Architecture and Digital Design of a flexible Correlation filter Processor

Author

Abdullah Aman Khan

2011-NUST-MS PhD-ComE-02

Supervisor

Dr. Saad Rehman

DEPARTMENT OF COMPUTER ENGINEERING

COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

ISLAMABAD

MAY, 2014

Architecture and Digital Design of a flexible Correlation filter Processor

Author

Abdullah Aman Khan

2011-NUST-MS PhD-ComE-02

A thesis submitted in partial fulfillment of the requirements for the degree of

MS Computer Engineering

Thesis Supervisor:

Dr. Saad Rehman

Thesis Supervisor's Signature: ______________________________________

DEPARTMENT OF COMPUTER ENGINEERING

COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY, ISLAMABAD

MAY, 2014

Declaration

I certify that this research work titled "Architecture and Digital Design of a flexible Correlation filter Processor" is my own work. The work has not been presented elsewhere for assessment. The material that has been used from other sources it has been properly acknowledged / referred.

Signature of Student

Abdullah Aman Khan

2011-NUST-MS PhD-ComE-02

Language Correctness Certificate

This thesis has been read by an English expert and is free of typing, syntax, semantic, grammatical and spelling mistakes. Thesis is also according to the format given by the university.

Signature of Student

Abdullah Aman Khan

2011-NUST-MS PhD-ComE-02

Signature of Supervisor

Dr. Saad Rehman

Copyright Statement

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Dr. Saad Rahman for the continuous support of my Masters Study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Masters study.

Besides my advisor, I would like to thank the rest of my thesis committee Brig. Dr. Shoab A Khan, Dr. Umer Munir, Dr. Usman Akram and Mr. Sajid Gul for their encouragement, insightful comments, and hard questions.

Last but not the least; I would like to thank my family supporting me spiritually throughout my life.

Abstract

Dedicated design for pattern recognition techniques can provide processing at higher speed, .Many Real time digital designs provide single functionality, but where speed, scalability and flexibility is required more extensive research is demanded. These special purpose and Application specific designs can provide real time procession for many applications. In this thesis the challenges, problems and design for a correlation patteren recognition processor is presented, The design is based on fixed point representation of binry numbers, further the fixed point representation is used to represent complex numbers as correlation filters are designed in frequency domain. The presented design is an educational purposes design, which provides mechanism to handle matrices on chip and also is capable of performing simple matrices operation like arithmetic and matrix handling. The matrices are represented with simple identifiers and using application specific micro instruction provided in the design assist in processing of many complex operations which are useful for solving bigger problems when summed up to gather in a meaning full manner. The design was successfully implemented and tested using VHDL language using Xilinx tool. The objective of this work was to get an area & time efficient architecture that can be used as a standalone processor with built in all resources necessary for an embedded pattern recognition application.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

Pattern recognition has been another challenging segment for man-made marvels, with the passing flow of time, requirements for Luxury and Necessity are demanding more extensive research in the field of pattern Recognition. From above a century mathematicians and scientists have devoted their whole lives to create and improve algorithms that can perform like Human Recognition System or even better. In a daily human life human brain carries out a lot of pattern recognition without even noticing [1], even it is an irregular pattern human recognition system is even then capable of recognizing. The human brain can also handle new situations even if it’s an unseen or a new case. Human brain is still capable of categorizing the object based on previously learned similarity and some logic that might be based on experience. We can even categories voices and recognize that what sort of voice is it i.e. the semantics and structure, Human brain is also capable of recognizing different persons be their face features and body language too, But there are some drawbacks also with the human recognition like humans recognition is slow, The other is that scaling is not easy for everyone and in some special cases human recognition is incapable of classifying. Likewise the case of finger print recognition, not all humans can perform finger print matching; only a highly trained person can perform this task but the efficiency and accuracy cannot be compared as of a machine. The Human brain lacks memory, in case of finding a single sample of fingerprint of a person in a large data, it might take an expert from days to even months to match a single print but comparatively a machine can do this job in the matter of seconds to minuets. The domain of pattern classification is involved in various filed of automations and it is being very extensively used in intelligent machines, some common application of Pattern classification are Voice recognition, fingerprint recognition, face recognition, Optical Character recognition (OCR) Object detection/recognition, Bio Medical Applications etc.

Almost everyone is familiar with the long extending benefits and the services of machines to mankind, the benefits that Machines can provide include speed, accuracy, cost saving and un-tiring service .Many automatic machines uses the concept of pattern recognition. For example a quality check at a high speed conveyor belt on a production line will require a trained eye to monitor the quality efficiently. A machine properly designed can work more accurately and at very high speeds thus cutting down the cost and the bottle neck that human supervision might create comparatively to the machine. With the span of time, the increasing demand for such machines has proved their worth.

The processors that are deliberately designed to perform specific tasks are dedicated processors, or in other words are built and designed for specific/dedicated purpose. These processors can perform almost all the basic operations like loops and arithmetic and logical operations. A processor that can perform logical operations like and, or and not and can loop can handle any kind of software related problem.

Security has been man’s nightmare from the beginning of time, whether it is an organization or an individual being on the safe side matters a lot .There has been a tremendous increase in the trend on relying on machines to provide self and automated security using vision or sensor systems. A simple camera attached to a machine capable of running a smart algorithm can provide a very effective and low cost security watch and the best part is that unlike human machine never sleeps and it cannot be bribed either, thus can provide a much safer environment. Whether it’s the matter of home land security or security of an individual these machines can provide greater services to mankind. In the field of defense modern radar systems are also providing a great deal of protection against enemy trespassing and attacks, there are several issues with old radar systems, one big problem was that they only provided data visually to the human eye, than the fate of the entity to be saved fully relied on the human and brain on the watch. Comparing to human, machines can work more efficiently and the efficiency dose not drop throughout the time as tiredness and boredom can severely effect humans work performance.

Correlation Pattern recognition has contributed a lot in the field of pattern recognition [1-3]. Usually digital machines are dealing with digital signal, Image and Audio can also be referred as a digital signal and the techniques of pattern recognition can be applied to these digital signals. The main concern is to properly classify an object or an entity so that it can be further worked upon. The following diagram represents some features of a fingerprint scan, features help in logically classify the object.

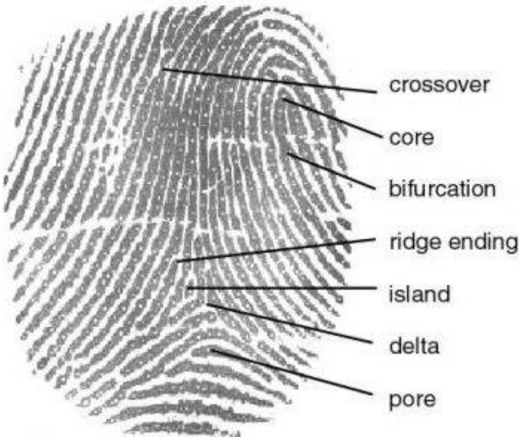


Figure 1-1: A fingerprint diagram with some salient features [1]

In pattern recognition usually an input is provided to a system, usually a system should perform some preprocessing to adjust the image so that the system can perform operations on the image in a more optimal way like removing the noise the input image that might have been added at the image acquisition phase or allying some algorithm that can enhance classification process. Afterwards the system extracts suitable number of required features and the works on the features of the provided pre-processed input, based on these features a decision is carried out whether to which class the input may belong to. The following diagram explains how a trivial Pattern classification mechanism works.

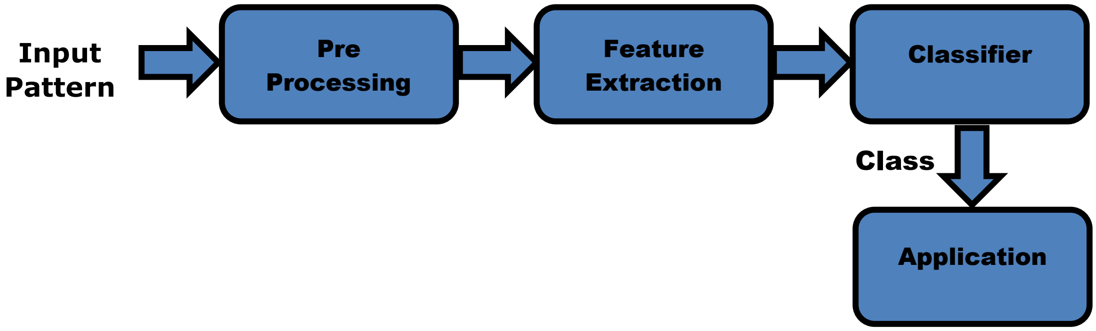


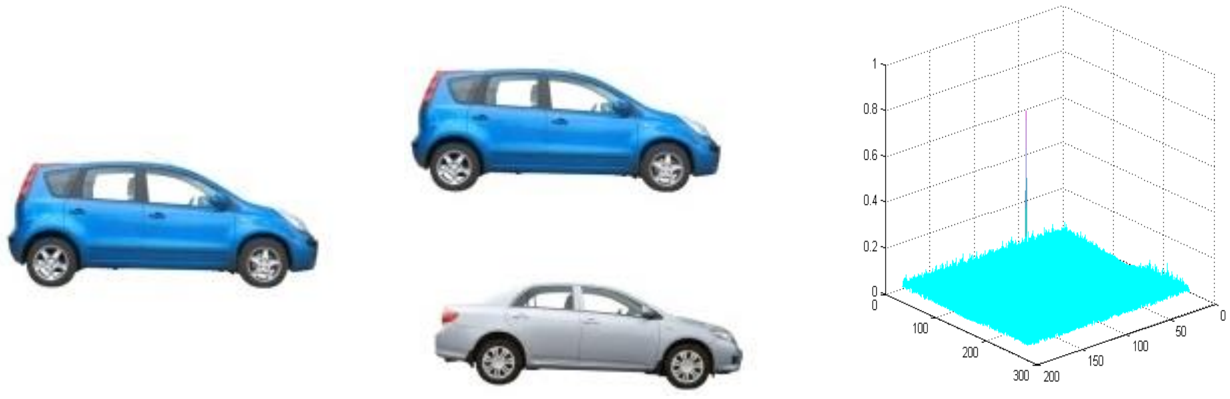
Figure 1-2: Typical Flow chart for typical Pattern Recognition system.

1.1 Typical Methods:

Following methods are generally used for recognizing a pattern

- Statistical
- Machine learning
- Artificial neural
- Correlation filters

The above mentioned approaches are mostly carried out in image spatial domain, where more significant advantage can be obtained in Frequency domain. All the input image might not be the same always there might be some sort of illumination changes in the input image, besides illumination there is a chance that multiple repetition of the pattern might occur in the input image, this situation should also be cratered .The desired pattern can exist anywhere in the scene. Another big challenge is that the desired pattern in the image might scale, rotated or translated. There must be an algorithm that that is robust of all the above mentioned problems .Luckily correlation filter can provide efficient and robust pattern classification. Straight forward correlation of test input can be carried out on a reference sample that will result in higher peaks in a correlation plane where the reference image exists.



(a) (b) (c)

Figure 1-3: (a) input pattern (b) Reference image (c) Correlation Plane (correlation results of (a) to (b))

The figure shows the correlation results of a test image over a reference image, in the resulting correlation plane the peak indicates the location of the test object in the reference image, sharper the peak the more is the probability of the test object to be present in the reference image. If the image has more than one existence of the test object in the reference image, the peaks will be replicated Straight forward Correlation technique lies under the field of statistics but is quite inefficient, Correlation filters designed in frequency domain have greater advantage of speed thus helping in building real time applications for pattern recognition .

Typically correlation filters are designed in frequency domain, there are many filter that are designed using FFT (Frequency Domain), mostly this technique is employed where Real time processing is required .Like the radar systems are critical, the modern aircrafts have almost broken the sound barrier and can travel up to or more than speed of sound, it is very critical to develop an algorithm that can process the sample gathered by the radar system in real time. If the processing is slow there are very high chances that the aircraft will reach its target unharmed and it might be too late to apply preventive measures.

Correlation filter are developed in frequency domain using the reference inputs, which are basically referred as the training sample or training data. A correlation filter is developed one filter per class, the training data contain the variations of the pattern o be detected, the variations include some rotation of the pattern. There also might be some scaling difference between the training samples. One filter developed results in a correlation plane when it is multiplied (correlated) to a Test input. If the object is present in the reference image it will produce peaks. That refers to the location of the pattern on the reference image. Furthermore algorithms and methods can be used to detect the pattern in the reference image using the obtained peak.

Many correlation filters have been developed with time, with grate performance. These filters include the *synthetic discriminant filter* (SDF) which was among the earlier developed correlation filter. Furthermore known filters like *Maximum Average Correlation Height* (MACH)[4-6], *Extended Maximum Average Correlation Height*

(EMACH)[7, 8], *Eigen Extended Maximum Average Correlation Height* (EEMACH)[9], *Maximum Average correlation energy* (MACE)[10], *minimum noise and average correlation plane energy* (MINACE)[11], *Minimum variance Synthetic Discriminant Function* (MVSDF) filter [12] etc. These filters are more robust and can perform the recognition at very high speeds.

1.2 Motivation

Over the years small form factor and power efficient flexible and programmable devices have captured the marked trends, These devices can work at very higher speed .custom built digital designs are more commonly implemented on these flexible hardware. For a mass production of a digital design semiconductor materials based chips are employed, the whole system can be fabricated on these chips these. To manufacture these semiconductor material dyes have to be manufactured first the manufacturing of these dyes itself is an expensive and pain giving procedure, but ultimately this method in mass production cuts down the cost of the digital design.

On the other hand if there is only one custom design that requires fewer amounts of copies to be produced, the simple flexible devices like FPGA, ASIC, CPLD can be employed. Although a single piece of custom design hardware will cost more than a single piece Mass produced hardware, but on the other hand if it is desired to manufacture a single piece of hardware using the mass production strategy that will increase the manufacturing cost at very high level. That is the main reason that the trend toward these flexible hardware has been increasing over the time. These electronic devices allow Design engineers and hobbyist to produce cost effective and very efficient and intelligent electronic devices.

Design Engineers around the world have been employing FPGA, CPLD etc. to implement and test their designs. In the last decade with the availability of FPGA with more memory has motivated the design engineers to also implement the image processing techniques on these devices .These devices can also be programed using *Hardware Descriptive language* (HDL).Some of the manufacture of these devices also provide pre built libraries that are tested and optimal speed wise, some open source languages to describe the design are have also been introduced into the digital design world. Verilog and VDHL are the most common languages used to program this flexible Hardware. The reasons why Design engineers use these technologies involve their small size, Low power consumption, flexibility and that they are capable of performing at higher speeds compared to a general Computer. Now a days top leading electronics devices manufactures are manufacturing the state of the art devices these days. A single digital camera and a Cellular phone uses up a very powerful processor capable of performing different tasks. The digital cameras these days are using small but simple image processing techniques to attract the customer. Smile and face detection are some examples, these camera are also capable of Enhancing the images using different enhancement techniques like filtering Histogram equalization, these techniques when applied pleases the human eye. The increasing trend in luxury requires more effort to be put into the field of embedded and dedicated systems.

Common correlation filters [1] can be implemented on general computing, but as mentioned in the above paragraphs that in some requirements small form factor devices, less power consumption and speed is a crucial matter. Face and smile detection can also be carried out on a general computer but it not always possible to carry a huge sized general computer. Besides this the correlation filters requires huge amount of computations .Besides this the general computing environment is usually running an operating system, the operating system is handling a lot of demons and application running in the background these applications require a lot of processing power, usually in general computer the processing power is shared among the tasks running on it where n the other hand dedicated hard ware does not require to share resources to other entities. FFT [13] its self is a compute intensive procedure and might cover much hardware cost itself. The discrete time Fourier transform is slower than FFT and also requires large amount of memory even on general computing. The main goal is to provide a very flexible architecture that is capable of performing all the tasks required by a correlation filter design, thus providing a Correlation Pattern recognition module that is flexible and provides great performance.

There are other Hardware Platforms available in the market at low costs that can provide a general computing environment with lesser power consumption, smaller size and the flexibility of a general computer .A simple microprocessor and some DSP Processors

[14] can also carry out the tasks required to perform a correlation Pattern recognition application. Besides these devices Raspberry Pi [15] is also providing a small sized ARM [16] based Credit card size Computer with necessary Peripherals. Some other companies are also manufacturing Flexible processors like Cortex, ARM etc.

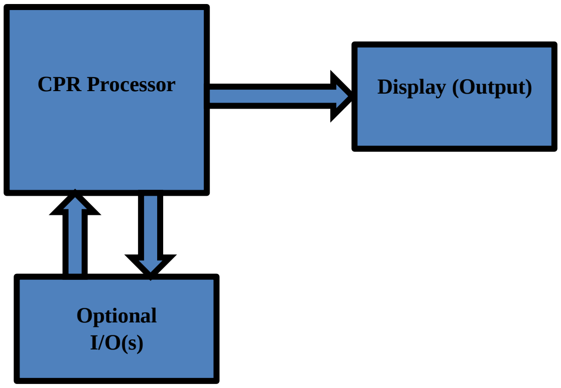


Figure 1-4: A standalone Model for CPR

1.3 Challenges

Designing an architecture that is capable of performing correlation pattern recognition requires to perform numerous small tasks that are entirely different in nature .There were several possibilities to realize this type of Digital Design. Some of the challenges faced in this research project are enlisted with some detail.

1.3.1 Flexibility

Most Digital design engineers while designing a dedicated system only focus on certain amount of specifications, for example a system is designed to work on an image of (128 X 128) pixels only .What if the end user has to change the size of the input image to (256 X 256) pixels. This problem has always been a question mark on the digital design industry. A digital design usually has to be replaced if no room is kept to tackle this size of problems. Replacing an old design has then its own requirements that can cost a great deal of time and money.

1.3.2 Cost Effectiveness

Another Big challenge in this case is to provide best performance at low cost. Usually Cost is directly proportional to performance. As the designers try to increase the performance of the machine the cost raises up to i.e. it consumes more hardware [17].

Besides performance, while increasing flexibility of a certain design also increases the hardware costs. It's nothing but a tradeoff which cannot be ignored. In case of EMACH Filter, this filter itself requires a lot of different operations like, Computing Fourier transform, Computing Eigen Values and Eigen vectors, Basic Matrix Operations like Matrix addition, subtraction, Multiplication. These operations themselves are a very big challenge to design individually. Such an architecture that has to perform all of these operations all together while saving cost and providing on chip flexibility is the biggest challenge.

1.3.3 Common Requirements

Many of the operations like finding the Fourier transform, computing the Eigen vector and values requires many operations like addition, subtraction Multiplication etc. If each module is merged together in a logical way the cost of the hardware will raise too much .some of the operations that a correlation filter design may require include .Frequency Domain Conversion (Computing FFT),Inverse Fourier Transform (Computing IFFT),Finding Eigen Values and Vectors, Finding Maximum and Minimum values, Implementation of some Numerical Methods, Maintain orientation of Matrix During operations, Matrix Addition, Matrix Subtraction, Matrix Multiplication (Scalar, Dot, Matrix),Matrix Division (scalar),Representing Complex Numbers scalar/matrix, Representation of conjugates of matrix, Calculating power of a matrix, Calculating Transpose of a matrix, Reshaping of a matrix , Provide a use of fractional Values, Provide Flexibility, Perform Logical Operations and Hazards Identification and removal. All the required parameters are quite challenging.

The above mentioned are the sub tasks that are required by the steps involved in the filter design, and the correlation. Each thing its self is a very big challenge to be implemented on hardware. One other Challenge was to provide these functionalities all to gather while cutting down the hardware size.

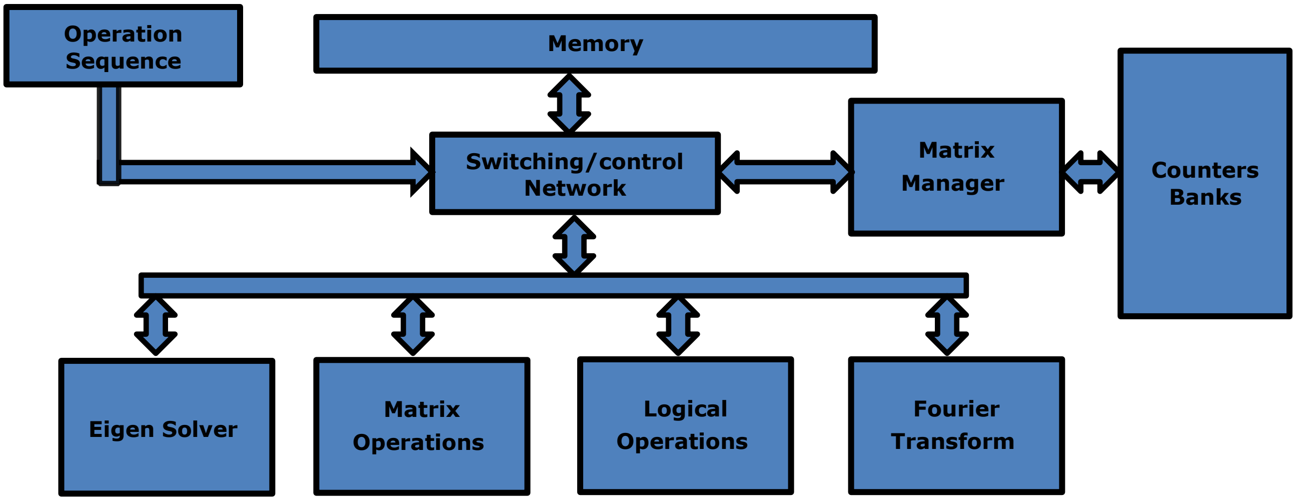


Figure 1-5: Diagram representing the requirements to perform CPR

The figure above shows the required operations to perform a correlation based pattern recognition. This is not the exact estimate but a near estimate that a design would require. This approach involves implementation of each module separately and then integrating the modules via an intelligent switching/Control network, this is one possibility, but to reside all the functional modules on a single chip will rise up the cost to a very large extent. Here one can clearly see the big challenge in the realization of this CPR generic Architecture. Thus it is intended to find common operations used by all the functional units (Subset of Operations) and Produce a design that is more cost effective.

1.4 Previous Works

Many researchers have contributed to the hardware design for image processing. Previously presented designs were based of different flexible platforms. These platform includes some well know hardware which boosts up image processing speeds. ASIC (Application Specific Integrated Chips), DSP (Digital Signal Processors) and some reconfigurable devices like FPGA (Field programmable Gate Arrays).Many image processing algorithms like edge detection using Sobel’s, Prewitt’s and canny were implemented on specific hardware implementation to FPGA manufacturers like Altera, Xilinx etc.

G.S. Richard [29] presented the idea of generating a parameterized program for convolution filters implemented on FPGA. Filter was constructed using a set of multipliers and adders generated from a canonical serial-parallel multiplier stage. Atmel proposed a 3 by 3 run-time reconfigurable convolution filter on FPGA. F.G. Lorca [30] presented an implementation of filters for 1D and 2D reducing memory usage and computational costs by half in software side and hardware. Nelson [31] presented and implementation of rand order filter, Morphological operations on Xilinx Virtix and Altera Flex 10K FPGA. Shinichi [32] realized the different image processing techniques for computation of image gravity center along with orientation detection using Hough's transform and radial projection. Fahad [33] proposed a high performance pipelined edge detection architecture for real time processing of images. Baran [34] implemented edge detection algorithm coded in impulse, further synthesizing the code for Altera Nios.

1.5 Orientation of Thesis

The reaming thesis is organized as described below. Chapter 2 contains a brief overview of the literature regarding to the processor, Chapter 3 explains the design and working of the individual components that builds up the processor. Chapter 4 explains the instruction set architecture along ith their use, Chapter 5 shows and explains the data path of different type of instructions, chapter 6 shows simulations results compared with actual results calculated by Matlab.

Chapter 2 Literature Review

The design of a digital architecture has a mandatory prerequisite, this is to study the problem thoroughly .The sub-problems of the main problem should be known very precisely to realize the digital version. The digital standalone architecture design of a CPR Processor requires to identify the sub operations and their detailed study.in this chapter an overview of the theoretical study of sub operations are presented.

2.1 Correlation Filters

Correlating filter has a significant edge in Pattern recognition due to their shift invariant and shift tolerant nature. Design process of a correlation filter involves the Fourier transform form of that particular image training samples. Fast Fourier Transform is usually employed for conversion from spatial domain to frequency domain. Figure 2-1 represents the common flow of a correlation filter. Some of the correlation filters mathematical back ground is explained in the next section.

