr='0') then
add_iod <= '1' ; -- io done signal
butterfly_iod <= but_fly ;
elsif(but_fly = 4 and io_mode = '0' and add_incr='1') then
butterfly_iod <= but_fly ;
add_iod <= '0' ;
add_staged <= '1' ; -- stage done signal
else
butterfly_iod <= but_fly ;
add_staged <= '0' ;
end if ;
end process ;

process(stage_no)
begin
if (stage_no=3) then
add_fftd <= '1' ; -- fft done signal
end if ;
end process ;

end rtl;
```

ioadd.vhd

```
-- IO ADDRESS GENERATOR
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity ioadd_gen is
port (
      io_butterfly : in std_logic_vector(3 downto 0) ;
      add_iomode , add_ip , add_op : in std_logic ;
      base_ioadd : out std_logic_vector(3 downto 0) ) ;
end ioadd_gen ;

architecture rtl of ioadd_gen is
begin
process(io_butterfly , add_iomode , add_ip , add_op)
variable out_data : std_logic_vector(3 downto 0) ;
begin
if(add_iomode = '1') then
 if (add_ip = '1') then
  out_data := io_butterfly(3 downto 0) ;
 elsif(add_op = '1') then
  if(io_butterfly(3) = '0') then -- ie, real part
  out_data := '0' & io_butterfly(0) & io_butterfly(1) &
io_butterfly(2) ;
  elsif(io_butterfly(3)='1') then -- ie, complex part
  out_data := '1' & io_butterfly(0) & io_butterfly(1) &
io_butterfly(2) ;
  end if ;
 end if ;
end if ;
base_ioadd <= out_data(3 downto 0) ;
end process ;
end rtl ;
```

divide.vhd

```
-- DIVIDE BY TWO UNIT. THIS FILE HOWEVER PASSED THE DATA
UNCHANGED
-- BECAUSE DIVISION IS REQUIRED ONLY IF SCALING IS USED TO AVOID
OVERFLOW.
--  NO SCALING WAS USED IN THIS PROJECT, SO THAT RESULTS OF
MATLAB MATCHED WITH OURS
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity divide is
 port (
       data_in : in std_logic_vector(31 downto 0) ;
       data_out : out std_logic_vector(31 downto 0) ) ;
end divide ;

architecture rtl of divide is
begin
process(data_in)
variable divide_exp : std_logic_vector(7 downto 0) ;
variable divide_mant : std_logic_vector(31 downto 0) ;
begin
if (data_in = "00000000000000000000000000000000") then
data_out <= "00000000000000000000000000000000" ;
elsif (data_in = "10000000000000000000000000000000") then
data_out <= "00000000000000000000000000000000" ;
else
divide_exp := data_in(30 downto 23) ;
divide_mant := data_in (31 downto 0) ;
divide_exp := divide_exp - "00000001" ;
--data_out <= divide_mant(31) & divide_exp(7 downto 0) &
divide_mant(22 downto 0)  ;
data_out <= data_in(31 downto 0) ; -- pass data unchanged
end if ;
end process ;
end rtl ;
```

dff.vhd

```
-- POSITIVE EDGE TRIGGERED FLIPFLOPS PLACED BEFORE THE DIVIDE BY
TWO UNIT
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity level_edge is
 port (
       data_edge : in std_logic_vector(31 downto 0) ;
       trigger_edge : in std_logic ;
       edge_out : out std_logic_vector(31 downto 0) ) ;
end level_edge ;

architecture rtl of level_edge is
begin
process(data_edge , trigger_edge)
begin
if (trigger_edge='1' and trigger_edge'event) then
edge_out <= data_edge(31 downto 0) ;
end if ;
end process ;
end rtl ;
```

cycles_but.vhd

```
--WAVEFORM GENERATOR
-- THE 4 BITS OF "DATA_OUT" ARE "C0 C1 C2 C3"
library ieee ;
use ieee.std_logic_1164.all ;
use work.butter_lib.all ;

entity cycles is
port (
      clock_main , preset , c0_en , cycles_clear : in std_logic ;
      waves : out std_logic_vector(3 downto 0) ) ;
end cycles ;
architecture rtl of cycles is
--type state_values is (st0 , st1 , st2 ,  st3) ;
--signal pres_state1 , next_state1 : state_values ;
shared variable data_out : std_logic_vector(3 downto 0)  ;
begin
process (clock_main , preset , c0_en,cycles_clear)
variable t : std_logic ;
begin
if (c0_en = '1') then
 if (preset = '1' and t='1')then
 pres_state1 <= st0 ;
 t := '0' ;
 elsif (clock_main'event and clock_main= '0') then
 pres_state1 <= next_state1 ;
 end if ;
end if ;
if(cycles_clear = '1') then
t := '1' ;
end if ;
end process ;

