FINAL REPORT on IMAGE MATCHING PROCESSOR on Xilinx FPGA

As Part of Project Work

For ECE 573

By

Satyanarayana Nekkalapu
Graduate Student
ID #: 947919900

Instructor: Dr. Marek Perkowski
Professor, ECE Department,
Portland State University
Winter 2007

Goal:

The goal of this project is to design an image matching processor, describe it in HDL (Verilog), synthesize, implement and test using Xilinx FPGA. The processor reads two images one is a sub set of the other and locates the subset in the superset image.

1. Introduction and Approach:

Image matching is the fundamental basis for many problems in computer vision. In object recognition, images in the object library are compared with the image under test. In panorama mosaic, global registration is performed to determine the relative projective mapping between different images. Other applications include location recognition in robot navigation, content-based image retrieval, 3D reconstruction, stereo image matching and motion tracking.

This project discusses one method of implementation of an image matching processor. The hardware model converts the images to frequency domain by performing FFT (Fast Fourier Transform) operation which allows reducing the convolution process to elementary matrix product. The resultant product matrix goes through the IFFT (Inverse Fast Fourier Transform) .The matrix obtained from IFFT is scanned for the highest intensity which indicates the high co-relation of the two images.

1.1 Introduction to Signals:

A Signal can be defined as a set of information or data. In general signal can be a function of one or more variables.

Example:
Speech: Speech is an audio signal that varies with time. This is an example of *one-dimensional* signals, as the signal is a function of only one variable, *time*. The time, here, varies continuously. So Speech can be represented as a function of time (t)

I.e. (Speech signal)F = f (t)

Picture: A Picture is a two-dimensional signal that depends on the spatial coordinates (*x* and *y* coordinates) of the picture plane. A picture, in general, is continuous in *x* and *y* directions and the strength/intensity of colors at each point. So picture can be represented as a function of coordinates x and y.

I.e. (Picture) P = f(x,y)

1.1.1 Classification of signals:

The are a number of ways in which the signals are classified, but in general they are classified into

1.1.1.1Continuous-time and discrete-time signals:

Signals that are continuous in time are called continuous time signals as shown in Fig 1. Signals that are discontinuous or discrete in time are said to be discrete-time signals as shown in Fig 1.2

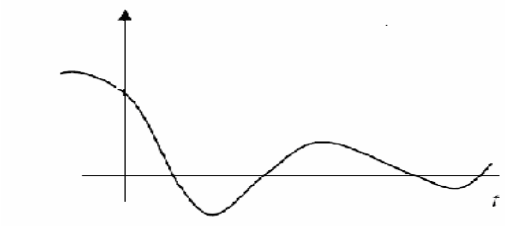


Fig 1.1

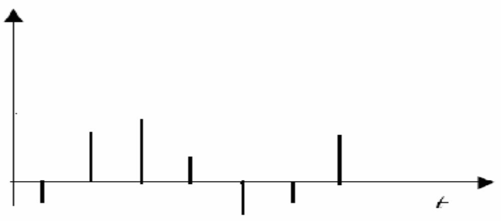


Fig1.2

1.1.1.2 Analogue and Digital signals:

- Analog signals are continuous both in time and amplitude; such as Fig. 1.1
- Digital signals are discrete both in time and amplitude, e.g. Fig. 1.2
- A signal which is discrete in time, but continuous in amplitude, is called sampled or discrete-time signal e.g., Fig. 1.3.

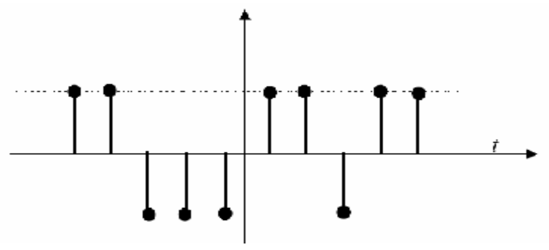


Fig 1.3

1.1.2 Useful Operations on Signals:

1.1.2.1 Time Shifting: k(t)=f(t ± T)

Figure 1.4 demonstrates the time shifting property.

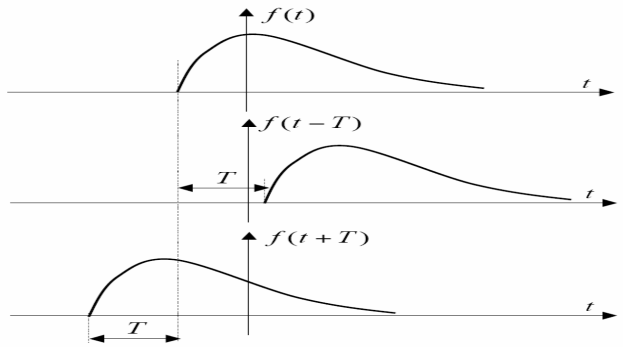


Fig 1.4

1.1.2.2 Time Scaling:
Figure 1.5 demonstrates the time shifting property.

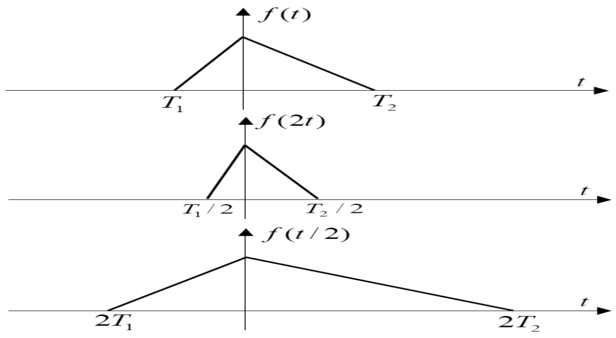


Fig 1.5

1.1.2.1 Time Inversion (Reversal): Ø (t) =f (-t)
Figure 1.6 demonstrates the time shifting property.

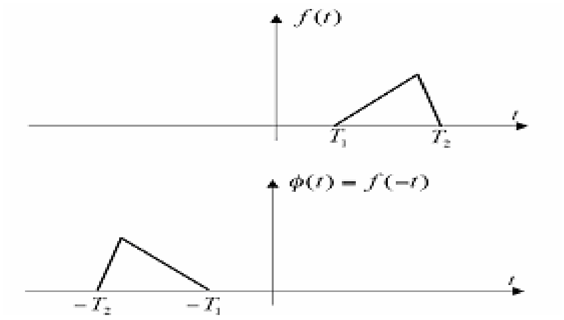


Fig 1.6

1.1.3 Signal Representation using Impulse Function:

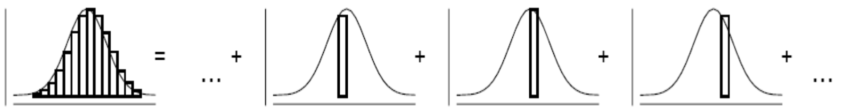


Fig 1.7

Any signal can be expressed as a sum of scaled and shifted unit impulses. We begin with the pulse or "staircase" approximation to a continuous signal, as illustrated in Fig. 1.7. Conceptually, this is trivial: for each discrete sample of the original signal, we make a pulse signal. Then we add up all these pulse signals to make up the approximate signal. Each of these pulse signals can in turn be represented as a standard pulse scaled by the appropriate value and shifted to the appropriate place. In mathematical notation:

$$\tilde{x}(t) = \sum_{k=-\infty}^{\infty} x(k\Delta)\,\delta_{\Delta}(t - k\Delta)\,\Delta.$$

As we let approach zero, the approximation becomes better and better, and the in the limit equal to the original signal.Therefore,

$$x(t) = \lim_{\Delta \to 0} \sum_{k=-\infty}^{\infty} x(k\Delta)\, \delta_{\Delta}(t - k\Delta)\, \Delta.$$

Also, as$\Delta \to 0$, the summation approaches an integral, and the pulse approaches the unit impulse:

$$x(t) = \int_{-\infty}^{\infty} x(s)\, \delta(t - s)\, ds. \qquad \text{eq 1}$$

In other words, we can represent any signal as an infinite sum of shifted and scaled unit impulses.

1.1.4. Linear Systems

In other words, we can represent any signal as an infinite sum of shifted and scaled unit impulses

$$y(t) = T[x(t)],$$

Where T denotes the transform, a function from input signals to output signals. Systems come in a wide variety of types. One important class is known as linear systems. To see whether a system is linear, we need to test whether it obeys certain rules that all linear systems obey. The two basic tests of linearity are homogeneity and additivity.

1.1.4.1 Homogeneity. As we increase the strength of the input to a linear system, say we double it, and then we predict that the output function will also be doubled. For example, if the current injected to a passive neural membrane is doubled, the resulting membrane potential fluctuations will double as well. This is called the *scalar rule* or sometimes the *homogeneity* of linear systems.

1.1.4.2 Additivity. Suppose we measure how the membrane potential fluctuates over time in response to a complicated time-series of injected current $x_1(t)$. Next, we present a second (different) complicated time-series$x_2(t)$. The second stimulus also generates fluctuations in the membrane potential which we measure and write down. Then, we present the sum of the two currents $x_1(t) + x_2(t)$ and see what happens. Since the system is linear, the measured membrane potential fluctuations will be just the sum of the fluctuations to each of the two currents presented separately.

1.1.4.3 Superposition. Systems that satisfy both homogeneity and additivity are considered to be

Linear systems. These two rules, taken together, are often referred to as the principle of superposition. Mathematically, the principle of superposition is expressed as:

$$T(\alpha x_1 + \beta x_2) = \alpha T(x_1) + \beta T(x_2)$$

Homogeneity is a special case in which one of the signals is absent. Additivity is a special case in which $\alpha = \beta = 1$.

1.1.4.4 Shift-invariance. Suppose that we inject a pulse of current and measure the membrane potential fluctuations. Then we stimulate again with a similar pulse at a

different point in time, and again we measure the membrane potential fluctuations. If we haven't damaged the membrane with the first impulse then we should expect that the response to the second pulse will be the same as the response to the first pulse. The only difference between them will be that the second pulse has occurred later in time, that is, it is shifted in time. When the responses to the identical stimulus presented shifted in time are the same, except for the corresponding shift in time, then we have a special kind of linear system called a *shift-invariant* linear system. Just as not all systems are linear, not all linear systems are shift-invariant.
In mathematical language, a system is shift-invariant if and only if:

1.1.5 Convolution:
To characterize a shift-invariant linear system, we need to measure only one thing: the way the system responds to a unit impulse. This response is called *the impulse response function* of the system. Once we've measured this function, we can (in principle) predict how the system will respond to any other possible stimulus.

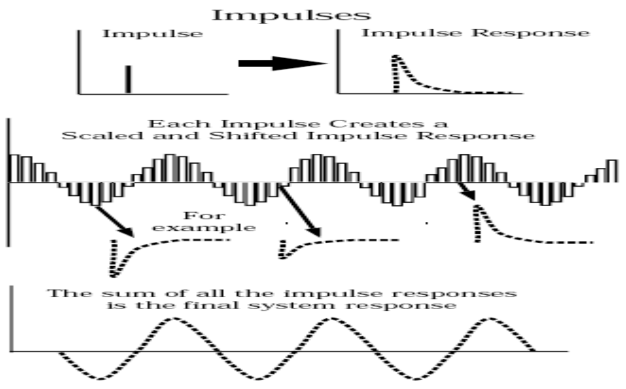

Impulses
Impulse
Impulse Response
Each Impulse Creates a
Scaled and Shifted Impulse Response
For
example
The sum of all the impulse responses
is the final system response

Fig1.8 characterizing a linear system using its impulse response.

The way we use the impulse response function is illustrated in Fig. xx. We conceive of the input stimulus, in this case a sinusoid, as if it were the sum of a set of impulses (Eq. 1). We know the responses we would get if each impulse was presented separately (i.e., scaled and shifted copies of the impulse response). We simply add together all of the (scaled and shifted) impulse responses to predict how the system will respond to the complete stimulus.

1.1.5.1 convolution integral. Begin by using Eq.1 to replace the input signal x (t) by its representation in terms of impulses:

$$\begin{aligned} y(t) = T[x(t)] &= T\left[\int_{-\infty}^{\infty} x(s)\,\delta(t-s)\,ds\right] \\ &= T\left[\lim_{\Delta\to 0}\sum_{k=-\infty}^{\infty} x(k\Delta)\,\delta_\Delta(t-k\Delta)\,\Delta\right] \end{aligned}$$

Using additivity,

$$y(t) = \lim_{\Delta\to 0}\sum_{k=-\infty}^{\infty} T[x(k\Delta)\,\delta_\Delta(t-k\Delta)\,\Delta].$$

Taking the limit,

$$y(t) = \int_{-\infty}^{\infty} T[x(s)\,\delta(t-s)\,ds].$$

Using homogeneity,

$$y(t) = \int_{-\infty}^{\infty} x(s)\,T[\delta(t-s)]\,ds.$$

Now let h (t) be the response of to the unshifted unit impulse, i.e., $h(t) = T[\delta(t)]$ Then by using shift-invariance,

$$y(t) = \int_{-\infty}^{\infty} x(s)\,h(t-s)\,ds. \qquad \text{eq (2)}$$

Notice what this last equation means. For any shift-invariant linear system T, we know its impulse response once we know its impulse response $h(t)$ (that is, its response to a unit impulse), we can forget about entirely, and just add up scaled and shifted copies of $h(t)$to calculate the response of it to any input whatsoever. Thus any shift-invariant linear system is completely characterized by its impulse response $h(t)$

The way of combining two signals specified by Eq.2 is know as convolution. It is such a widespread and useful formula that it has its own shorthand notation,* .For any two signals *x* and *y* there will be another signal *z* obtained by convolving *x* with *y*,

$$z(t) = x * y = \int_{-\infty}^{\infty} x(s)\,y(t-s)\,ds.$$

1.2. Time and Frequency Representation

The most common representation of signals and waveforms is in the time domain. However, most signal analysis techniques work only in the frequency domain. The concept of the frequency domain representation of a signal is quite difficult to understand when one is first introduced to it.

1.2.1. Time and Frequency Domains

The frequency domain is simply another way of representing a signal. For example, consider a simple sinusoid in Fig1.9

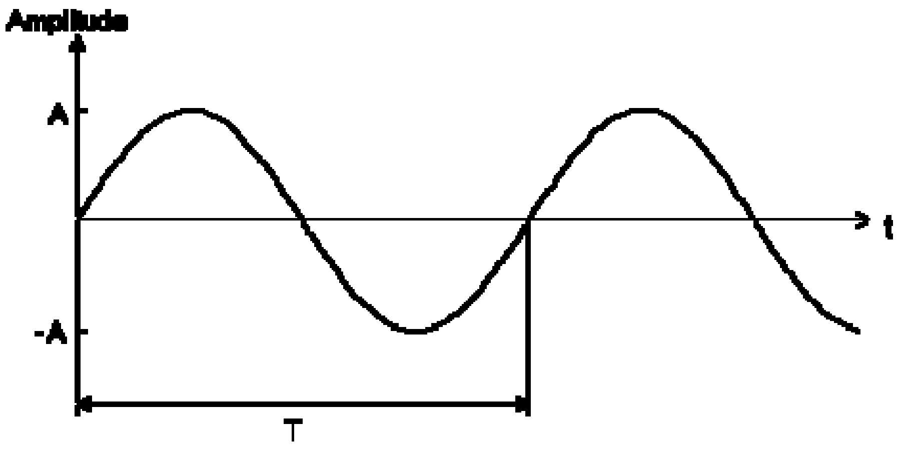


Fig1.9

The time - amplitude axes on which the sinusoid is shown define the *time plane*. If an extra axis is added to represent frequency, then the sinusoid would be as illustrated below in figure 2.0.

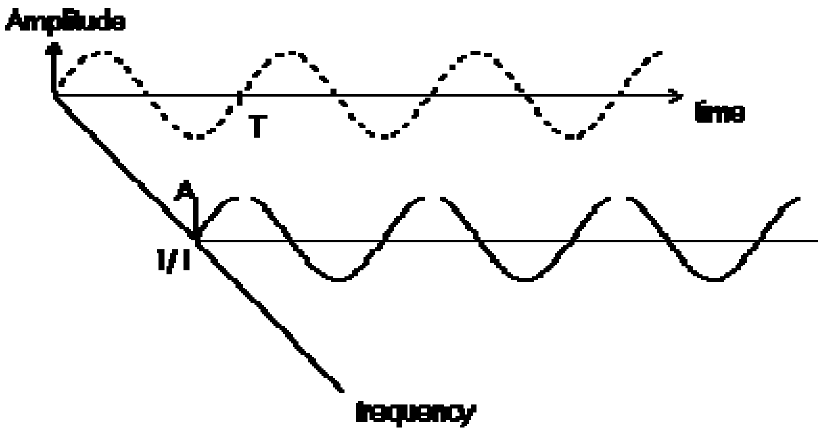


Fig2.0

The frequency - amplitude axes define the *frequency plane* in a manner similar to the way the time plane is defined by the time - amplitude axes. This frequency plane is what is represented when the spectrum of a signal is shown. The frequency plane is orthogonal to the time plane, and intersects with it on a line which is the amplitude axis.

Note that the time signal can be considered to be the projection if the sinusoid onto the time plane (time - amplitude axes). The actual sinusoid can be considered to be as existing some distance along the frequency axis away from the time plane. This distance along the frequency axis is the frequency of the sinusoid, equal to the inverse of the period of the sinusoid.

The waveform also has a projection onto the frequency plane. If you imagine yourself standing on the frequency axis, looking toward the sinusoid, you would see the sinusoid as simply a line. This line will have a height equal to the amplitude of the sinusoid. So, the projection of the sinusoid onto the frequency plane is simply a line equal to the amplitude of the sinusoid. These two projections mean that the sinusoid appears as a sinusoid in the time plane (time - amplitude axes), and as a line in the frequency plane (frequency - amplitude axes) going up from the frequency of the sinusoid to a height equal to the amplitude of the sinusoid. It should be noted very carefully that all the information about the sinusoid (frequency, amplitude and phase) is represented in the time plane projection, but all phase information is lost in the projection onto the frequency plane. If the full signal is to be reconstructed from the frequency representation then an additional graph called the *phase diagram* is needed. The phase diagram is simply a graph of the phase versus frequency, similar to the amplitude versus frequency graph obtained from the frequency plane. Although it examined only for a sinusoidal waveform, it is relevant to all waveforms because any non-sinusoidal waveform can be expressed as the sum of various sinusoidal components. This is achieved by Fourier series expansion.

1.3. FOURIER SERIES:

The **Fourier series** is a mathematical tool used for analyzing periodic functions by decomposing such a function into a weighted sum of much simpler sinusoidal component functions sometimes referred to as **normal Fourier modes**, or simply **modes** for short. The weights, or coefficients, of the modes, are one-to-one mapping of the original function. Fourier series serve many useful purposes, as manipulation and conceptualization of the modal coefficients are often easier than with the original function.