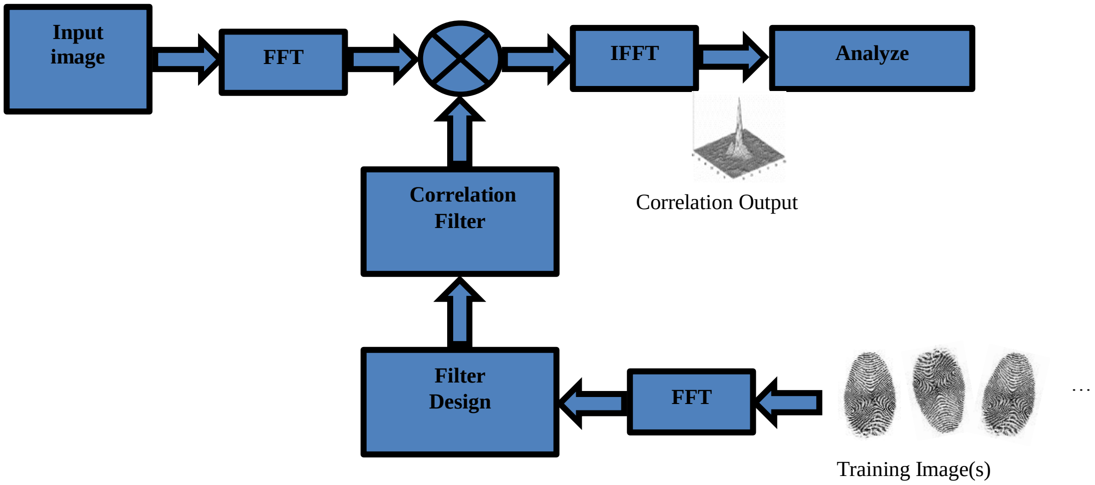


Figure 2-1: A Correlation Filter application flow

2.1.1 MACH Filter

MACH Filter is among recently developed filters. The design of this filter maximizes the peak energy using the mean of the training sample of provided image(s).in this section a brief overview of this filter design is presented the detailed version can be found at [5] .Like other correlation filters MACH Filter is also completely designed in frequency domain. The first step in the development of this filter is to compute the Fourier transform of a 2 Dimensional Image (2D Matrix) is computed. The size of one training image is assumed as d x d pixels. For computational ease the results are then converted to column vector, by scanning from left to right. After this the scanning is done from top to bottom sequence. The resulting 2D filter $\boldsymbol{h}$ can also be expressed in the same way. The i^{th} training image's correlation output can be represented by equation (2.1.1)

$$\mathrm{c}\,(0,0)\ = h^{+}\,\mathrm{x} \qquad (2.1)$$

Peak intensity of average training image can be expressed by the expression in 2.2, also referred as *Average Correlation Height* (ACH) [5]. The notation **m** represents the Average of all the *Fourier Transforms* (FTs) of all the training images, where the total number of images is denoted by **N.**

$$|\bar{c}(0{,}0)|^2 = \left|\frac{1}{N}\sum_{i=1}^{N} h^{+}x_i\right|^2 = |h^{+}m|^2 \qquad (2.2)$$

A tolerance characterizing measure metric Average Similarity Measure *(ASM)* is introduced which basically shows the dissimilarity between correlation output, training images and the average of the training images.ASM can be expressed as follows

$$ASM = \frac{1}{N}\sum_{i=1}^{N}\left|g_i - f_{opt}\right|^2 \qquad (2.3)$$

$$Where\ g_i = X_i h^{*}\ and\ f_{opt} = M h^{*}$$

X_i Represents a diagonal Matrix, the diagonal entries contains the elements of the FT vector along the diagonal. Matrix $\boldsymbol{M}$ is also a diagonal matrix that has the elements of $\boldsymbol{m}$ along the diagonal. The size of these tow diagonal matrix will be $\boldsymbol{d^2}$ **x** $\boldsymbol{d^2}$**.**Where $\boldsymbol{g_i}$ represents the correlation output of $\boldsymbol{i^{th}}$ training image. The term $\boldsymbol{f_{opt}}$ refers to optimal reference correlation output resulting in minimum mean square error this is usually found by measuring the gradient of ASM with respect to $\boldsymbol{f_{opt}}$ .Afterwards the putting the expression equal to zero and solving the expression for $\boldsymbol{f_{opt}}$.

By substituting the values of $\boldsymbol{f_{opt}}$ and $\boldsymbol{g_i}$ in equation (2.3) will yield the following expression.

$$ASM = h^{+} S_x h \qquad (2.4)$$

$$where\ S_x = \frac{1}{N} \sum_{i=1}^{N} (X_i - M)^{+}\, (X_i - M) \qquad (2.5)$$

Up till here S_x is a diagonal matrix with size $\boldsymbol{d^2}$ **x** $\boldsymbol{d^2}$.The small *x* in the subscript of the equation (2.4) indicates the dependence on the training image of a certain class *x*. By using equation (2.4) and (2.2) will result in equation (2.6) shown below.

$$J(h) = \frac{|\bar{c}(0,0)|^2}{h^{+} I h + h^{+} S_x h} = \frac{h^{+} m m^{+} h}{h^{+} (I + S_{x)h}} \qquad (2.6)$$

Noise also has a role in the images which usually adds at the input, the term $\mathbf{h^{+} I h}$ represents the *output noise variance (ONV)* [18].This makes this system noise tolerant. The filter is then computed by taking the gradient of equation (2.6) w.r.t. $\boldsymbol{h}$ and afterwards putting the expression equal to zero. The resultant is given in the following equation

$$h = (I + S_x)^{-1} m \qquad (2.7)$$

In the above section MACH filter was described theoretically, this filter has its own limitations. The design of this filter over depends upon the mean of the training images (Mean of FTs).The mean of the images results in loss of some small and important details which are necessary for a good clutter rejection. Mean of the training images results in another clutter image which is quite different from original image [5].

2.1.2 EMACH Filter

EMACH filter is among the modern Developed filter of this era. The over reliance on the average training images in MACH filter lead to the development of Extended Maximum Average Correlation Height (EMACH) Filter which introduced a new a new metric value *All Image Correlation Height* (AICH) [5].

$$AICH = \frac{1}{N}\sum_{i=1}^{N}|h^{+}x_i|^2 - \alpha|h^{+}x_i|^2 \quad (2.8)$$

The range of the parameter α lies between 0 and 1 .This parameter helps in controlling the emphasis on a training image individually. The above mentioned equation can be re written as follows after substituting$\alpha = 2\beta - \beta^2$, where the value of β lies between 0 and 1.

$$AICH = \frac{1}{N}\sum_{i=1}^{N}|h^{+}x_i - \beta h^{+}m|^2 = h^{+}C_x^{\beta}h \qquad (2.9)$$

Where

$$C_x^{\beta} = \frac{1}{N}\sum_{i=1}^{N}(X_i - \beta M)(X_i - \beta M)^{+} \qquad (2.10)$$

In equation (2.9) AICH becomes the mean of the peak intensities in the correlation of the **N** Training Images. The MACH filter treated the training images as exemplar, but in this case an exemplar is expressed by $(\boldsymbol{x_i - \beta m})$.To match the correlation out puts of all training images as well another metric *Modified Average Similarity Measure* (MASM)[8] is introduced which is defined as follows

$$MASM = \frac{1}{N}\sum_{i=1}^{N}\left|g_i - f_{opt}\right|^2 \qquad (2.11)$$

Where

$$g_i = (X_i - \beta M)h^{*} \; and \; f_{opt} = (1-\beta)Mh^{*} \qquad (2.12)$$

Solving the equation (2.4) Yields as below, $\boldsymbol{f_{opt}}$ is determined by using the method described in section 2.1.

$$MASM = h^{+}S_x^{\beta}h \qquad (2.13)$$

$$where\ S_x^{\beta} = \frac{1}{N}\sum_{i=1}^{N}(X_i - (1-\beta)M)^{+}\ (X_i - (1-\beta)M) \qquad (2.14)$$

By minimizing the value of MASM the system becomes more tolerant towards distortion .This minimization allows the transformed i^{th} training image to resemble more to the reference images (transformed exemplars).Conclusively the EMACH filter should maximize AICH value while minimize the MASM value. Also adding the noise (ONV) to the denominator to carter the effect of noise, the following expression is obtained [5].

$$J(h) = \frac{AICH}{ONV+MASM} = \frac{h^{+}C_x^{\beta}h}{h^{+}\left(I+ S_x^{\beta}\right)h} \qquad (2.15)$$

After saving the above expression using gradient with respect to $\boldsymbol{h}$ the following expression is concluded.

$$(I +\ S_x^{\beta})^{-1}C_x^{\beta} = J(h)h \qquad (2.16)$$

Where ***J* (h)** is the Eigen value and $\boldsymbol{h}$ is the Eigen vector of the matrix $(I +\ S_x^{\beta})^{-1}C_x^{\beta}$

2.2 The Fourier Transform

A mathematician named Joseph Fourier introduced a method for inter conversion of signals between spatial domain and frequency domain. This basically a transformation .The best beauty of this transformation is that it is a reversible transform. This method has provided many services in the field of engineering and Physics.

Fourier claimed that a signal can be represented as an ordered (by Frequencies) sum of complex sinusoidal. These sinusoidal can be of different frequencies. The Fourier transform basically gives the information of frequencies present in a signal, which is quite helpful in analyzing the signal in another perspective. A signal can exist as a sum of sine waves as shown in the next diagram.

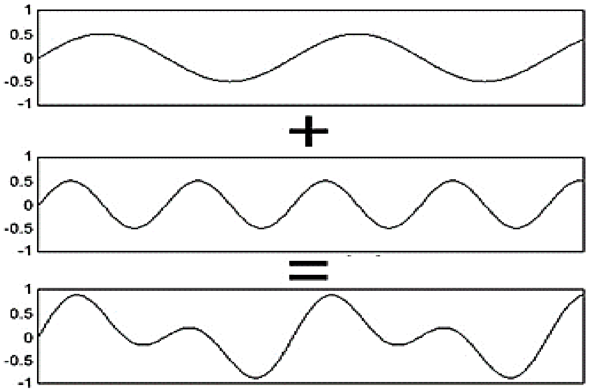


Figure 2-2 : Signal represented as a sum of sinusoidal [13]

For a continuous signal, mathematically this transformation can be written as

$$\hat{f}(\varepsilon) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \varepsilon} \, dx \quad (2.17)$$

Discrete Time Fourier Transform (DTFT) is used to convert Discrete Values of a spatial domain signal to frequency domain signal. DTFT has a very large number in practical applications these days [1].In time domain any physical quantity or signal is a function that is sampled among time. An image in time domain contains pixel intensity among a row or column of a matrix. DTFT of a vector can be calculated using the following expression

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \qquad K = 0,1,\dots,N-1 \quad (2.18)$$

In the above mentioned expression X_k is the Fourier transform value for K^{th} element place of the resulting Fourier transform vector. The sign $-\boldsymbol{i}$ is a complex number representation,x_n is the n^{th} element of the vector. The inverse Fourier transform can be obtained using the inverse calculation expression as follows.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k\frac{n}{N}} \qquad K = 0,1,\dots,N-1 \qquad (2.19)$$

The expressions mentioned in the above section are capable of inter conversion of a 1D array (vector) .But in practice image is a 2D array .the following expression can be used to calculate the DTFT and the inverse DTFT of a multidimensional array[19].

$$\hat{h}(k,l) = \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} e^{-i(\omega_k n+\omega_l m)}\ h(n,m) \qquad (2.20)$$

$$h(n.m) = \frac{1}{NM}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1} e^{i(\omega_k n+\omega_l m)}\ \hat{h}(k,l) \qquad (2.21)$$

Although DTFT Provides Fourier transform of the signal, but the method is too slow for computation .DTFT has a complexity ofN^2, later faster algorithms were developed. *Fast Fourier Transform* (FFT) is a method developed with much lower complexity than DTFT .The complexity of this method was reduced to $N\ log_2 N$ which is much lighter and faster the DTFT. This method is considered as the most important Algorithm of all time [20].A detailed explanation of FFT can be found at [21].

2.3 Fourier Transform Matrix Representation

As described in the above section, the Fourier Transform of a discrete signal can be computed using a summation operator and thus yielding a single value off Fourier Transform for the required Point $\boldsymbol{k}$. The point to be noted here is that this single value of the K^{th} Fourier Value has the effect of all the values present in the given vector. Similarly the Fourier Transform (Discrete) of *N* Points can be represented in the form of a matrix, When this Matrix is multiplied by an input signal it yields the DTFT of the input signal Vector. To compute the DFTF of *N* Points signal the transformation matrix will have *N* X *N* elements.

The DTFT Calculation can be performed as $X = Wx$ where X is the resultant Fourier Transform, W is the DTFT Transformation matrix and x is the vector of which the DTFT is to be computed using

For a vector with N points the transformation matrix of size N X N can be represented in the following way.

$$W = \left(\frac{\omega^{jk}}{\sqrt{N}}\right)_{j,k=0,\dots,N-1} \quad (2.22)$$

The value $\omega = e^{-\frac{2\pi i}{N}}$ represents the primitive N^{th} root of unity, also a complex value that is $i = \sqrt{-1}$ and $1/\sqrt{N}$ is the normalizing factor.expanding the above equation in the matrix form can be represented as.

$$W = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}, \quad (2.23)$$

Multiplication of this matrix with, a N point vector will yield the DFT of the input vector. This technique can be employed for computation of DTFT like a stored table. To find the inverse DTFT the sign of the exponent can be reverse to find the Inverse Transform another way to compute the inverse DTFT is to multiply the matrix with the conjugate of the transformation matrix $x = WX$.

An image is represented as a matrix or a 2D vector for the case of images the same mechanism can be employed to calculate the Fourier transform of images. In general first the Fourier transform can be computed using the DFT equation. The resulting Matrix is transposed and again the Fourier transform is computed and then again the transpose of the resulting is the final DFT of the 2D matrix.

$$X = DFT(DFT(x)^{Transpose})^{Transpose} \qquad (2.24)$$

Similarly by Matrix Multiplication the Fourier Transform of a 2D Matrix (image) can be computed in the following manner.

$$X = W \ \times W \quad (2.25)$$

2.4 Fixed Point Representation of Binary Numbers

In real life problem mathematical numbers usually have a fractional part, for example to represent a number half can be expressed by adding a fractional part after a decimal point. i.e. (1/2=0.5).Fixed point is a simple and easy way to represent Fractional numbers in binary representation for a machine.[22] Another way to represent fractional numbers is the Floating point Method which is capable of representing fractional numbers with a good precision. The floating point representation will require a separate hardware or module for conversion i.e. Floating Point Unit (FPU). In many of the cases speed is more critical than accuracy. Fixed point representation can be used in such cases to represent fractional numbers in binary. In old computers the calculations were only done on integers, and programmers used a software based method to deal with fractional numbers [23].Fractional part of a number falls between 0 and 1, in digital signal processing the use of real number is essential. The fixed point Binary number representation is a light weight method in terms of speed and hardware requirements. This method can also be employed in the digital design of a certain application where representation of real numbers is mandatory.

The binary point remains stationary at the same position where the number of binary digits in each word remains the same. In the case of floating point, position of the decimal point is determined at the time of processing. As compared to the fixed point, Floating point has much greater range of numbers that it can expressed. For fixed point representations ***m*** bits can be used for the whole part and ***n*** bits can be used for the fractional part this is also refers as Qm.n format.

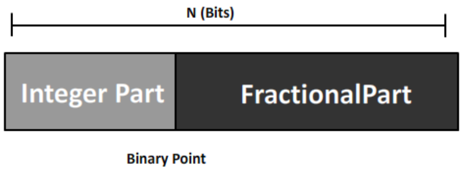


Figure 2-3: Word Representation of fixed point binary [22]

The total number of bits required to express the real number will be the number of bits of the Whore part plus the Number of bits used to represent the fractional part. For example a Q3.3 where m and n are equal to 3, the number of bits required to express the number will be 6.To represent a fractional number two parts are required, the integral part and the fractional part. The integral portion can be signed or unsigned .The following representation shown for signed and unsigned numbers.

An N-bit(s) Fixed Point Number in binary can be interpreted as an integer or a fractional number .For example for an unsigned 4-bit number with Q2.2, 2bits Integer part and 2 bits for fractional part. “0100” represents a fractional number 1.0, where as if it’s viewed as an integer it represents 4.

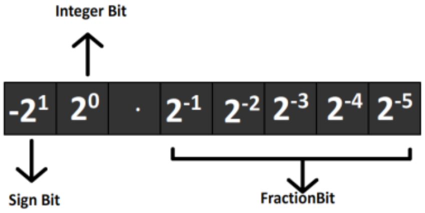


Figure 2-4 : Representation of fixed point numbers

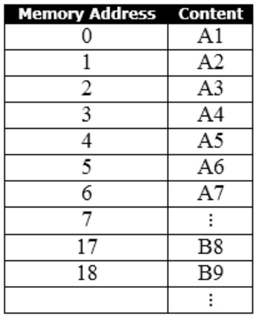
Chapter 3 Implementation

As described in the previous chapters, functional requirements for realizing a CPR processor. In this chapter the implementation and reasons of the modules is explained in detail. As the nature of major operations required to calculate the filter are far different from each other, it is then feasible to cut out basic operations and implement them in such a manner that the functional parts can be reused over and over instead of integration standalone modules, in this way the flexibility of the modules can be retained . For example the Fourier Transform conversion requires the addition and subtraction operation, whereas the calculation for Eigen vectors and values also requires an Adder and Subtractor. It is possible that we can use a single module for Addition and Subtraction and impose some control instructions through which operations can be controlled. The internal Architecture of the components is not added because of intellectual property of the author. The modules and their individually functionalities are explained briefly in the next section. These modules are further used to construct a processor capable of performing many complex computations on hardware.

3.1 Main Memory

The main memory is basically simple memory that is designed to hold the contents of the matrix. The memory is liner in nature .It can be assumed of an array holding the contents without any logic or in other words the items or elements are place randomly, the memory will be un familiar that to which matrix the elements belongs to. For example the memory is kept of 64 Kbytes.it means that we can store 64 Thousand values with the width of 1 Byte (8-Bits).Suppose we are need to store 2 matrix of size 3 x 3 in this memory, the values will be stored in a continuous manner.

$$A = \begin{bmatrix} A1 & A2 & A3 \\ A4 & A5 & A6 \\ A7 & A8 & A9 \end{bmatrix}, B = \begin{bmatrix} B1 & B2 & B3 \\ B4 & B5 & B6 \\ B7 & B8 & B9 \end{bmatrix}$$



Memory Address	Content
0	A1
1	A2
2	A3
3	A4
4	A5
5	A6
6	A7
7	⋮
17	B8
18	B9
	⋮

Figure 3-1: Elements of two Matrices placed in same memory (a memory overview)

In general computing the memory is usually managed by the operating system, this management is transparent to the user. In case of this processor a liner memory will be used and the management of the memory will also be transparent to the end user .The user will only have to write the micro instruction like “ADDm C, A, B”, The user will only specify whether he wants to add a scalar value or a matrix point to point. There are many orientations of memory that can be used, a dual port read memory design like a MIPS32 processor has a register file with output ports of data and three input port for address, another port for data input plus some Control pins like read write control. In this case a similar type of memory will be used, the major benefit of this memory is that it can provide two operands and a target location and can complete one instruction in a single cycle. The main aim is to keep the processor work in a single cycle per micro instruction. The memory layout is shown in the following diagram with its detailed pin in and pin out. R/W shows the read or writes signals. The contents of the memory are in mixed format, it will store fixed point representation and binary point representation together.

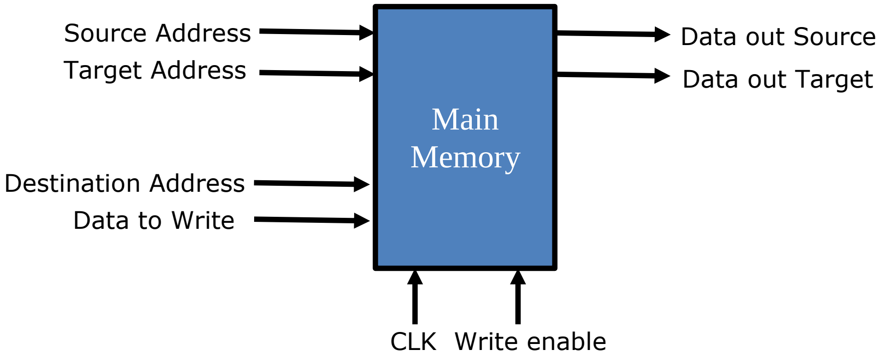


Figure 3-2 : Main memory module layout

The source and target address fetches from the same common pool of memory at the same time, and the write data will write to the same pool of memory at the address specified on the destination address port. The memory is restricted to one operation at an edge of clock cycle. The Memory is restricted to read on the rising edge of the clock cycle and allowed to write at the falling edge, this methodology is adopted to avoid hazards that will be discussed later.

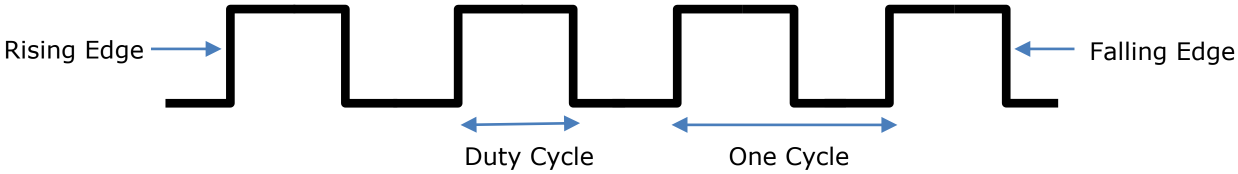


Figure 3-3: Description of common features of clock cycle.

To write the memory the write enable must be set to '1', then on the negative edge of the clock cycle with write enable at set position the Data available on the data input will be written to the destination address. The size of the address in bits depends on the memory size, for a memory of sized N requires an address line of $log_2(N)$ bits. For example a memory of 32 Locations will require an address of $log_2(32)$ bits, which is 5 bits. Thus to access a memory of 32 locations will require a 5 bit wide address line. Images usually have large number of pixels. Each pixel is a gray scale value at a certain row, column of a matrix location. As an image is represented by a large sized matrix, therefore to serve the purpose of image processing application a large amount of memory will be required. For example to store 8 gray scale images of sized 64 x 64 pixels, the depth of memory require will be (8x64x64) , which is 32768 locations, the width of the memory will be 8bits for a gray scale image. So the total amount of memory required will be 32,768 Bytes of 262,144 bits. This is very large amount of memory. The address bus required for this memory size will be $log_2(32768\ )$ which 15 bits is. A detailed mechanism of saving and reusing memory will be providing in chapter 5.

3.2 Memory Manager

To provide flexibility to the processor and ease of use to the user an on chip mechanism is adopted to provide the dynamic control of functionality. In terms of image size we need to specify a size of image that will allow the processor to learn that what image size it is working on. A module named memory manager is introduced to the design. This module is basically a lookup table that can hold three type of information about a matrix. It will hold the base address of the matrix, the total Number of rows and column of a matrix. The memory manager will also provide the ease of micro programming. The instructions and guidelines to the working of the processor will be flexible.

In a general code matrix handling mechanism could be written on the software side, but addition of this functionality will cost more in terms of CPU cycles as the number of decisions and comparisons will be increased, in general computing the main memory of the RAM is the major bottle neck, anything that requires to fetch something from the memory for an arithmetic use or a logical use can cost many machine cycle, some General computing systems even uses cache and virtual memory, in such a case where the required memory location is not available on the cache it will cost even more cycles to fetch, the beauty of the dedicated systems is that custom designs can overcome these limitations hence providing more performance, but in some cases the cost in terms of silicon may rise up to a very great extent. A mechanism for conversion of matrix from its subscript index to a single index is adopted in the realization of this processor the detailed reason for using this mechanism is explained in the next section. The processor provides an on chip mechanism for handling matrices, for this purpose a sparse memory will be used .the layout of this small sized memory is explained below and the module is shown in the figure.

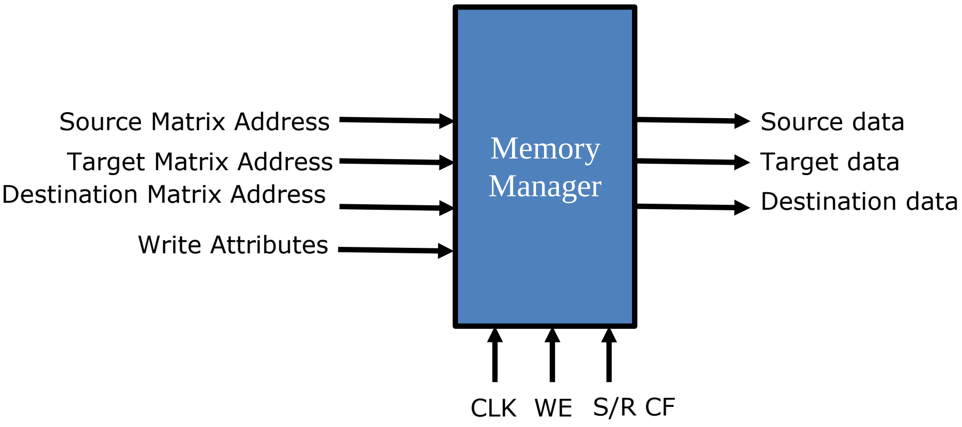
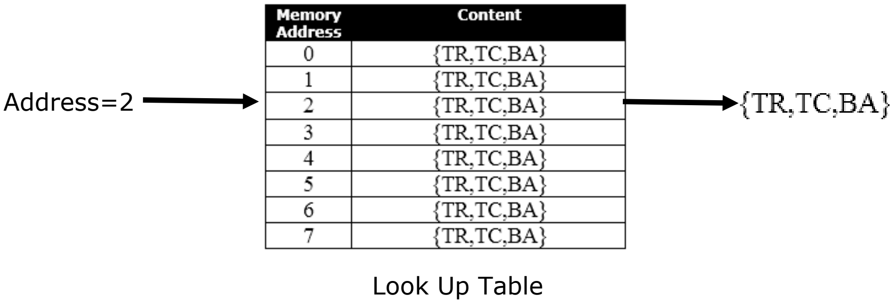


Figure 3-4: Layout of memory manager inputs and outputs.

Suppose that the requirement is to store 8 matrices in the memory of size(m, n), the height of this memory will be 8, and the address lines must be of 3 bits. Memory manager is a form of a lookup table which can be written by setting the Write Enable to ‘1’ on the negative edge of the clock .The data will be accessed from a common pool and will be available on the outputs. In the figure shown below the working mechanism of the this module is shown, an input address is 2, the module will output the data available on the address location 2, The second location points to the properties of Matrix Number 2, i.e. *Total Rows* (TR), *Total Columns* (TC) and *Base Address* (BA).