process(pres_state1 , c0_en , clock_main)
variable temp_clock : std_logic ;
begin

 case pres_state1 is
  when st0 =>
   data_out := "1000" ;
   next_state1 <= st1 ;

  when st1 =>
    data_out :=  "0100" ;
    next_state1 <= st2 ;

 when st2 =>
   data_out := "0010" ;
   next_state1 <= st3 ;

 when st3 =>
```

```
   data_out := "0001" ;
   next_state1 <= st0 ;

 when others =>
    next_state1 <= st0 ;

end case ;

waves <= data_out ;

end process ;
end rtl ;
```

counter.vhd

```
-- THIS FILE COUNTS THE NUMBER OF CYCLES AFTER FFT COMPUTATION IS
ENABLED.
-- THIS IS REQUIRED BECAUSE WRITING INTO THE RAM BEGINS ONLY
AFTER 5 CYCLES (DURING C1)
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity counter is
port (
      c : out std_logic_vector(2 downto 0) ;
      disable , clock_main , reset : in std_logic) ;
end counter ;

architecture rtl of counter is
begin
process (reset , clock_main , disable)
variable temp : std_logic_vector(2 downto 0) ;
begin
 if (disable <= '0') then
 if(reset = '1') then
 c <= "000" ;
 temp := "000" ;
 elsif(clock_main = '1' and clock_main'event) then
 c <= (temp + 1) ;
 temp := temp + 1 ;
 end if ;
end if ;
end process ;
end rtl ;
```

controller.vhd

```
-- CONTROL UNIT OF THE PROCESSOR
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity cont_gen is
port (
      con_staged , con_iod , con_fftd , con_init : in std_logic ;
      con_ip , con_op , con_iomode , con_fft : out std_logic ;
      con_enbw , con_enbor , c0_enable , con_preset : out
std_logic ;
      con_clear , disable : out std_logic ;
      c0 , clock_main : in std_logic ;
      en_rom , en_romgen , reset_counter : out std_logic ;
      con_clkcount : in std_logic_vector(2 downto 0) ) ;
end cont_gen ;

architecture rtl of cont_gen is
type state is (rst1,rst2,rst3,rst4,rst5,rst6,rst7) ;
signal current_state , next_state : state ;
shared variable counter , temp2 : std_logic_vector(1 downto 0) :=
"00" ;
begin
process (current_state ,con_staged , con_iod , con_fftd ,
con_clkcount , c0)

begin
case current_state is
 when rst1 =>
 con_iomode <= '1' ; -- set mode to io.
 con_ip <= '1' ; -- input mode
 con_clear <= '1' ; -- clear all blocks
 con_enbw <= '1' ; -- enable write to RAM
 con_enbor <= '0' ; -- disable read
 c0_enable <= '0' ; -- disable cycles unit
 disable <= '1' ; -- disable counter
 next_state <= rst2 ;

 when rst2 =>
  con_clear <= '0' ; -- bring clear signal back to zero
  next_state <=rst3 ;

 when rst3 =>
  if(con_iod = '1') then
   con_preset <= '1' ; -- reset cycles
   reset_counter <= '1' ; -- reset counter
   c0_enable <= '1' ; -- enable cycles
   con_iomode <= '0' ; -- set io mode to '0'
```

```
  con_fft <= '1' ; -- fft mode
  en_rom <= '1' ; -- enable ROM
  en_romgen <= '1' ; -- enable ROM address generator
  con_clear <= '1' ; -- clear all blocks
  con_enbw <= '0' ; -- disable write to RAM
  con_enbor <= '1' ; -- enable read from ROM
  disable <= '0' ; -- enable  counter unit.
  next_state <= rst4 ;
 else
 next_state <= rst3 ;
 end if ;

when rst4 =>
 con_preset <= '0' ; -- reset for cycles
 reset_counter <= '0' ; -- reset for counter
 con_clear <= '0' ; -- clear all signals
 if (con_clkcount = 5) then -- check whether 4 or not
  con_enbw <= '1' ; -- enable write to ROM
  disable <= '1' ; -- disable counter
  reset_counter <= '1' ; -- reset counter
  next_state <= rst5 ;
 else
  next_state <= rst4 ;
 end if ;

when rst5 =>

 if (con_fftd = '1') then
 disable <= '0' ; -- enable counter
 reset_counter <= '0' ;
 con_clear <= '1' ; -- clear butterfly generator
 con_fft <= '0' ; -- disable fft address generator
 if (con_clkcount = 4) then
   disable <= '1';
   con_enbw <= '0' ;
   con_iomode <= '1' ;
   con_op <= '1' ;
   con_ip <= '0' ;
   next_state <= rst6 ;
   else
  next_state <= rst5 ;
  end if ;
 else
 next_state <= rst5 ;
end if ;