If $f(t)$ is a periodic function, with period *T,* and $\omega = \frac{2\pi}{T}$, then

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(n\omega t) + b_n \sin(n\omega t)$$

where

$$a_n = \frac{2}{T} \int_{-T/2}^{T/2} f(t) \cos(n\omega t)\, dt \quad \text{and} \quad b_n = \frac{2}{T} \int_{-T/2}^{T/2} f(t) \sin(n\omega t)\, dt$$

for $n = 0, 1, 2, \ldots$

1.4. Fourier Transform

The Fourier transform is used to transform a continuous time signal into the frequency domain. It describes the continuous spectrum of a non periodic time signal. The Fourier transform X*(f)* of a continuous time function *x(t)* can be expressed as

$$X(f) = \int_{-\infty}^{\infty} x(t)\, e^{-i2\pi ft}\, dt$$

The inverse transform is

$$x(t) = \int_{-\infty}^{\infty} X(f)\, e^{i2\pi ft}\, df$$

The *Fourier Transform* provides the means of transforming a signal defined in the time domain into one defined in the frequency domain. When a function is evaluated by numerical procedures, it is always necessary to sample it in some fashion. This means that in order to fully evaluate a Fourier transform with *digital operations*, it is necessary that the time and frequency functions be sampled in some form or another. Thus the digital or *Discrete Fourier Transform* (DFT) is of primary interest.

1.4.1. Discrete Fourier Transform

This is used in the case where both the time and the frequency variables are discrete (which they are if digital computers are being used to perform the analysis). Let $x(n\mathrm{T})$ represent the discrete time signal, and let $\mathrm{X}(m\mathrm{F})$ represent the discrete frequency transform function. The Discrete Fourier Transform (DFT) is given by

$$X(mF) = \sum_{n} x(nT)\, e^{-i n m 2\pi F T}$$

where

$$x(nT) = \frac{1}{N}\sum_{m} X(mF)\, e^{jnm2\pi FT}$$

1.4.2. Fast Fourier Transform

The *fast Fourier transform* (FFT) is simply a class of special algorithms which implement the discrete Fourier transform with considerable savings in computational time. It must be pointed out that the FFT is not a different transform from the DFT, but rather just a means of computing the DFT with a considerable reduction in the number of calculations required.

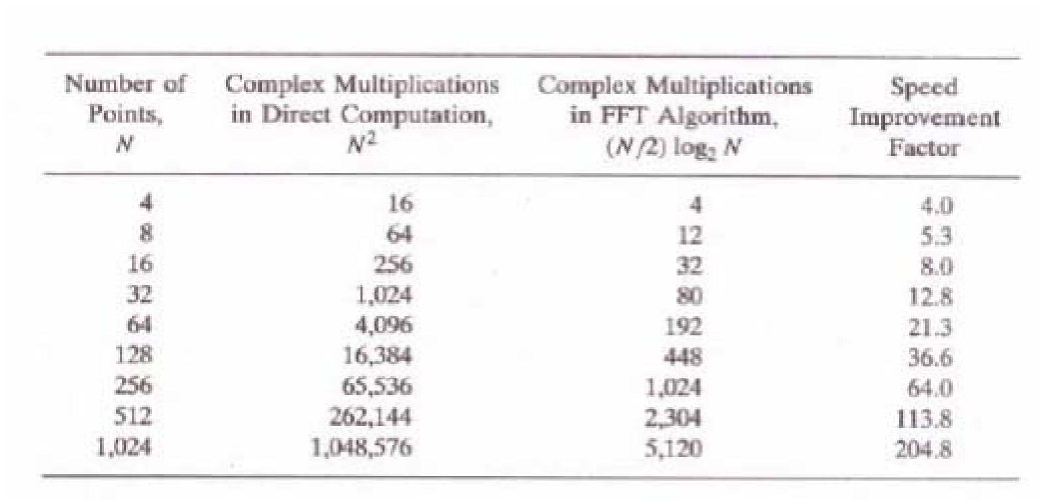
The number of complex multiplication and addition operations required by the simple forms both for the Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT) is of order N^2 as there are N data points to calculate, each of which requires N complex arithmetic operations.

For length n input vector x, the DFT is a length n vector X, with n elements:

$$f_j = \sum_{k=0}^{n-1} x_k e^{-(2\pi i/n)jk} \qquad j = 0, \ldots, n-1.$$

In computer science jargon, we may say they have algorithmic complexity O(N) 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 applications. 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.
The 'Radix 2' algorithms are useful if N is a regular power of 2 (N=2). 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 Fig. 2.1, ratio of execution times for the (DFT) vs. (Radix 2FFT) (denoted as 'Speed Improvement Factor') 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 Computation, N^2	Complex Multiplications 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	1,024	80	12.8
64	4,096	192	21.3
128	16,384	448	36.6
256	65,536	1,024	64.0
512	262,144	2,304	113.8
1,024	1,048,576	5,120	204.8

Fig 2.1

The radix-2 decimation-in-frequency FFT is an important algorithm obtained by the divide-and-conquer approach. The Fig. 2.2 below shows the first stage of the 8-point DIF algorithm.

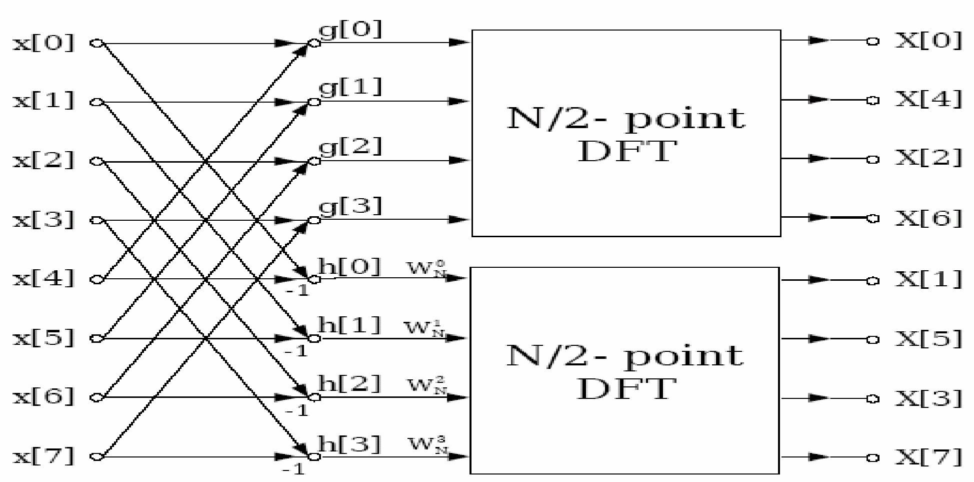


Fig 2.2: Decimation in frequency of a length-*N* DFT into two length- *N*

The decimation, however, causes shuffling in data. The entire process involves $\nu = \log_2 N$ stages of decimation, where each stage involves $N/2$ butterflies of the type shown in the Fig. 2.3.

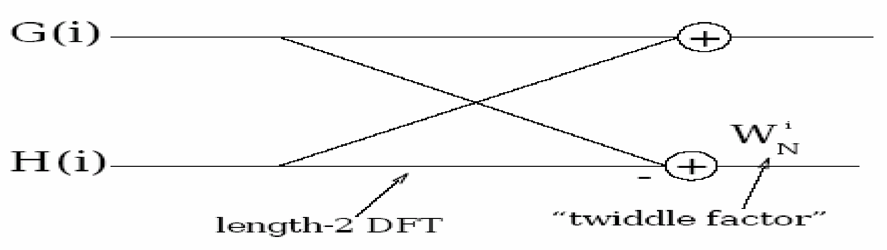


Fig 2.3

Here $W_N = e^{-j\,2\Pi/N,}$ is the Twiddle factor.

Consequently, the computation of N-point DFT via this algorithm requires $(N/2)\log_2 N$ complex multiplications. For illustrative purposes, the eight-point decimation-in frequency algorithm is shown in the Figure below 2.4. 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.

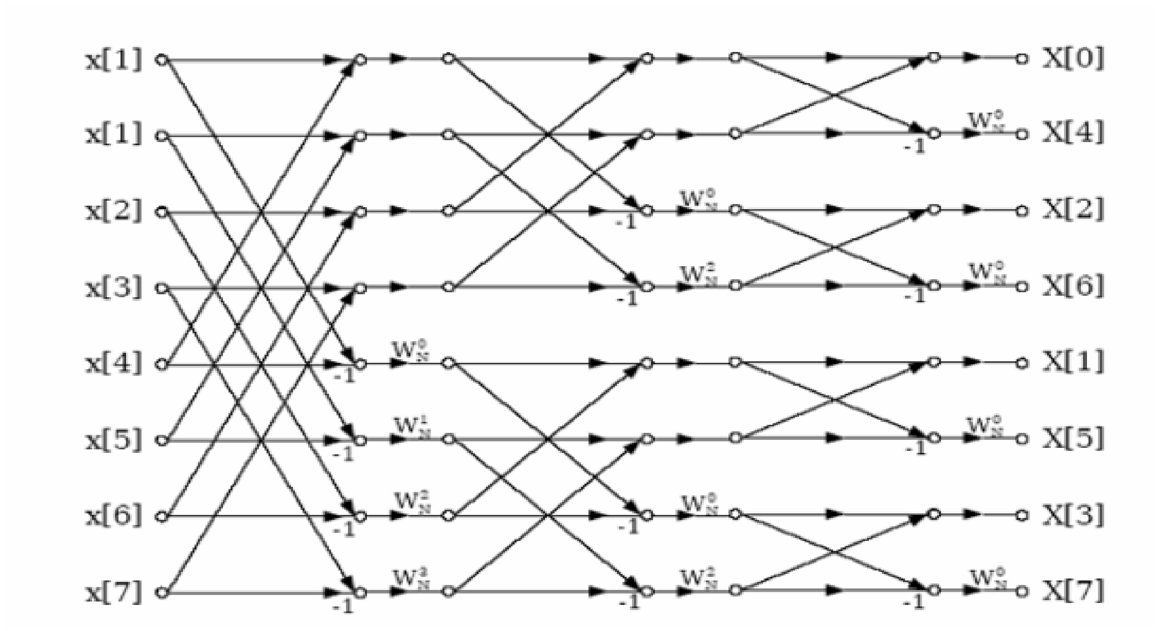


Fig 2.4

1.4.2. Multidimensional FFT:

An interesting property of the Fourier transform is its separability. When we extend the DFT to two dimensions we can rewrite it as

$$f[x,y] = \sum_{v=0}^{M-1} \left(\sum_{u=0}^{N-1} F[u,v] e^{2\pi jux/N} \right) e^{2\pi jvy/M}$$

$$F[u,v] = \frac{1}{M} \sum_{y=0}^{M-1} \left(\frac{1}{N} \sum_{x=0}^{N-1} f[x,y] e^{-2\pi jux/N} \right) e^{-2\pi jvy/M}$$

with $0 \le u, x \le N-1$ and $0 \le v, y \le M-1$.

I.e. we can compute a multi-dimensional Fourier transform by performing one-dimensional transforms in each dimension consecutively.

f(x,y) ↔ F(u.y) ↔ F(u,v)

This important result implies that the 2D DFT **F (u,v)** can be obtained by

- performing single dimension Fourier transform for function f(x,y) on variable x would give an transformed function F(u,y) as shown in fig 2.5

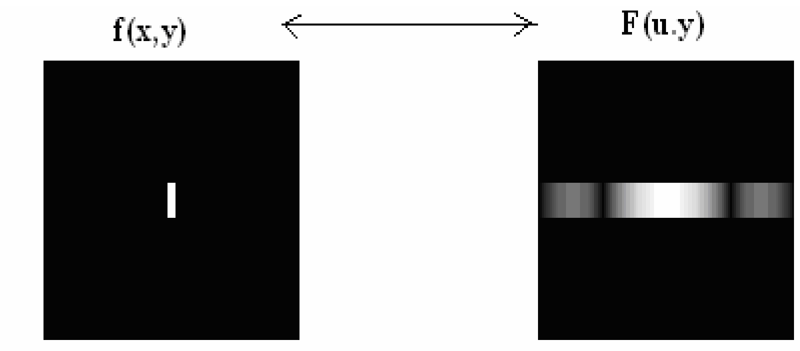


Fig 2.5

- performing single dimension Fourier transform for F(u,y) on variable y would give an transformed function F(u,v) as shown in fig 2.6

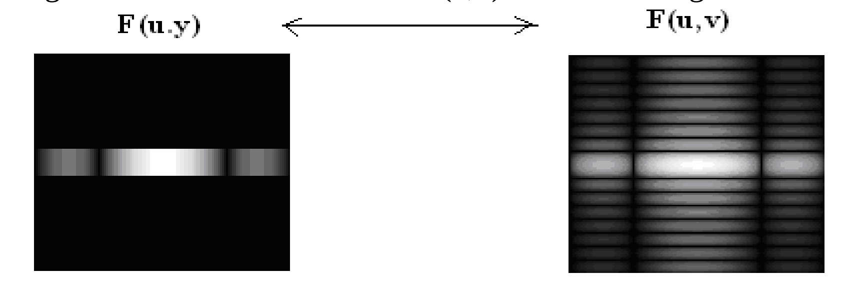


Fig 2.6

The same separable form also applies for the inverse 2D DFT.

1.5. CONVOLUTION THEOREM:

The convolution in time domain is equivalent to the product in frequency domain.

$$\boxed{x \circledast y \longleftrightarrow X \cdot Y.}$$

Proof:

$$\begin{aligned}
\mathrm{DFT}_k(x \circledast y) &\triangleq \sum_{n=0}^{N-1} (x \circledast y)_n e^{-j2\pi nk/N} \\
&\triangleq \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} x(m) y(n-m) e^{-j2\pi nk/N} \\
&= \sum_{m=0}^{N-1} x(m) \sum_{n=0}^{N-1} \underbrace{y(n-m) e^{-j2\pi nk/N}}_{e^{-j2\pi mk/N} Y(k)} \\
&= \left(\sum_{m=0}^{N-1} x(m) e^{-j2\pi mk/N} \right) Y(k) \quad \text{(by the Shift Theorem)} \\
&\triangleq X(k) Y(k)
\end{aligned}$$

2.0 FPGA DESIGN FLOW:

The complete flow of the design process is shown in fig 3.0. It has many steps involved in it.

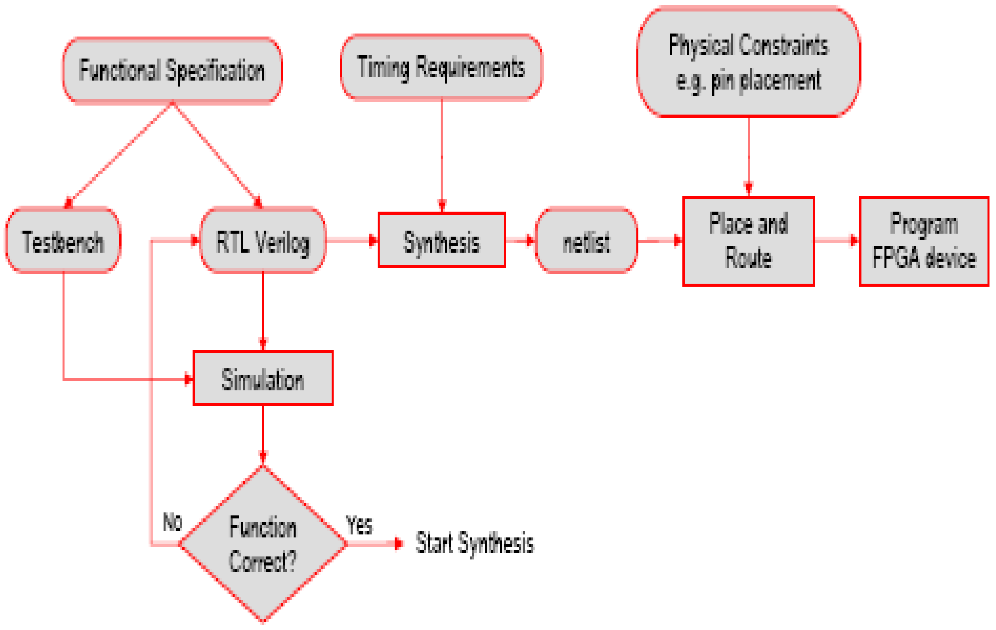


Fig 3.0

2.1 Functional Specification:

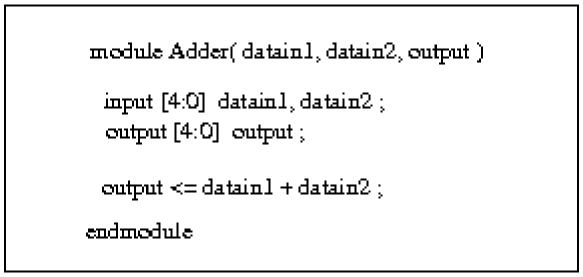
In this stage the designer need to understand the requirements of the system and specifying the abstract functionality of the system.

2.2 RTL Description/Simulation:

ASIC design descriptions are written by designers at different levels of abstraction. Most common hardware description languages used by designers are Verilog and VHDL. Both these languages are equally capable of providing complex constructs to describe complex functionality. Behavioral modeling forms highest level of abstraction.

2.2.1. Behavioral description

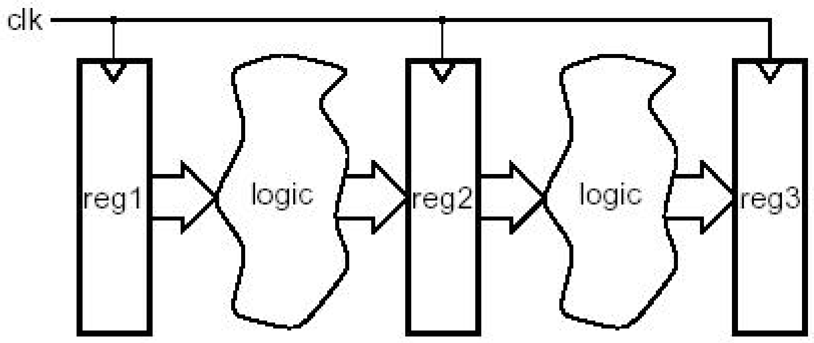
At initial stage of the design process the designer provides a Behavioral description of the functionality intended. Behavioral model does not care about the structure of the design, combinational or sequential elements used in the design, clock signal and the timing constraints involved. It captures the intended behavior of the design. The below given example describes the behavior of an adder that adds two four digit inputs to return and output. It is important to note that this description does not capture timing information.



```
module Adder( datain1, datain2, output )

 input [4:0]  datain1, datain2 ;
 output [4:0]  output ;

 output <= datain1 + datain2 ;

endmodule
```

2.2.2. RTL description

RTL stands for Register Transfer Level. In this model the entire design is split into registers with flow of information between these registers at each clock cycle. RTL description captures the change in design at each clock cycle. All the registers are updated at the same time in a clock cycle. Typically an RTL description divides design into registers and the logic blocks that join those registers together. RTL captures the data flow but fails to give a good description of control flow.