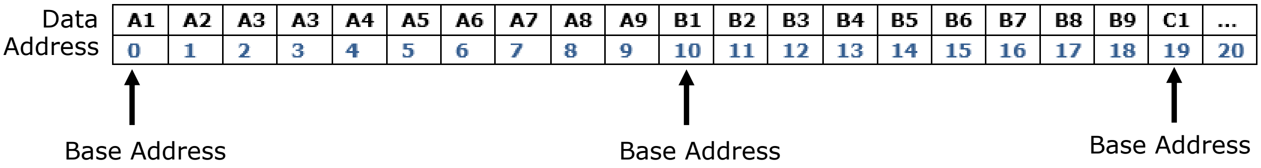
Memory Address	Content
0	{TR,TC,BA}
1	{TR,TC,BA}
2	{TR,TC,BA}
3	{TR,TC,BA}
4	{TR,TC,BA}
5	{TR,TC,BA}
6	{TR,TC,BA}
7	{TR,TC,BA}

Figure 3-5 : Working Mechanics of Memory Manager

The total size (in bits) of the width of single row in memory manager is the sum of Total Bits (TR) + Total Bits (TC) +Total Bits (BA).The base Address is determined on chip, The base address of the first of size 3x3 matrix stored in the linear memory will be kept to zero. The base address of the second of size 3 x 3 matrix stored in the memory will be

.Kept 9, indicating the elements of the second matrix is starting from the ninth location in the memory.

$$A = \begin{bmatrix} A1 & A2 & A3 \\ A4 & A5 & A6 \\ A7 & A8 & A9 \end{bmatrix}, B = \begin{bmatrix} B1 & B2 & B3 \\ B4 & B5 & B6 \\ B7 & B8 & B9 \end{bmatrix}, C = \begin{bmatrix} C1 & C2 & C3 \\ C4 & C5 & C6 \\ C7 & C8 & C9 \end{bmatrix}$$



Data	A1	A2	A3	A3	A4	A5	A6	A7	A8	A9	B1	B2	B3	B4	B5	B6	B7	B8	B9	C1	...
Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Figure 3-6: Linear storage of matric in memory showing their base addresses

3.3 Sub to Index

As explained in the above section that we are going to use a linear memory that will contain all the matrix placed all to gather without any boundary marking, the module memory manager will hold the information about the marking, that what is what on the main memory. The mechanism is that an element is a matrix can be addressed by its location number. The following will explain the location index of a matrix. Trivially an element of a matrix can be expressed by $\mathrm{A}(\mathrm{x}, \mathrm{y})$ where x is the row index and y is the column index this is usually known as the sub-script notation of an element, this unique element can also be represented by a unique index $A(Z)$.

In other words it can be written as $\mathrm{A}(\mathrm{x}, \mathrm{y}) = \mathrm{A}(\mathrm{Z}),$ let us consider an example of a below 3 x 3 matrix. Given a matrix A with values as given below

$$A = \begin{bmatrix} A(1,1) & A(1,2) & A(1,3) \\ A(2,1) & A(2,2) & A(2,3) \\ A(3,1) & A(3,2) & A(3,3) \end{bmatrix}$$

(a)

$$A = \begin{bmatrix} A(1) & A(2) & A(3) \\ A(4) & A(5) & A(6) \\ A(7) & A(8) & A(9) \end{bmatrix}$$

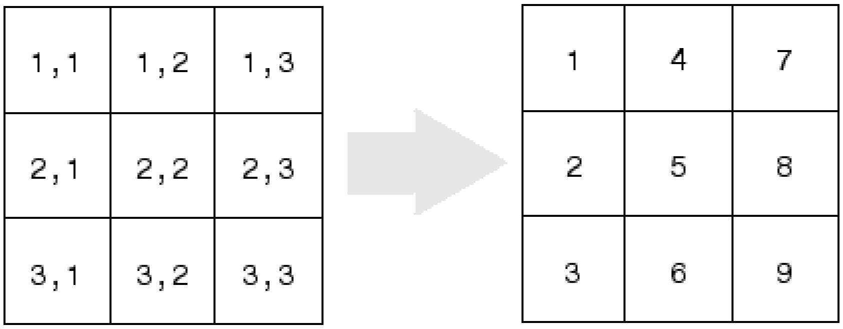
(b)

$$A = \begin{bmatrix} A(1) & A(4) & A(7) \\ A(2) & A(5) & A(8) \\ A(3) & A(6) & A(9) \end{bmatrix}$$

(c)

Figure 3-7: (a) Representation of Matrix in Subscript (b) Single index representation (row oriented) (c) Single index representation (column oriented)

The benefit of using the index notation is that we can put all the items of a matrix in a continuous order even in a straight memory and can address the element with index representation. To explain the concept in a more feasible let us consider the example of MATLAB function **sub2ind** $(\mathbf{x}, \mathbf{y})$**.**This function converts the subscript i.e. the row column value to a single index.



1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

1	4	7
2	5	8
3	6	9

Figure 3-8: Conversion of Subscript index to Single Index

To realize this conversion a methodology was adopted. The expression which helps in converting the subscript values to single index values. Mathematically the methodology can be written as.

$$\mathrm{A}(\mathrm{x},\mathrm{y}) = \mathrm{A}(\mathrm{Z}) = A(((x-1)*A_{TC})+y) \qquad (3.1)$$

$$\mathrm{Z} = ((x-1)*A_{TC})+y) \qquad (3.2)$$

(a) Row Oriented

$$\mathrm{A}(\mathrm{x},\mathrm{y}) = \mathrm{A}(\mathrm{Z}) = A\left(((y-1)*A_{TR})+x\right) \quad (3.3)$$

$$\mathrm{Z}(((y-1)*A_{TR})+x) \qquad (3.4)$$

(b) Column Oriented

The expression (a) can be used to compute the single index in row orientation and the expression (b) can be employed to calculate a single index in column orientation like MATLAB. A simple MATLAB simulation code is presented here to justify the index conversion, the first provided code work in Row orientation and the second sample code works in a column oriented manner.

```
% simulating for subscript to single index conversion
% Author Abdullah Aman Khan
% 01 - January -2013


% Specify the total number of row and col for a random matrix
TR=3;
TC=3;


Matrix=rand (TR, TC);

% Setting Up a loop that will traverse thorough all the values of
the Generated Random Matrix

for x=1:TR
    for y=1:TC

           single_index= ((y-1)*TR)+x;
           element_subscript=Matrix (sub2ind([TR TC],x,y))
           element_single_index=Matrix (single_index)
     end
end
```

```
% simulating for subscript to single index conversion
% Author Abdullah Aman Khan
% 01 - January -2013


% Specify the total number of row and col for a random matrix
TR=3;
TC=3;


Matrix=rand (TR, TC);
Matrix_tp=Matrix';
% Setting Up a loop that will traverse thorough all the values of
the Generated Random Matrix


for x=1:TR
    for y=1:TC

           single_index= ((x-1)*TC) +y;
           element_subscript=Matrix (sub2ind([TR TC],x,y))
           element_single_index=Matrix_tp(single_index)



    end
end
```

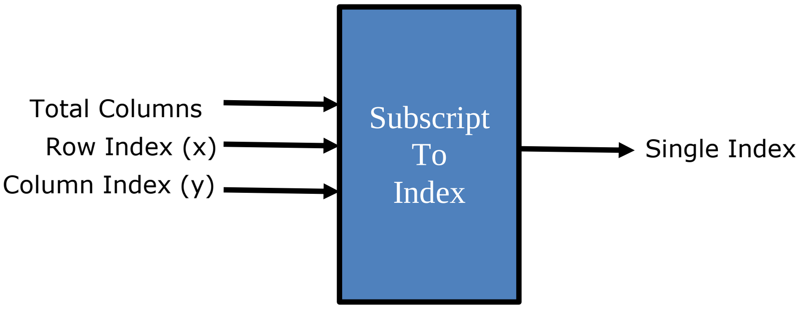


Figure 3-9: Conversion of Subscript index to Single Index

3.4 Sequence Guide

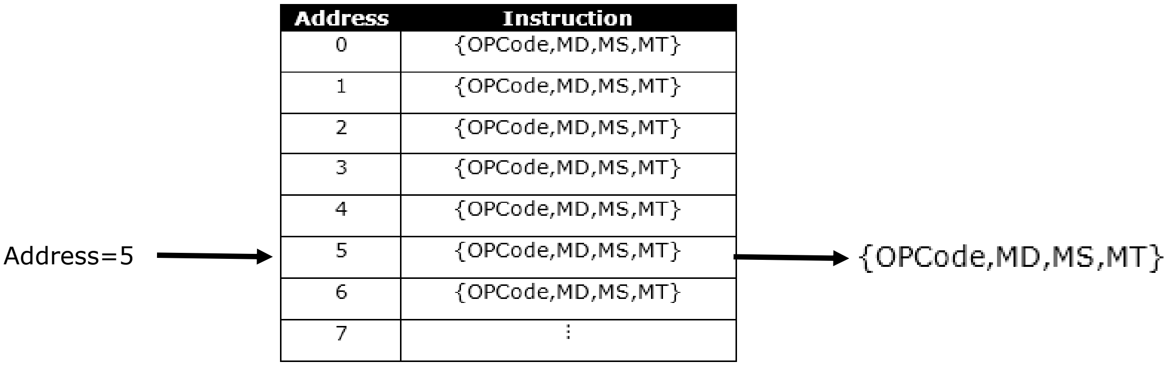
Sequence Guide is a type of instruction memory [24] that contains the instruction or operation to be carried out; one slice or a row contains the OPcode, the Source, Destination and the target Identifier of the matrix / scalar to be worked upon. The OPcode allows differentiating between the types of instructions according to which the control sequence will be generated. The nature of the instruction can be different from other. The types of instruction are explained in chapter 4. The flow control of a sequence is controlled by special register *Program Counter* (PC).The size of the Program Counter Register depends upon the Height of the instruction memory .Suppose if the height of the Instruction memory is 128 Locations, then the PC register size will be $log_2(128) = 7$ Bits. The operating mechanism of this guide is similar to the memory manager. MD is the Destination Matrix, MS is the Source Matrix and MT is the Target matrix.

$$A = \begin{bmatrix} A1 & A2 & A3 \\ A4 & A5 & A6 \\ A7 & A8 & A9 \end{bmatrix}, B = \begin{bmatrix} B1 & B2 & B3 \\ B4 & B5 & B6 \\ B7 & B8 & B9 \end{bmatrix}$$

C = A + B

Destination = Source + Target

Figure 3-10 : Source, Destination and Target Description



Address	Instruction
0	{OPCode,MD,MS,MT}
1	{OPCode,MD,MS,MT}
2	{OPCode,MD,MS,MT}
3	{OPCode,MD,MS,MT}
4	{OPCode,MD,MS,MT}
5	{OPCode,MD,MS,MT}
6	{OPCode,MD,MS,MT}
7	⋮

Figure 3-11 : The Operation Guide Contents view (Matrix Arithmetic)

The program counter old the address of the current instruction to be carried out. When a single instruction is executed the address of the next instruction is stored on the PC register, thus allowing a flow of instruction in a sequence. The inputs and outputs of the module are described in this section.

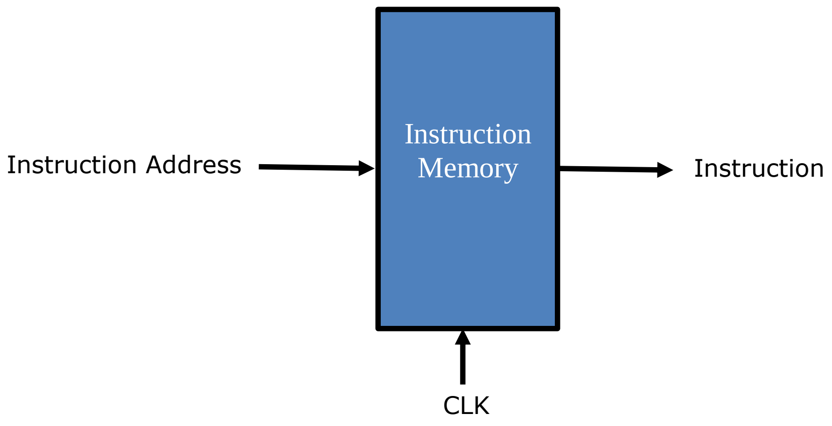


Figure 3-12: The Operation Guide Contents view

The instruction can be broken accordingly, The Size of the MS, MT and MD depends upon the Height of the memory manager. The memory manager size is kept 8 then the size of the MS, MT and MD will be kept $log_2(8) = 3$ bits per field. The size of the OPcode depends upon the number of instruction types, if there are 64 types of instructions the size of OPcode will be $log_2(64) = 6$ bits. The outputs of the instruction memory will be in binary. The total size of the (width) instruction will be then 15 bits.

3.5 Arithmetic and Logic Unit

To provide a re-use mechanism of the functional components a central unit is introduced which contains the main adder and is capable of performing logical decisions. The arithmetic and logic unit has two outputs, which are single elements. Arithmetic and Logic operations can be carried out on these operands.

The ALU OPcode identifies that what type of operations are to be performed on the operands, It can be logical or arithmetic like Add, Subtract, Divide .The ALU performs Arithmetic and logical operation on Fixed Point Binary (complex) and Simple Binary (complex) Notations .The Processor requires to operate on Simple Binary Numbers and Fixed Point Binary Numbers Together, The interpretation is different but the arithmetic operations resemble in nature. Fast Adders can be implemented to minimize the delay .Similar components can be internally Implemented to maximize the throughput. This module is a combinatory circuit; it is not dependent on clock cycles.

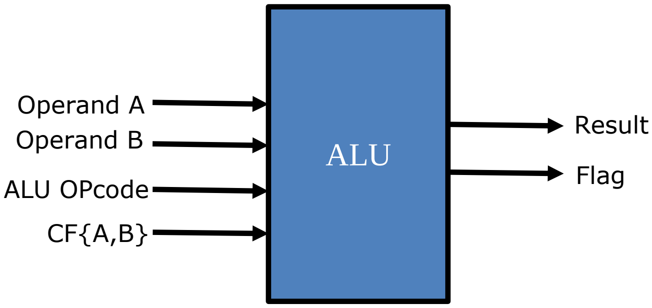


Figure 3-13 : Layout of the ALU

3.6 Control Unit

The control Unit will dispatch the concerned control signals based on the nature of instruction in use .This dispatch of instruction is based on the OPcode. The Control Unit is also a combinatory circuit, it is also a combinatory circuit and it is not dependent on clock cycles. It is a like a look up table that will output the corresponding set of control vector, the control signals then will be supplied to the to the corresponding fictional unit in the architecture.

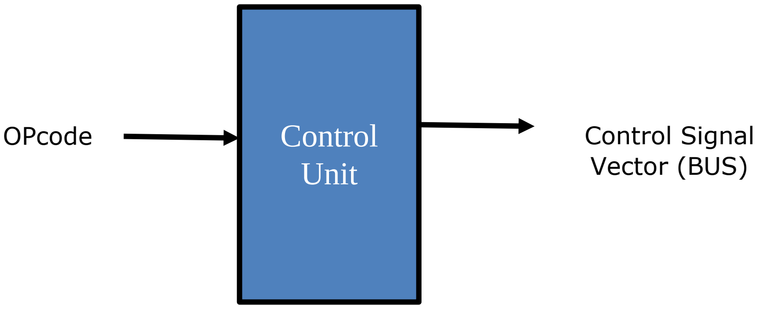
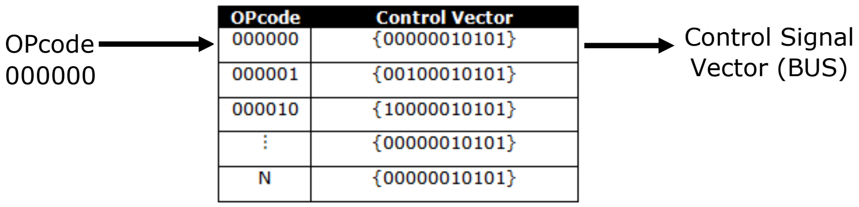


Figure 3-14: Control Unit layout

The control Unit operates like a lookup table, the control vector corresponding to the given OPcode will be available on the bus.



OPcode	Control Vector
000000	{00000010101}
000001	{00100010101}
000010	{10000010101}
⋮	{00000010101}
N	{00000010101}

Figure 3-15 : Inside view of the lookup table for Control Panel

3.7 Counter and Counter banks

In general computing usually loops are used to perform matrix, addition, subtraction etc. For example to traverse a full Matrix the following sample code will be handy. This code will traverse all the elements of the matrix. To provide a control mechanism for the loop a special modified counter is built to give a variable to increment over a clock cycle.

```
% simulating for Matrix Traversal
% Author: Abdullah Aman Khan
% 01 - January -2013

Matrix=rand (10, 10);

for x=1:10
    for y=1:10

                  Matrix(x,y)

    end
end
```

The counter designed have a filled that specifies the end point on which the counter will reset and raise a flag that the counter has completed one loop. This flag can be used to trigger other modules in the circuit. The initial value of the counter starts from one, on every clock cycle the initial value is incremented by one. These counter uses up an adder and some flip-flops internally.

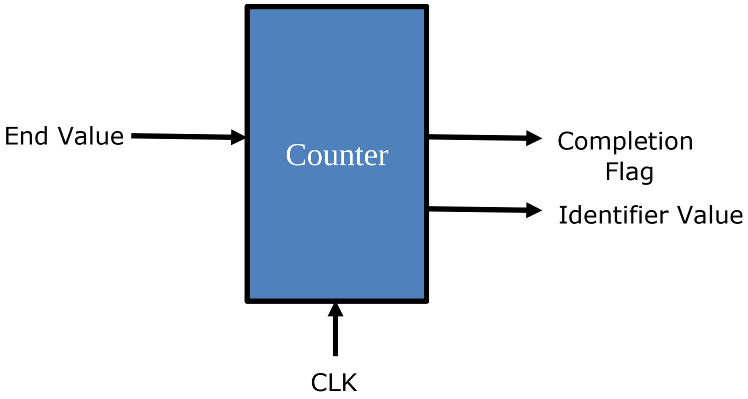


Figure 3-16 : Control Unit layout

A computer program mechanism is presented to explain the usage of the loop of n^3 complexity, this mechanism is provided to support on chip loops for matrix operations. Whenever the end limit of the loop identifier y is reached the identifier is reset to its initial position and then the upper identifier x is incremented until the upper identifier reaches its end limit. Here if we assume that the indexes are accessible through a memory, we need to derive x and y identifiers that will help in computing the results the diagram below explains the concept of using counter for and identifier. These generated indexes can be used further with a subscript to single index generating mechanism to generate singe addresses for the liner memory. Subtraction and scalar multiplications can be carried out in the same way.

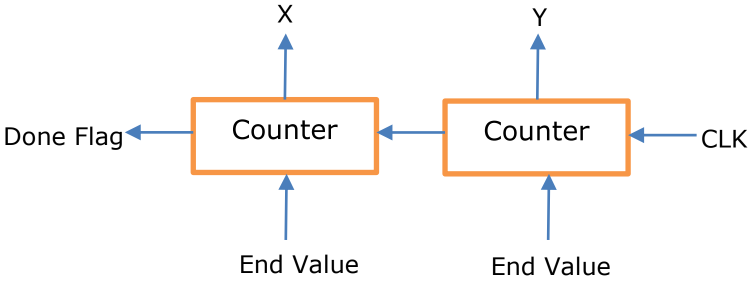


Figure 3-17 : Mechanism for generating index values for a matrix

```
% Simulation of Matrix Addition
% Author: Abdullah Aman Khan
% 01 - January -2013

A = rand (10, 10);
B = rand (10, 10);
C =[];

for x=1:10
    for y=1:10

                      C(x,y)=A(x,y)+B(x,y);

    end
end
```

Following figure shows the internal view of the counter used in the design, the module just increments the value starting from one, until the last value. On the last value the comparators computers the current and End value, if the current value is the last value then an end flag (signal) is raised to inform that one iteration is completed.

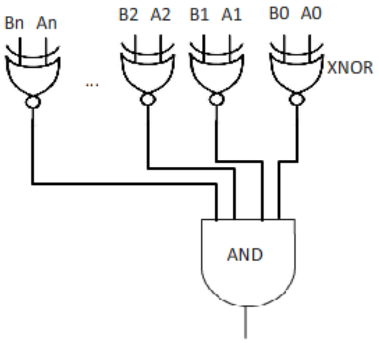


Figure 3-18 : Gate Level view of the comparator

The comparator can be implemented using XNOR Gates and then feeding the outputs of each XNOR gate to a common and Gate.

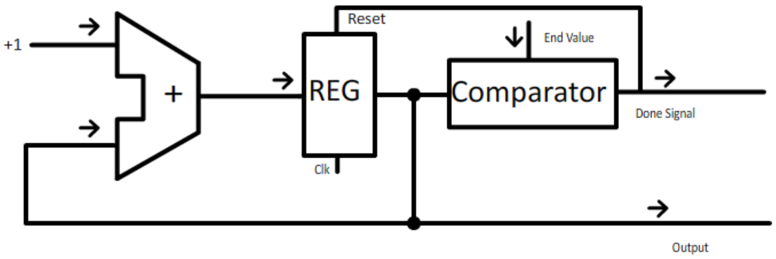


Figure 3-19 : Internal view of counter

A schoolbook matrix multiplication has high complexity, and requires n^3 operations .some other algorithms that have lesser complexity, but the complexity isn't much different from schoolbook multiplication algorithm. The next code shown uses up three loops with identifiers *(z, x, y)* .These nested loops are necessary for mutilation of matrix in school book manner. The diagram next to the code shows that counters can be used to generate index that can be used further .A switching network that uses up different multiplexers. From which we can control the flow of and instruction and manage the timing of the signal flow.

```
% Simulation of Matrix Multiplication
% Author: Abdullah Aman Khan
% 01 - January -2013

a=rand(3,9);
b=rand(9,3);


[r1 c1]=size(a);
[r2 c2]=size(b);

tmp=zeros(r1,c2);

for z=1:r1
    for x=1:c2
           t=0;
        for y=1:c1
                t=(a(z,y)*b(y,x))+t;
        end
        tmp(i,k)=t;
    end
end
```

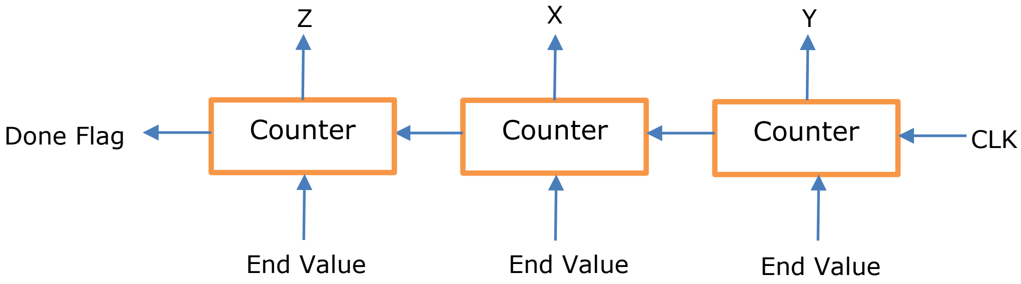


Figure 3-20 : Mechanism for generating index values for matrix multiplications

3.8 Multiplexer

Multiplexer is one of the core components of digital design, this components can save a lot of resources in terms of hardware. Multiplexers can transfer or select data available on its inputs to the output port, based on the selection which can be selected from selection switch. For N numbers of input lines the required size of selection switch will be $log_2(N)$ more details for a multiplexer design and working can be found at[25, 26].The number of inputs required for each multiplexer may vary according the required size. This component costs low in terms of hardware and has very low latency, a multiplexer helps in saving huge amount of buses and eventually helps in lowering the cost.

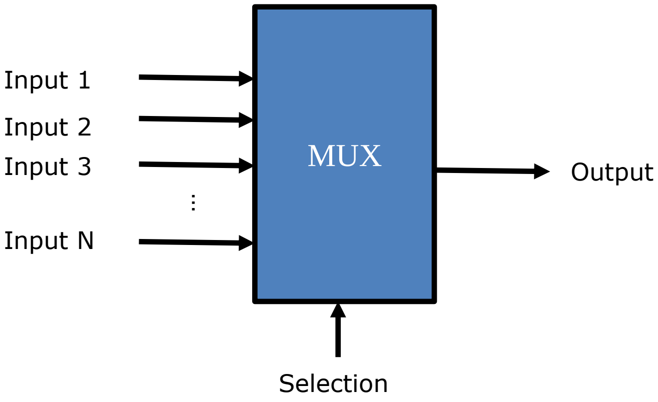


Figure 3-21: A Multiplexer in-out Description.

Chapter 4 Instruction Set Architecture

As the nature of operations required compute the filters are different in nature, the operations will be carried out by specifying instruction in the operation guide. By adopting this methodology the chances of encountering error will decrease .Besides this factor the flexibility will be increase, adopting this methodology will allow this processor to realize much other application, but the main aim is to calculate and use correlation filters.

The Filter calculation can be achieved by using the Instruction Set Architecture is explained in the next section. The instructions are classified according to their use. The instruction are classified as follows

1. Matrix with Matrix
2. Matrix with Scalar
3. Scalar with Scalar
4. Control Instructions
5. Matrix Manipulation

Matrix Multiplication with Matrix

This instruction Multiples a matrix M1 to a matrix M2 and saves the results to a new Matrix Location M0.

M0=M1*M2

MULm Destination Matrix, Source Matrix, Target Matrix

MULm M0, M1, M2

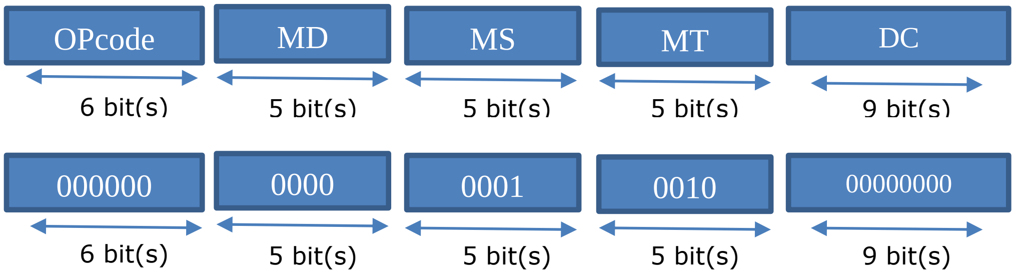


Figure 4-1: Instruction Vector Bits lay out for Matrix Multiplication with Matrix

Matrix dot multiplication with Matrix

This instruction dot multiples a matrix M1 to a matrix M2 and saves the results to a new Matrix Location M0.