when rst6 =>
 con_clear <= '0' ;
 next_state <= rst7 ;

when rst7 =>
 if(con_iod = '1') then
```

```
   con_clear <= '1' ;
   con_preset <= '1' ;
   con_enbor <= '0';
  else
  next_state <= rst7 ;
  end if ;


 when others =>
  next_state <= rst1 ;

end case ;
end process ;

process(clock_main , con_init)
begin
if(con_init = '1') then
current_state <= rst1 ;
elsif (clock_main'event and clock_main = '0') then
current_state <= next_state ;
end if ;
end process ;
end rtl ;
```

butter_lib.vhd

```
-- THIS FILE DECLARES THE SIGNALS USED IN THE PROCESSOR
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use ieee.std_logic_unsigned.all ;

package butter_lib is

signal
ram_data,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15,d16,d17,
d18,d19,out_data : std_logic_vector(31 downto 0) := (others =>
'0') ;
signal data_rom , rom_ff : std_logic_vector(31 downto 0) ;
signal clock_main , reset , enbl , clock : std_logic := '0' ;
signal c0 , c1 , c2 , c3 , c0_c1 , c2_c3 , c0_c2 , c1_c3 :
std_logic ;
signal
c0_and,c1_and,c2_and,c3_and,c0_c1and,c2_c3and,c0_c2and,c1_c3and :
std_logic ;
signal reset_count : std_logic ;
type state is (reset1 , reset2 , reset3 , reset4 , reset5 ,
reset6 , reset7) ;

signal final_sum : std_logic_vector (31 downto 0) := (others =>
'0') ;
signal shift , finish_sum , signbit , normalise , end_all ,
a_small , addsub ,sum_out2 , shift_done , done , num_rec , setbit
,  addpulse : std_logic := '0' ;
signal shift_outa , swap_num2 : std_logic_vector ( 31 downto 0 )
:= (others => '0') ;
signal swap_num1 , sum_out : std_logic_vector (32 downto 0) :=
(others => '0') ;
signal sub2 : std_logic_vector (8 downto 0) := (others => '0') ;
signal suba : std_logic_vector (8 downto 0) := (others => '0') ;
signal exp : std_logic_vector (7 downto 0) := (others => '0') ;
signal rst , enswap , ensub , enshift , finsub , finswap ,
finshift , numzero : std_logic := '0' ;
signal zerodetect : std_logic_vector(1 downto 0) ;
signal changea : std_logic ;

signal final_sumb : std_logic_vector (31 downto 0) := (others =>
'0') ;
signal shiftb , finish_sumb , signbitb , normaliseb , end_allb ,
a_smallb , addsubb,sum_out2b , shift_doneb , doneb , num_recb ,
setbitb ,  addpulseb , clockb : std_logic := '0' ;
signal shift_outb , swap_num2b : std_logic_vector ( 31 downto 0 )
:= (others => '0') ;
signal swap_num1b , sum_outb : std_logic_vector (32 downto 0) :=
(others => '0') ;
signal sub2b : std_logic_vector (8 downto 0) := (others => '0') ;
```

```
signal subb : std_logic_vector (8 downto 0) := (others => '0') ;
signal expb: std_logic_vector (7 downto 0) := (others => '0') ;
signal rstb , enswapb , ensubb , enshiftb , finsubb , finswapb ,
finshiftb , numzerob , clock_mainb , resetb , enblb : std_logic
:= '0' ;
signal zerodetectb : std_logic_vector(1 downto 0) ;
signal changeb : std_logic ;

signal incr , clear , io_mode , staged , iod : std_logic ;
signal
butterfly,fftadd_rd,shift1,shift3,shift4,shift5,shift6,ram_wr,ram
_rd,io_add : std_logic_vector(3 downto 0) := (others => '0') ;
signal fftd , fft_en , ip , op , init : std_logic ;
signal stage : std_logic_vector(1 downto 0) ;
--signal clock_main,c0,c1,c2,c3,c0_c1,c2_c3,c0_c2,c1_c3 :
std_logic ;
signal preset,disable,c0_en,rom_en,romgen_en : std_logic ;
signal clk_count : std_logic_vector(2 downto 0) ;
signal enbw , enbor : std_logic ;
signal data_io : std_logic_vector(31 downto 0) := (others => '0')
;
signal rom_add : std_logic_vector(2 downto 0) ;
type state_values is (st0 , st1 , st2 ,  st3) ;
signal pres_state1 , next_state1 : state_values ;

signal butterfly_iod : std_logic_vector(3 downto 0) ;
signal cyc_clear : std_logic ;
signal add_rd , add_wr : std_logic_vector(3 downto 0) ;