Register Transfer Level description of a design.

2.2.3. Structural Description

Structural description consists of a network of instances of logic gates and registers described by a technology library as shown in fig 3.1 . Technology library is provided by fabrication houses. Technology library is a description of simple AND, OR, NOT and complicated multiple functionality cells. The description of a cell includes its geometry, delay and power characteristics. Structural modeling describes circuit in form of instances of cells and interconnects between those cells.

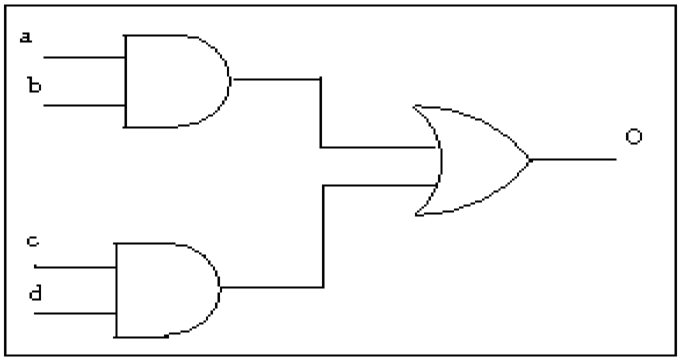


Fig 3.1: structural modeling.

2.2.4. Simulation:

Logic simulation is an essential part of digital circuit design. Logic simulation and verification are used to verify the functionality described by a design description against output values expected at the output ports of a digital integrated circuit.

There are mainly three classes of logic simulators:

- compiled code logic simulators
- event-driven logic simulators and
- compiled-code event driven simulators.

2.3. Synthesis:

Synthesis is the process of converting a design expressed in register-transfer level Hardware Description Language (HDL) into a netlist of gates or logic primitives that can be mapped to the cells in the technology library.
Synthesis involves three major steps:

- Transition from RTL description into gates and flip-flops
- Optimization of logic, and
- Placement and routing of optimized netlist.
-

Most of the intelligence resides in optimization stage but modern synthesis tools apply many smart techniques while converting RTL description into gates in order to reduce the number of gates in the design.

The synthesis tool generates various report file

2.3.1. Area report: shows the designer how much of the resources of the chip the design has consumed. The designer can tell if the design is too big for a particular chip and the designer needs to target a larger chip, if the design should go into a smaller chip, or if the current chip will work fine. The designer can also get a relative size of the design to use in later stages of the design process

2.3.2. Timing report: shows the timing of critical paths or specified paths of the design. The designer examines the timing of the critical paths closely because these paths ultimately determine how fast the design can run. If the longest path is a timing critical part of the design and is not meeting the speed requirements of the designer, then the designer may have to modify the HDL code or try new timing constraints to make the path meet timing.

The most important type of output data is the netlist for the design in the target technology. This output is a gate or macro level output in a format compatible with the place and route tools that are used to implement the design in the target chip. For instance, most place and route tools for FPGA technologies take in an EDIF netlist as an input format. The primitives used in the netlist are those used in the synthesis library to describe the technology. The place and route tools understand what to do with these primitives in terms of how to place a primitive and how to route wires to them.

2.4. Place & Route:

Place-and-route (P&R) describes several processes where the netlist elements are physically places and mapped to the FPGA physical resources, to create a file that can be downloaded in the FPGA chip.
Place and route tools are used to take the design netlist and implement the design in the target technology device. The place and route tools place each primitive from the netlist into an appropriate location on the target device and then route signals between the

primitives to connect the devices according to the netlist. One input to the place and route tools is the netlist in EDIF or another netlist format. Another input to some place and route tools is the timing constraints, which give the place and route tools an indication about which signals have critical timing associated with them and to route these nets in the most timing efficient manner. These nets are typically identified during the static timing analysis process during synthesis. These constraints tell the place and route tool to place the primitives in close proximity to one another and to use the fastest routing. The closer the cells are, the shorter the routed signals will be and the shorter the time delay. Some place and route tools allow the designer to specify the placement of large parts of the design. This process is also known as floor planning. Floor planning allows the user to pick locations on the chip for large blocks of the design so that routing wires are as short as possible. The designer lays out blocks on the chip as general areas. The floor planner feeds this information to the place and route tools so that these blocks are placed properly. After the cells are placed, the router makes the appropriate connections.
After all the cells are place and routed, the output of the place and route tools consists of data files that can be used to implement the chip. In the case of FPGAs, these files describe all of the connections needed to fuse FPGAs macro cells to implement the functionality required. Anti-fuse FPGAs use this information to burn the appropriate fuses while reprogrammable devices download this information to the device to turn on the appropriate transistor connections. The other output from the place and route software is a file used to generate the timing file. This file describes the actual timing of the programmed FPGA device or the final ASIC device. This timing file, as much as possible, describes the timing extracted from the device when it is plugged into the system for testing. The most common format of this file for most simulators is the SDF (Standard Delay Format).

3. VERILOG:

Is a hardware description language that can describe hardware not only at the gate level and the register-transfer level, but also at the algorithmic level.There are two general styles of description: *behavioral* and *structural*. Structural Verilog describes how a module is composed of simpler modules or of basic primitives such as gates or transistors. Behavioral Verilog describes how the outputs are computed as functions of the inputs. There are two general types of Statements used in behavioral Verilog. *Continuous assignment* statements always imply combinational logic. *Always* blocks can imply combinational logic or sequential logic, depending how they are used. It is critical to partition your design into combinational and sequential components and write Verilog in such a way that you get what you want. If you don’t know whether a block of logic is combinational or sequential, you are very likely to get the wrong thing.

3.1. Modeling with Continuous Assignments:

A Verilog module is like a “cell” or “macro” in schematics. It begins with a description of the inputs and outputs, which in this case are 32 bit busses. In the structural description style, the module may contain

Assign statements, always blocks, or calls to other modules.

```
module adder(a, b, y);
  input [31:0] a, b;
  output [31:0] y;
    assign y = a + b;
  endmodule
```

3.1.2 Operators:

Verilog has three types of operators, they take either one, two or three operands. *Unary* operators appear on the left of their operand, *binary* in the middle, and *ternary* separates its three operands by two operators.

```
clock = ~clock;// ~ is the unary bitwise negation operator,
c = a || b;  // || is the binary logical or, a and b are the operands
r = s ? t : u;    // ?: is the ternary conditional operator, which
                      // reads r = [if s is true then t else u]
```

3.1.3 Reduction Operators:

Reduction operators imply multiple-input gate acting on a single bus. For example, the following module describes an 8-input AND gate with inputs A[0], A[1], …, A[7].

```
module and8(a, y);
  input [7:0] a;
   output y;
     assign y = &a;
     endmodule
```

As one would expect, |, ^, ~&, and ~| reduction operators are available for OR, XOR, NAND, and NOR as well. Recall that a multi-bit XOR performs parity, returning true if an odd number of inputs are true.

3.2. Useful Constructs:

3.2.1. Internal Signals:

Often it is convenient to break a complex calculation into intermediate variables. For example, in a full adder, we sometimes define the propagate signal as the XOR of the two inputs A and B. The sum from the adder is the XOR of the propagate signal and the carry in. We can name the propagate signal using a wire statement, in much the same way we use local variables in a programming language.

```
module fulladder(a, b, cin, s, cout);
input a, b, cin;
output s, cout;
wire prop;
```

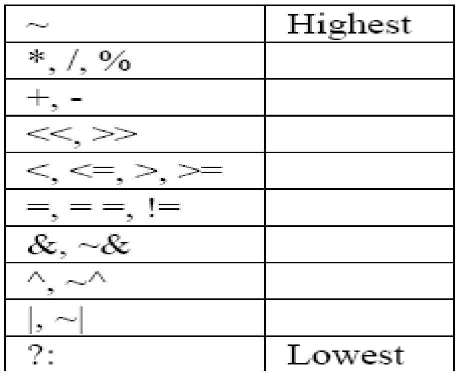
```
assign prop = a ^ b;
assign s = prop ^ cin;
assign cout = (a & b) | (cin & (a | b));
endmodule
```

Technically, it is not necessary to declare single-bit wires. However, it is necessary to declare multi-bit busses. It is good practice to declare all signals. Some Verilog simulation and synthesis tools give errors that are difficult to decipher when a wire is not declared.

3.2.2. Precedence:

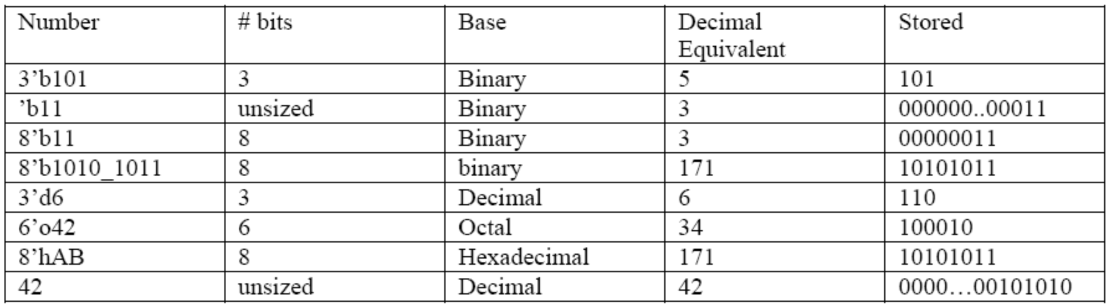
The order of precedence is important
Assign cout = a&b | cin&(a|b)

The operator precedence from highest to lowest is much as you would expect in other languages. AND has precedence over OR.



~	Highest
*, /, %	
+, -	
<<, >>	
<, <=, >, >=	
=, = =, !=	
&, ~&	
^, ~^	
|, ~|	
?:	Lowest

3.2.3. Constants:
Constants may be specified in binary, octal, decimal, or hexadecimal.



Number	# bits	Base	Decimal Equivalent	Stored
3'b101	3	Binary	5	101
'b11	unsized	Binary	3	000000..00011
8'b11	8	Binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	Decimal	6	110
6'o42	6	Octal	34	100010
8'hAB	8	Hexadecimal	171	10101011
42	unsized	Decimal	42	0000…00101010

3.2.4. Tristates:
It is possible to leave a bus floating rather than drive it to 0 or 1. This floating value is called 'z in Verilog. For example, a tri-state buffer produces a floating output when the enable is false.

```
module tristate(a, en, y);
input [3:0] a;
input en;
output [3:0] y;
assign y = en ? a : 4’bz;
endmodule
```

Floating inputs to gates cause undefined outputs, displayed as ’x in Verilog. At startup, state nodes such as the internal node of flip-flops are also usually initialized to ’x, as we will see later.

3.2.5. Bit Swizzling:

The {} notation is used to concatenate busses. For example, the following 8x8 multiplier produces a 16-bit result, which is, placed on the upper and lower 8-bit result busses.

```
module mul(a, b, upper, lower);
input [7:0] a, b;
output [7:0] upper, lower;
assign {upper, lower} = a*b;
endmodule
```

3.3 Modeling with Always Blocks:

Assign statements are reevaluated every time any term on the right hand side changes. Therefore, they must describe combinational logic. Always blocks are reevaluated only when signals in the header change. Depending on the form, always blocks may imply sequential or combinational circuits.

3.3.1 Flip-Flops:

Flip-flops are described with an always @(posedge clk) statement:

```
module flop(clk, d, q);
input clk;
input [3:0] d;
output [3:0] q;
reg [3:0] q;
always @(posedge clk)
q <= d;
endmodule
```

The body of the always statement is only evaluated on the rising (positive) edge of the clock. At this time, the output q is copied from the input d. The <= is called a nonblocking assignment.

3.3.2 Latches:

Always blocks can also be used to model transparent latches, also known as D latches. When the clock is high, the latch is transparent and the data input flows to the output. When the clock is low, the latch goes opaque and the output remains constant.

```
module latch(clk, d, q);
input clk;
input [3:0] d;
output [3:0] q;
reg [3:0] q;
always @(clk or d)
if (clk) q <= d;
endmodule
```

The latch evaluates the always block any time either clk or d change. If the clock is high, the output gets the input. Notice that even though q is a latch node, not a register node,

3.4. Combinational Logic:

Always blocks imply sequential logic when some of the inputs do not appear in the @ stimulus list or might not cause the output to change. For example, in the flop module, d is not in the @ list, so the flop does not immediately respond to changes of d. In the latch, d is in the @ list, but changes in d are ignored unless clk is high. Always blocks can also be used to imply combinational logic if they are written in such a way that the output always is reevaluated given changes in any of the inputs. The following code shows how to define a bank of inverters with an always block.

```
 module inv(a, y);
   input [3:0] a;
    output [3:0] y;
    reg [3:0] y;
     always @(a)
      y <= ~a;
        endmodule
```

3.5. Memories:

Verilog has an array construct used to describe memories. The following module describes a 64 word x 16 bit RAM that is written when wrb is low and otherwise read.

```
 module ram(addr, wrb, din, dout);
     input [5:0] addr;
      input wrb;
       input [15:0] din;
        output [15:0] dout;
         reg [15:0] mem[63:0]; // the memory
           reg [15:0] dout;
            always @(addr or wrb or din)
                          if (~wrb) mem[addr] <= din;
                              else dout <= mem[addr];
   endmodule
```

3.6 Blocking and Nonblocking Assignment:

Verilog supports two types of assignments inside an always block. *Blocking assignments* use the = statement. *Nonblocking assignments* use the <= statement. Do not confuse either type with the assign statement, which cannot appear inside always blocks at all.

```
   module shiftreg(clk, sin, q);
   input clk;
    input sin;
          output [3:0] q;
    // This is a correct implementation using nonblocking assignment
      reg [3:0] q;
       always @(posedge clk)
        begin
         q[0] <= sin; // <= indicates nonblocking assignment
          q[1] <= q[0];
           q[2] <= q[1];
            q[3] <= q[2];
                     end
  endmodule
```

the synthesized would be as shown in figure 3.2

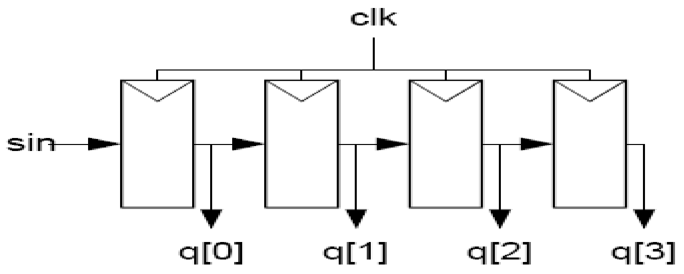


Fig 3.2

Consider the same module using blocking assignments. When clk rises, the Verilog says that q [0] should be copied from sin. Then q [1] should be copied from the new value of q [0] and so forth. All four registers immediately get the sin value which is not which we intended to design.

4.0. IMPLEMENTATION:

The basic block diagram for the implementation is shown in the figure 4.0

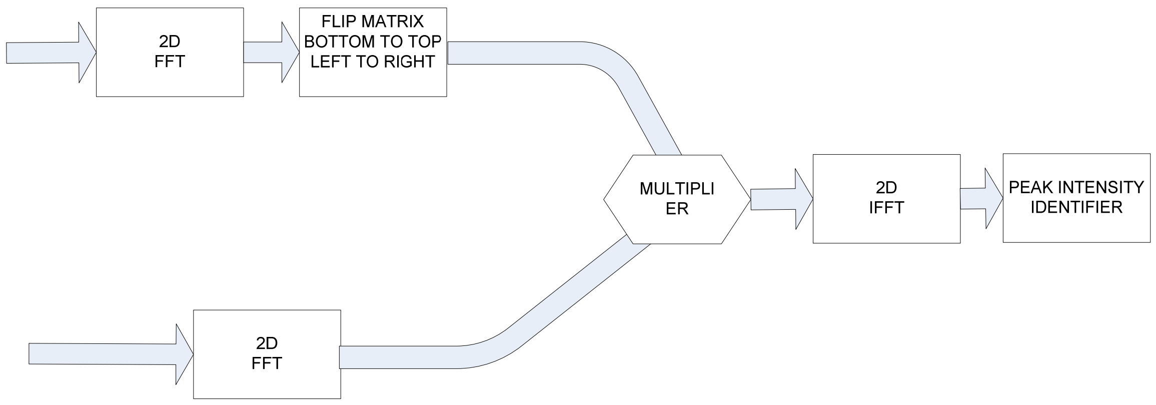

2D FFT
FLIP MATRIX BOTTOM TO TOP LEFT TO RIGHT
MULTIPLI ER
2D IFFT
PEAK INTENSITY IDENTIFIER
2D FFT

Fig 4.0

2D FFT:
This block converts the image which is a 2 dimensional NxN matrix from time domain to the frequency domain. Then the frequency domain image is forwarded to the further stages.

Flip Matrix: This flips the obtained matrix left to right and top to bottom. One of the matrices must be flipped in order to counteract the inherent flip present in convolution.

Multiplier: The block computes the product of the matrices element by element and forwards to the further stages.

2D IFFT: This computed the inverse Fast Fourier Transform of the convolved (frequency domain) matrix and converts into time domain which is nothing but the convolution of the two images.

Peak Intensity Identifier: This block scans the whole image and identifies the co-ordinates of the highest intensity which identifies the location of the image.