M0=M1.*M2

MULmd Destination Matrix, Source Matrix, Target Matrix

MULmd M0, M1, M2

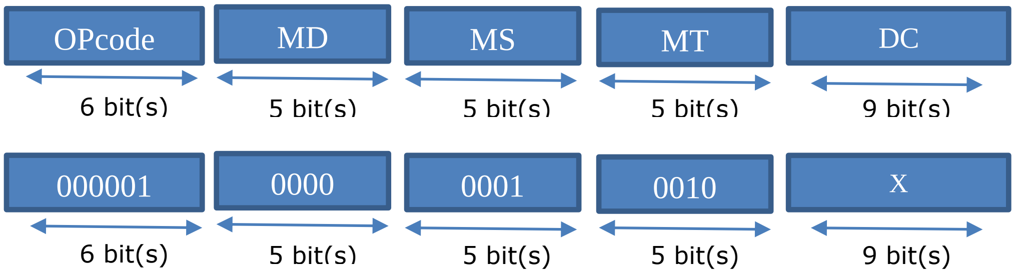


Figure 4-2: Bits lay out Matrix dot multiplication with Matrix

Matrix dot multiplication with immediate Value

This instruction dot multiples a matrix M1 to a scalar Value and saves the results to a new Matrix Location M0.

M0=M1.*2

MULmi Destination Matrix, Source Matrix, “Immediate Value”

MULmi M0, M1, 2

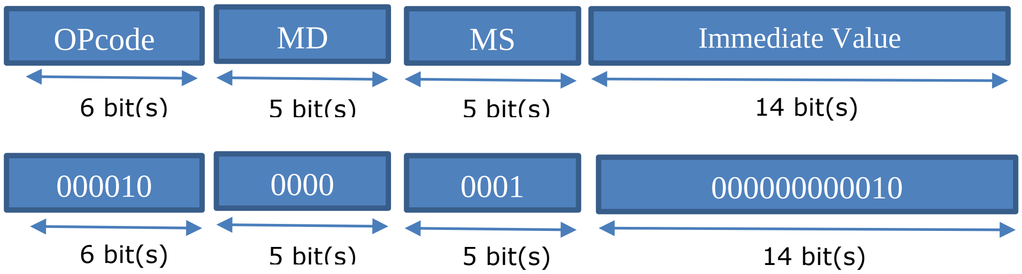


Figure 4-3: Bits lay out for Matrix dot multiplication with immediate Value

Matrix dot multiplication with stored Value

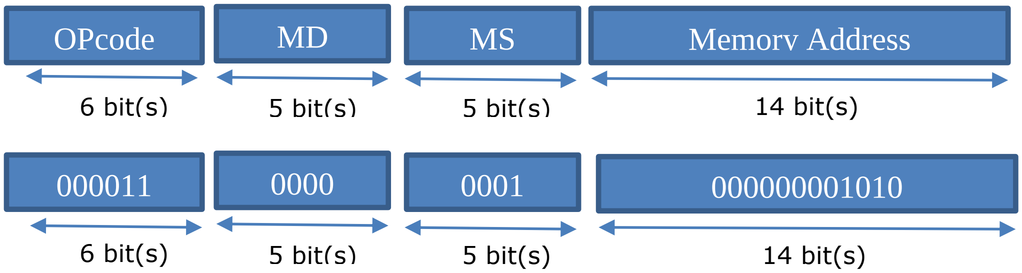
This instruction dot multiples a matrix M1 to a value stored at the provided address and saves the results to a new Matrix Location M0.

A=5;

M0=M1.*A

MULmv Destination Matrix, Source Matrix, Location Address in Memory

MULmv M0, M1, 10



OPcode	MD	MS	Memory Address
6 bit(s)	5 bit(s)	5 bit(s)	14 bit(s)
000011	0000	0001	000000001010
6 bit(s)	5 bit(s)	5 bit(s)	14 bit(s)

Figure 4-4 : Bits lay out for Matrix dot multiplication with stored Value

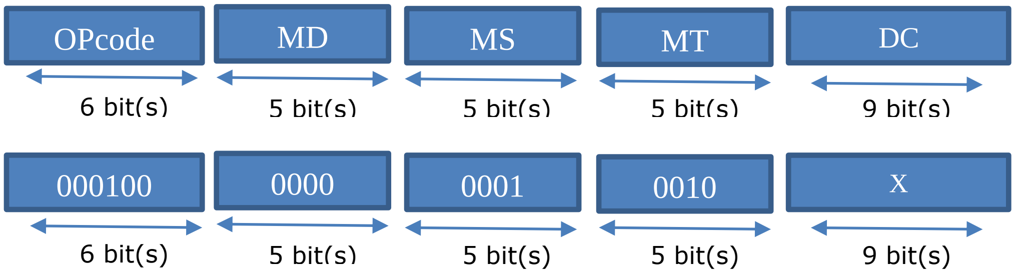
Matrix dot division with Matrix

This instruction dot divides a matrix M1 to a matrix M2 and saves the results to a new Matrix Location M0.

M0=M1. /M2

DIVm Destination Matrix, Source Matrix, Target Matrix

DIVm M0, M1, M2



OPcode	MD	MS	MT	DC
6 bit(s)	5 bit(s)	5 bit(s)	5 bit(s)	9 bit(s)
000100	0000	0001	0010	X
6 bit(s)	5 bit(s)	5 bit(s)	5 bit(s)	9 bit(s)

Figure 4-5: Bits lay out of Matrix dot division with Matrix

Matrix dot Division with stored Value

This instruction dot divides a matrix M1 to a value stored at the provided address and saves the results to a new Matrix Location M0.

A=5;

M0=M1. /A

DIVmv Destination Matrix, Source Matrix, Location Address in Memory

DIVmv M0, M1, 10

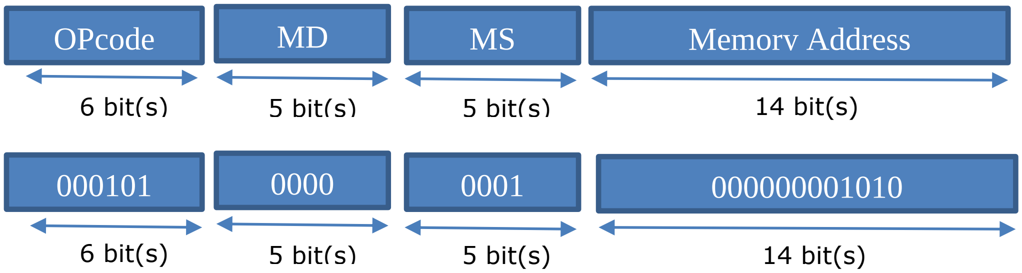


Figure 4-6: Bits lay out for Matrix dot Division with stored Value

Matrix dot Division with Immediate Value

This instruction dot divides a matrix M1 to an immediate value provided and saves the results to a new Matrix Location M0.

M0=M1. /2

DIVmi Destination Matrix, Source Matrix, Location Address in Memory

DIVmi M0, M1, 10

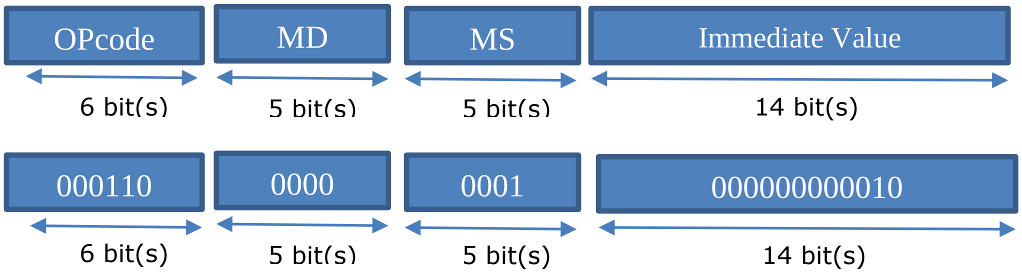


Figure 4-7: Bits lay out for Matrix dot Division with Immediate Value

Matrix Addition with Matrix

This instruction adds a matrix M1 to a matrix M2 and saves the results to a new Matrix Location M0.

M0=M1+M2

ADDm Destination Matrix, Source Matrix, Target Matrix

ADDm M0, M1, M2

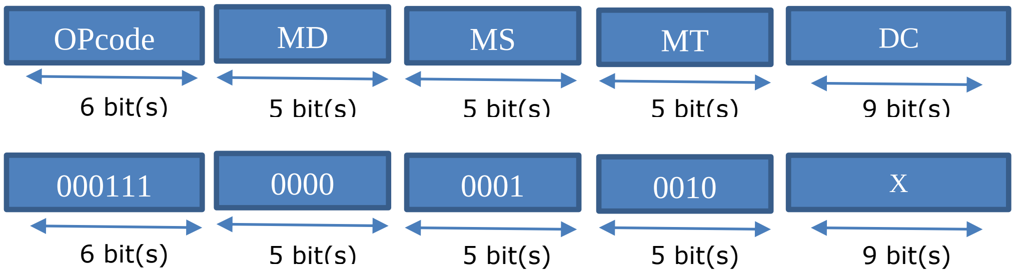


Figure 4-8: Bits lay out for Matrix Addition with Matrix

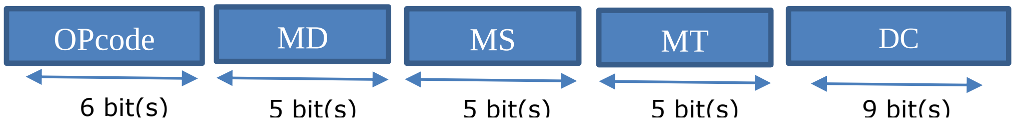
Matrix Subtraction with Matrix

This instruction subtracts a matrix M1 to a matrix M2 and saves the results to a new Matrix Location M0.

M0=M1-M2

SUBm Destination Matrix, Source Matrix, Target Matrix

SUBm M0, M1, M2



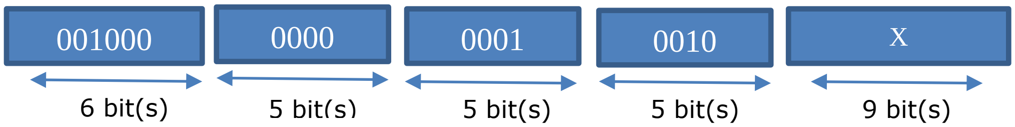


Figure 4-9 : Bits lay out Matrix Subtraction with Matrix

Matrix Addition with stored Value

This instruction Adds a matrix M1 to a value stored at the provided address and saves the results to a new Matrix Location M0.

A=5;

M0=M1+A

ADDmv Destination Matrix, Source Matrix, Location Address in Memory

ADDmv M0, M1, 10

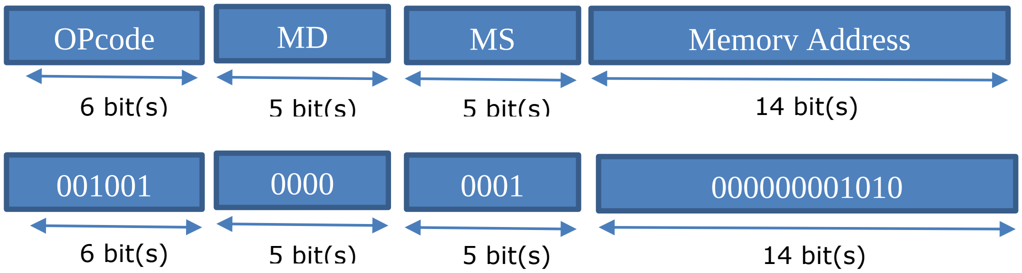


Figure 4-10: Bits lay out Matrix Addition with stored Value

Matrix Subtraction with stored Value

This instruction subtracts a matrix M1 to a value stored at the provided address and saves the results to a new Matrix Location M0.

A=5;

M0=M1-A

SUBmv Destination Matrix, Source Matrix, Location Address in Memory

SUBmv M0, M1, 10

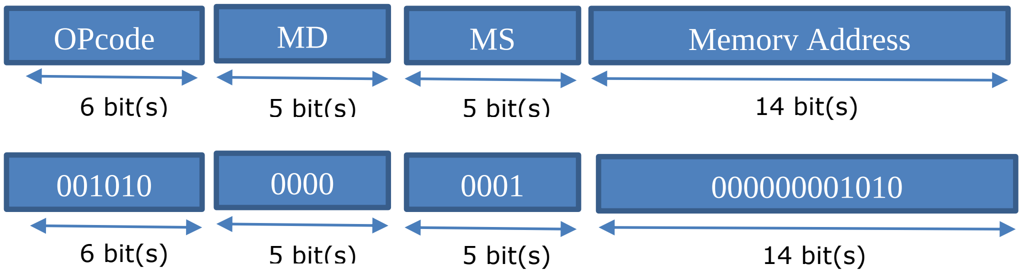


Figure 4-11 : Bits lay out of Matrix Subtraction with stored Value

Matrix dot Power with stored Value

This instruction calculates the power of all the elements in matrix M1 to a value stored at the provided address and saves the results to a new Matrix Location M0.

A=5;

M0=M1. ^A

POWmv Destination Matrix, Source Matrix, Location Address in Memory

POWmv M0, M1, 10

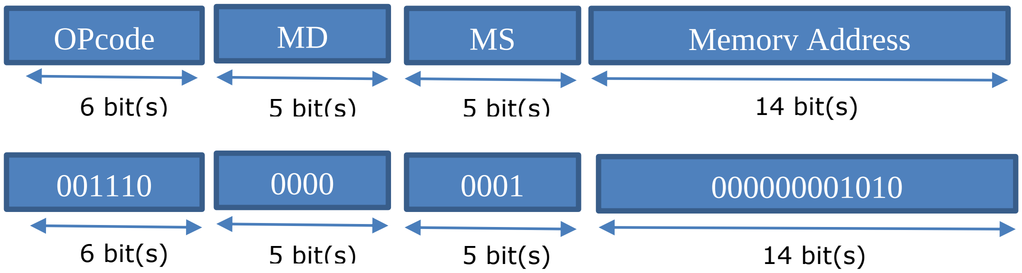


Figure 4-12 : Bits lay out Matrix dot Power with stored Value

Matrix Subtraction with immediate Value

This instruction subtracts a matrix M1 to a scalar Value and saves the results to a new Matrix Location M0.

M0=M1-2

SUBmi Destination Matrix, Source Matrix, “Immediate Value”

SUBmi M0, M1, 2

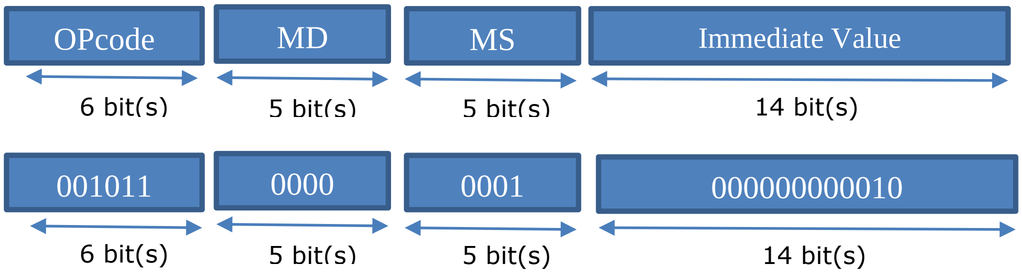


Figure 4-13: Bits lay out Matrix Subtraction with immediate Value

Matrix Addition with immediate Value

This instruction adds a matrix M1 to a scalar Value and saves the results to a new Matrix Location M0.

M0=M1+2

ADDmi Destination Matrix, Source Matrix, “Immediate Value”

ADDmi M0, M1, 2

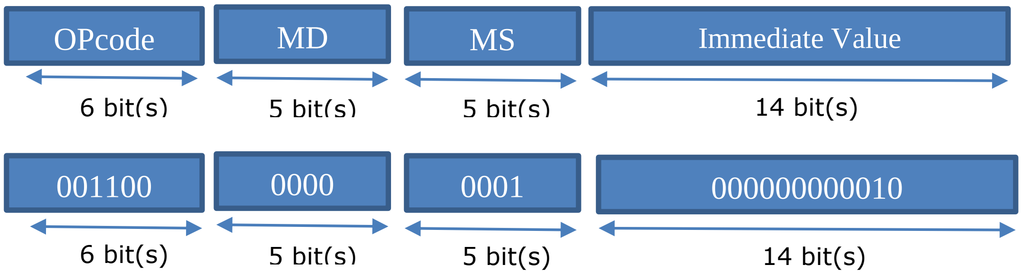


Figure 4-14: Bits lay out Matrix Addition with immediate Value

Matrix Raised to the Power of immediate Value

This instruction calculates the elements of matrix M1 to the power of a scalar Value and saves the results to a new Matrix Location M0.

M0=M1. ^2

POWmi Destination Matrix, Source Matrix, “Immediate Value”

POWmi M0, M1, 2

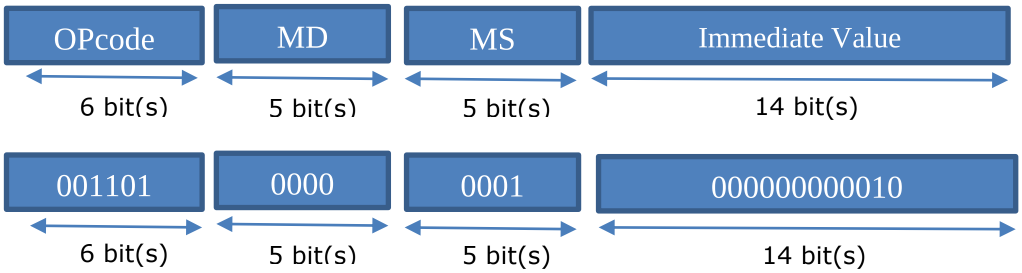


Figure 4-15: Bits lay out Matrix Raised to the Power of immediate Value

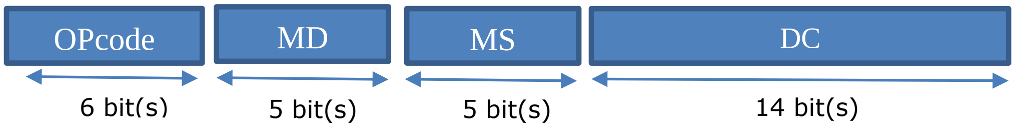
Transpose of Matrix

This instruction calculates the transpose of matrix M1 and saves the results to a new Matrix Location M0.

M0=M1’

TP Destination Matrix, Source Matrix

TP M0, M1



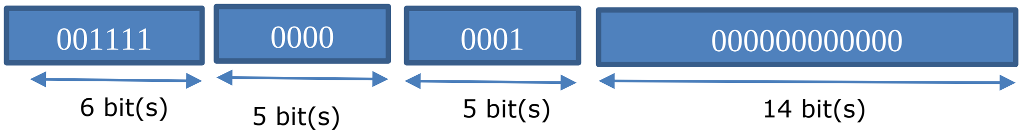


Figure 4-16: Bits lay out Transpose of Matrix

Set Conjugate Flag

This instruction sets a flag indication that the given matrix is conjugate.

SCONJ Destination Matrix

SCONJ M0

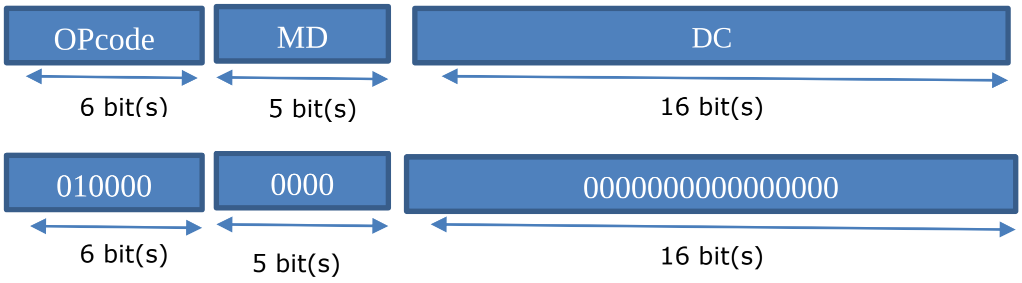


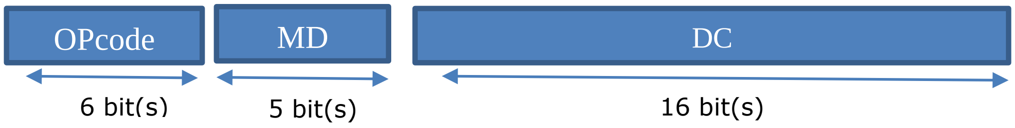
Figure 4-17: Bits lay out Set Conjugate Flag

Re-Set Conjugate Flag

This instruction resets a flag indication that the given matrix is not conjugate.

RCONJ Destination Matrix

RCONJ M0



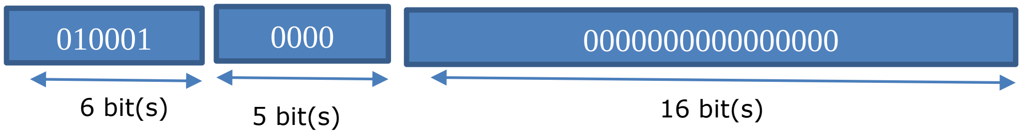


Figure 4-18: Bits lay out Re-Set Conjugate Flag

Fourier Transform First Pass

This instruction multiplies the matrix to a stored table, which computes the Fourier Transform and stored to the destination.

FTm Destination Matrix, Source Matrix

FTm M0, M1

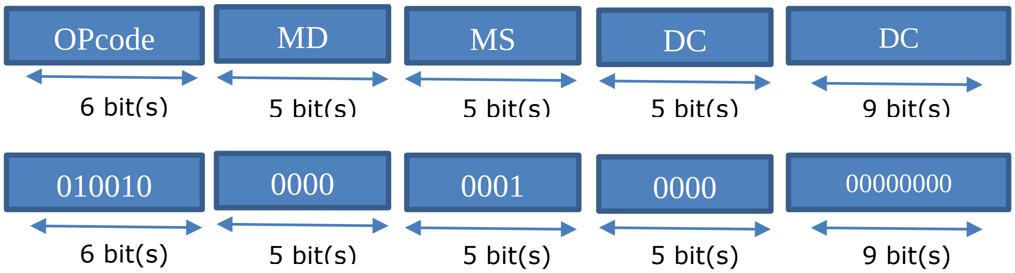


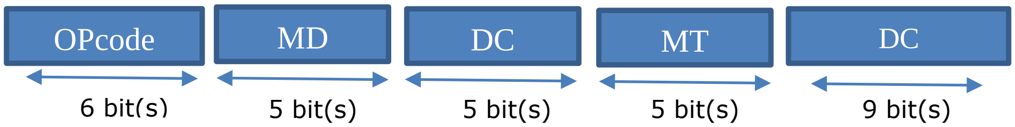
Figure 4-19: Bits lay out Fourier Transform First Pass

Fourier Transform Second Pass

This instruction multiplies the matrix to a stored table, which computes the Fourier Transform. Provided M1 contains the first pass results and is stored to M0.

FTmm Destination Matrix, Source Matrix

FTmm M0, M1



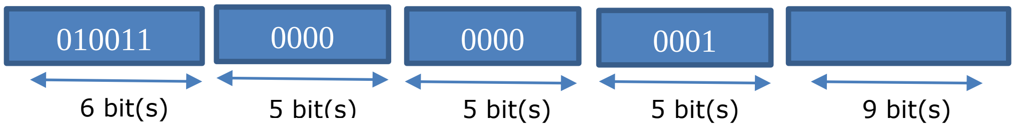


Figure 4-20: Bits lay out Fourier Transform Second Pass

Scalar Multiplication

This instruction Multiples scalar Value R1 to a matrix R2 and saves the results to a new memory Location R0.

R0=R1*R2

MUL Destination, Source, Target

MUL R0, R1, R2

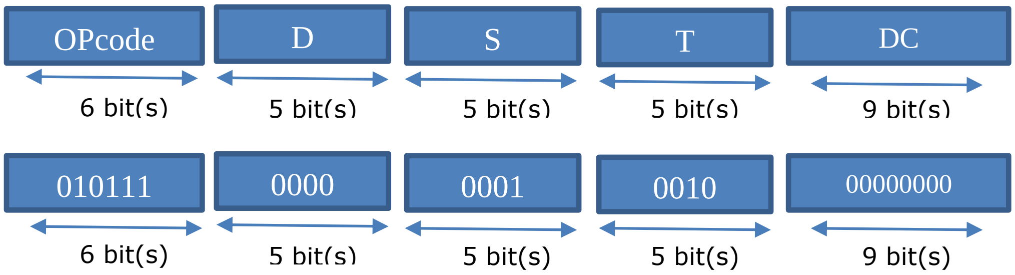


Figure 4-21: Bits lay out Scalar Multiplication

Scalar Addition

This instruction adds scalar Value R1 to a matrix R2 and saves the results to a new memory Location R0.

R0=R1+R2

ADD Destination, Source, Target

ADD R0, R1, R2

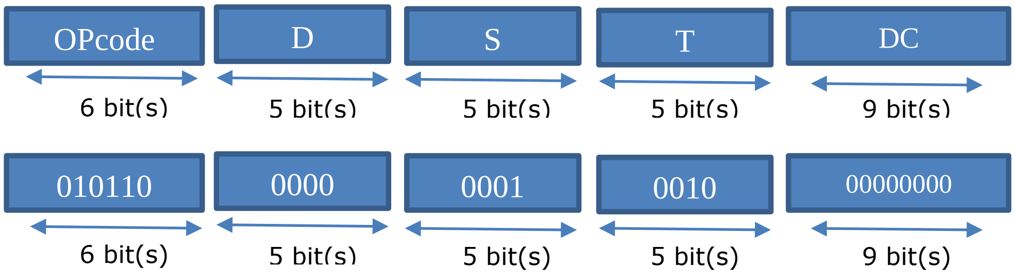


Figure 4-22: Bits lay out Scalar Addition

Scalar Subtraction

This instruction subtracts scalar Value R1 to a matrix R2 and saves the results to a new memory Location R0.