end butter_lib ;
```

but.vhd

```
-- BUTTERFLY GENERATOR
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.butter_lib.all ;
use ieee.std_logic_unsigned.all ;

entity but_gen is
port (
      add_incr , add_clear , stagedone : in std_logic ;
      but_butterfly : out std_logic_vector(3 downto 0) ) ;
end but_gen ;

architecture rtl of but_gen is
begin
process(add_clear , add_incr , stagedone)
variable cnt : integer ;
variable count : std_logic_vector(3 downto 0) ;
begin
if(add_clear = '1' or stagedone = '1') then
count := "0000" ;
but_butterfly <= "0000" ;
elsif (add_incr'event and add_incr = '1') then
but_butterfly <= (count + 1) ;
count := count + 1 ;
end if ;
end process ;
end rtl ;
```

baseindex.vhd

```
--BASE INDEX GENERATOR
library ieee;
use ieee.std_logic_1164.all;
use work.butter_lib.all ;

entity baseindex is
 port(
   ind_butterfly: in std_logic_vector(3 downto 0);
   ind_stage: in std_logic_vector(1 downto 0);
   add_fft: in std_logic;
   fftadd_rd: out std_logic_vector(3 downto 0);
   c0,c1,c2,c3: in std_logic);
end baseindex;

architecture rtl of baseindex is
begin
process(ind_butterfly,ind_stage,add_fft,c0,c1,c2,c3)
variable out_sig : std_logic_vector(3 downto 0);
begin
if (add_fft='1') then
if(c2='1') then -- address for 'x'. Since this is the real part,
case ind_stage is -- M.S.B is '0'.
   when "00" => out_sig := "00" & ind_butterfly(1 downto 0);
   when "01" => out_sig := '0' & ind_butterfly(1) & '0' &
ind_butterfly(0);
-- when "10" => out_sig := '0' & '1' & '1' & ind_butterfly(3);
   when "10" => out_sig := '0' & ind_butterfly(1 downto 0) & '0';
   when others => out_sig := "0000";
end case;

elsif(c0='1') then -- address for 'y'.
case ind_stage is
  when "00" => out_sig := "01" & ind_butterfly(1 downto 0);
  when "01" => out_sig := '0' & ind_butterfly(1) & '1' &
ind_butterfly(0);
  when "10" => out_sig := '0' & ind_butterfly(1 downto 0) & '1';
  when others => out_sig := "0000";
end case;

elsif(c1='1') then -- addresss for 'Y'
case ind_stage is
  when "00" => out_sig := "11" & ind_butterfly(1 downto 0);
  when "01" => out_sig := '1' & ind_butterfly(1) & '1' &
ind_butterfly(0);
  when "10" => out_sig := '1' & ind_butterfly(1 downto 0) & '1';
  when others => out_sig := "0000";
end case;

elsif(c3='1') then -- address for 'X'
case ind_stage is
```

```
 when "00" => out_sig := "10" & ind_butterfly(1 downto 0);
  when "01" => out_sig := '1' & ind_butterfly(1) & '0' &
ind_butterfly(0);
  when "10" => out_sig := '1' & ind_butterfly(1 downto 0) & '0';
  when others => out_sig := "0000";
--else
--out_sig := "ZZZZ";
end case;
end if;
end if;
fftadd_rd <= out_sig (3 downto 0) ;
end process;
end rtl;
```

and_gates.vhd

```
--THIS FILE RECEIVES THE OUTPUT OF THE WAVEFORM GENERATOR AND
--OUTPUTS THE REQUIRED CYCLES

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.butter_lib.all ;

entity and_gates is
port (
      waves_and : in std_logic_vector(3 downto 0) ;
      clock_main , c0_en : in std_logic ;
      c0,c1,c2,c3 : out std_logic ;
      c0_c1,c2_c3,c0_c2,c1_c3 : out std_logic ) ;
end and_gates ;

architecture rtl of and_gates is
begin
process(clock_main,waves_and)
begin
if (c0_en = '1' and clock_main='1') then

c0 <= waves_and(3) ;
c1 <= waves_and(2) ;
c2 <= waves_and(1) ;
c3 <= waves_and(0) ;
c0_c1 <= waves_and(3) or waves_and(2) ;
c0_c2 <= waves_and(3) or waves_and(1) ;
c2_c3 <= waves_and(1) or waves_and(0) ;
c1_c3 <= waves_and(0) or waves_and(2) ;
else
c0 <= '0' ;
c1 <= '0' ;
```

```
c2 <= '0' ;
c3 <= '0' ;
c0_c1 <= '0' ;
c0_c2 <= '0' ;
c2_c3 <= '0' ;
c1_c3 <= '0' ;
end if ;
end process ;
end rtl ;
```