4.1. MATLAB IMPLIMENTATION:

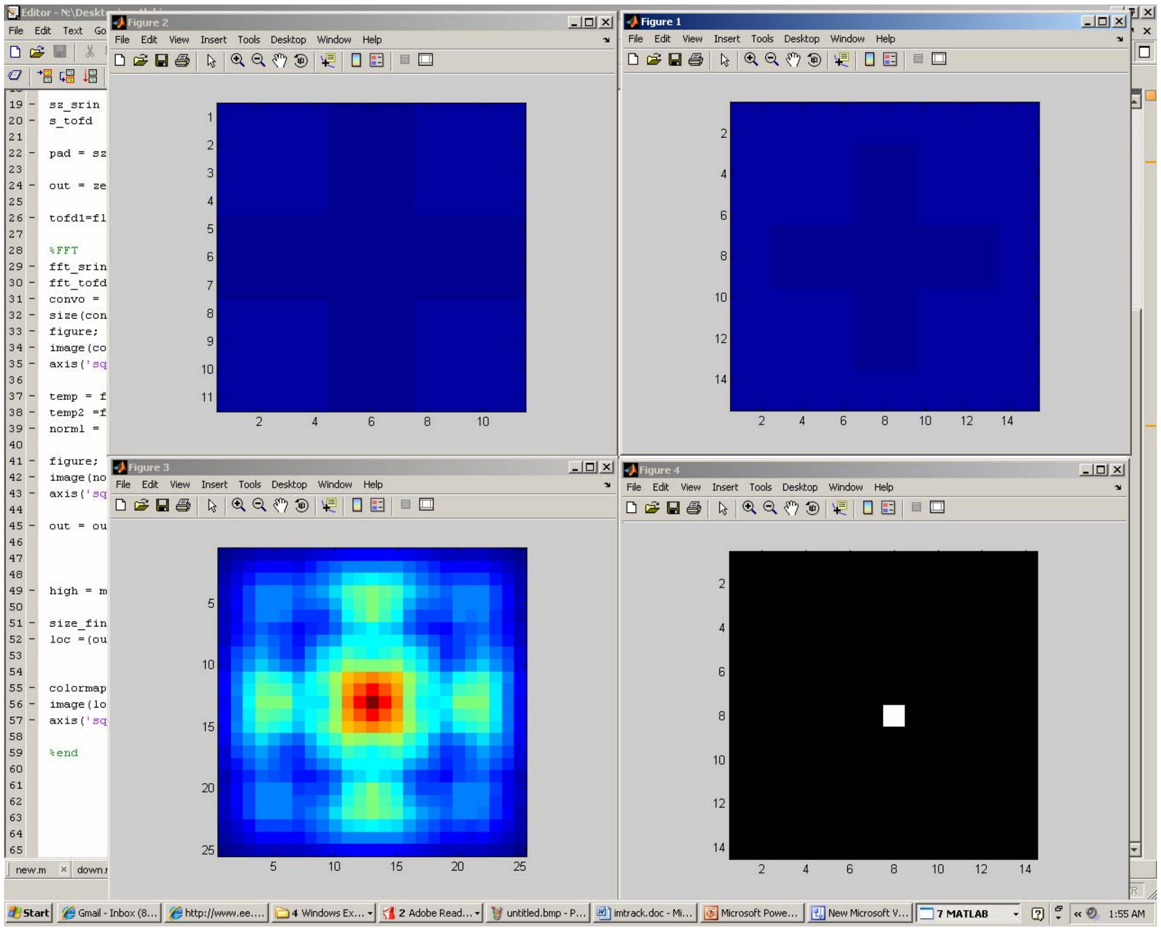
As Mat Lab tool has readily available built in functions for 2DFFT, IFFT and multiplication the above model has been prototyped for verifying the functionality.
The model has been simulated in Mat Lab and the results are plotted.

4.1.1.Matlab code:
```
clear
close all
%picv   =imread('complx.bmp');
picin  = double(imread('sao1.bmp'));
headin = double(imread('sao.bmp'));
figure;
image(uint8(headin));
axis('square');
% This is the threshold value. It is used to locate the point of
% highest correlation between the head and the picture. Typical
% values are 98 or 99 percent (.98, .99).
th = .98;
% Take the sizes of the two matrices.
size_pic = size(picin)
```

```
size_head = size(headin)
size_pic = size_pic(1:2);
size_head = size_head(1:2);
% To achieve a linear convolution, must zero-pad the matrices
% to a size 1 less than the sum of the two sizes.
pad = size_pic + size_head - [1 1];
 % Initialize final output matrix.
fin = zeros(pad);

% A convolution involves flipping the matrix up-down and
% left-right. We do not want this, so pre-flip the head
% matrix.
headin=flipud(fliplr(headin));

% Compute the linear convolution of the head with the picture by
% multiplying the fourier coefficients together and then taking
% the inverse fourier transfrom of it.
fft_pic = fft2(picin, pad(1), pad(2))
fft_head = fft2(headin, pad(1), pad(2))
convo = ifft2(fft_pic.*fft_head);
size(convo)

figure;
image(convo);
axis('square');

% A high amount of correlation will be detected wherever there is
% a large amount of brightness. To normalize this: compute the
% convolution of the picture squared(to make differences more
% extreme) with a matrix of ones the same size of the head matrix,
% then divide the previous convolution matrix with this one.
temp = fft2(picin,pad(1),pad(2));
temp2 =fft2(ones(size_head),pad(1),pad(2));
norml = ifft2(temp.*temp2).^.5;
fin = fin + convo./norml;
%end
 % Find the largest value in the final matrix.
high = max(max(fin));
 % Construct a new matrix, setting all values greater than a certain
% threshold equal to white and everything else equal to black (on a
% gray(2) colormap.
%loc = (fin >= high - .05) + 1;
size_fin = size(fin);
loc =(fin(ceil(size_head(1)/2):size_fin(1)-
ceil(size_head(2)/2),ceil(size_head(2)/2):size_fin(2)-
ceil(size_head(2)/2)) >= high*th) +1;
figure;
image(uint8(picin));
axis('square');
figure;
colormap(gray(2));
image(loc);
axis('square');
```

4.1.2. RESULTS FROM MATLAB SIMULATION:

SIMULATION1:

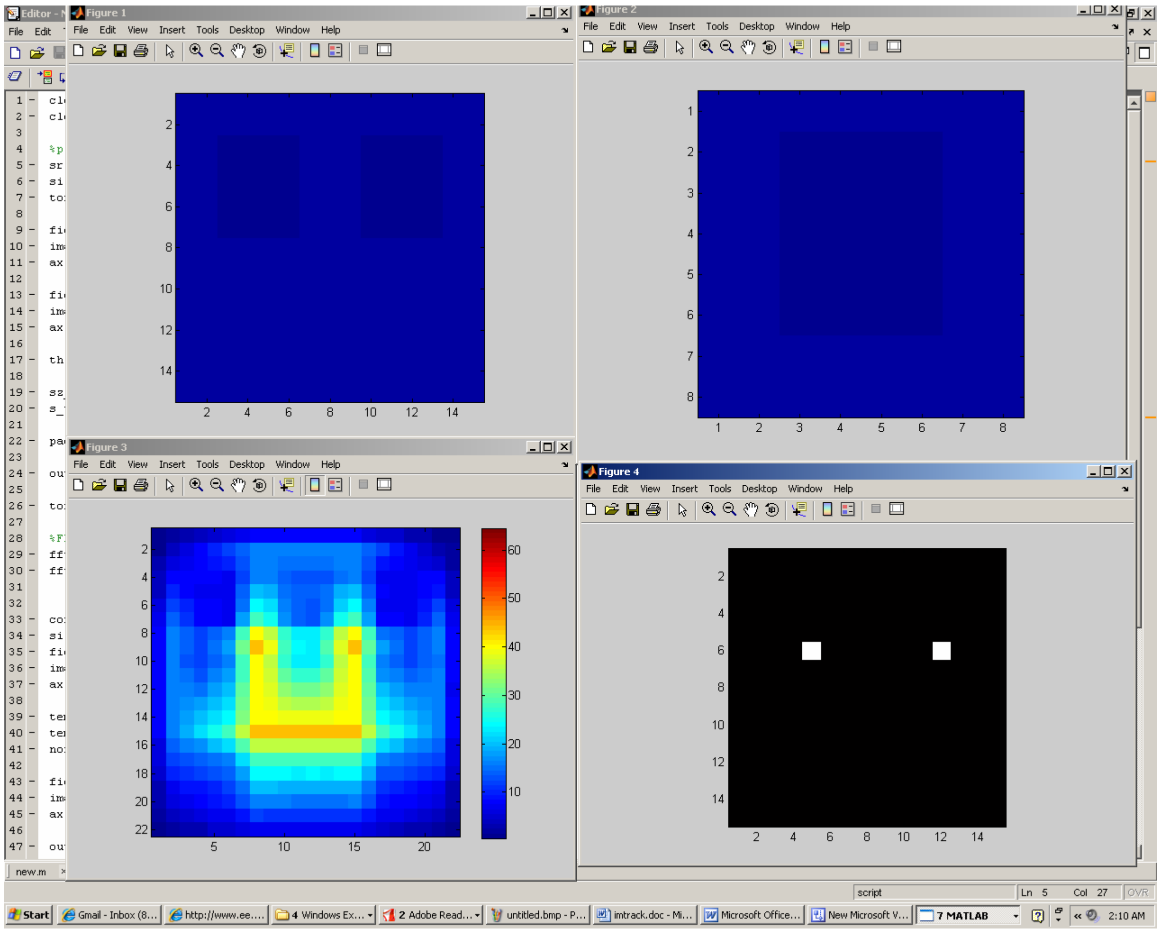


```
sz_srin
s_tofd

pad = sz

out = ze

tofd1=fl

%FFT
fft_srin
fft_tofd
convo =
size(con
figure;
image(co
axis('sq

temp = f
temp2 =f
norml =

figure;
image(no
axis('sq

out = ou



high = m

size_fin
loc =(ou


colormap
image(lo
axis('sq

%end
```

Window 1

From the Window1 above:
Figure 1: Is the image in which we need to search for match.
Figure 2: Is the Kernel image which is to be traced in Fig 1.
Figure 3: Is the convoluted image of images in figure1 and figure 2 .The dark red indicated the highest intensity.
Figure 4: shows the position where the kernel image is located in Figure1.

SIMULATION2:

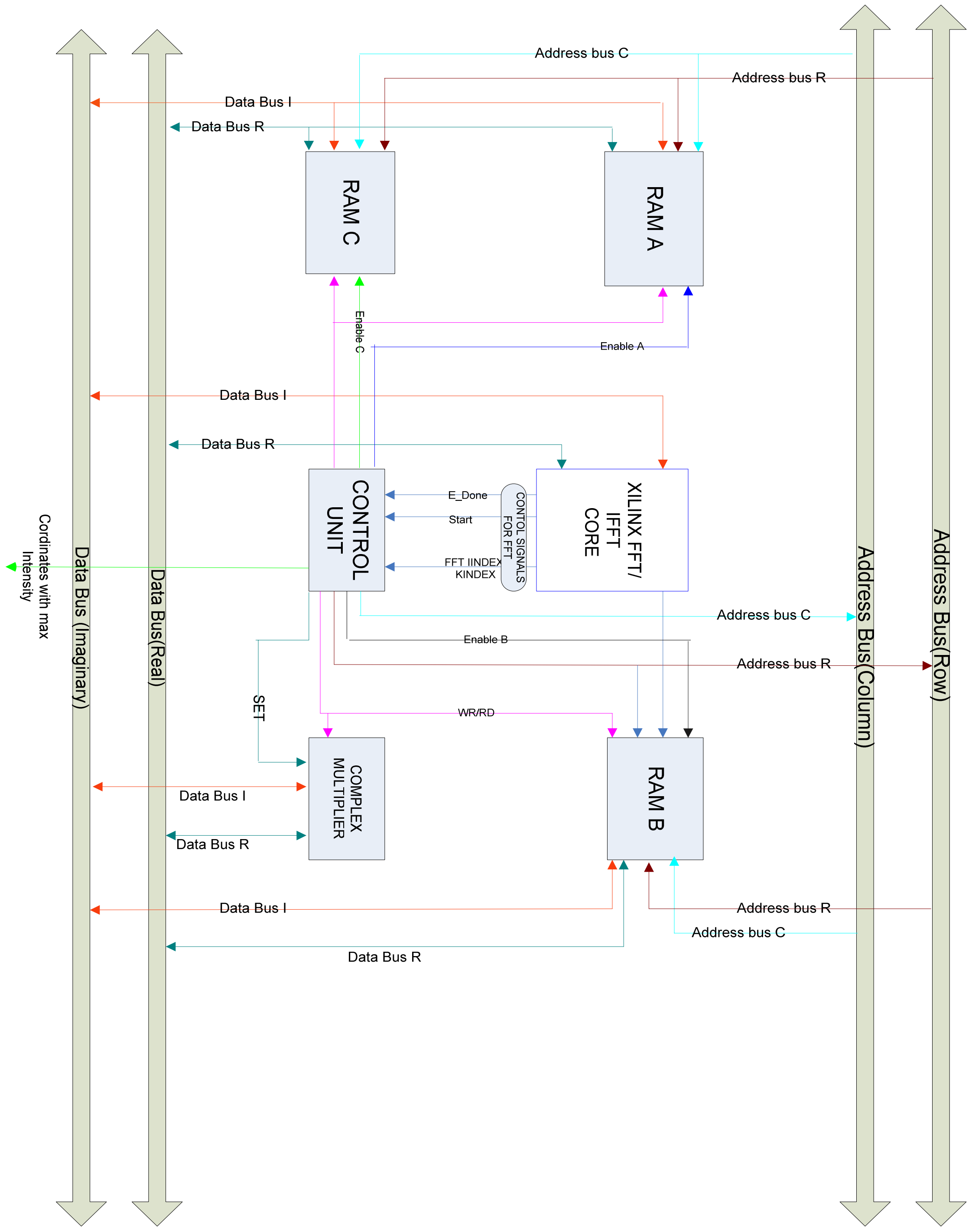


Window 2

From the window2
Figure 1: Is the image in which we need to search for match.
Figure 2: Is the Kernel image which is to be traced in Fig 1.
Figure 3: Is the convoluted image of images in figure1 and figure 2 .The dark red indicated the highest intensity.
Figure 4: shows the position where the kernel image is located in Figure1.

4.2. HW IMPLIMENTATION:

4.2.1 BLOCK DIAGRAM:

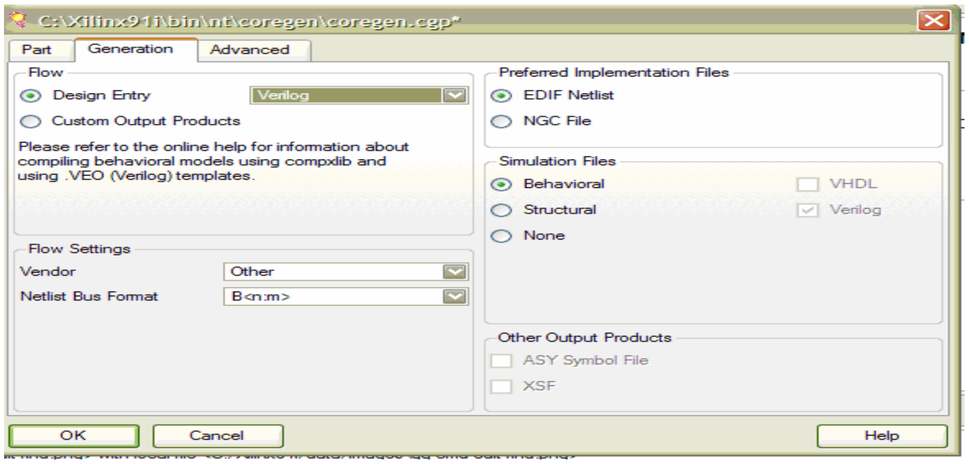


4.2.2. FFT:

To perform the FFT operation I am using the readily available Xilinx core (IP), with the configurations described in section 4.2.3

4.2.3. CORE GENERATION:

The FFT core is generated by using a Xilinx utility core generator. The core generator has a good graphical interface through which user can enter his configuration settings. The graphical view is shown in the Fig 4.1 below


C:\Xilinx91i\bin\nt\coregen\coregen.cgp*

Part | Generation | Advanced

Flow
- (●) Design Entry: Verilog
- ( ) Custom Output Products

Please refer to the online help for information about compiling behavioral models using compxlib and using .VEO (Verilog) templates.

Flow Settings

Vendor	Other
Netlist Bus Format	B<n:m>

Preferred Implementation Files
- (●) EDIF Netlist
- ( ) NGC File

Simulation Files
- (●) Behavioral
- ( ) Structural
- ( ) None
- ☐ VHDL
- ✓ Verilog

Other Output Products
- ☐ ASY Symbol File
- ☐ XSF

Fig 4.1

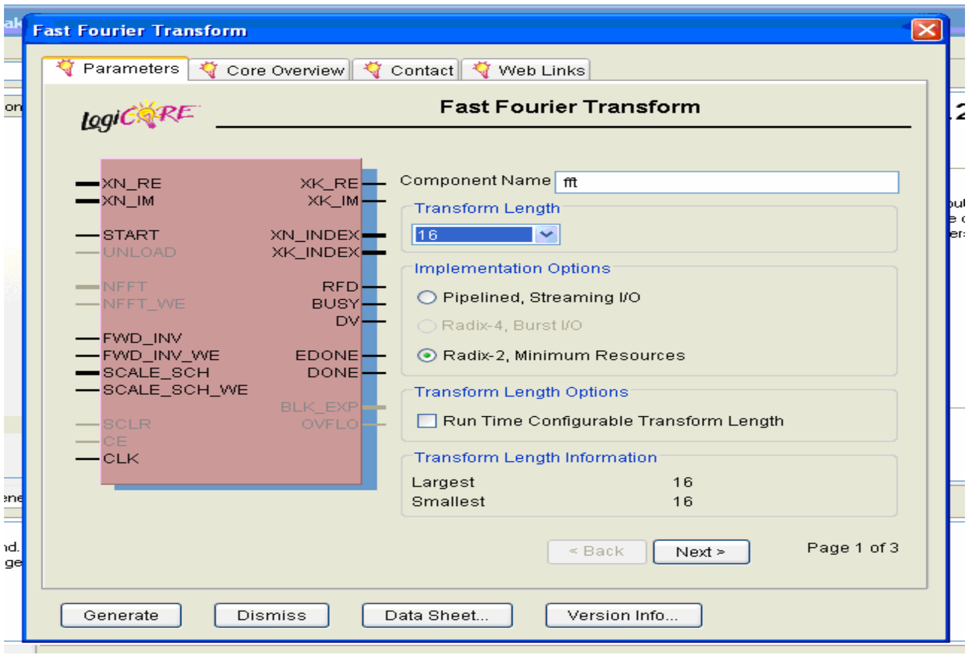
The Project options that the user should provide are the
Device/Family name: The name of the device for which the core is to be generated.
Output file type:

- Verilog/VHDL (Behavioral/Structural) file
- EDIF/NCG netlist file

The Parameters for generating the FFT core are

- Transform length
- Input data width
- Out put data width
- Pipeline implementation
- Scaling factor
- Use of block ram or distributed ram available in the device

The snap shot of the tool and configurable options is shown in fig 4.2



Fast Fourier Transform

Component Name fft

Transform Length

16

Implementation Options

( ) Pipelined, Streaming I/O

( ) Radix-4, Burst I/O

[✓ Radix-2, Minimum Resources]

Transform Length Options

☐ Run Time Configurable Transform Length

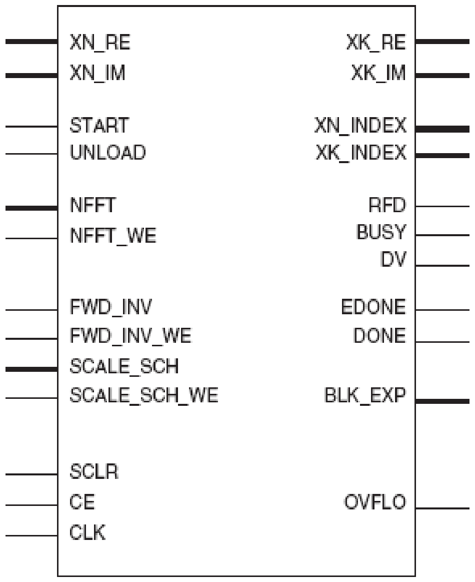
Transform Length Information

Largest	16
Smallest	16

Page 1 of 3

Fig 4.2

4.2.3.1. Core symbol and port definition:



Port Name	Port Width	Direction	Description
XN_RE	b_{xn}	Input	Input data bus: Real component (b_{xn} = 8 - 24) in two's complement format
XN_IM	b_{xn}	Input	Input data bus. Imaginary component (b_{xn} = 8 - 24) in two's complement format
START	1	Input	FFT start signal (Active High): START is asserted to begin the data loading and transform calculation (for the burst I/O architectures). For streaming I/O, START will begin data loading, which proceeds directly to transform calculation and then data unloading.
UNLOAD	1	Input	Result unloading (Active High): For the burst I/O architectures, UNLOAD will start the unloading of the results in normal order. The UNLOAD port is not necessary for the Pipelined, Streaming I/O architecture or for bit/digit reversed unloading.
NFFT	5	Input	Point size of the transform: NFFT can be the size of the transform or any smaller point size. For example, a 1024-point FFT can compute point sizes 1024, 512, 256, and so on. The value of NFFT is $\log_2$ (point size). This port is only used with run-time configurable transform length.