R0=R1-R2

SUB Destination, Source, Target

SUB R0, R1, R2

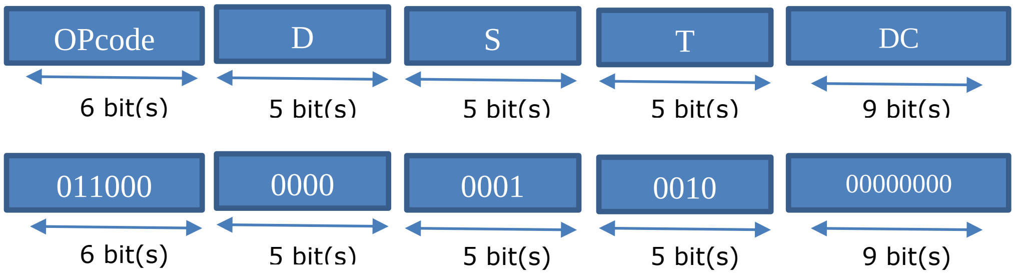


Figure 4-23: Bits lay out Scalar Subtraction

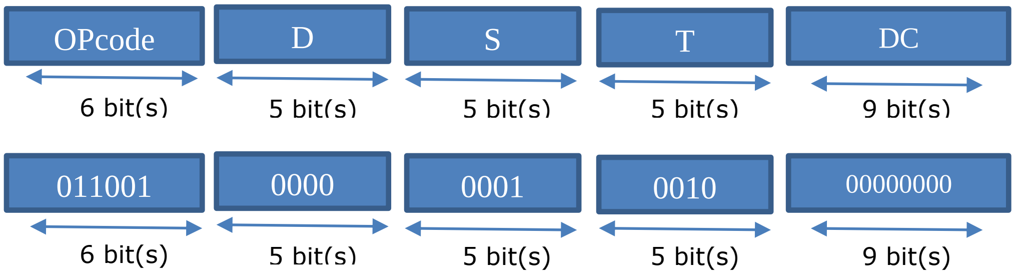
Scalar Division

This instruction Divides scalar Value R1 to a R2 and saves the results to a new memory Location R0.

R0=R1/R2

DIV Destination, Source, Target

DIV R0, R1, R2



OPcode	D	S	T	DC
6 bit(s)	5 bit(s)	5 bit(s)	5 bit(s)	9 bit(s)
011001	0000	0001	0010	00000000
6 bit(s)	5 bit(s)	5 bit(s)	5 bit(s)	9 bit(s)

Figure 4-24: Bits lay out Scalar Division

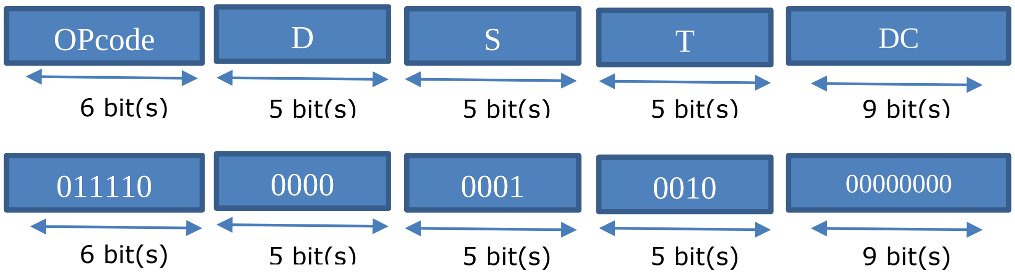
Scalar Power

This instruction computes the exponent of scalar Value R1 to a scalar R2 and saves the results to a new memory Location R0.

R0=R1^R2

POW Destination, Source, Target

POW R0, R1, R2



OPcode	D	S	T	DC
6 bit(s)	5 bit(s)	5 bit(s)	5 bit(s)	9 bit(s)
011110	0000	0001	0010	00000000
6 bit(s)	5 bit(s)	5 bit(s)	5 bit(s)	9 bit(s)

Figure 4-25: Bits lay out Scalar Power

Scalar Addition with immediate Value

This instruction adds the source R1 with an immediate provided value and save to Location R0.

R0=R1+2

ADDi Destination Matrix, Source Matrix, “Immediate Value”

ADDi R0, R1, 2

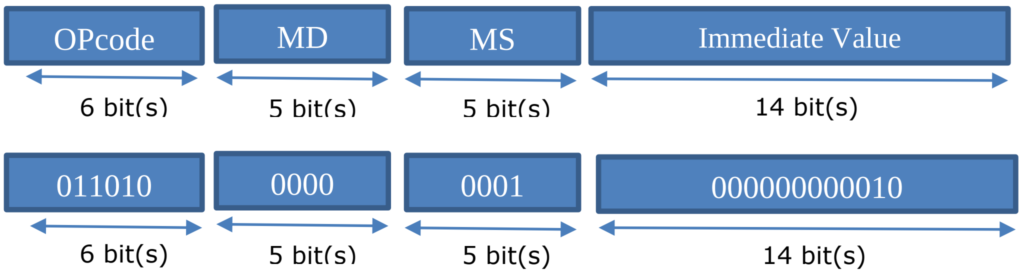


Figure 4-26: Bits lay out Scalar Addition with immediate Value

Scalar subtraction with immediate Value

This instruction adds the source R1 with an immediate provided value and save to Location R0.

R0=R1-2

SUBi Destination Matrix, Source Matrix, "Immediate Value"

SUBi R0, R1, 2

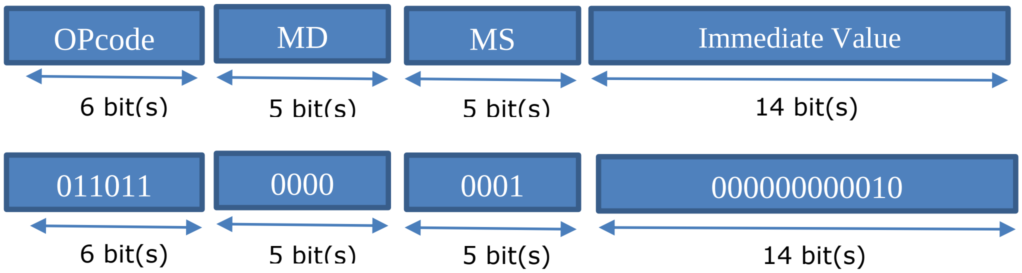


Figure 4-27: Bits lay out Scalar subtraction with immediate Value

Scalar Division with immediate Value

This instruction add the source R1 with an immediate provided value and save to Location R0.

R0=R1/2

DIVi Destination Matrix, Source Matrix, “Immediate Value”

DIVi R0, R1, 2

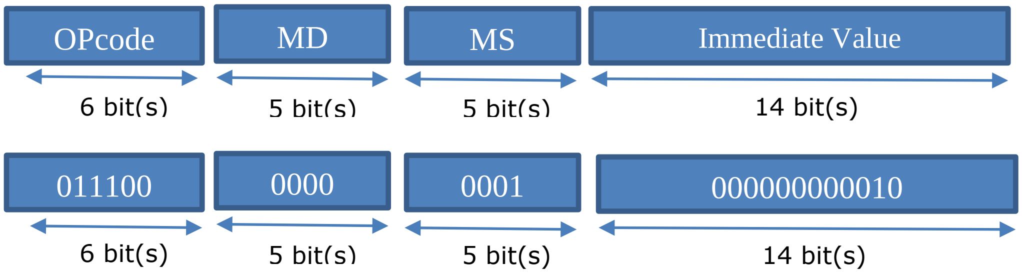


Figure 4-28: Bits lay out Scalar Division with immediate Value

Scalar Power with immediate Value

This instruction adds the source R1 with an immediate provided value and save to Location R0.

R0=R1^2

POWi Destination Matrix, Source Matrix, “Immediate Value”

POWi R0, R1, 2

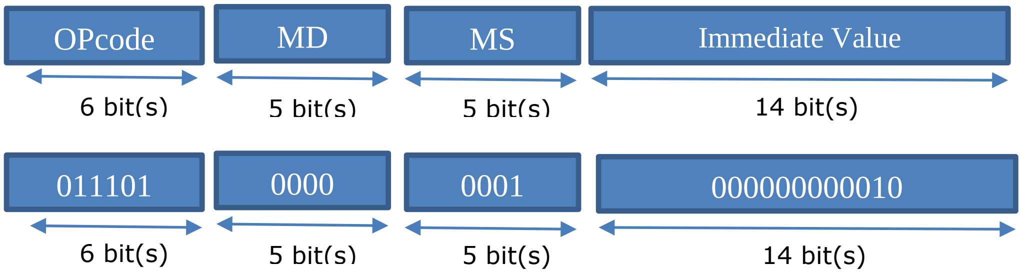


Figure 4-29: Bits lay out Scalar Power with immediate Value

Jump if equal

This instruction performs a branch to specified location if the operands R1, R2 are Equal.

JEQ Destination Matrix, Source Matrix, “Jump Address”

JEQ R0, R1, 2

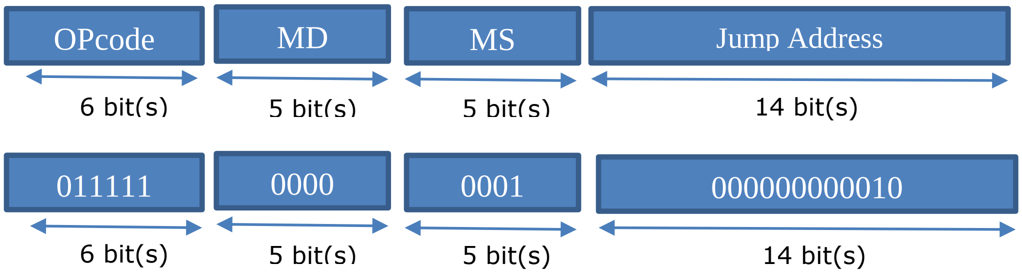


Figure 4-30: Bits lay out Jump if equal

Jump if not equal

This instruction performs a branch to specified location if the operands R1, R2 are not equal.

JNE Destination Matrix, Source Matrix, “Jump Address”

JNE R0, R1, 2

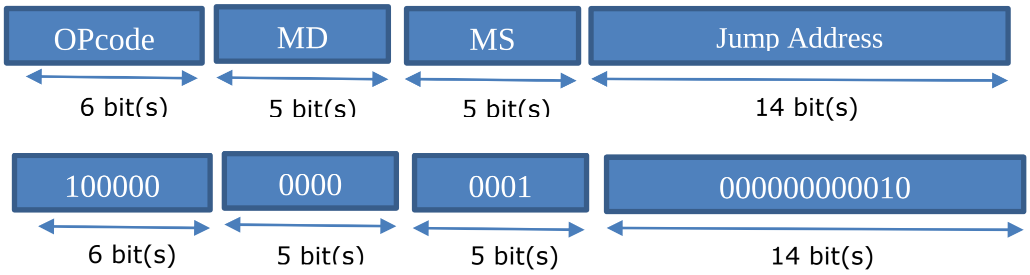


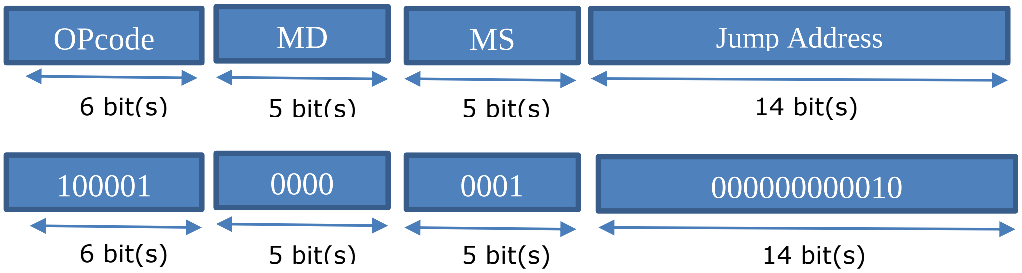
Figure 4-31: Bits lay out Jump if not equal

Jump if is less than or equal

This instruction performs a branch to specified location if the operands R1 is less than or equal to R2.

JLE Destination Matrix, Source Matrix, “Jump Address”

JLE R0, R1, 2



OPcode	MD	MS	Jump Address
6 bit(s)	5 bit(s)	5 bit(s)	14 bit(s)
100001	0000	0001	000000000010
6 bit(s)	5 bit(s)	5 bit(s)	14 bit(s)

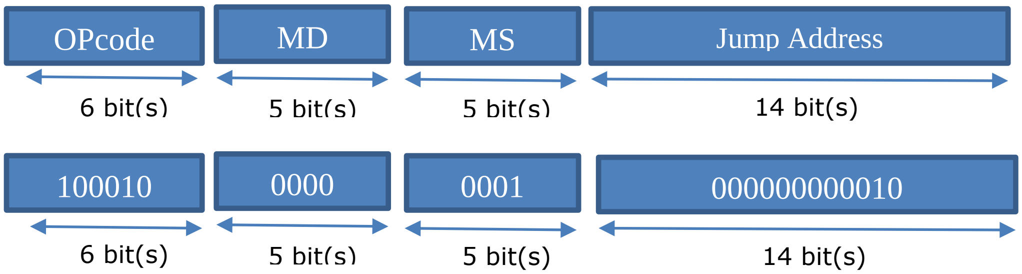
Figure 4-32: Bits lay out Jump if is less than or equal

Jump if is Greater than or equal

This instruction performs a branch to specified location if the operands R1 is Greater than or equal to R2.

JGE Destination Matrix, Source Matrix, “Jump Address”

JGE R0, R1, 2



OPcode	MD	MS	Jump Address
6 bit(s)	5 bit(s)	5 bit(s)	14 bit(s)
100010	0000	0001	000000000010
6 bit(s)	5 bit(s)	5 bit(s)	14 bit(s)

Figure 4-33: Bits lay out Jump if is Greater than or equal

Unconditional Jump

This instruction performs a branch to specify without any condition.

JMP “Jump Address”

JMP 2

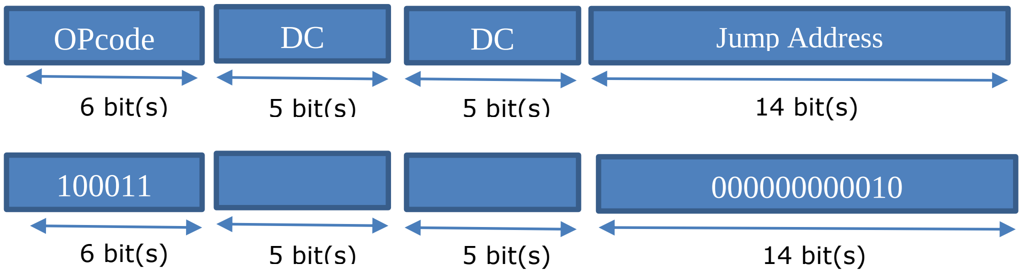


Figure 4-34: Bits lay out for Unconditional Jump

Jump if is less than

This instruction performs a branch to specified location if the operands R1 are less than R2.

JLT Destination Matrix, Source Matrix, “Jump Address”

JLT R0, R1, 2

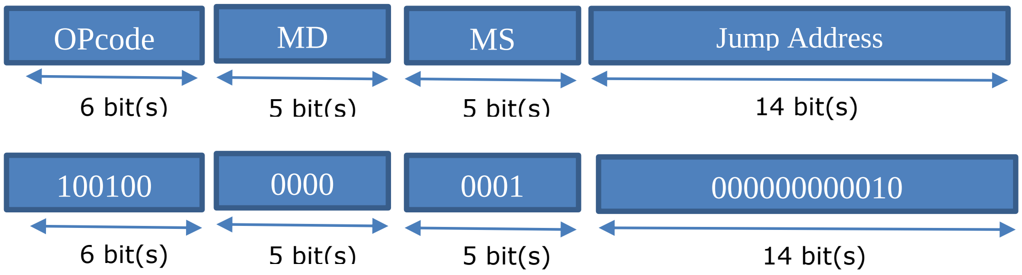


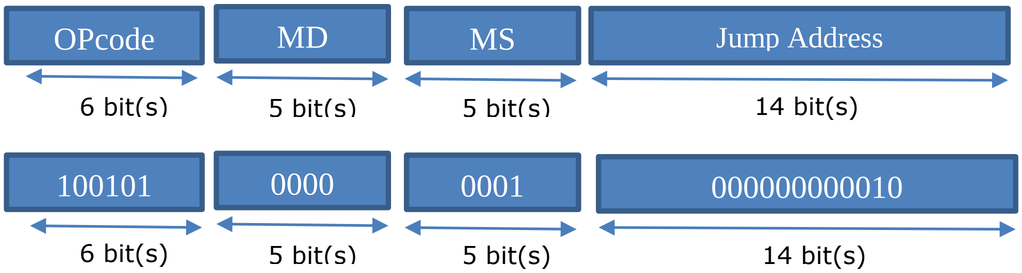
Figure 4-35: Bits lay out for Jump if is less than

Jump if is Greater than

This instruction performs a branch to specified location if the operands R1 is Greater than R2.

JGT Destination Matrix, Source Matrix, “Jump Address”

JGT R0, R1, 2



OPcode	MD	MS	Jump Address
6 bit(s)	5 bit(s)	5 bit(s)	14 bit(s)
100101	0000	0001	000000000010
6 bit(s)	5 bit(s)	5 bit(s)	14 bit(s)

Figure 4-36: Bits lay out for Jump if is Greater than

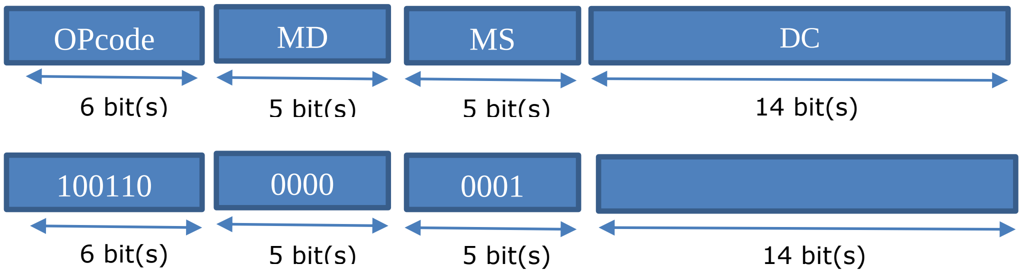
Copy real part of matrix

This instruction copies the real part vales of elements of a matrix and stores to Destination and the complex part is assumed to be zero.

M0=real (M1);

CREAL Destination Matrix, Source Matrix

CREAL M0, M1



OPcode	MD	MS	DC
6 bit(s)	5 bit(s)	5 bit(s)	14 bit(s)
100110	0000	0001	
6 bit(s)	5 bit(s)	5 bit(s)	14 bit(s)

Figure 4-37: Bits lay out Copy real part of matrix

Loading a value to a memory Location

This instruction loads a value to a memory location, this value can be binary or fixed point depends upon the usage.

LOD=5

LOD R0, 5

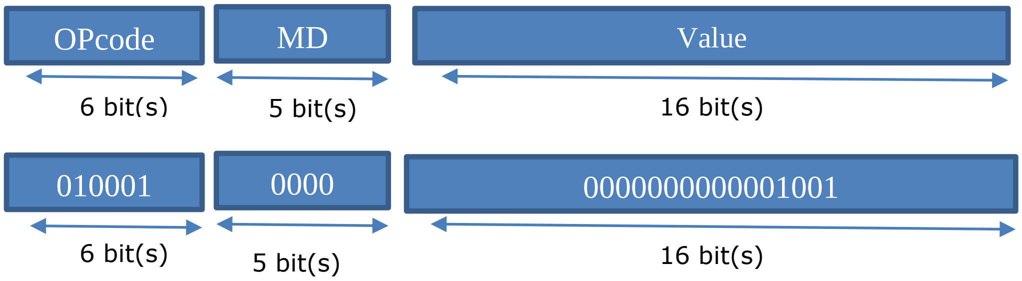


Figure 4-38: Bits lay out for loading a value to a memory Location

Loading negative a value to a memory Location

This instruction loads a negative value to a memory location, this value can be binary or fixed point depends upon the usage.

LODn = -5

LODn R0, -5

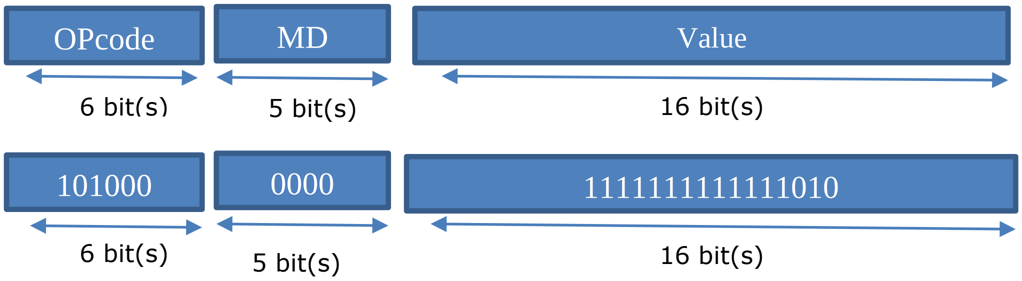


Figure 4-39: Bits lay out Loading negative a value to a memory Location

Copy a memory location to a new location

This instruction copies the real part vales of elements of a matrix and stores to Destination.

R0=R1

CPY Destination Matrix, Source Matrix

CPY R0, R1

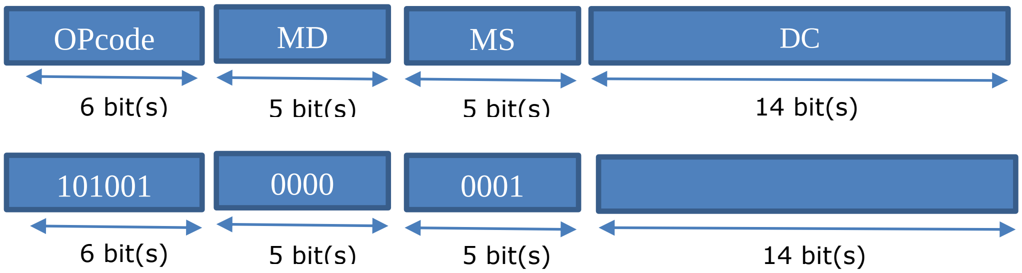


Figure 4-40: Bits lay out Copy a memory location to a new location

Copy a Matrix location to a new Matrix location

This instruction copies the real part vales of elements of a matrix and stores to Destination.

M0=M1

CPYm Destination Matrix, Source Matrix

CPYm M0, M1

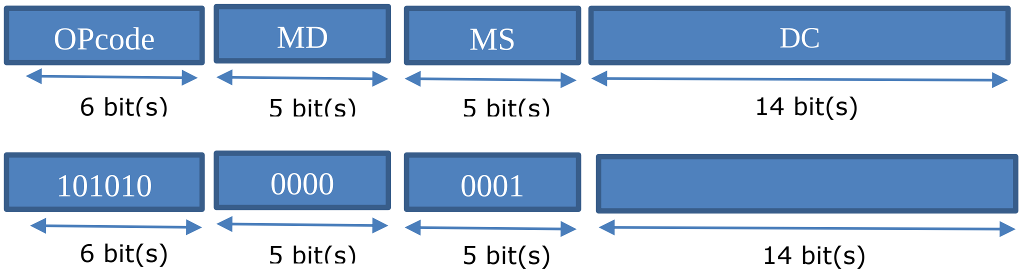


Figure 4-41: Bits lay out Copy a Matrix location to a new Matrix location

Copy a Column of a matrix

This instruction copies the real part vales of elements of a matrix and stores to Destination.

R0=1;

M0=M1 (: R0)

CPYc Destination Matrix, Source Matrix, Target Memory Location

CPYc M0, M1, R0

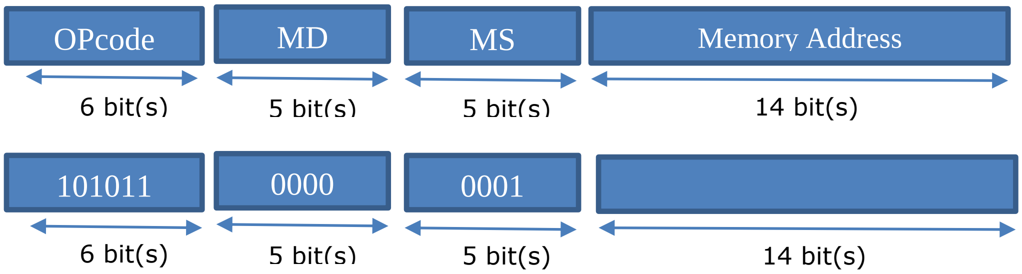


Figure 4-42: Bits lay out Copy a Column of a matrix

Load Memory content to special register

This instruction copies the memory content of specified location to a special register.

LSR Source

LSR R0

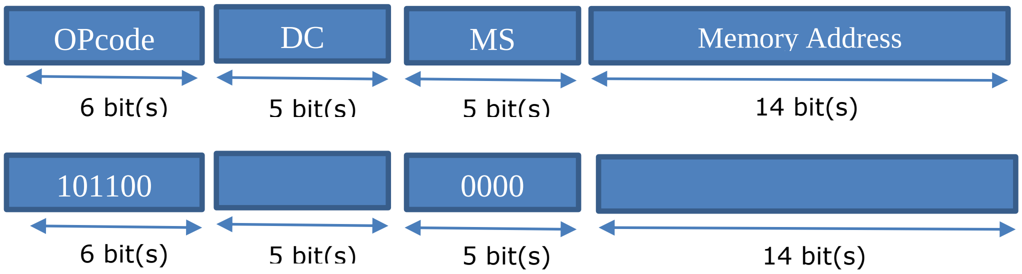
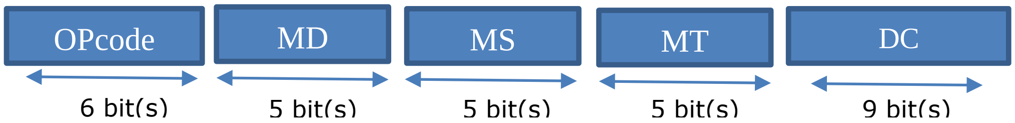


Figure 4-43: Bits lay out Load Memory content to special register

Write from special register

This instruction copies the content of special register to specified location in a matrix.