Port Name	Port Width	Direction	Description
RFD	1	Output	Ready for data (Active High): RFD is High during the load operation.
BUSY	1	Output	Core activity indicator (Active High): This signal will go High while the core is computing the transform.
DV	1	Output	Data valid (Active High): This signal is High when valid data is presented at the output.
EDONE	1	Output	Early done strobe (Active High): EDONE goes High one clock cycle immediately prior to DONE going active.
DONE	1	Output	FFT complete strobe (Active High): DONE will transition High for one clock cycle when the transform calculation has completed.
BLK_EXP	5	Output	Block exponent: The number of bits scaled for every point in the data frame. Available only when block-floating point is used.
OVFLO	1	Output	Arithmetic overflow indicator (Active High): OVFLO will be High during result unloading if any value in the data frame overflowed. The OVFLO signal is reset at the beginning of a new frame of data. This port is optional and only available with scaled arithmetic.

Port Name	Port Width	Direction	Description
NFFT_WE	1	Input	Write enable for NFFT (Active High): Asserting NFFT_WE will automatically cause the FFT core to stop all processes and to initialize the state of the core to the new point size on the NFFT port. This port is only used with run-time configurable transform length.
FWD_INV	1	Input	Control signal that indicates if a forward FFT or an inverse FFT is performed. When FWD_INV=1, a forward transform is computed. If FWD_INV=0, an inverse transform is performed.
FWD_INV_WE	1	Input	Write enable for FWD_INV (Active High).
SCALE_SCH	$2 \times ceil\left(\frac{NFFT}{2}\right)$ for PIpelined Streaming I/O and Radix-4 Burst I/O architectures or $2 \times NFFT$ for Radix-2 Minimum Resources where $NFFT$ is $\log_2$ (point size) or the number of stages	Input	Scaling schedule: For Burst I/O architectures, the scaling schedule is specified with two bits for each stage, starting at the two LSBs. The scaling can be specified as 3, 2, 1, or 0, which represents the number of bits to be shifted. An example scaling schedule for N=1024, Radix-4 Burst I/O is [1 0 2 3 2]. For N=128, Radix-2 or Radix-2-Lite, one possible scaling schedule is [1 1 1 1 0 1 2]. For Pipelined Streaming I/O architecture, the scaling schedule is specified with two bits for every pair of Radix-2 stages, starting at the two LSBs. For example, a scaling schedule for N=256 could be [2 2 2 3]. When N is not a power of 4, the maximum bit growth for the last stage is one bit. For instance, [0 2 2 2 2] or [1 2 2 2 2] are valid scaling schedules for N=512, but [2 2 2 2 2] is invalid. The two MSBs of SCALE_SCH can only be 00 or 01. This port is only available with scaled arithmetic (not unscaled or block-floating point).
SCALE_SCH_WE	1	Input	Write enable for SCALE_SCH (Active High): This port is available only with scaled arithmetic.
SCLR	1	Input	Master synchronous reset (Active High): Optional port.
CE	1	Input	Clock enable (Active High): Optional port.
CLK	1	Input	Clock
XK_RE	b_{xk}	Output	Output data bus: Real component in two's complement format. (For scaled arithmetic and block floating point arithmetic, $b_{xk}=b_{xn}$. For unscaled arithmetic, $b_{xk}=b_{xn}+NFFT+1$)
XK_IM	b_{xk}	Output	Output data bus: Imaginary component in two's complement format. (For scaled arithmetic and block floating point arithmetic, $b_{xk}=b_{xn}$. For unscaled arithmetic, $b_{xk}=b_{xn}+NFFT+1$)
XN_INDEX	$\log_2$ (point size)	Output	Index of input data.
XK_INDEX	$\log_2$ (point size)	Output	Index of output data.

4.2.3.2. Pipelined, Streaming I/O:

The core has been configured for Radix2 streaming I/O. The Pipelined, Streaming I/O solution pipelines several Radix-2 butterfly processing engines to offer continuous data processing. Each processing engine has its own memory banks to store the input and intermediate data. The core has the ability to simultaneously perform transform calculations on the current frame of data, load input data for the next frame of data, and unload the results of the previous frame of data. The user can continuously stream in input data and, after the calculation latency, can continuously unload the results. If preferred, this design can also calculate one frame by itself or frames with gaps in between. This architecture supports unscaled full-precision and scaled fixed point arithmetic methods. In the scaled fixed point mode, the data is scaled after every pair of Radix-2 stages. The unloaded output data can either be in bit reversed order or in natural order. By choosing the output data in natural order, additional memory resource will be utilized. The timing diagram for this implementation is shown in figure 4.3

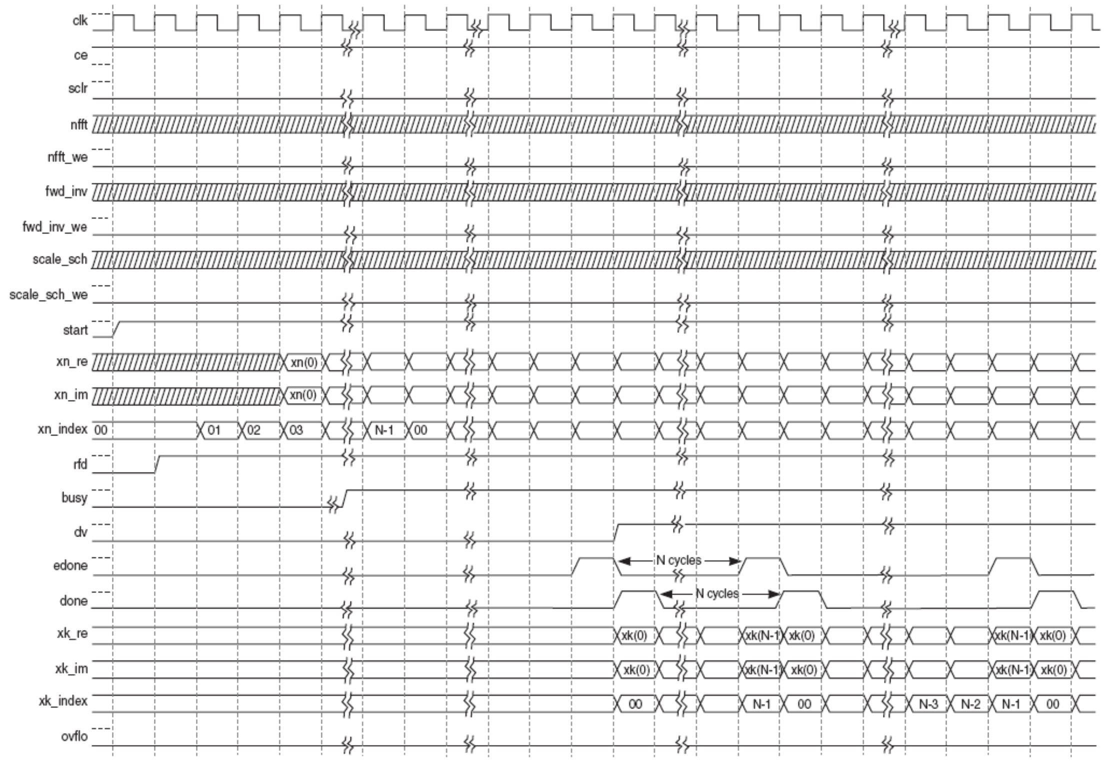
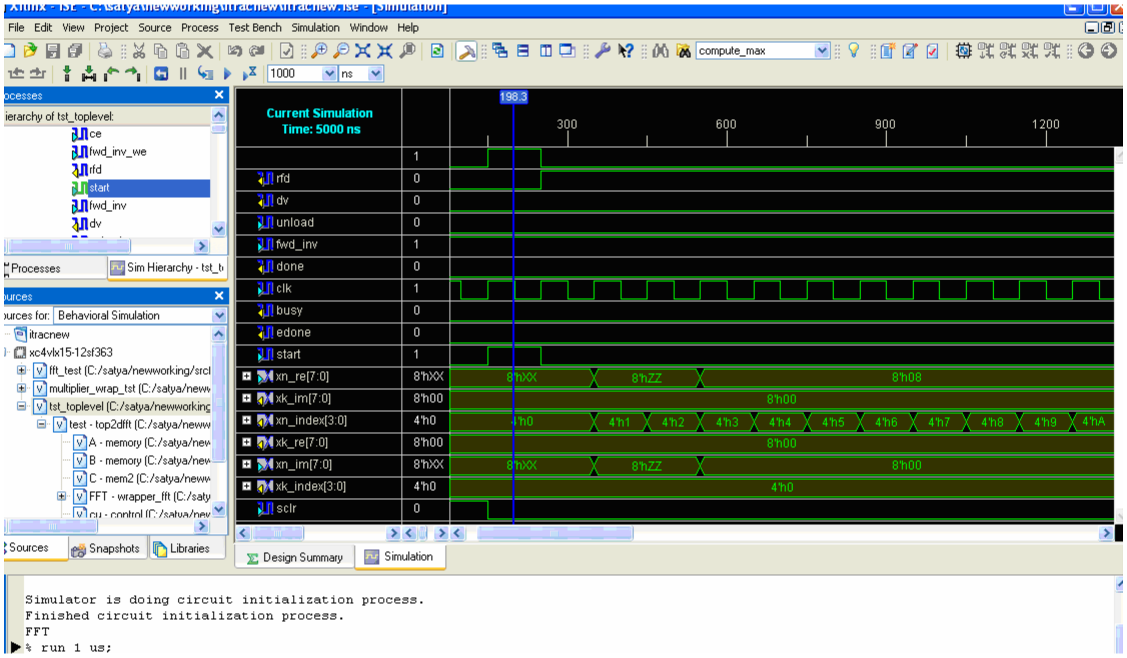


Fig 4.3

4.2.3.3. Timing diagram from simulations:



```
Simulator is doing circuit initialization process.
Finished circuit initialization process.
FFT
% run 1 us;
```

4.2.4. 2D FFT:

The two dimensional FFT is implemented by performing single dimensional FFT on the rows first and then performing single dimensional FFT on the columns of the resultant matrix. In order to optimize the area I am reusing the same FFT core to perform 2D FFT. The block diagram for this implementation is shown in fig 4.4.

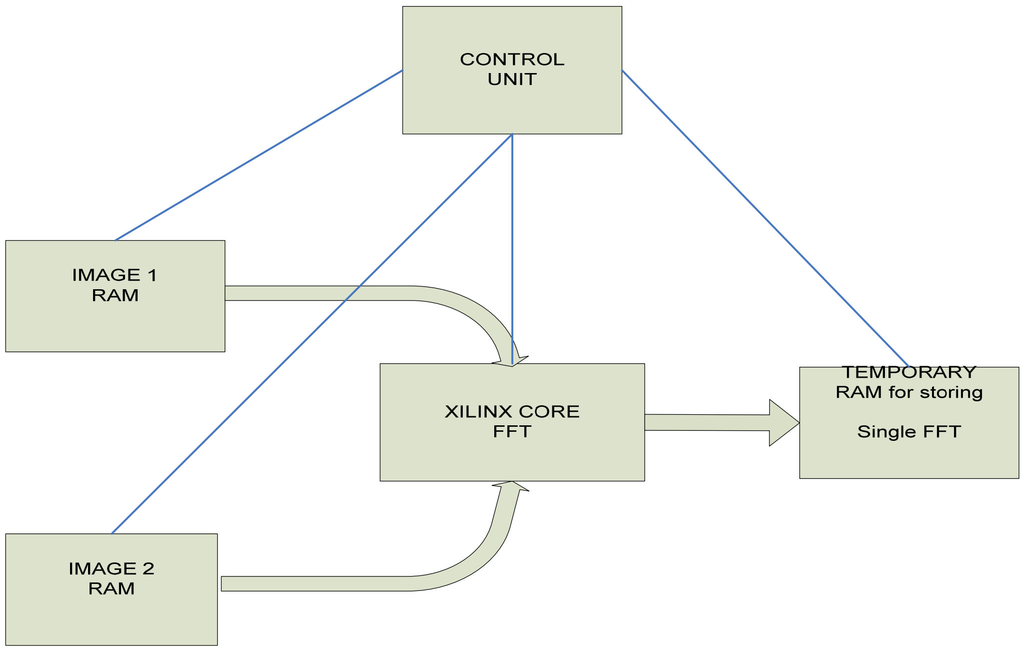


Fig 4.4

The control unit generates the control signals to perform 2D FFT using ID FFT. Initially the Images are loaded into Image 1 Ram and Image 2 Ram, now Xilinx core FFT reads the matrix in Image ram1 row by row performs FFT and stores in Transformed image ram1 column by column after completing all rows the Xilinx core FFT reads the matrix in Transformed Image ram1 row by row performs FFT and stores in image ram1 column by column. Once it is done with Image 1 its starts with the Image 2.

4.2.5. CONTROLUNIT:

The control unit is the heart of the design it co ordinates all the events by generating the corresponding control signals. The main tasks that are to be co ordinate by the control unit are

- Compute 2D FFT for IMAGE 1
- Compute 2D FFT for IMAGE 2
- Compute element by element matrix product of the two transformed Images
- Compute 2D IFFT of the resultant product matrix.
- Compute the coordinates of max value

Since the design shares a common address/data buss the control unit should also act as the bus arbitration. The control unit has 4 state machines and all the above tasks are performed sequentially. The logic is designed such that the control is passed form one state machines to other state machines which generate the respective control signal for getting the task to be done. The schematic is shown in fig 4.5 is obtained from the synthesized RTL viewer from the tool.

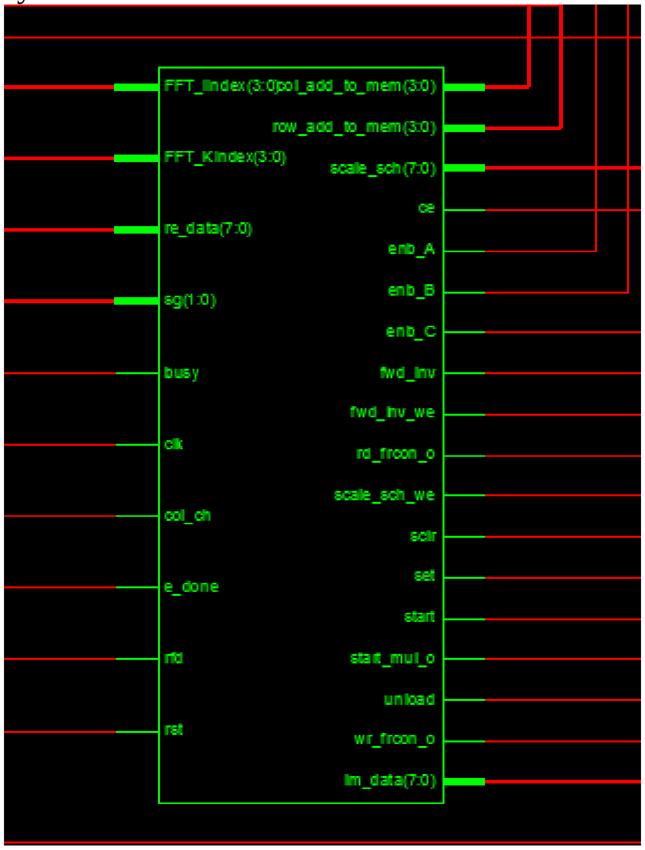


Fig 4.5

4.2.5.1 Pin details:

Output Ports:

Enb_A,
Enb_B,
Enb_C : The enable signal for Ram. (Active low signal)
Start : when asserted the FFT/IFFT block would start reading the vector on which the transformed is to be performed.
Sclr : reset to FFT.
Unload: on asserting it, FFT would start writing the transformed to ram locations.
Row_Add_to_mem
Col_Add_to_mem: Address bus
Rst : external reset (Active high)

Inputs Ports:

CLK: global clock signal
Rfd : output from FFT when asserted indicates the FFT is ready to read the data.
Sg[1:0] : out put from the multiplication unit used as an acknowledgment for changing state of state machine .
Busy: output from FFT indicates that FFT is busy in performing the transform.
FFTIINDEX/FFTKINDEX: Out put from FFT indicates the index of element that's is being written or read from.

Inout:

Re_data[7:0] :real data bus.
Im_data[7:0] :imaginary data bus.

4.2.5.2. State Diagram for Computing 2d FFT/IFFT:

The main difference between FFT/IFFT is asserting/disserting a signal FWD_INV.
If FED_INV = 1 the core computes FFT
If FED_INV = 0 the core computes IFFT

The state diagram is shown in fig 4.6.