WFSR M0, R0, R1



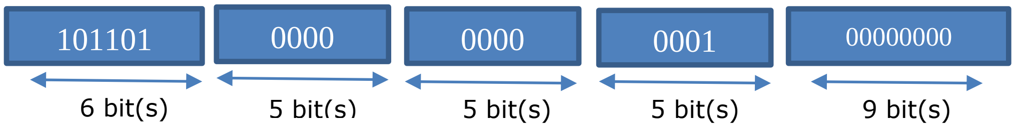


Figure 4-44: Bits lay out Write from special register

Load row and column Address

This instruction copies the row and column index stored in memory location R0, R1.

LRC R0, R1

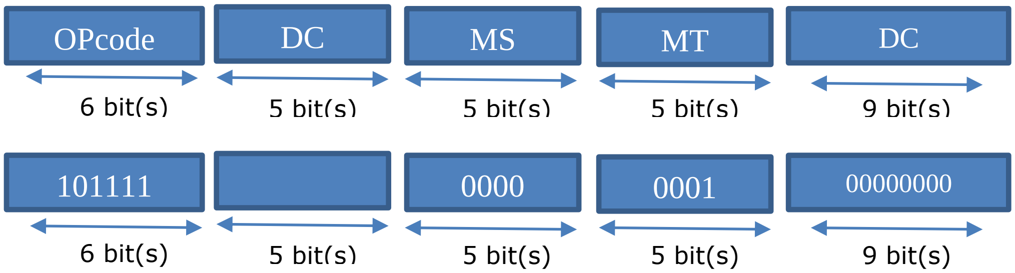


Figure 4-45: Bits lay out Load row and column Address

BITREV R0, R1

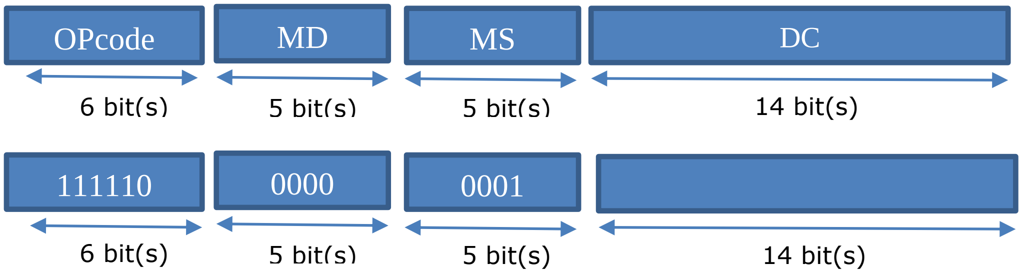
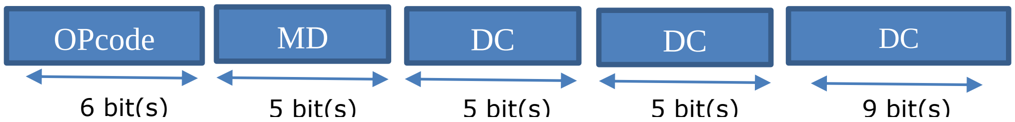


Figure 4-46: Instruction layout for bit reversal

Write to special register (row and column)

This instruction copies the content of special register to specified location in a matrix.

WTSR M0



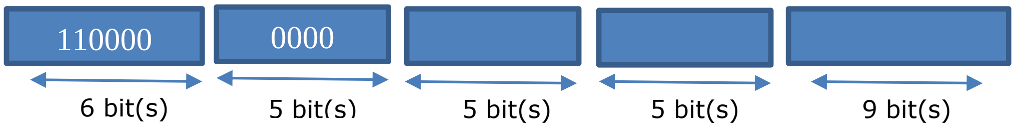


Figure 4-47: Bits lay out Write to special register (row and column)

Save from Special Register

This instruction copies the content of special register to specified location in a matrix.

SFSR R0

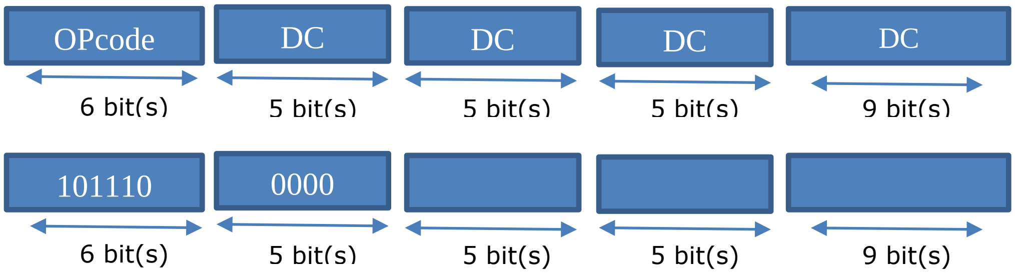


Figure 4-48: Bits lay out Save from Special Register

Initialize a new matrix

This matrix initializes a new matrix

IMAT M0, 5, 5

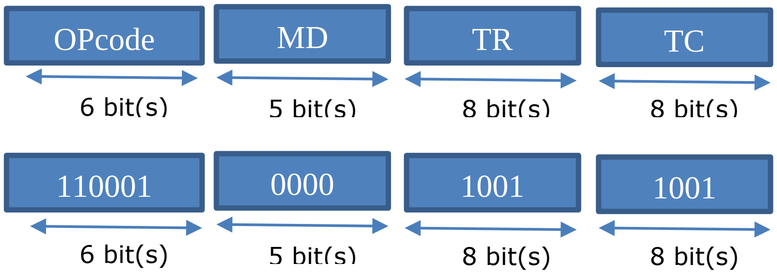


Figure 4-49: Bits lay out Initialize a new matrix

Reshape Matrix

Change matrix size, these instructions helps in changing the Attributes of a matrix, the roc and columns of a matrix can be changed according to need.

RSHP M0, 5, 5

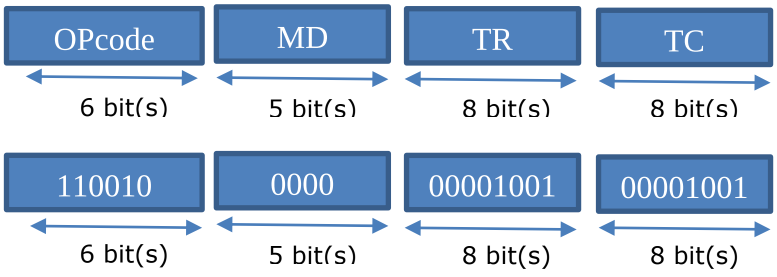


Figure 4-50: Bits lay out Reshape Matrix

Copy real part of matrix

This instruction bit reverses the contents of the specified memory location. This operation can only be performed on exact integer values used for matrix index manipulation.

CREAL M0, M2

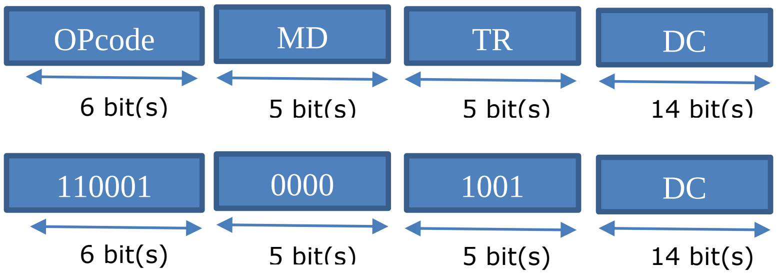


Figure 4-51: Bits lay out Initialize a new matrix

The following table presents the details of available instruction

Table 4-1 list of available instructions

Instruction	Syntax	Example	Machine Cycles	Description
MULm	MULm MD,MS,MT	MULm M0, M1, M2	NA x NB x MB	Multiplies two matrices
MULmd	MULmd MD,MS,MT	MULmd M0, M1, M2	N x M	Multiplies two matrices element by element.
MULmi	MULmi MD,MS,Value	MULmi M0, M1, 2	N x M	Matrix with Scalar multiplication with immediate Value
MULmv	MULmv MD, MS, R1	MULmv M0, M1, 10	N x M	Matrix with Scalar multiplication with stored Value
DIVm	DIVm MD, MS, MT	DIVm M0, M1, M2	N x M	Matrix division with Matrix element by element
DIVmv	DIVmv MD, MS, RT	DIVmv M0, M1,	N x M	Matrix dot Division with stored Value

		R1		
DIVmi	DIVmi MD, MS, Value	DIVmi M0, M1, 10	N x M	Matrix dot Division with Immediate Value
ADDm	ADDm MD, MS, MT	ADDm M0, M1, M2	N x M	Matrix Addition with Matrix
SUBm	SUBm MD, MS, MT	SUBm M0, M1, M2	N x M	Matrix Subtraction with Matrix
ADDmv	ADDmv MD, MS, RT	ADDmv M0, M1, 10	N x M	Matrix Addition with stored Value
SUBmv	SUBmv MD, MS, RT	SUBmv M0, M1, 10	N x M	Matrix Subtraction with stored Value
POWmv	POWmv MD, MS, RT	POWmv M0, M1, 10	N x M	Matrix dot Power with stored Value
SUBmi	SUBmi MD, MS, Value	SUBmi M0, M1, 2	N x M	Matrix Subtraction with immediate Value
ADDmi	ADDmi MD, MS, Value	ADDmi M0, M1, 2	N x M	Matrix Addition with immediate Value
POWmi	POWmi MD, MS, Value	POWmi M0, M1, 2	N x M	Matrix Raised to the Power of immediate Value
TP	TP MD, MS	TP M0, M1	N x M	Transpose of Matrix
SCONJ	SCONJ MD	SCONJ M0	1	Set Conjugate Flag
RCONJ	RCONJ MD	RCONJ M0	1	Re-Set Conjugate Flag
FTm	FTm MD, MS	FTm M0, M1	NA x NB x MB	Fourier Transform First Pass
FTmm	FTmm MD, MS	FTmm M0, M1	NA x NB x MB	Fourier Transform Second Pass
MUL	MUL RS, RT, RD	MUL R0, R1, R2	1	Scalar Multiplication
ADD	ADD RS, RT, RD	ADD R0, R1, R2	1	Scalar Addition
SUB	SUB RS, RT, RD	SUB R0, R1, R2	1	Scalar Subtraction
DIV	DIV RS, RT, RD	DIV R0, R1, R2	1	Scalar Division
POW	POW RS, RT, RD	POW R0, R1, R2	1	Scalar Power
ADDi	ADDi RS, RT,Value	ADDi R0, R1, 2	1	Scalar Addition with immediate Value
SUBi	SUBi RS, RT,Value	SUBi R0, R1, 2	1	Scalar subtraction with immediate Value
DIVi	DIVi RS, RT,Value	DIVi R0, R1, 2	1	Scalar Division with immediate Value
POWi	POWi RS, RT,Value	POWi R0, R1, 2	1	Scalar Power with immediate Value
JEQ	JEQ RS, RT, Address	JEQ R0, R1, 2	1	Jump if equal
JNE	JNE RS, RT, Address	JNE R0, R1, 2	1	Jump if not equal
JLE	JLE RS, RT, Address	JLE R0, R1, 2	1	Jump if is less than or equal
JGE	JGE RS, RT, Address	JGE R0, R1, 2	1	Jump if is Greater than or equal
JMP	JMP Address	JMP 2	1	Unconditional Jump
JLT	JLT RS, RT, Address	JLT R0, R1, 2	1	Jump if is less than
JGT	JGT RS, RT, Address	JGT R0, R1, 2	1	Jump if is Greater than
CREAL	CREAL MD, MS	CREAL M0, M1	1	Copy real part of matrix
LOD	LOD RD, Value	LOD R0, 5	1	Loading a value to a memory Location
LODn	LODn RD, Value	LODn R0, -5	1	Loading negative a value to a memory Location
CPY	CPY RD, RS	CPY R0, R1	1	Copy a memory location to a new location
CPYm	CPYm MD, MT	CPYm M0, M1	N x M	Copy a Matrix location to a new Matrix location

CPYc	CPYc MD, MS, Column	CPYc M0, M1, R0	M	Copy a Column of a matrix
LSR	LSR R0	LSR R0	1	Load Memory content to special register
WFSR		WFSR M0, R0, R1	1	Write from special register
LRC		LRC R0, R1	1	Load row and column Address
WTSR		WTSR M0	1	Write to special register (row and column)
SFSR		SFSR R0	1	Save from Special Register
IMAT		IMAT M0, 5, 5	1	Initialize a new matrix
RSHP		RSHP M0, 5, 5	1	Reshape Matrix
BITREV		BITREV R0, R1	1	Copy real part of matrix

Where MS is the source matrix, MT is the target matrix, MD is the destination matrix, RS is the source memory location, RT is target memory location, RD is the destination memory location, N is the number of rows, M is the total number of columns, NA is total rows of source matrix, NB is total rows of target matrix and MB is total numbers of column in matrix.

Chapter 5 Data paths and Design

5.1 Requirement Details

The design phase requires intense study of the operations to be done individually. The following flow graph explains the flow of the design of the processor. In the case of EMACH filter, the design phase of this filter requires some operations, all of these operations are quite different from each other .Like for a set of training samples first its Fourier Trans formed is required, Further there are some arithmetic operations that are to be done on the .In the first section of this chapter some brief details of the functional units required by the operations are discussed. The main aim to describe them individually is to grab the existence of common requirements and to use them only once to reduce the cost of the hardware.

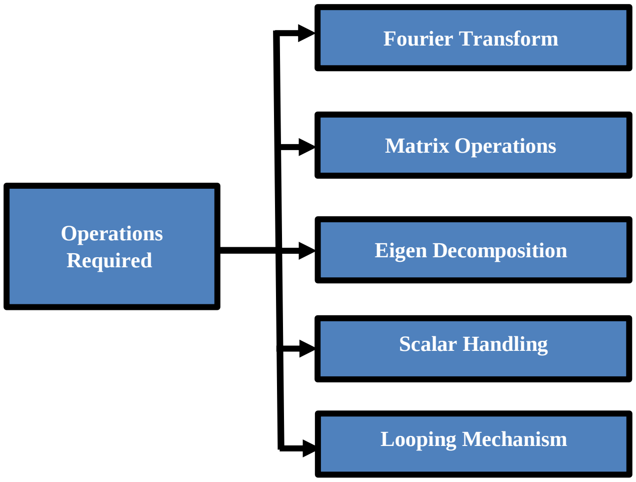


Figure 5-1 : The sub operations required by Major Operations

5.2 Fourier Transform

The design of some correlation filters are carried out in frequency domain, so this is the first basic requirement to convert the image to frequency domain, the major point is that the image is a two dimension matrix at this point a gray scale image is considered only .Different methodologies for computing Fourier Transform exists, some Flexible device manufactures are also providing state of the art prebuilt Fourier Transform Mechanism. These core provided by the manufacture can also be used if the Implementation is only restricted to the platform provided by that manufacture or similar platforms. The Fourier Transform method that details will be discussed in next section. The Operations required to complete Fourier transform are shown in the following figure

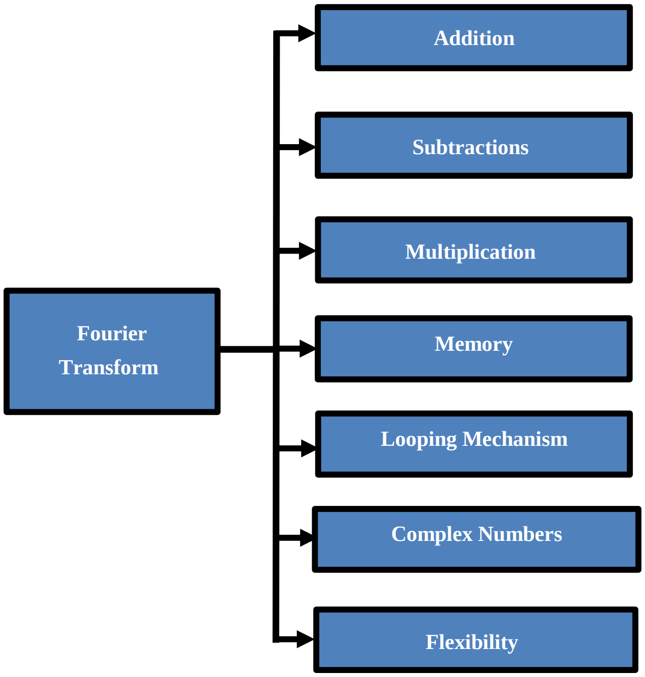


Figure 5-2 : Operations required to perform a Fourier transform

5.3 Basic Matrix Operations

Besides the above specified operations, the hardware should be able to handle matrices .in most of the processors the design is usually restricted to a fixed length size. The libraries provided by the manufacturer can be generated using a simple for, while the generation of the module it is a asked for what size the library should be generated, if a user selects say 64 points, the user will be restricted to 64k point only Although it is possible to compute the Fourier transform of a 32k point Fourier transform by padding zeros. In this case the aim is to produce flexibility on the hardware size .Matrix management provided with in the digital system that can provide more flexibility and the ease of use.

The basic Matrix operations include Arithmetic operations of matrix to matrix and matrix to scalar. Some of the Basic matrix operations are as shown below

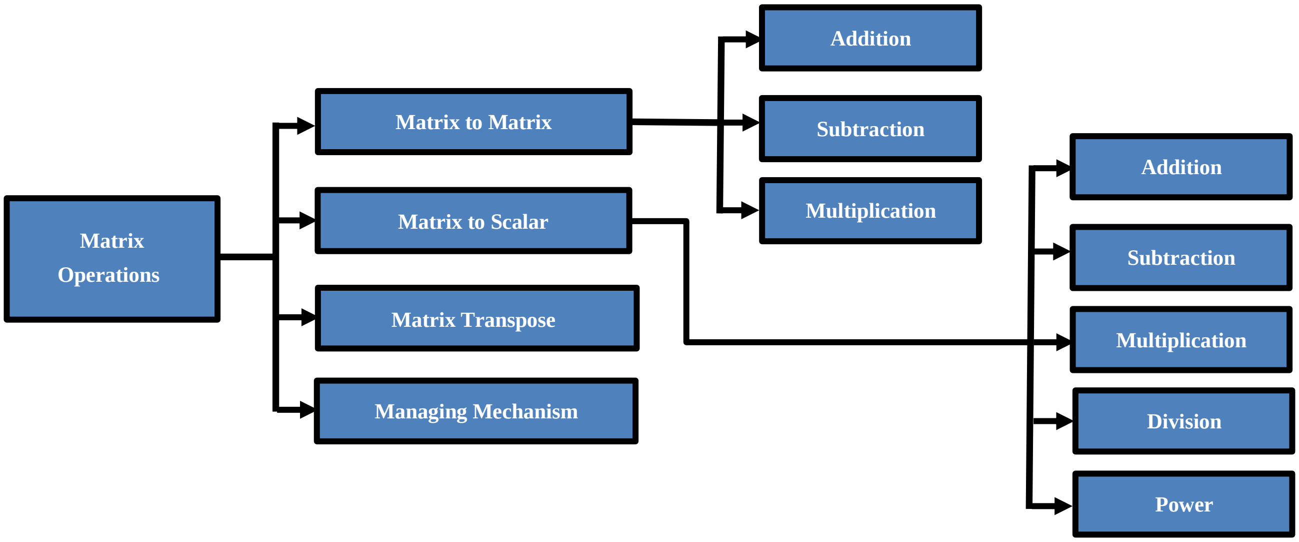


Figure 5-3 : Matrix Operations

5.4 Scalar Arithmetic

To provide a flexible control mechanism and for other requirements some scalar arithmetic functions are required.in a general loop of a program,

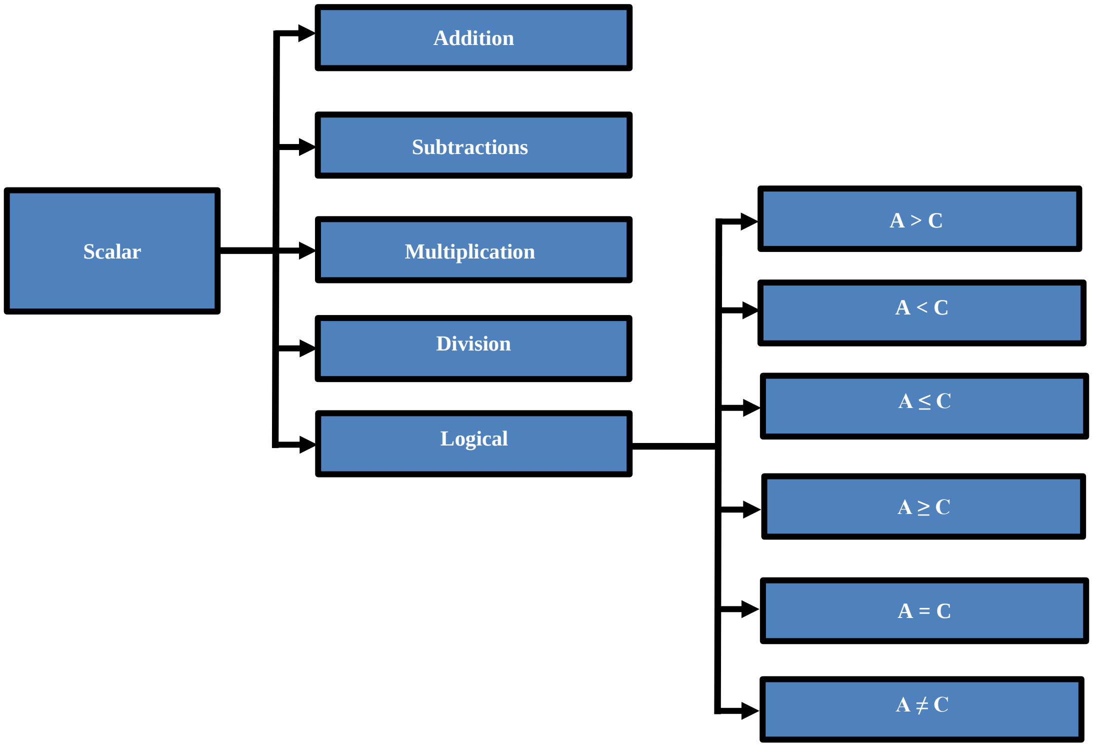


Figure 5-4 : Scalar Operations required

The requirements mention in the above section shows some the operations required to fulfill a correlation Pattern Classification.

5.5 Fourier Transform Design

The second chapter describes the methodology to compute the DTFT, Matrix multiplication can be employed to compute the Fourier transform of a given matrix, as described a general matrix can be pre computed and can be further used to compute the DTFT by multiplying the matrix with the input vector the resultant will be the DTFT (Fourier Transform) of the input vector.

An image is represented as a matrix or a 2D vector, The DFT can also be computed by using the Matrix multiplication Method. The Design already requires Matrix Handling i.e. Matrix Addition, Multiplication, Subtraction. The purpose of employing Matrix multiplication for computing the DTFT is to save extra Hardware, in the next chapter a brief detail of pros and cons is provided. A MATLAB Simulation for Generation for the DTFT Transformation Matrix and its use is shown below; the code Snippet basically computes the transformation matrix for the size of the vector (a random Provided Vector).Afterwards the provided input vector is then multiplied with the input vector, the product of these two will be the DTFT. The matrix Multiplication method will be used to compute the DFT of vectors with the number of elements can be represented in the exact power of two $N = 2^n$. For number of elements others zero padding will be required. A transformation matrix with other Number of points can also be computed and detailed version can be found at [27].

```
% Simulation: DFT of vector using Matrix Multiplication
% Author Abdullah Aman Khan

Vector=rand(1,8);

N=size(Vector,2);% The total Number Of point in the vector.

% Generating an N X N DFT Transformation Matrix
for x=0:N-1
    for y=0:N-1
            ft(x+1,y+1)=exp((-2*pi*i*x*y)/N);
            %Ft is the DFT Transformation Matrix
    end
end

%Computing DFT with Matrix Multiplication
DFT= Vector * ft

%MATLAB DTFT Results
DFT_MATLAB=fft(Vector)
```

For a two dimensional matrix of size N X N a transformation matrix will still be of the size N X N (for a square Matrix).After computing the Transformation Matrix the DTFT of the Two dimensional Matrix (Image) can be computed as described in above section. Another MATLAB code snippet that shows the simulation results of a 2D Matrix

(Image).The code snippet shows the simulation for computing a Transform Matrix and its usage. The Results of MATLAB prebuilt functions for computing DTFT is also computed along with the Transformation Matrix results for the purpose of comparison.

```
% Simulation: DFT of 2D Matrix using Matrix Multiplication
% Author Abdullah Aman Khan

% DTFT Matrix for Square Matrix

Vector=rand(8,8);
N=size(Vector,2);% The total Number Of point in the Matrix.

% Generating an N X N DFT Transformation Matrix
for x=0:N-1
    for y=0:N-1



        ft(x+1,y+1)=exp((-2*pi*i*x*y)/N);
        % Ft is the DFT transformation Matrix



    end
end
 %Computing DFT with Matrix Multiplication

DFT= ft * Vector * ft
%MATLAB DTFT Results
DFT MATLAB=fft2(Vector)
```

The attached MATLAB Code snippets show the computation of the DTFT for a vector and two dimensional matrix (image).Here a problem arises that the matrix is only computed for a size of N points, If it is required to compute DFT of points Greater than N, The transformation Matrix has to be computed again, the DFT computation is already compute intensive .Computing another Transformation Matrix for this new sized vector will require more Computation and storage in the memory.

There is a possibility that a general DFT Transformation matrix can be employed, to construct a generic Matrix, a mechanism is required to for truncation of values from a bigger pre-computed matrix. Suppose that the Generic Transformation Matrix is computed for a N X N .This matrix will be capable of computing the DFT of N Point Vector. Now a situation occurs that it is required to compute the DFT of M Points vector$Where\ M < N$ a simple truncation can be done in the Generic matrix already computed for N points. A new matrix can be produced form a larger generic matrix, the Truncation can be done from the first row till M^{th} row and from first column till M^{th} column.

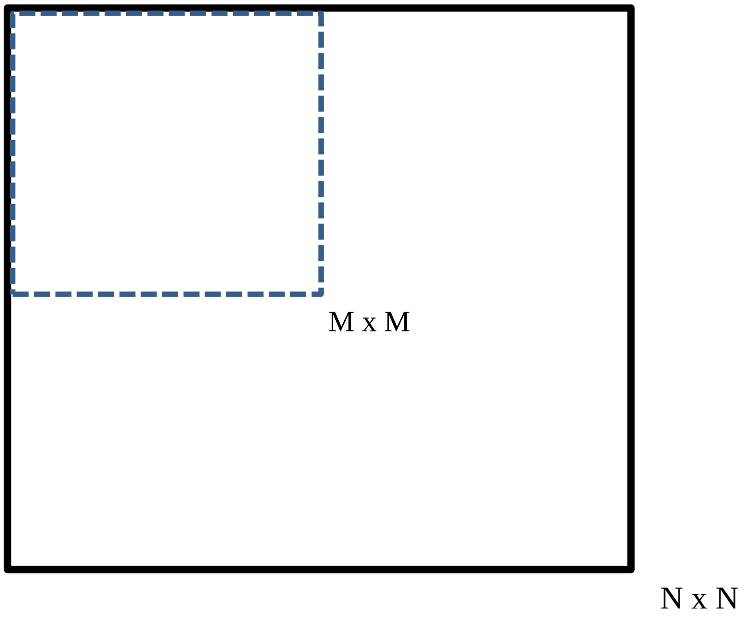


Figure 5-5 : Truncation of Values from Larger Generic Matrix for computation of DFT

In the above figure the bigger Bold Square represents the generic matrix, and the dotted square represents the smaller truncated new transformation Matrix. The purpose of using this mechanism is to provide a flexible stored table DTFT calculation Matrix .As described in section chapter 1, $\omega = e^{-\frac{2\pi i}{N}}$ was stored in the transformation Matrix, Here we can clearly see that it hold the effect of N, The goal is to create a general DTFT matrix, that can be further truncated according to the required points. So to achieve this we have to remove the effect of N in the equation. Instead of storing the Value $\omega = e^{-\frac{2\pi i}{N}}$ we can simply store the value $-2\pi i$ after Truncation the new matrix can be divided by the total Number of new point M and afterwards taking the exponential. This will yield a new Transformation Matrix.

$$W = \omega^{jk} \qquad (5.1)$$

$$\omega = e^{-2\pi i}/_{N} \qquad (5.2)$$

$$\log(\omega) = \log(e^{-2\pi i}/_{N}) \quad (5.3)$$

$$\log(\omega) = \frac{-2\pi i}{N} \qquad (5.4)$$

$$\mathrm{N}\log(\omega) = -2\pi i \qquad (5.5)$$

After Truncation

$$\frac{N\ log(\omega)}{N} = \frac{-2\pi i}{N} \qquad (5.6)$$

$$N\ log(\omega) = \frac{-2\pi i}{N} \qquad (5.7)$$

Taking Exponential of above equation the expression reverts back.

$$\omega = e^{\frac{-2\pi i}{N}} \qquad (5.8)$$

```
% Simulation: DFT of vector using Matrix Multiplication
% Author Abdullah Aman Khan

%Generating a general DFT Matrix that can be cropped according to a new
%General Matrix row and columns

Gr=256;
Gc=256;

% DTFT Matrix
for x=0:Gr-1
    for y=0:Gc-1


        ft(x+1,y+1)=(-2*pi*i*x*y);
        % The table basically stores -j2pi =N ln (dftmatrix)


    end
end

a=rand(1,4)
[r c]=size(a);
% truncating generic matrix

mat=exp(ft(1:c,1:c)/c);

% Calculating DFT
 DFT=a*mat
 DFT=fft(a)
```

There can be many different possibilities for calculating the Transformation Matrix, the first option is to store the transformation Matrix like a stored table implementation of DFT, The second method is to calculate the transformation Matrix for a specified size of input vector, The third Method can Employee the truncation Method from a generic Method and the fourth Method can me of computing the DFT using the trivial addition method as described in the Fourier transform in the first chapter.

The Generation of a new Transformation Matrix can save computation and also speed up the procedure, For example the A transformation Matrix Generated for computation of 256 Points , This matrix can also be employed to calculate the DTFT of 128 Points Matrix by zero Padding, The DTFT was to be calculated 0f 128 point But after zero Padding the Vector will be resized to 256 Points thus the calculations required for the 128 Point vector will be exactly equal to the calculations required for a 256 Point Vector.

5.6 Representation of a complex Number

The correlation filters are usually designed in frequency domain and the calculations are mostly in the frequency domain too. Fourier Transform requires a complex number to express the frequency components. Thus it is necessary to express every number as a complex number r, Representing every number as a complex number will require very large amount of memory, for every N bit Number an M bit Imaginary part is included with the real Number . The Implementation is kept flexible in the descriptive language, the size of the Binary Fixed point can be changed at compile time i.e. the Number of bits of the Fractional Part and the Number of bits of the Whole Part of the number.

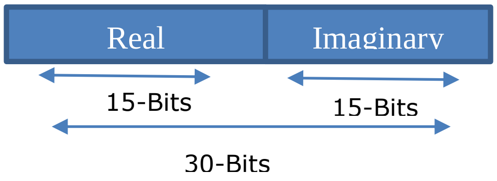


Figure 5-6 : Representation of a complex Number (Memory Location View)

Each Location in the memory will have two parts, the real part and the imaginary part. The ALU will deal with the Real and Imaginary Parts separately as the arithmetic operations are different for Complex Numbers as compared to Real Numbers. The real and imaginary parts are them self-Fixed point notations.

The following Diagram explains the representation of the Memory that actually stores K number of bits in the memory, The memory is unaware of what type of data is stored in the memory, the real and imaginary part of the memory are fixed point values of M bits. These M bits are considers as scalar Binary Values for scalar operations and Fixed Point where there is a need to represent fractional values. The OPcode differentiates whether to handle the number as complex, real, fixed point or as binary Number.

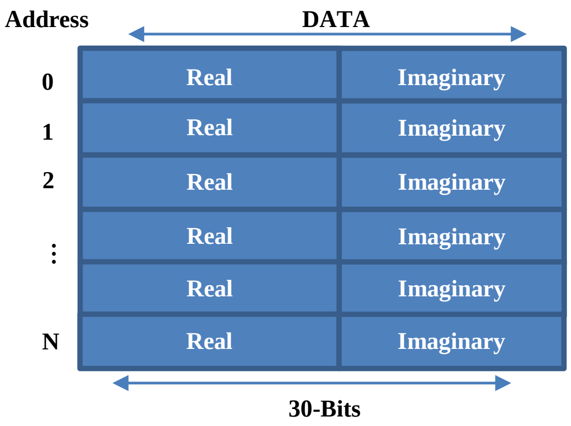


Figure 5-7 : Representation of a complex Number (Memory Location View)

5.7 Data Paths for Individual instruction Types

A detailed description of the data path for the ISA is presented in this section,

5.8 The Program Counter

For every processor which has an instruction memory requires a mechanism to fetch the next instruction from the Instruction memory. Usually a special Register known as

Program Counter [28] is employed to hold the address of the next instruction. In a normal program flow the starting address ‘0’ is loaded on the program counter register and on every clock cycle available to the register the instruction is incremented by one .this schemes employs an adder to fulfill this increment. The current address is incremented by the adder and is available to the in putt of the program counter register which will be written on the next available clock cycle. In some cases Branching might be required, the branch instruction directly holds the branch address, which is available on the Multiplexer available, on logical operations for branching, and the ALU raises a flag which actually works as the selection line for the multiplexer placed it the instruction fetching mechanism. When the ALU Flag is set to high, the Multiplexer will select the forced input (branch Address) to the Micro Instruction memory, and the next address relative to the branch address will be written on the Program Counter Register.

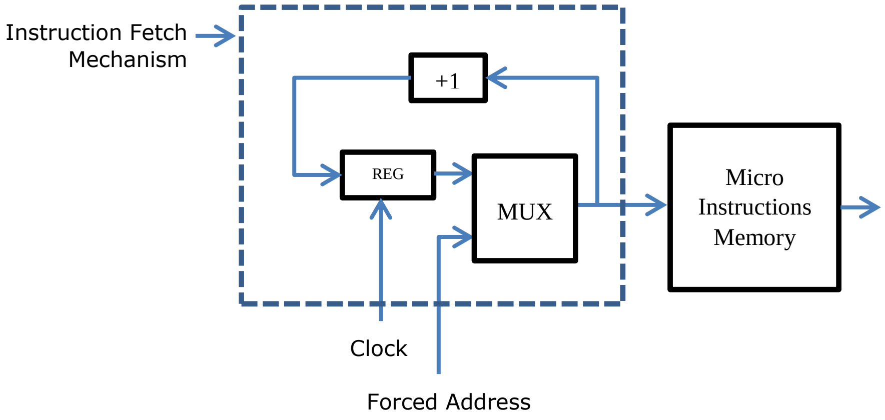


Figure 5-8 : The micro instruction Fetch Mechanism along the instruction Memory

The dotted area shows the components and working of the next instruction fetching scheme, for simple processors this scheme is very commonly used, in the next sections the Instruction Fetching mechanism will be represented by dotted line which indicates the same organization.

5.9 On chip Memory Manager

The proposed method comprises of an on chip memory manager the major benefit using this unit is that it will provide a very large amount of flexibility as explained in the introduction section the biggest challenge in implementing High level programming techniques to the hardware. Designing fixed size hardware restricts the flexibility to a very large extent. Functionality of much bigger instruction can be implemented using small scalar instructions, But Complex functions like Matrix handling (size, Addition, Subtraction, and Multiplication etc.) are provided on chip. To provide this functionality on chip a scheme shown in next figure is employed.

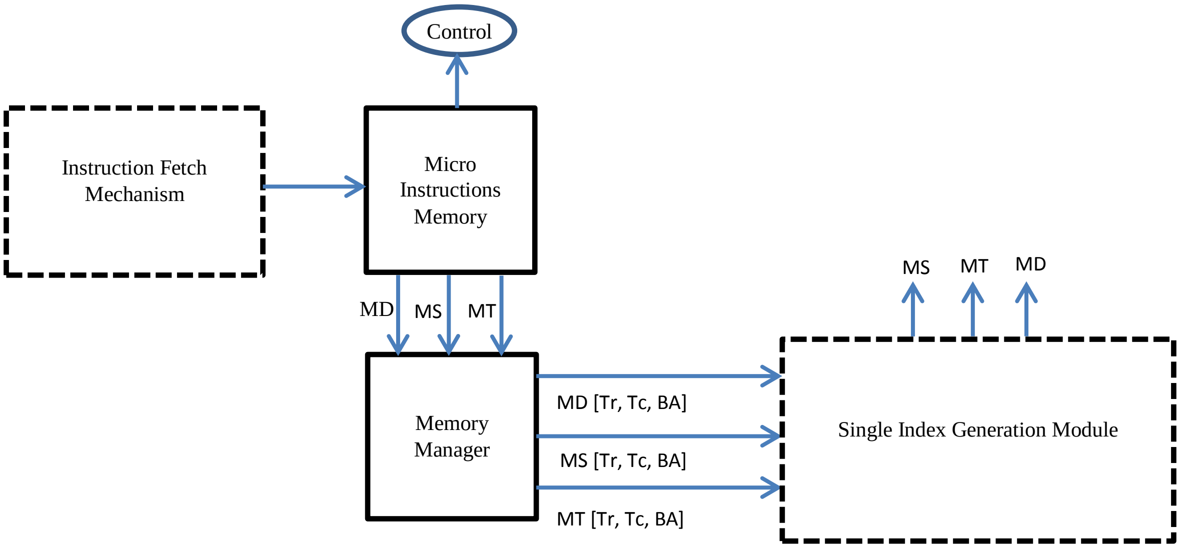

Control
Instruction Fetch Mechanism
Micro Instructions Memory
MD
MS
MT
Memory Manager
MD [Tr, Tc, BA]
MS [Tr, Tc, BA]
MT [Tr, Tc, BA]
Single Index Generation Module
MS
MT
MD

Figure 5-9 : Data path of the Memory Manager

The instruction Fetching mechanism will fetch the next instruction, The instruction holds basically the address of the matrix, suppose we can store 10, 15 matrix on our Memory then the memory manager will hold the information for the size and base address of the specified matrix. This information about the total rows and columns of a matrix can also be manipulated afterwards to save matrix locations. The information of the required corresponding Matrices is then transfer to the next section, which will compute Sub index to a single index form.

5.10 Sub-Index to single index Generation

As described in the earlier that the counters can generate (X, Y, Z) indices used in a loop of a high level language. The counter will be provided with an end value, which will determine whether where to stop. The counter is initialized with a value one which is the default value of the program. This is done to match the index assignment like MATLAB, the matrix sub-indices start from 1 instead of 0 unlike other programming language.

The control unit sends a control signals to the multiplexers, which helps in selecting the right end address for the counter. The major purpose of using these multiplexers are because the nature of operation are different, Addition and Subtraction may have different Index Generation as compared to matrix multiplication, thus multiplexer are used here to provide multi functionality. For operations like matrix Addition, Transpose and Subtraction only two counters can play the trick, but to provide on chip matrix multiplication a third counter has to be employed for the third index generation as explained in the next section. The counters are provided with the end address, on every clock cycle the counter starts moving towards its end point, each time the counter is incremented by one, as the identifier index of a loop is incremented on every successful completion. Another set of multiplexers are provided with the outputs of the counters, these multiplexers are then used to create a valid Sub Index Address if the matrix location according to the operation in progress.

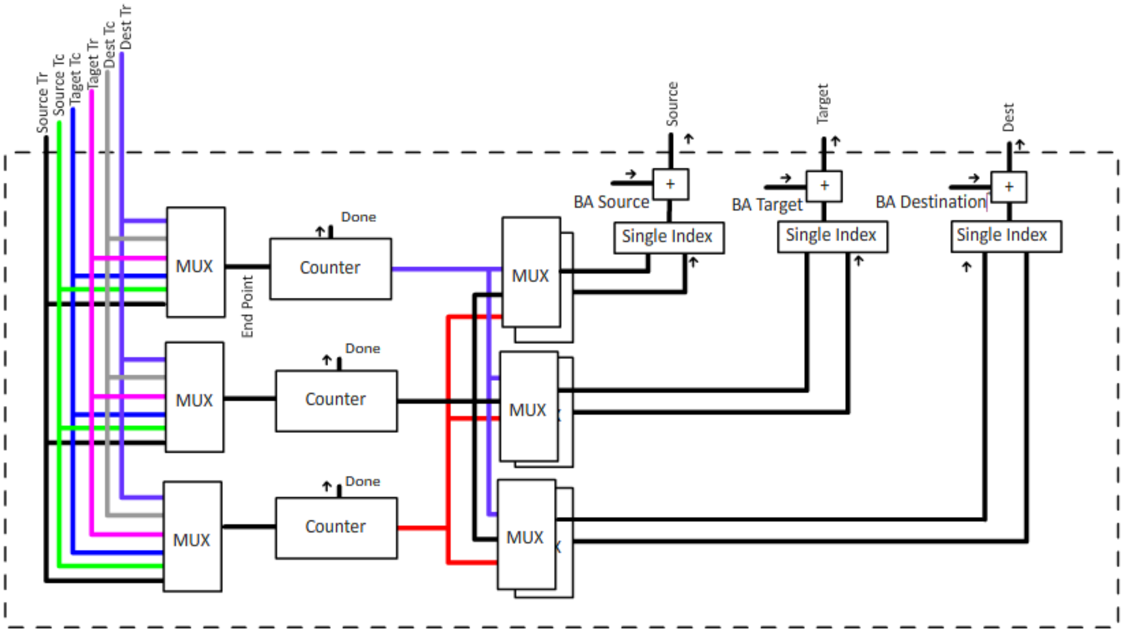


Figure 5-10 : Index Generation Data path

The single index basically converts the Sub index to a single index as explained earlier in the Sub to Index section in the previous chapters. This module will convert the subscript to a single index, then for the specific matrix the single index is added to the base address of the matrix, the resultant will be the exact address of the element in the main memory. The major benefit of providing an on chip index conversion will provide flexibility and the ease of use to the programmer, although this mechanism will cost more in terms of hardware .But will increase the functionally to a very large extent.

5.11 Counter and Program Counter clock Selection:

Providing the on chip looping scheme is quite challenging, each counter has to be triggered on the right time. The timing issues can be resolved by using an intelligent control methodology, on each instruction depending on the type of instruction the control panel will decide that which signal should be the clock of the counter, which will trigger the next value an intelligent switching network is embedded which determine the triggering pulse to a specific counter. Suppose in case of Matrix Addition the Sub indexes are the same for the operands and the destination and can be done in two loops using a high level programming language, the end signal of the Y index counter can trigger the next counter to go to the next state.

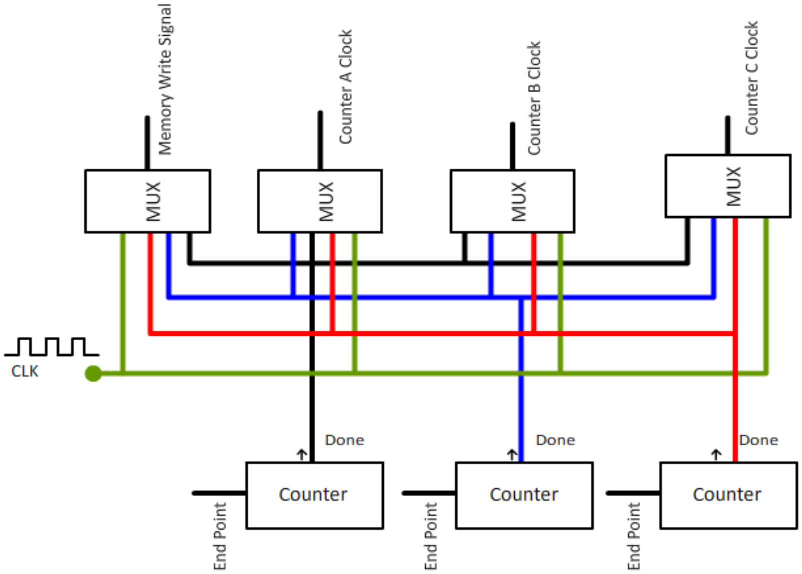


Figure 5-11 : Clock feeding to Counters

5.12 The Matrix Addition/Subtraction data path

Sub script index to the single index conversions makes it easier and flexible to work with variable sized matrices. As shown in the data path diagram below, a normal flow of instruction will continue. The Instruction will be fetched in a normal manner; the OPcode of the current instruction will determine that is a matrix operation like addition or subtraction etc. The instruction will also hold the address of source, target and destination addresses of the matrices. These addresses are provided to the memory manager which will yield out the required information of matrix to the single index generation module. This module will then derive exact address of the elements of the source, target and Destination matrices. And the data of the operands will be provided to the Arithmetic and logic unit (ALU) which will further work on the provided data either to add, subtract the data etc.

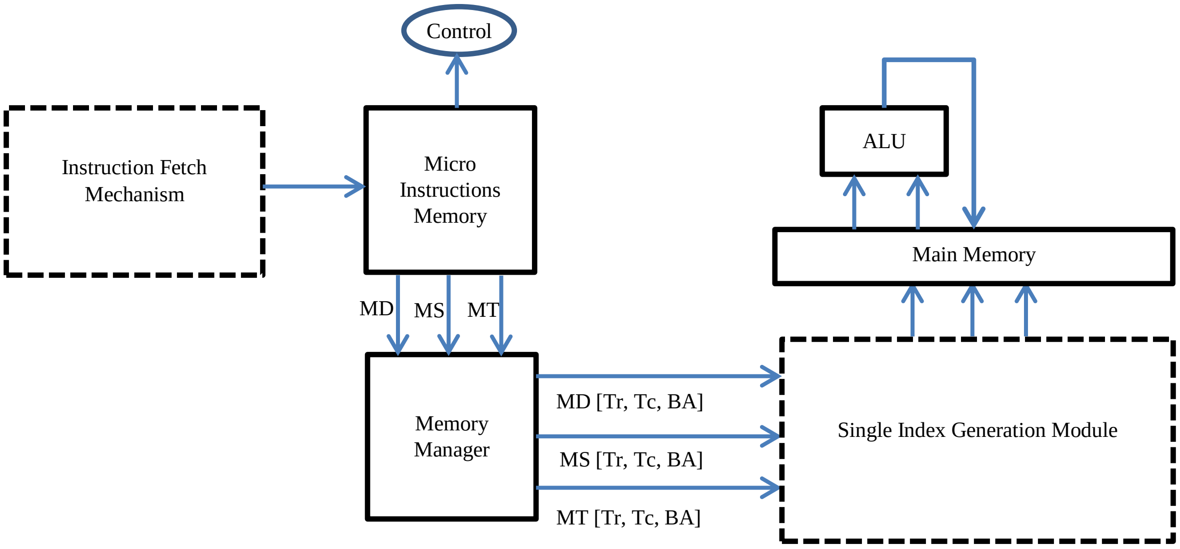


Figure 5-12 : Data Path for Matrix Addition/ Subtraction

Matrix addition and subtraction is quite straight forward, only two indexes can represent the Source, target and destination. The following figure shows that only two counters can be employed to generate matrix indices. Usually Matrix addition and subtraction can be expressed as $Dest(X,Y) = Source(X,Y) \pm Target(X,Y)$. It is very clear that only two identifiers can be used to traverse the whole matrices and luckily the increment in the same order, whereas index generation for Matrix multiplication is quite complex as compared to matrix Addition and Subtraction.

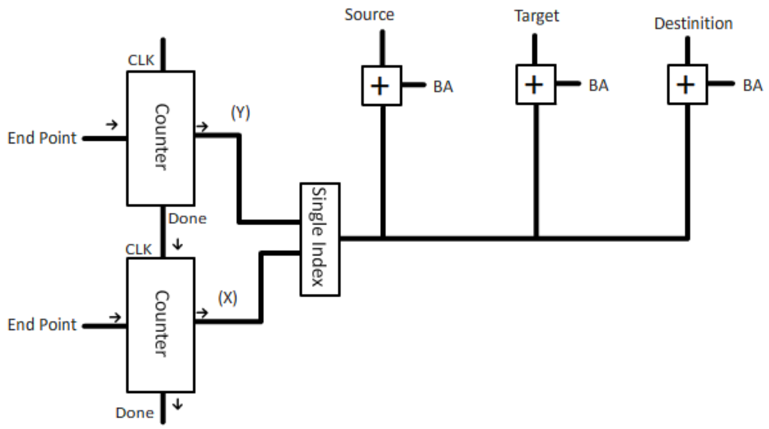


Figure 5-13 : Index Generation for Matrix Addition / Subtraction

5.13 The Matrix Multiplication

Matrix Multiplication is not straight forward as Matrix addition and subtraction. A regular schoolbook Multiplication algorithm requires at least three loops. In other works it requires three identifiers i.e. X, Y, K.The Matrix Multiplication for a single resulting element can be expressed as

$$C_{(x,y)} = \sum_{x=1}^{y} A_{(x,y)}.B_{(y,x)} \qquad (5.9)$$

these three identifiers requires three counters to implement the full matrix multiplication, that's why the single index generation module is designed with three counters, each counters contains a register with an Adder .

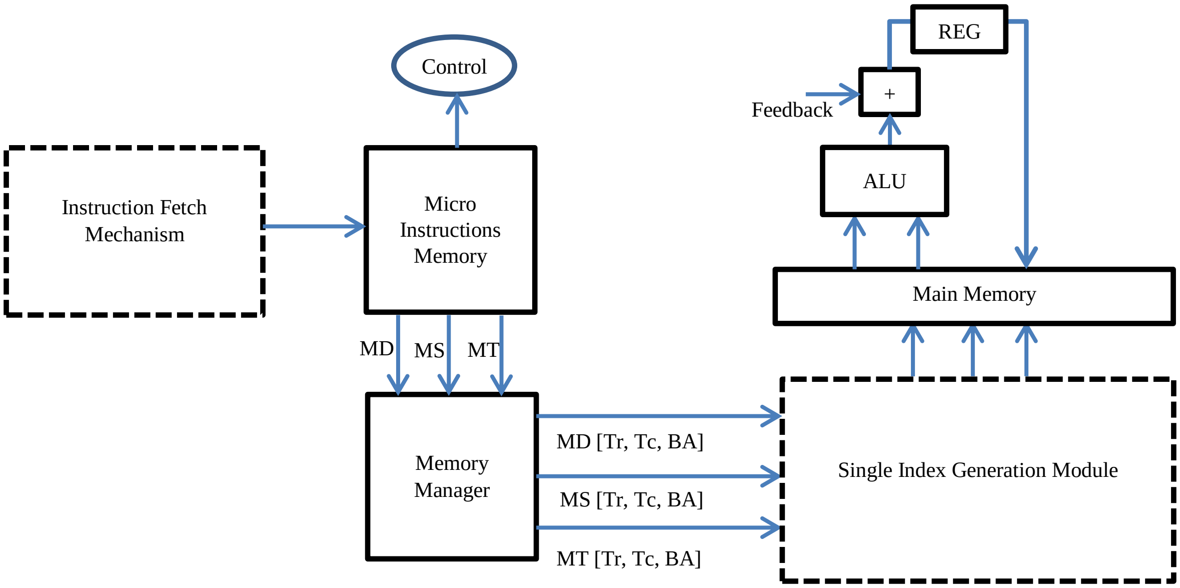


Figure 5-14 : Index Generation for Matrix Multiplication

The control unit will dispatch the control signal according to the iterations required for multiplying a matrix multiplication. The first counter will be fed with a system clock; the second counters clock in put will be triggered by the End flag of the first counter and the third counter will be fed by the end flag of the second counter .In this way the sub index required for the matrix multiplications will be generated and then further element by element the index will be accessed from the main memory, thus capable of providing a full matrix multiplication. The single index generator will generate the exact address of the contents. To sum up the product of two elements another adder is employed to save machine cycles, this adder along with an adder keeps on adding the result of each multiplications like $\mathrm{X} = \mathrm{X} + (\mathrm{A}.\mathrm{B})$, where X is the value of the register and A and B are the elements of the source and target matrix respectively. This adder uses a feedback from the registers and keeps on updating results as required.