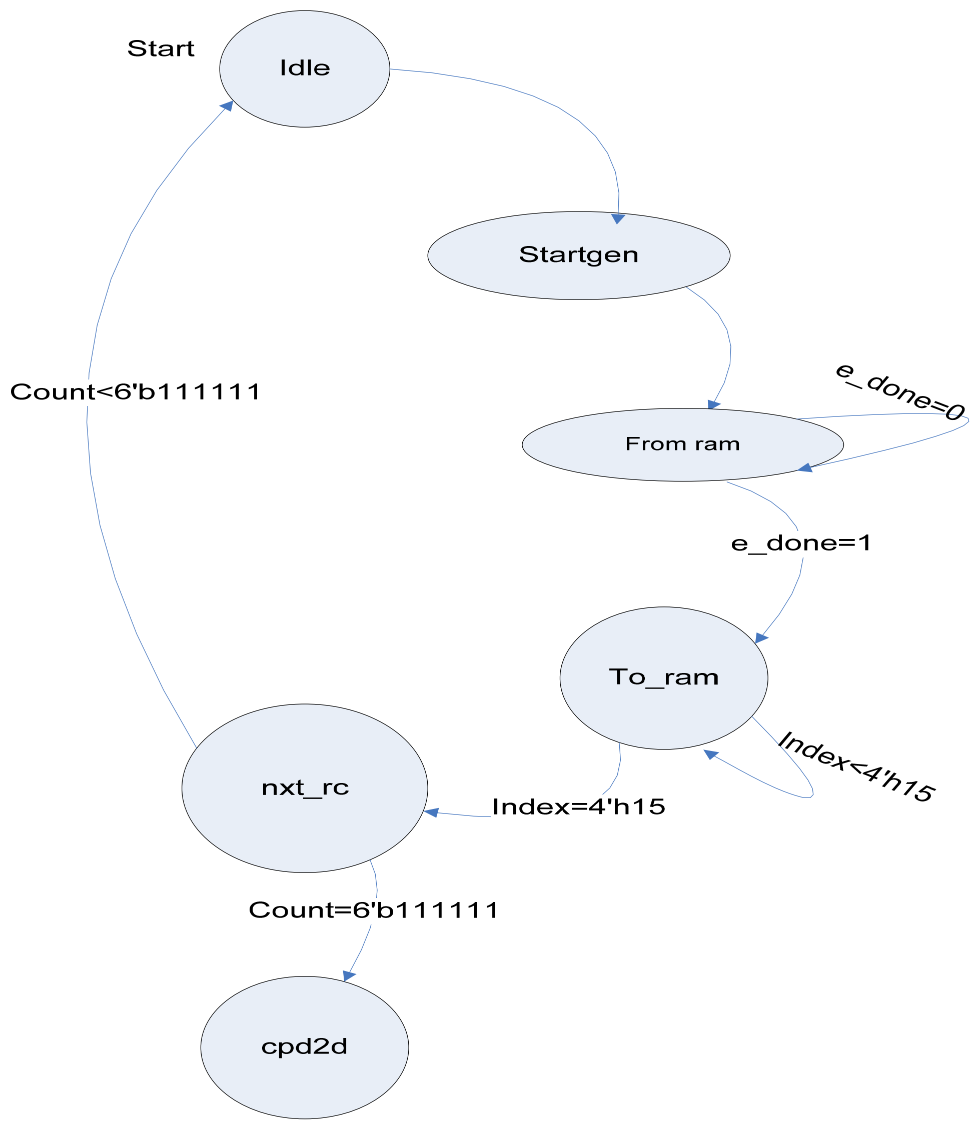


Fig 4.6

Description:
The brief explanation of the importance of each state, transactions of states and the out puts that are generated are described below
All the transaction are Synchronized
Idle: In this state the machine resets all the conditions like enables, FFT forward inverter, and Scaling factor for FFT. On the positive edge of the clock the machine moves to the next state Startgen.
Startgen: As the name indicates this state would initiate the computation of FFT by asserting a start signal. On the positive edge of the clock the machine moves to the next state Fromram.
Fromram: This state generates the address and read signal to ram and the data is fed to FFT for performing transformation. Machine waits in this state until it receives a signal (e_done) from FFT indicating that computation of the transform is done and is ready to issue results.

To_ram: In this state the machine generates the address and write signal, enable to ram and the data is stored in the ram, based on the counter value is (transform length) the machine moves to nxt_rc state.
nxt_rc: This state decides whether the transformation of an image is completed or not if not it restarts the cycle again .If completed it goes to a completed state (Cpd2d).
Cpd2d: in this state it asserts signals to activate other state machines for further processing.

4.2.5.3. State Diagram for convolution (MULTIPLICATION):

The convolution is performed by element by element product of the transformed images in the ram. The state diagram is shown in fig 4.7. The steps involved in the state machine are

- Read a data from an address in ramA(image1) feed the data to multiplier
- Read a data from an bit inverted address in ramC(image2) feed the data to multiplier
- Generate control signal to initiate multiplication
- Wait for the acknowledgment from the multiplier (Result computed)
- Write the data back to ramA

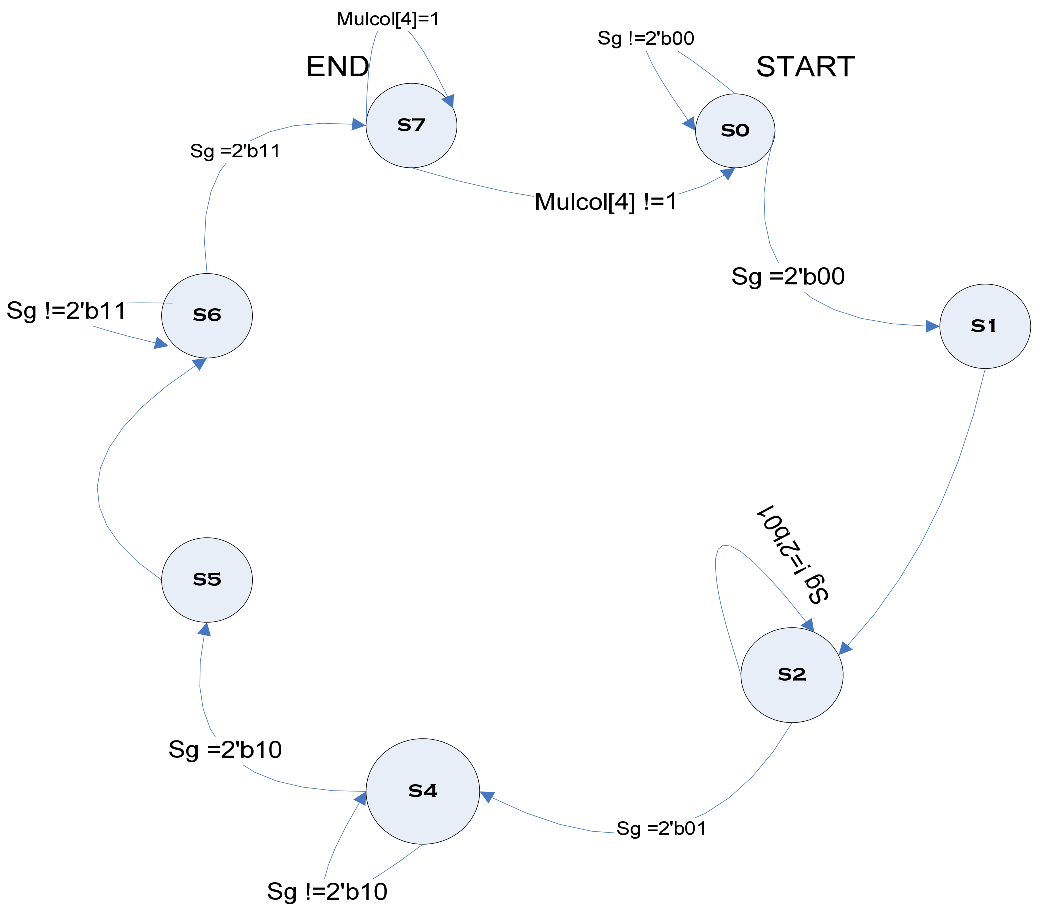


Fig 4.7

Description:

S0: is the start state
Generates address/control signal to read data from ram A and stores it in a register in multiplier.
S1: is the delay state for data to be latched properly.
S2: Generates address/control signal to read data from ram C and stores it in other register in multiplier.
S4: waits for acknowledgement (sg) from multiplier that result is computed. (It would take 4 cycles**).**
S5: Initiates the process of writing back the result to the ramA.
S6: waits until the data is written properly.
S7: checks whether multiplication for whole array elements are done if not start again by proceeding to state S0.

4.2.5.4. State Diagram for computing maximum value of the resultant matrix:
This state machine is invoked after IFFT is done. This block scans whole array and remembers the co ordinates (Row,columm) which has the max value. This is done by a simple start machine as shown in fig

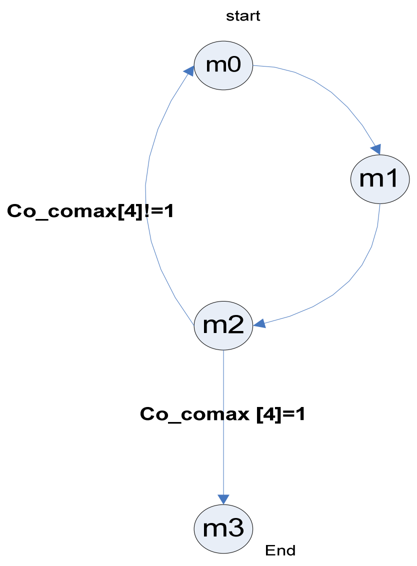
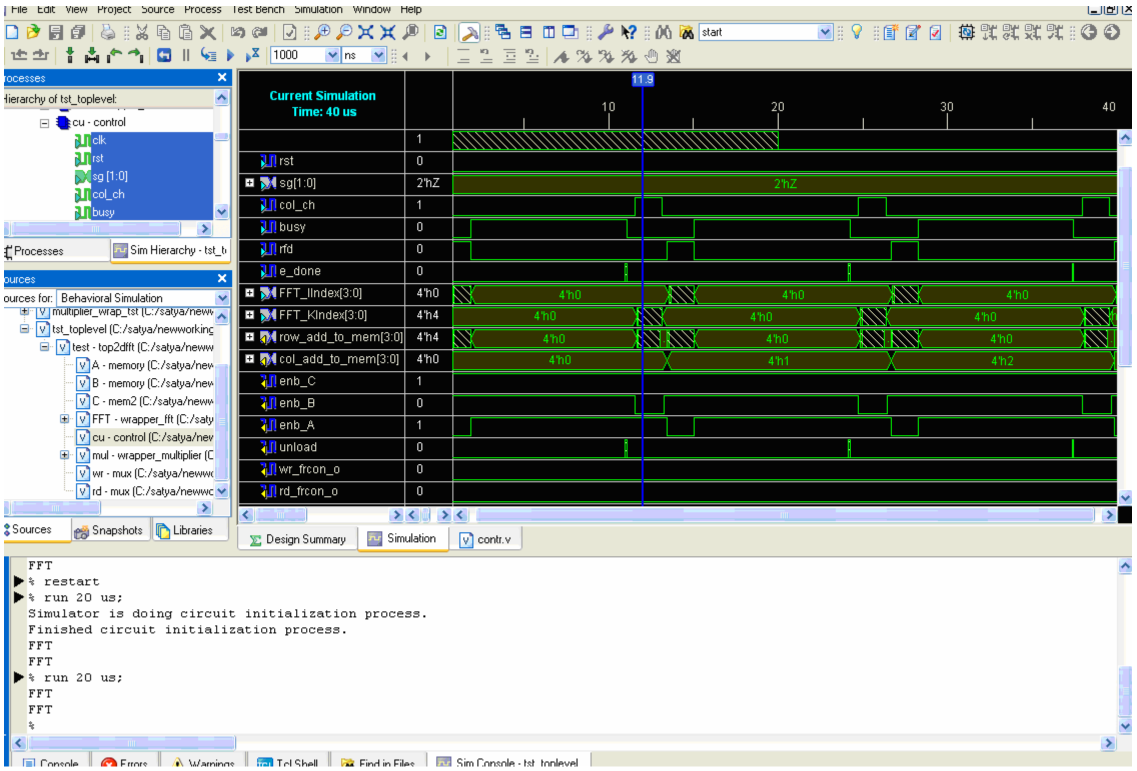


Fig 4.8

Mo: In this state the control unit generates address; enable signal to the ram read the data.
M1: The read signal is generated.
M2: The data received is compared with the Max value if data > Max value the max value is updated and co ordinates are remembered, and test is performed to check if whole array is compared, if so it proceeds to m3 or it goes to m0 to check other elements.
M3: It is the end and indicates the max value coordinates are computed.

Simulation Timing diagram:



```
FFT
% restart
% run 20 us;
Simulator is doing circuit initialization process.
Finished circuit initialization process.
FFT
FFT
% run 20 us;
FFT
FFT
%
```

4.2.6. MULTIPLIER:

Since the outputs from the FFT are complex numbers a complex multiplier must be designed which performs matrix product element by element (convolution).The complex multiplier is generated by using Xilinx coregen. As the design shares single data bus the multiplier should get the operands sequentially. The schematic of multiplier is shown in fig

The Multiplication has three stages

- Read operand 1 from an **address** from Ram1
- Read operand 2 from an **inverted !(address)** from Ram2
- Perform product and write back to address of operand1 in Ram1.

The corresponding rd/wr and address/! (Address) are generated by control unit, but the multiplies unit should acknowledge the events. The multiplier unit uses signal Sg[1:0] as an acknowledgement to the control unit based on which the state changes are done in control unit..

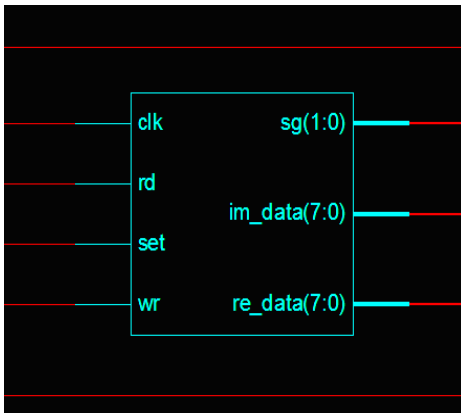


Fig 4.9

Pin Details:

Input:
Clk: global clock signal
Rd: signal for reading the computed result.
Wr: signal for writing the operand values
Set: active high reset.

Output:
Sg[1:0]: an out put signal to control unit acts as an acknowledgment ,by which the control unit changes state

Inout:
Im_data(7:0) :imaginary data bus
Re_data(7:0) : real data bus.

4.2.6.1. Xilinx Core implementation:

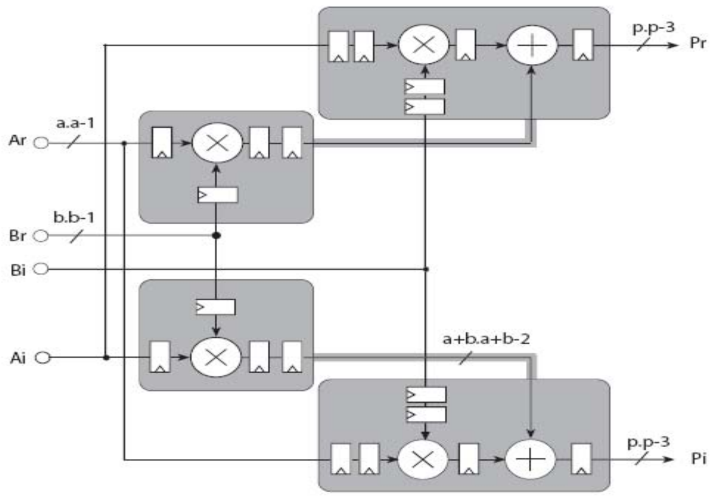


Fig 4.10

The Xilinx core using 4 real multipliers is shown in the figure Fig 4.10
Since the core accepts both the operands at a time to perform computation, but since we have a single data bus we could access single operand at a time so a wrapper multiplier is used to read the operand one by one and provide both the operands at a time to the core unit. So the block diagram would look like that as shown in figure 4.11

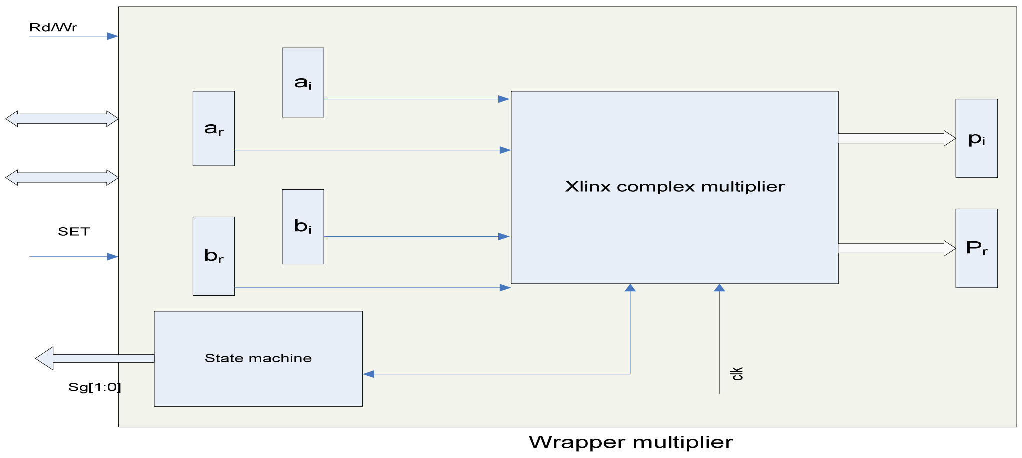


Fig 4.11

4.2.6.2 State Diagram:

The state machine shown in the wrapper function helps in acknowledging the control unit by writing the operands data into the registers, generating a control signal to initiate the multiplication and invoke the control unit when the operation is completed indicating the result is ready to read. The state diagram is shown in fig 4.12. The state machine is having 7 states and each state is explained in brief.

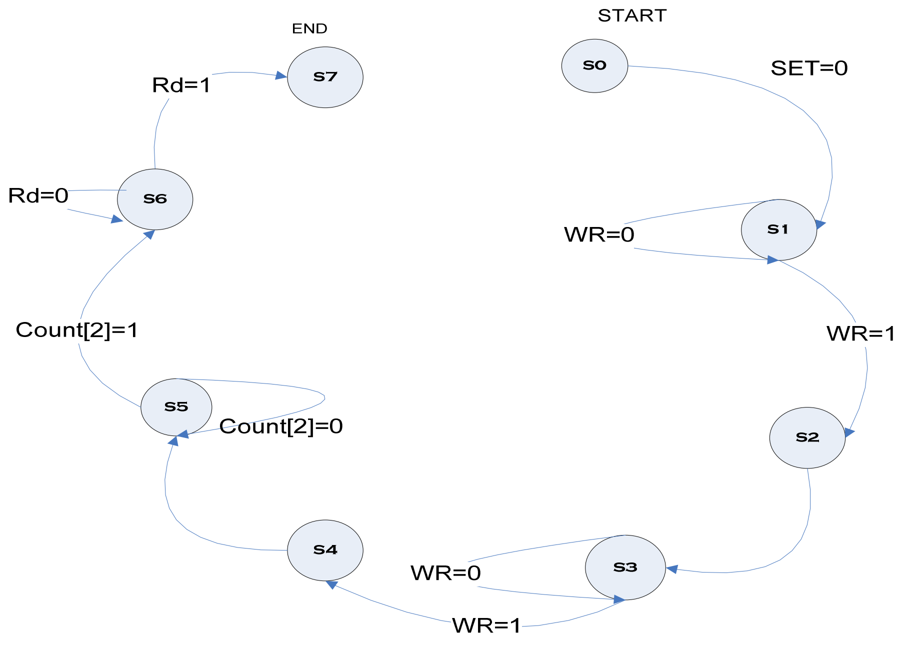
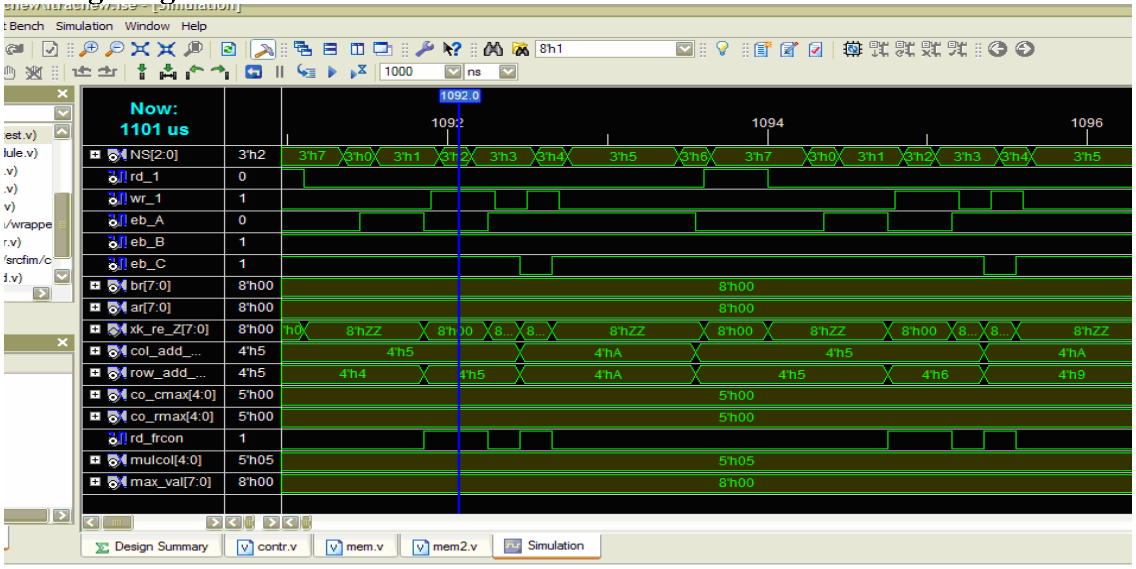


Fig 4.12

S0: Is a start state stays in it unless set value driven by the control unit is zero.
S1: in this state the data from the bus is copied to the register A.
S2: Is an ideal state inserted to get the date properly latched into register A.
S3: in this state the data from the bus is copied to the register B.
S4: Is an ideal state inserted to get the date properly latched into register B
S5: wait unit the result is computed it would take 4 cycles.
S6: data is placed in the bus for control unit to read it.
S7: is the ideal end state.

The multiplied result is truncated to 8 bits preserving the signed bit.

Timing Diagram:



4.2.7. RAM:

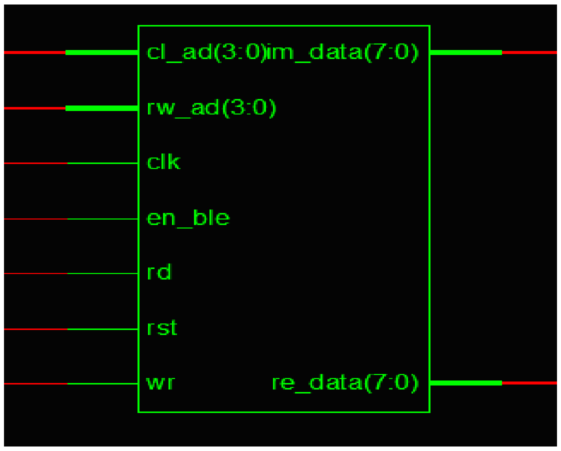


Fig 4.13

The schematic of the ram module implemented in the design is shown in fig 4.13 .The ram has 2 sub modules Real and Imaginary Block as shown in fig 4.14, both the blocks share common address bus; control signal but both of them have a different data bus. The memory in the design is declared as a register of 16X16 elements and each element can be 8 bits, upon reset the image is loaded into the ram.

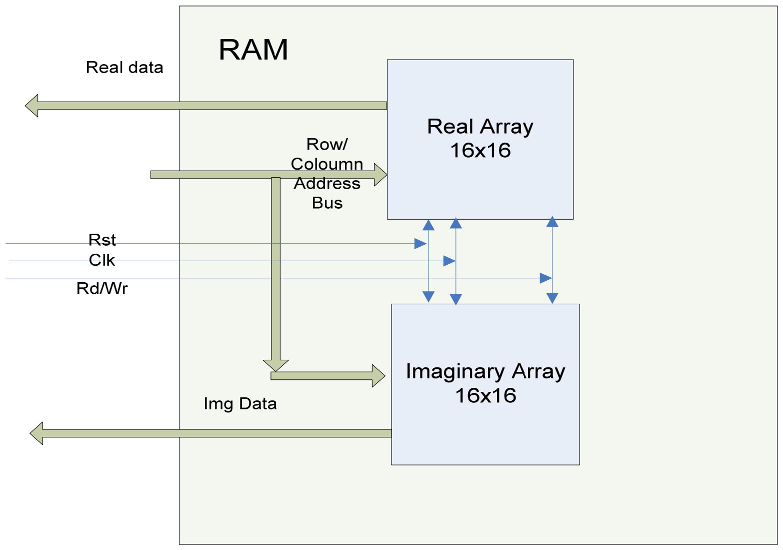
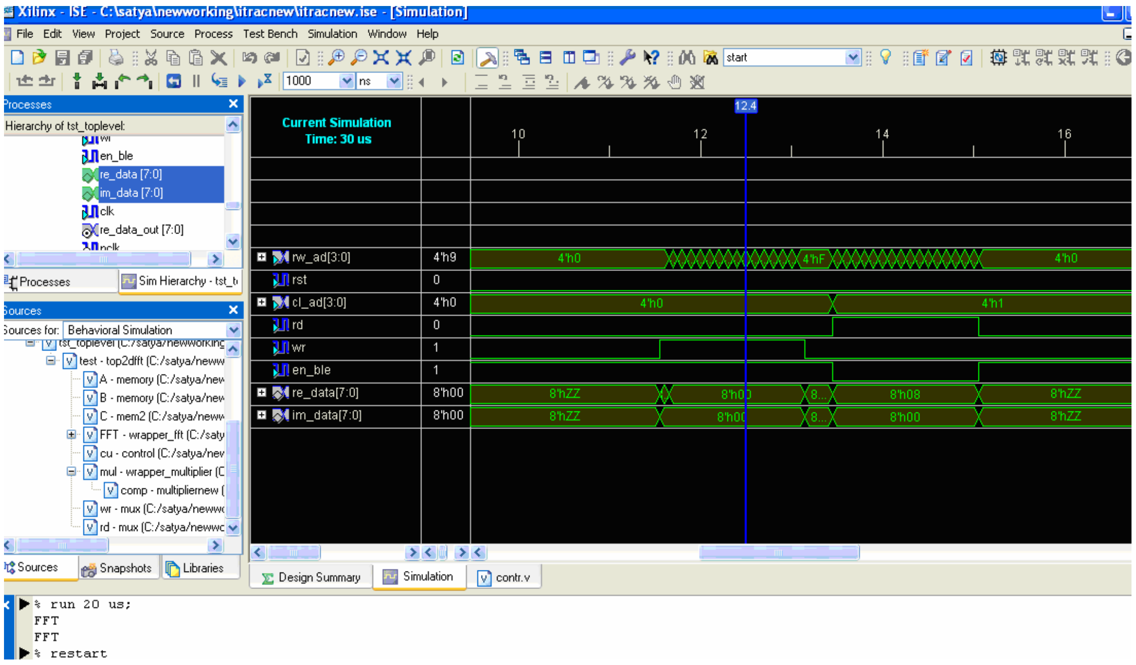


Fig 4.14

Input:
Clk: global clock signal
Rd: signal for reading from ram (active high).
Wr: signal for writing from ram (active high).
rst: active high reset.
en_ble: Is the signals which enable the ram (active low).
Rw_ad(3:0) : Row address.
Cl_ad(3:0) : column address.

Inout
Im_data(7:0) :imaginary data bus
Re_data(7:0) : real data bus.

Timing Diagram:



```
% run 20 us;
FFT
FFT
% restart
```

5. Simulation:

5.1. Simulation Setup: The simulation set up is a shown in the figure 4.15. The major two components are

- The design under test (Image matching Processor –DUT)
- Signal Generator: generates continuous clock signal and reset signal.

All the internal signals can be monitored on dynamic simulation. The out put from the DUT indicates the coordinate having max correlation.

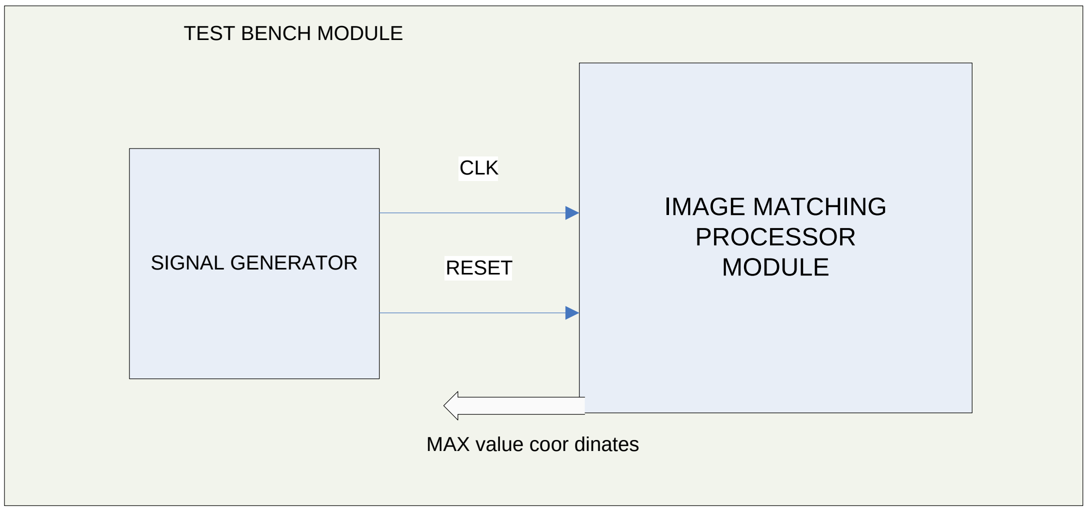


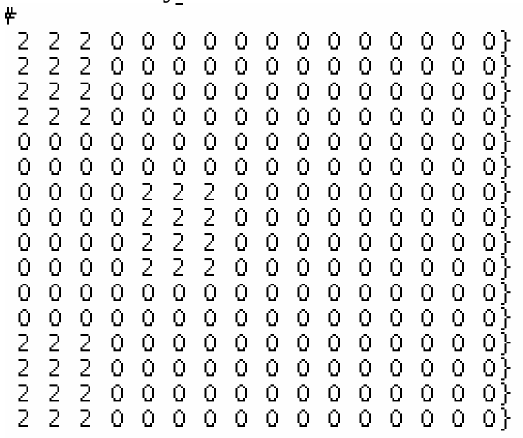
Fig 4.15

5.2 Results

The memory elements are printed dynamically by performing simulations.

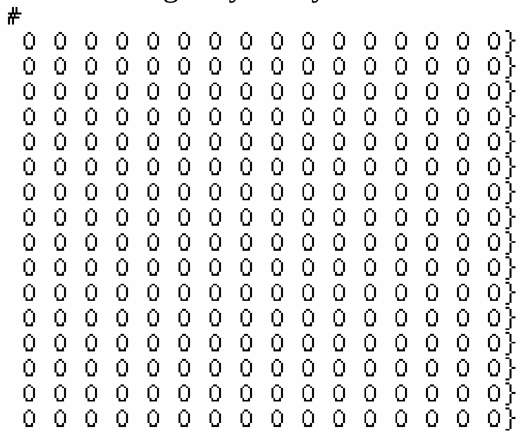
IMAGE 1:(RAM !)

Real Array



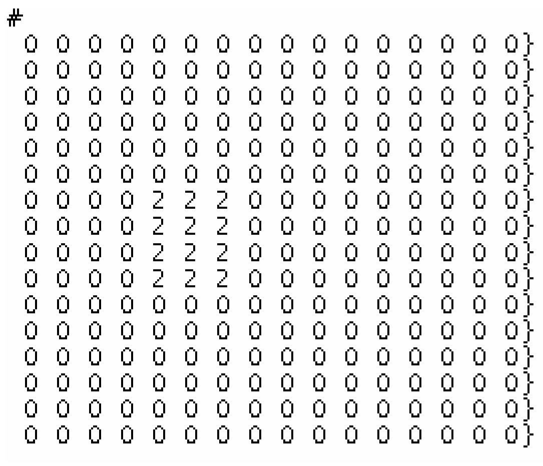
```
#
 2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0}
 2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0}
 2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0}
 2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  2  2  2  0  0  0  0  0  0  0  0  0}
 0  0  0  0  2  2  2  0  0  0  0  0  0  0  0  0}
 0  0  0  0  2  2  2  0  0  0  0  0  0  0  0  0}
 0  0  0  0  2  2  2  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0}
 2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0}
 2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0}
 2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0}
```

Imaginary Array

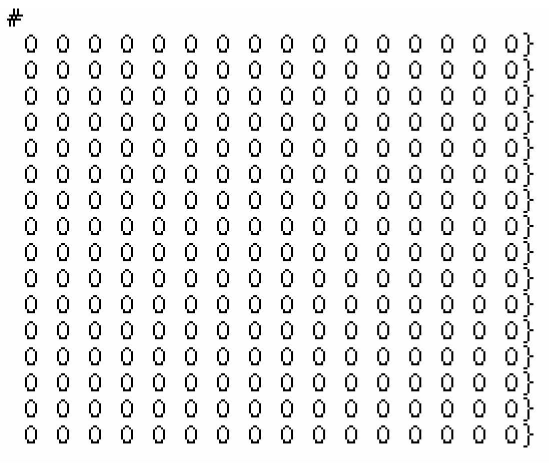


```
#
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
```

IMAGE 2: (RAM C)

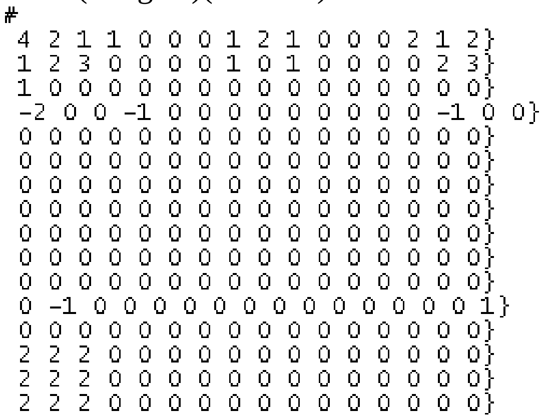


```
#
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  2  2  2  0  0  0  0  0  0  0  0  0}
 0  0  0  0  2  2  2  0  0  0  0  0  0  0  0  0}
 0  0  0  0  2  2  2  0  0  0  0  0  0  0  0  0}
 0  0  0  0  2  2  2  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
```

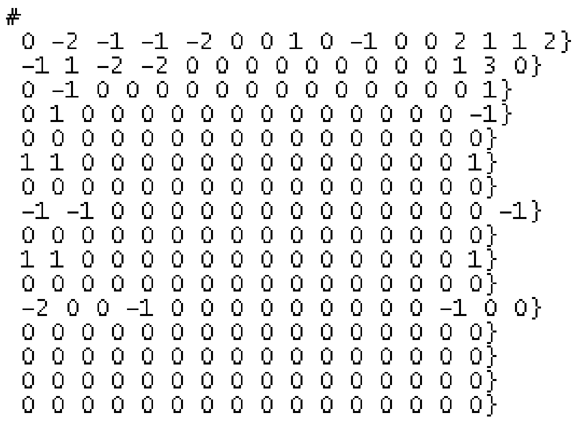


```
#
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
```

FFT2(Image 1)(RAM A)

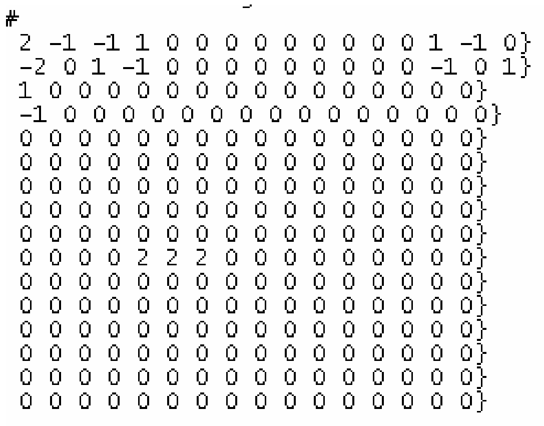


```
#
 4  2  1  1  0  0  0  1  2  1  0  0  0  2  1  2}
 1  2  3  0  0  0  0  1  0  1  0  0  0  0  2  3}
 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 -2  0  0  -1  0  0  0  0  0  0  0  0  0  -1  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  -1  0  0  0  0  0  0  0  0  0  0  0  0  0  1}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0}
 2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0}
 2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0}
```

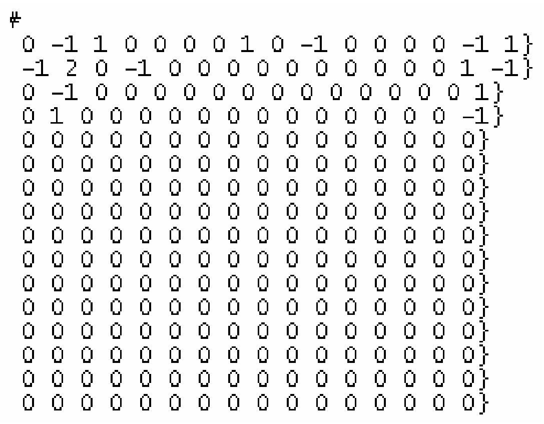


```
#
 0  -2  -1  -1  -2  0  0  1  0  -1  0  0  2  1  1  2}
 -1 1  -2  -2  0  0  0  0  0  0  0  0  0  1  3  0}
 0  -1  0  0  0  0  0  0  0  0  0  0  0  0  0  1}
 0 1  0  0  0  0  0  0  0  0  0  0  0  0  0  -1}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 1 1  0  0  0  0  0  0  0  0  0  0  0  0  0  1}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 -1  -1  0  0  0  0  0  0  0  0  0  0  0  0  0  -1}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 1 1  0  0  0  0  0  0  0  0  0  0  0  0  0  1}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 -2  0  0  -1  0  0  0  0  0  0  0  0  0  -1  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0}
```

FFT2(Image 2) (RAM C)

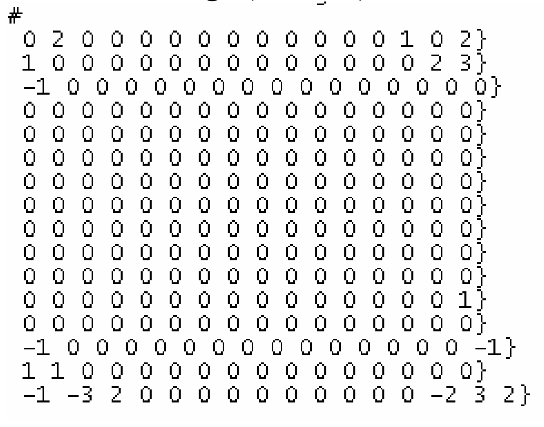


```
#
 2 -1 -1 1 0 0 0 0 0 0 0 0 0 1 -1 0}
 -2 0 1 -1 0 0 0 0 0 0 0 0 0 -1 0 1}
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 2 2 2 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
```

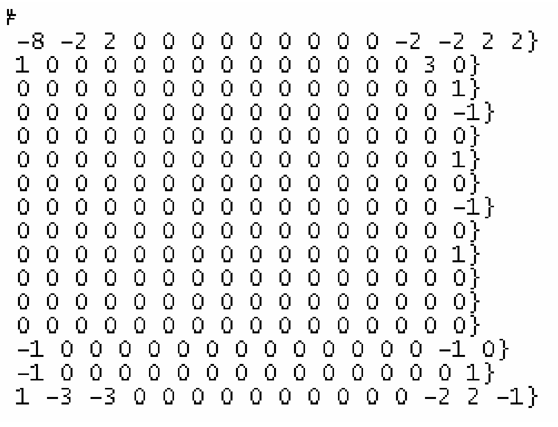


```
#
 0 -1 1 0 0 0 0 1 0 -1 0 0 0 0 -1 1}
 -1 2 0 -1 0 0 0 0 0 0 0 0 0 0 1 -1}
 0 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 1}
 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 -1}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
```

Convolved Image:(RAM A)

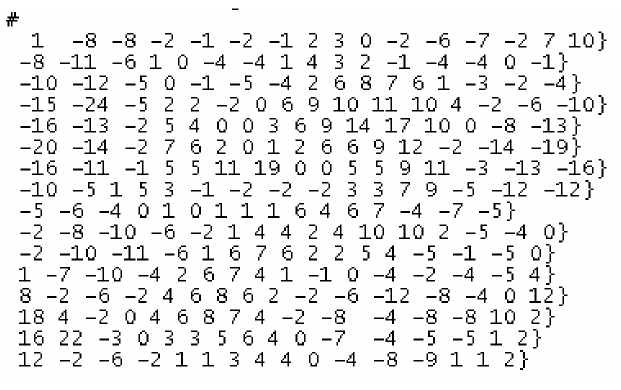


```
#
 0 2 0 0 0 0 0 0 0 0 0 0 0 1 0 2}
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 2 3}
 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1}
 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 -1 -3 2 0 0 0 0 0 0 0 0 0 0 -2 3 2}
```

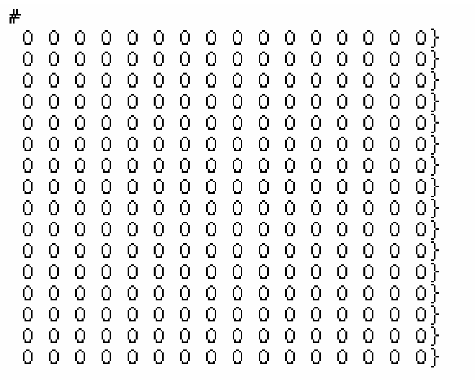


```
#
 -8 -2 2 0 0 0 0 0 0 0 0 0 -2 -2 2 2}
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0}
 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1}
 1 -3 -3 0 0 0 0 0 0 0 0 0 0 -2 2 -1}
```

IFFT2(Convolved Image) (RAM A



```
#
  1  -8 -8 -2 -1 -2 -1 2 3 0 -2 -6 -7 -2 7 10}
 -8 -11 -6 1 0 -4 -4 1 4 3 2 -1 -4 -4 0 -1}
 -10 -12 -5 0 -1 -5 -4 2 6 8 7 6 1 -3 -2 -4}
 -15 -24 -5 2 2 -2 0 6 9 10 11 10 4 -2 -6 -10}
 -16 -13 -2 5 4 0 0 3 6 9 14 17 10 0 -8 -13}
 -20 -14 -2 7 6 2 0 1 2 6 6 9 12 -2 -14 -19}
 -16 -11 -1 5 5 11 19 0 0 5 5 9 11 -3 -13 -16}
 -10 -5 1 5 3 -1 -2 -2 -2 3 3 7 9 -5 -12 -12}
 -5 -6 -4 0 1 0 1 1 1 6 4 6 7 -4 -7 -5}
 -2 -8 -10 -6 -2 1 4 4 2 4 10 10 2 -5 -4 0}
 -2 -10 -11 -6 1 6 7 6 2 2 5 4 -5 -1 -5 0}
 1 -7 -10 -4 2 6 7 4 1 -1 0 -4 -2 -4 -5 4}
 8 -2 -6 -2 4 6 8 6 2 -2 -6 -12 -8 -4 0 12}
 18 4 -2 0 4 6 8 7 4 -2 -8  -4 -8 -8 10 2}
 16 22 -3 0 3 3 5 6 4 0 -7  -4 -5 -5 1 2}
 12 -2 -6 -2 1 1 3 4 4 0 -4 -8 -9 1 1 2}
```



```
#
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
```

Result
Max value – 24
Coordinates (4,2)

The wave form for total simulation is shown in fig 4.16

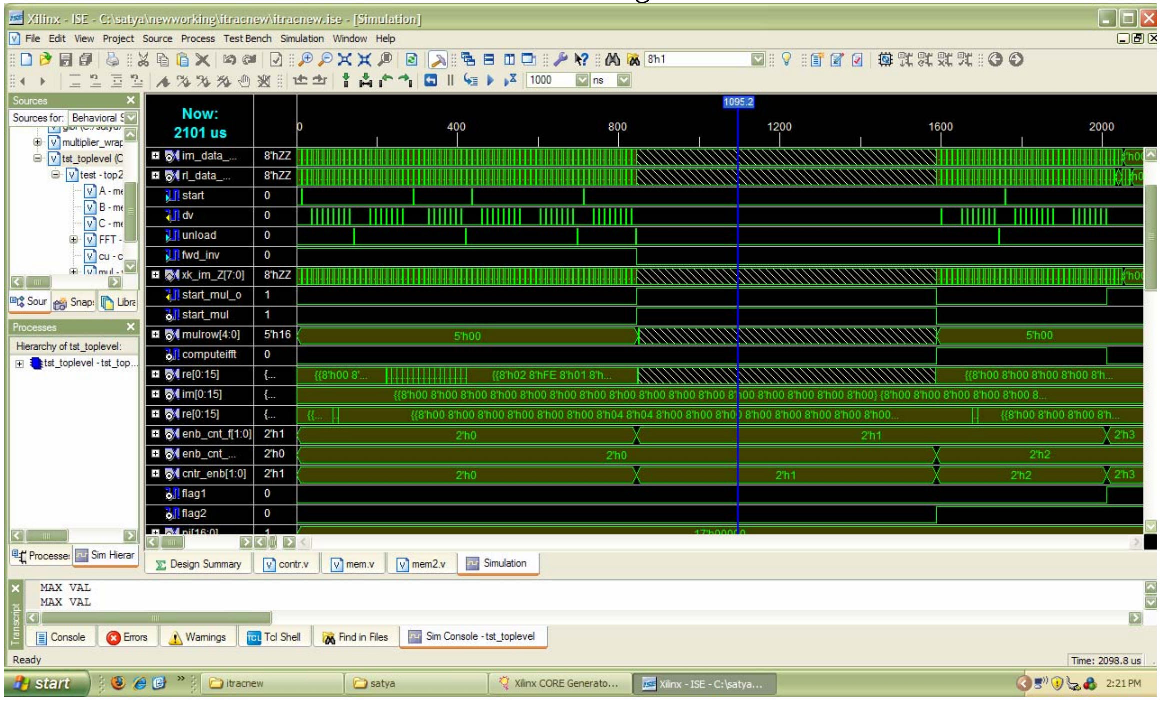


Fig 4.16

6.Synthesis Report:

```
Synthesizing Unit <top2dfft>.
    Related source file is "../../srcfim/topmodule.v".
.
Unit <top2dfft> synthesized.


=======================================================================
===
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                    : 1
 5-bit adder                                            : 1
# Counters                                              : 6
 3-bit up counter                                       : 1
 5-bit up counter                                       : 4
 6-bit up counter                                       : 1
# Registers                                             : 1583
 1-bit register                                         : 16
 2-bit register                                         : 1
 4-bit register                                         : 5
 6-bit register                                         : 3
 8-bit register                                         : 1558
# Latches                                               : 14
 1-bit latch                                            : 7
 4-bit latch                                            : 3
 6-bit latch                                            : 3
 8-bit latch                                            : 1
# Tristates                                             : 19
```

```
 1-bit tristate buffer                                 : 8
 2-bit tristate buffer                                 : 1
 8-bit tristate buffer                                 : 10
# Xors                                                 : 1
 1-bit xor2                                            : 1

======================================================================

=====================================================================
*                       Advanced HDL Synthesis
*
======================================================================

Loading device for application Rf_Device from file '4vlx15.nph' in
environment C:\Xilinx91i.
Reading core <fft_dat_8.ngc>.
Reading core <multipliernew.ngc>.
Loading core <fft_dat_8> for timing and area information for instance
<FFTk>.
Loading core <multipliernew> for timing and area information for
instance <comp>.

======================================================================
===
Advanced HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 1
 5-bit adder                                           : 1
# Counters                                             : 6
 3-bit up counter                                      : 1
 5-bit up counter                                      : 4
 6-bit up counter                                      : 1
# Registers                                            : 12513
 Flip-Flops                                            : 12513
# Latches                                              : 14
 1-bit latch                                           : 7
 4-bit latch                                           : 3
 6-bit latch                                           : 3
 8-bit latch                                           : 1
# Xors                                                 : 1
 1-bit xor2                                            : 1

======================================================================
===

======================================================================
===
*                         Low Level Synthesis
*
======================================================================
===
Optimizing unit <top2dfft> ...

Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block top2dfft, actual
ratio is 178.
```

```
Optimizing block <top2dfft> to meet ratio 100 (+ 5) of 6144 slices:
WARNING:Xst:2254 - Area constraint could not be met for block
<top2dfft>, final ratio is 177.
FlipFlop cu/enb_cnt_m_2 has been replicated 1 time(s)

Final Macro Processing...

Processing Unit <top2dfft>:
      Found 3-bit shift register for signal <FFT/xn_im_2_0>.
Unit <top2dfft> processed.

======================================================================
Final Register Report

Macro Statistics
# Registers                                                : 12489
 Flip-Flops                                                : 12489
# Shift Registers                                          : 16
 3-bit shift register                                      : 16


======================================================================
*                           Partition Report                         *
======================================================================

Partition Implementation Status
-------------------------------

  No Partitions were found in this design.

-------------------------------

======================================================================
*                             Final Report                           *
======================================================================

Final Results
RTL Top Level Output File Name     : top2dfft.ngr
Top Level Output File Name         : top2dfft
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 2

Cell Usage :
# BELS                             : 15080
#      BUF                         : 1
#      GND                         : 6
#      INV                         : 36
#      LUT2                        : 607
#      LUT2_D                      : 2
#      LUT3                        : 626
#      LUT3_D                      : 2
#      LUT3_L                      : 1
#      LUT4                        : 7146
#     LUT4_D                    : 95
#     LUT4_L                    : 8
#     MULT_AND                  : 5
#     MUXCY                     : 6278
#     MUXF5                     : 111
#     MUXF6                     : 4
#     MUXF7                     : 2
#     VCC                       : 5
#     XORCY                     : 145
# FlipFlops/Latches             : 13248
#     FD                        : 46
#     FD_1                      : 2
#     FDC                       : 33
#     FDCE                      : 12370
#     FDE                       : 631
#     FDP                       : 5
#     FDPE                      : 69
#     FDR                       : 3
#     FDRE                      : 45
#     FDRS                      : 5
#     FDS                       : 1
#     FDSE                      : 5
#     LD                        : 27
#     LDE_1                     : 6
# RAMS                          : 3
#     RAMB16                    : 3
# Shift Registers               : 123
#     SRL16                     : 16
#     SRL16E                    : 106
#     SRLC16E                   : 1
# Clock Buffers                 : 2
#     BUFGP                     : 2
# DSPs                          : 6
#     DSP48                     : 6
===================================================================

Device utilization summary:
---------------------------

Selected Device : 4vlx15sf363-12

 Number of Slices:                    10929  out of   6144   177% (*)
 Number of Slice Flip Flops:          13248  out of  12288   107% (*)
 Number of 4 input LUTs:               8646  out of  12288    70%
 Number used as logic:                8523
 Number used as Shift registers:  123
 Number of IOs:                        2
 Number of bonded IOBs:                2  out of    240     0%
 Number of FIFO16/RAMB16s:             3  out of     10    30%
  Number of GCLKs:                     2  out of     32     6%
 Number of DSP48s:                     6  out of     32    18%


WARNING:Xst:1336 -  (*) More than 100% of Device resources are used

Timing Report:
```

```
Clock Information:
------------------
-----------------------------------------------------------+---------
----------------------+-------+
Clock Signal                                               | Clock
buffer(FF name)          | Load  |
-----------------------------------------------------------+---------
----------------------+-------+
clk                                                        | BUFGP
| 13319 |
cu/inc                                                     |
NONE(cu/mulrow_2)             | 9     |
cu/col_ch_inv(cu/col_ch_inv1:O)                            |
NONE(*)(cu/count_1)           | 10    |
cu/inc1                                                    |
NONE(cu/co_rmax_3)            | 9     |
rst                                                        | BUFGP
| 6     |
cu/row_add_to_mem_f_not0001(cu/row_add_to_mem_f_not000158:O)|
NONE(*)(cu/row_add_to_mem_f_0)| 8     |
cu/enb_cnt_f_not0001(cu/enb_cnt_f_not0001:O)               |
NONE(*)(cu/enb_cnt_f_3)       | 2     |
cu/enable_f_not0001(cu/enable_f_not0001175_f5:O)           |
NONE(*)(cu/enable_f_5)        | 3     |
cu/NS_I_not0001(cu/NS_I_not0001115:O)                      |
NONE(*)(cu/NS_I_1)            | 6     |
cu/fwd_inv_we_not0001(cu/fwd_inv_we_not0001:O)             |
NONE(*)(cu/fwd_inv_we)        | 3     |
cu/enb_Ce_or0000(cu/enb_Ce_or00001:O)                      |
NONE(*)(cu/enb_Ce)            | 1     |
cu/start_not0001(cu/start_not0001_f5:O)                    |
NONE(*)(cu/start)             | 1     |
cu/unload_not0001(cu/unload_not000158:O)                   |
NONE(*)(cu/unload)            | 1     |
cu/scale_sch_we_not0001(cu/scale_sch_we_not000130:O)       |
NONE(*)(cu/scale_sch_we)      | 1     |
cu/flag1_not0001(cu/flag1_not00011:O)                      |
NONE(*)(cu/flag1)             | 1     |
-----------------------------------------------------------+---------
----------------------+-------+
(*) These 11 clock signal(s) are generated by combinatorial logic,
and XST is not able to identify which are the primary clock signals.
Please use the CLOCK_SIGNAL constraint to specify the clock signal(s)
generated by combinatorial logic.
INFO:Xst:2169 - HDL ADVISOR - Some clock signals were not
automatically buffered by XST with BUFG/BUFR resources. Please use the
buffer_type constraint in order to insert these buffers to the clock
signals to help prevent skew problems.

Asynchronous Control Signals Information:
----------------------------------------
--------------------------------------------+------------------------
--+-------+
Control Signal                              | Buffer(FF name)
| Load  |
--------------------------------------------+------------------------
--+-------+
rst                                         | BUFGP
| 12334 |
FFT/FFTk/sig00000001(FFT/FFTk/blk00000002:G)|
NONE(FFT/FFTk/blk000004fb)| 136   |
cu/set(cu/set:Q)                            | NONE(mul/CS_0)
| 7     |
cu/count_or0000(cu/count_or00001:O)         | NONE(cu/count_1)
| 6     |
--------------------------------------------+-------------------------
--+-------+

Timing Summary:
---------------
Speed Grade: -12

   Minimum period: 13.917ns (Maximum Frequency: 71.857MHz)
   Minimum input arrival time before clock: 6.555ns
   Maximum output required time after clock: No path found
   Maximum combinational path delay: No path found

======================================================================
===

Process "Synthesize" completed successfully
```

7. Observations:

Initially the device is targeted for Spartan 3 A, but from the synthesis report the logical (Resource) utilization for the device is 533% which implies the design could not fit into the FPGA. So the target is changed to Vertex Series and with optimizations the design is re synthesized, but could minimize to 177% of logic utilization.

When the (EDIF)netlist file generated from the synthesis tool is fed to Place and Route Tool since the design is bigger than it is supposed to fit in the chip it couldn't identify the partitions for placement and routing. So the design could no be routed and downloaded into FPGA.

8. Scope for Improvements.

- The Logic can be further optimized (state machine minimization) for better performance.
- The Ram modules can be implemented out side of FPGA so that design fits in FPGA.
- Due approximations and Scaling down the value the identification of the exact location where match is present is deviating this can be rectified either by increasing the data width or by modifying the logic.
- More parallelism can be introduced to improve the through put.

Bibliography:

Xilinx Documents:
http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp

Verilog HDL Synthesis A Practical Premier by J.Bhaskar.

Signals and Systems by BP Lathi.

Digital Signal processing by JN chitode

Matlab documents