5.14 The Matrix Transpose

Transpose of a matrix is simply computed by reversing the Sub-indices i.e. $\mathrm{W}(\mathrm{X},\mathrm{Y}) = \mathrm{w}\ (\mathrm{Y},\mathrm{X})$ the methodology used is quite simple. For this purpose a pair of counters can be used to generate the indices. The diagram below shows that how a single pair of counter can be used to generate indices.

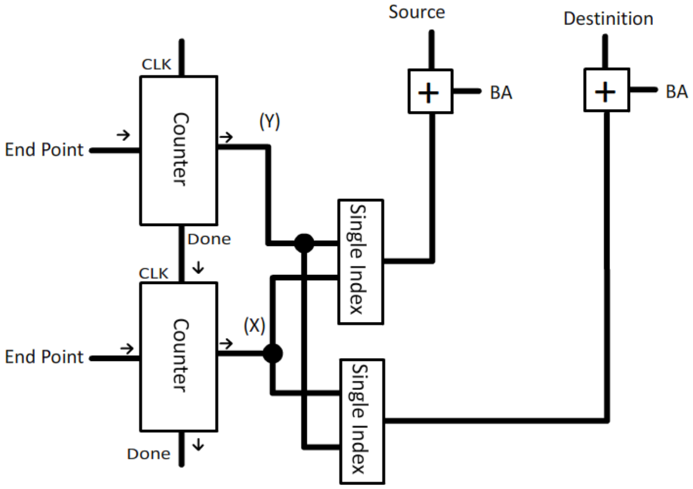


Figure 5-15 : Index Generation for Transpose of a Matrix

5.15 The Matrix Operations with a Scalar Value place in memory

In certain situation it is required to perform operations with a scalar value, this value is stored in the main memory. To get the data available on the location an address is required to access the value from the main memory. This address can be provided in the instruction .To multiply the matrix M0 with the value stored on the first location of the memory say (R0) with address 0.The location address can be specified in the instruction which is stored in the instruction memory, The following data path can perform Matrix to scalar addition, Matrix to Scalar Subtraction, Matrix to scalar Multiplication, Matrix to scalar Division, Matrix to scalar exponent .All these operation composes the following data path, the only difference occurs at the ALU OPcode, the OPcode will let the ALU to know that what sort of arithmetic operation has to be performed i.e. Addition, subtraction, Multiplication, Division etc.

Again based on the OPcode the control panel will determine the control signals and the OPcode for the ALU operation. The single index generation will work as before, the output of counter A, B will be useful, The counter A clock input will be feed with the system clock, and the clock input of the second counter will be fed with End signal of the first counter (A). In this way the index can be generated. $Dest(X,Y) = Z \pm Target(X,Y)$ Where Z is single value (scalar) stored at some memory location. For example intensities of an image have to be doubled, and then the 2D matrix (Image) can be multiplied with a scalar factor i.e. 2.Although this not the exact method for contrast stretching, but multiplying the image with 2 will stretch the intensity level.

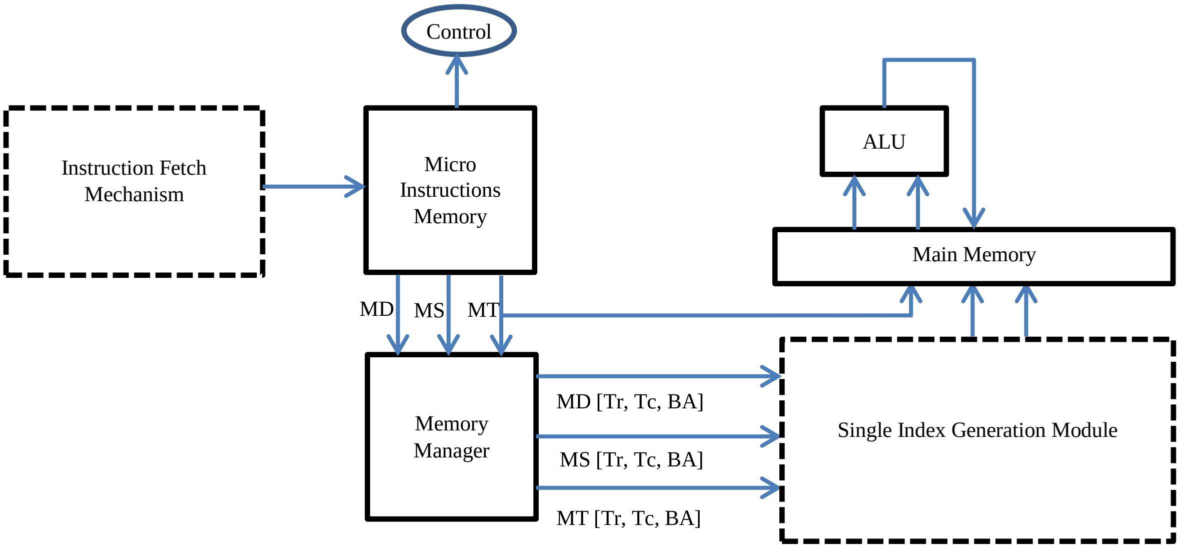


Figure 5-16 : Matrix with Scalar Operations

5.16 Matrix Operations with an Immediate Value

The data path is almost the same as of a stored Value operation with a matrix, the only difference is at the immediate value .which is defined in the instruction. The Target will be treated as the destination address. The index generation will be the same as for other matrix operations.

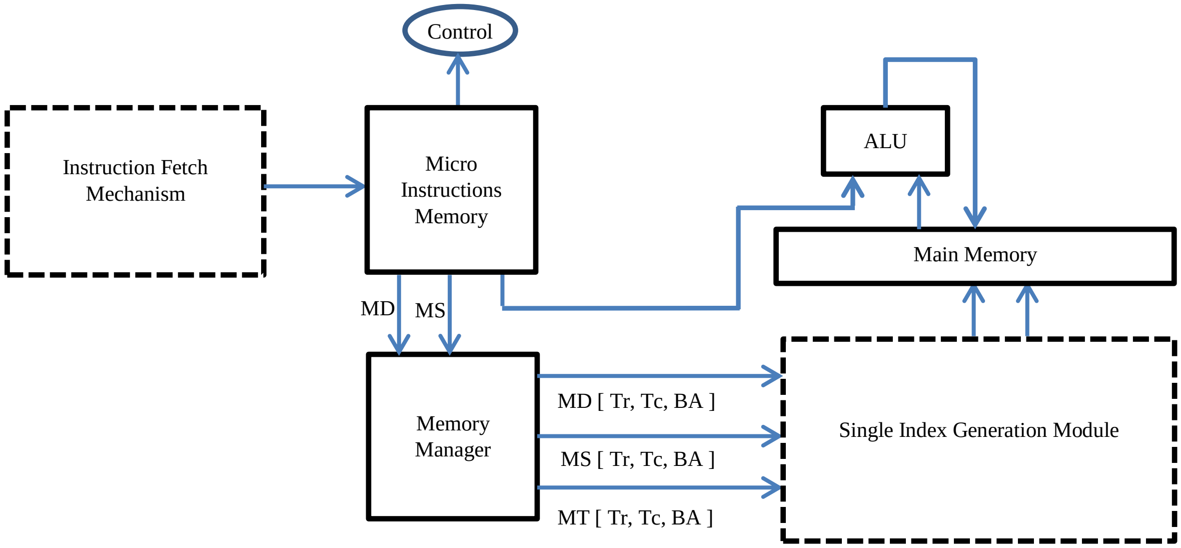


Figure 5-17 : Matrix with Scalar Operations

5.17 Scalar Operations (In Memory)

Besides Matrix operations, it is compulsory to provide scalar (single Value) operations. The scalar value handling does not require single index conversion. The direct memory address of the operand can be specified in the instruction. The first 32 locations of the memory are reserved for scalar operation, so that they can act as register but are actually part of the same common memory. A sign adjustment is required to access the memory location. One big challenge is to integrate the matrix and scalar computation on the same hardware by re utilizing the components.

The instruction fetch mechanism will fetch the instruction, now this instruction will contain the address of the memory location on which operations are to be performed, these operation can be logical or arithmetic. The design provides logical operations only to scalar values. The main purpose is to provide Branching operations based on these logical decisions and to find maximum and minimum values in the matrices.

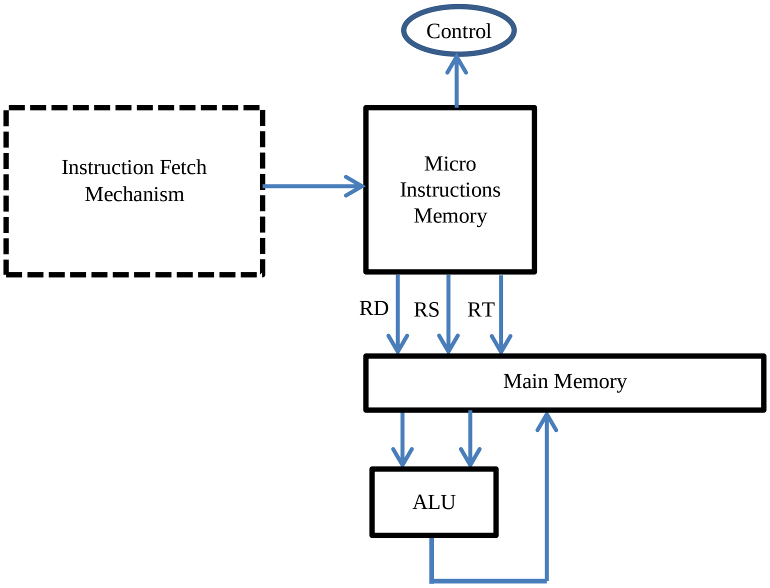


Figure 5-18 : Matrix with Scalar Operations

5.18 Scalar Operations Immediate

Scalar with immediate value operations works almost same as the scalar operations, only the target address is extracted from the instruction.

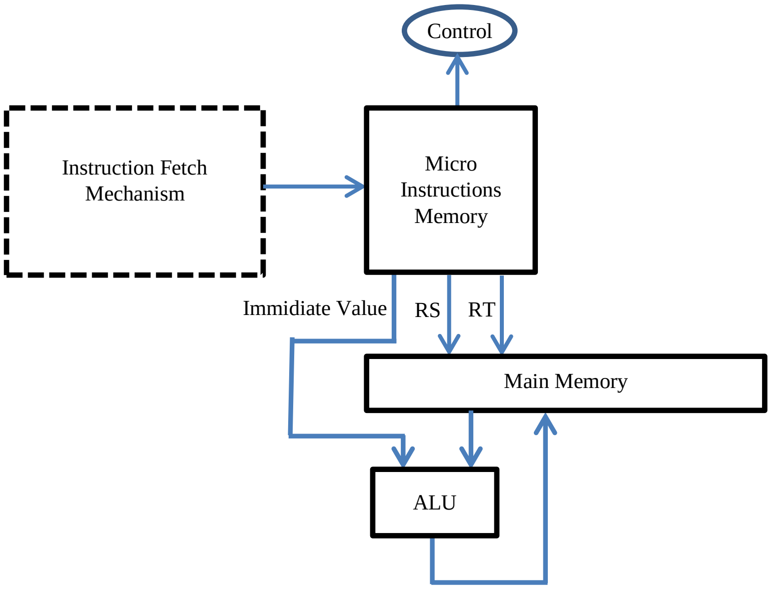


Figure 5-19 : Matrix with Scalar Operations

5.19 Jumps and Branching Instructions

The branching instruction transfers control to the specified line number of the instruction memory. The Jump address is stored in the instruction along with two operands address stored. The ALU will set the flag to high if the condition is true (For conditional Jump).For unconditional Jump the ALU will set the flag.

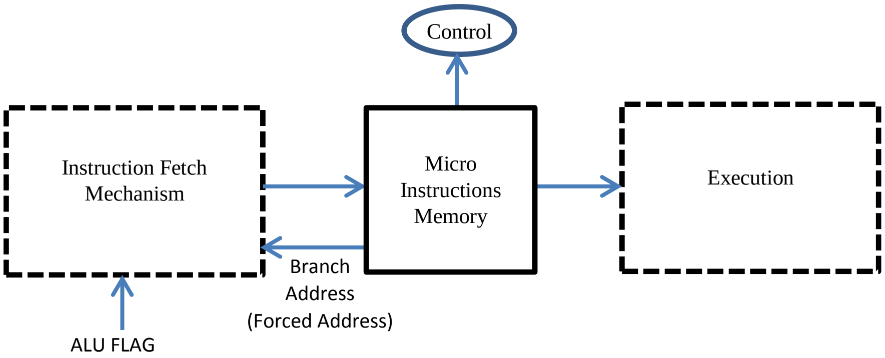

Control
Instruction Fetch Mechanism
Micro Instructions Memory
Execution
Branch Address (Forced Address)
ALU FLAG

Figure 5-20 : Branching and looping

5.20 Variable Clock Cycle Implementation along with single cycle

The processor implements complex and simple instructions to gather, the instructions for scalar values requires a single cycle for one instruction, whereas the handling for matrices requires multi cycle for completion on one instruction.

The processor is performing operations like Instruction Fetch, Instruction Decode, Instruction Execute and Write Back in the same common clock cycle. At the first step the instruction Fetch Mechanism fetches the instruction from the instruction memory. After wards it is time to decode the instruction, that what type of instruction it is and how many operands it has to tackle with, this stage also identifies the operands which can be scalars or matrix in this case. Operations like Addition Subtraction multiplication is done in the execution stage. After the execution the result is written back to the memory. All of these stages are covered in a single cycle for a scalar operation. The next instruction is only fetch when the current instruction has completed.

The determination of length of one clock cycle can be determined by the following expression, where CCT is the Clock Cycle time and D is expressing some other latency like, data Arrival delay, clock skews adjustment etc.

$$(\mathbf{CCT}) = \sum \boldsymbol{All\ Delays\ of\ components} + \boldsymbol{D} \tag{5.10}$$

5.21 Main Architecture Design

After merging all the data paths together, the following design can perform all the above explained functionality in one single design.

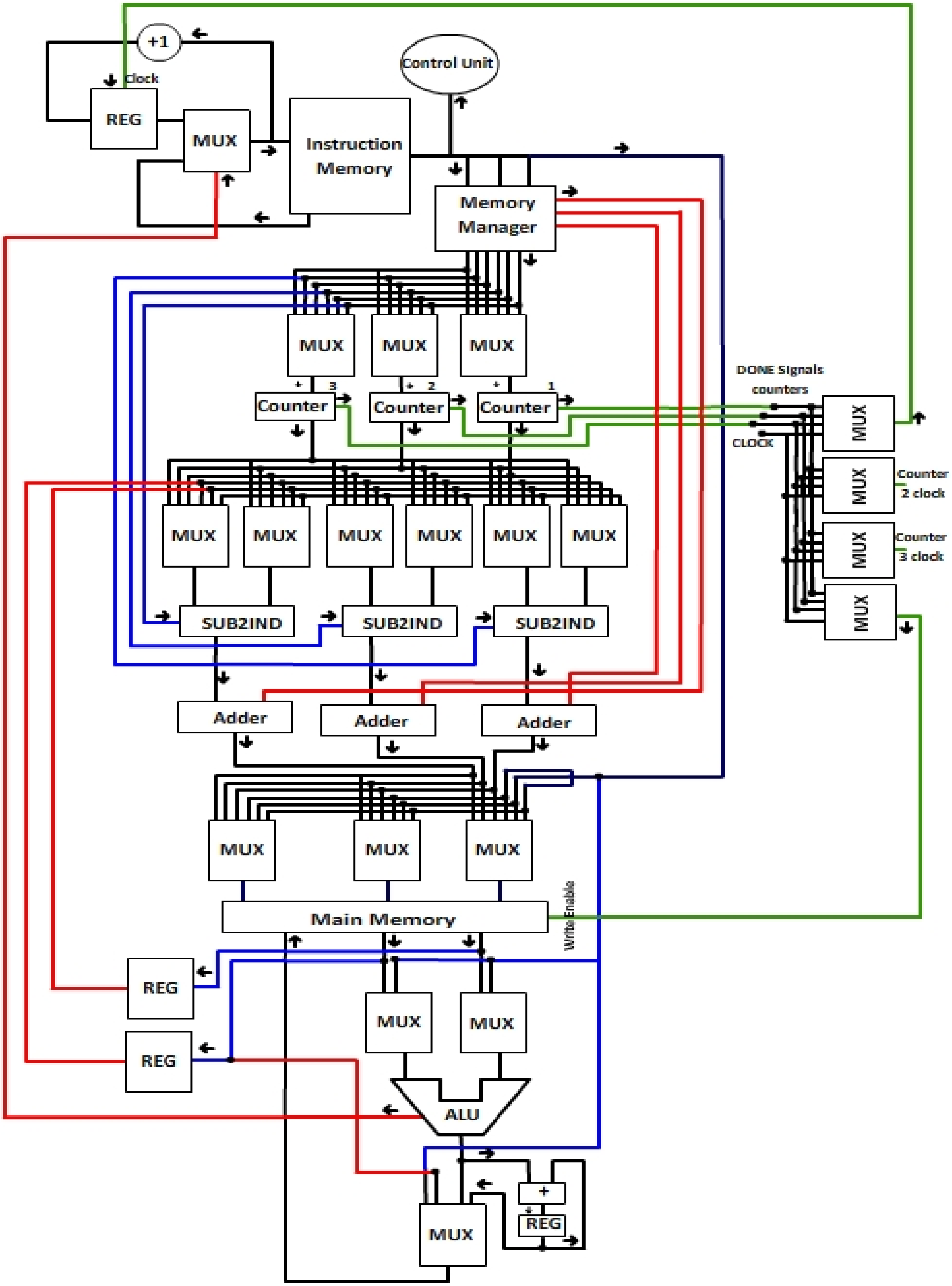
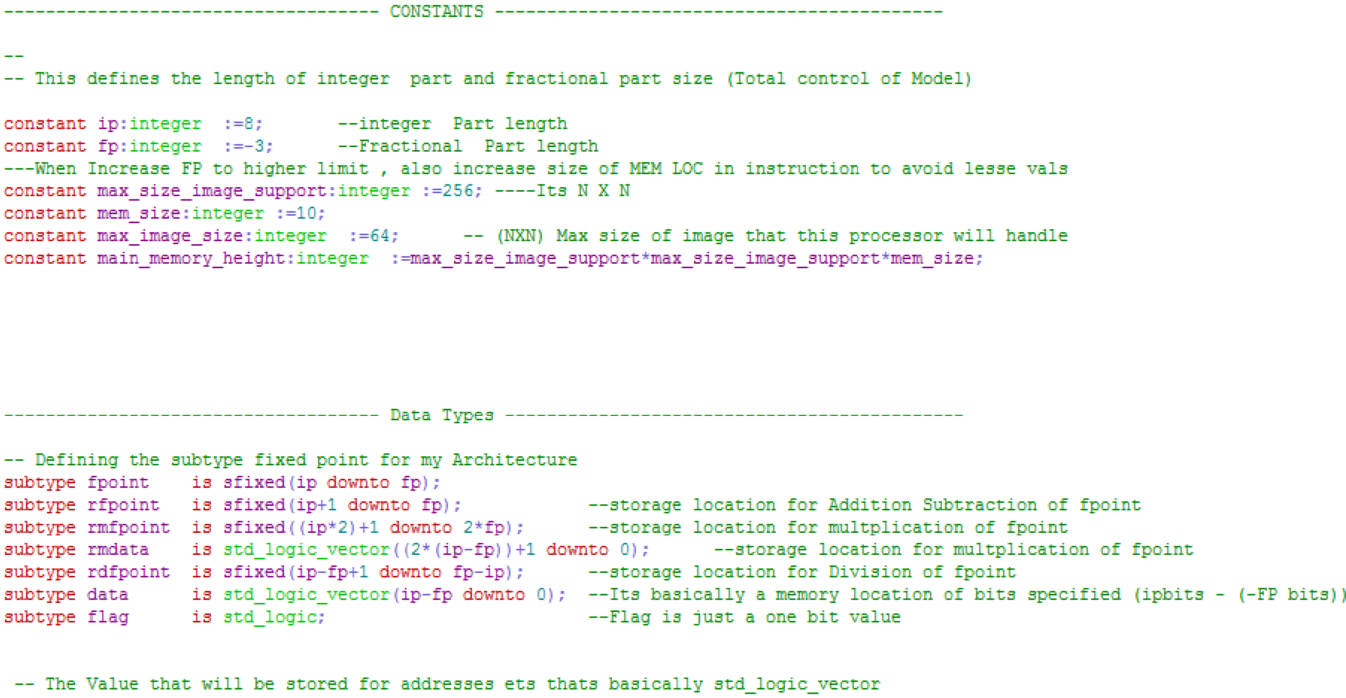


Figure 5-21 All Data Paths Integrated (main Architecture)

Chapter 6 Simulation Results

Calculating the exact value is a quite curtail task in digital design, the values if not represented in exact decimal places, it can arises a huge possibility that the reverse transformation can be corrupted. The design was simulated and tested using VHDL (Verilog Hardware Descriptive Language).

To see all the aspects of the increasing and decreasing a main a main configuration files holds the size of all the buses and Memory module to everything, By changing simple Values in the configuration file, the bit size of the modules can be changed so that it can produce a variety of regression texting for different number of Numbers (different size of bits and Fixed Point).A snapshot of the configuration file is presented below.



```
------------------------------------ CONSTANTS -------------------------------------------

--
-- This defines the length of integer  part and fractional part size (Total control of Model)

constant ip:integer  :=8;       --integer  Part length
constant fp:integer  :=-3;      --Fractional  Part length
---When Increase FP to higher limit , also increase size of MEM LOC in instruction to avoid lesse vals
constant max_size_image_support:integer :=256; ----Its N X N
constant mem_size:integer :=10;
constant max_image_size:integer  :=64;      -- (NXN) Max size of image that this processor will handle
constant main_memory_height:integer  :=max_size_image_support*max_size_image_support*mem_size;




------------------------------------ Data Types --------------------------------------------

-- Defining the subtype fixed point for my Architecture
subtype fpoint    is sfixed(ip downto fp);
subtype rfpoint   is sfixed(ip+1 downto fp);           --storage location for Addition Subtraction of fpoint
subtype rmfpoint  is sfixed((ip*2)+1 downto 2*fp);     --storage location for multplication of fpoint
subtype rmdata    is std_logic_vector((2*(ip-fp))+1 downto 0);      --storage location for multplication of fpoint
subtype rdfpoint  is sfixed(ip-fp+1 downto fp-ip);     --storage location for Division of fpoint
subtype data      is std_logic_vector(ip-fp downto 0); --Its basically a memory location of bits specified (ipbits - (-FP bits))
subtype flag      is std_logic;                        --Flag is just a one bit value


 -- The Value that will be stored for addresses ets thats basically std_logic_vector
```

Figure 6-1 : The configuration file snippet

Computing DFT

The design is capable of computing Fourier Transforms of variable sized of points; initially a matrix is stored into the memory in a linear format, The matrix **A** contain the elements as shown. All the values are in real numbers and have no imaginary part with them. A test case is taken of 3 x 3 matrix, as the design is flexible and can carter variable sized of matrices.

$$a = \begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{bmatrix}$$

The Fourier transform of this matrix can be calculated using the prescribed instruction, the simulation results in the form of wave are also provided in the wave diagrams below.

$$A = \begin{bmatrix} 45\ +\ \ 0\mathrm{i} & 4.5\ -\ \ 2.5981\mathrm{i} & 4.5\ +\ \ 2.5981\mathrm{i} \\ -13.5\ +\ \ 7.7942\mathrm{i} & 1.7764\mathrm{e}-15\ +\ 1.1102\mathrm{e}-15\mathrm{i} & 8.8818\mathrm{e}-16\ +\ 2.8866\mathrm{e}-15\mathrm{i} \\ -13.5\ -\ \ 7.7942\mathrm{i} & 8.8818\mathrm{e}-16\ +\ 2.2204\mathrm{e}-16\mathrm{i} & -2.2204\mathrm{e}-15\ +\ 1.1102\mathrm{e}-15\mathrm{i} \end{bmatrix}$$

Table 6-1 DFT Results comparison with MATLAB

Matrix SUB Index	Processor Results	MATLAB Results
(1,1)	45 + 0i	45 + 0i
(1,2)	4.5 - 2.5979i	4.5 - 2.5981i
(1,3)	4.5 + 2.5979i	4.5 + 2.5981i
(2,1)	-13.5+ 7.7937i	-13.5 + 7.7942i
(2,2)	0 + 0i	1.7764e-15 + 1.1102e-15i
(2,3)	0 + 0i	8.8818e-16 + 2.8866e-15i
(3,1)	-13.5 - 7.7937i	-13.5 - 7.7942i
(3,2)	0 + 0i	8.8818e-16 + 2.2204e-16i
(3,3)	0 + 0i	-2.2204e-15 + 1.1102e-15i

Following is the code snippet of the calculation of the Fourier transform in the processor, As described earlier for matrix operations a single instruction is completed in multiple clock cycles, the arrow numbered (3) shows the normal flow of clock, which can be considered as the main system clock, the arrow identifier (2) shows the write signal which is a single bit, when the sum of the multiplication of one row and the required column is ready the write Enable flag of the memory is raised, This enabled the memory to be written on the specified address. The identifier shows the results of a 2D DFT .The results of DFT are calculated in a row oriented manner by the processor. The Identifier (1) shows the first four results of the Fourier Transform calculated by the processor. Some difference is that the general computing environment works on floating point which can provide high precision as compared to fixed point. Difference is due to use of two different representation scheme, MATLAB is using floating point technique and the implemented processor is using fixed point, Fixed point cannot provide the exact amount of precision until the number of bits for the fractional part is increased.

Below is provide the snapshot of the wave form for the simulations, Only the inputs on the memory are shown, For testing purpose the real and the imaginary parts of the value are shown separately for the comparison, A non-synthesizable function is used to convert the fixed point binary value to a Real number also eases to constantly monitor the values at the input of the memory. In the next diagrams the results provided by the execution unit to the input of the memory are shown.

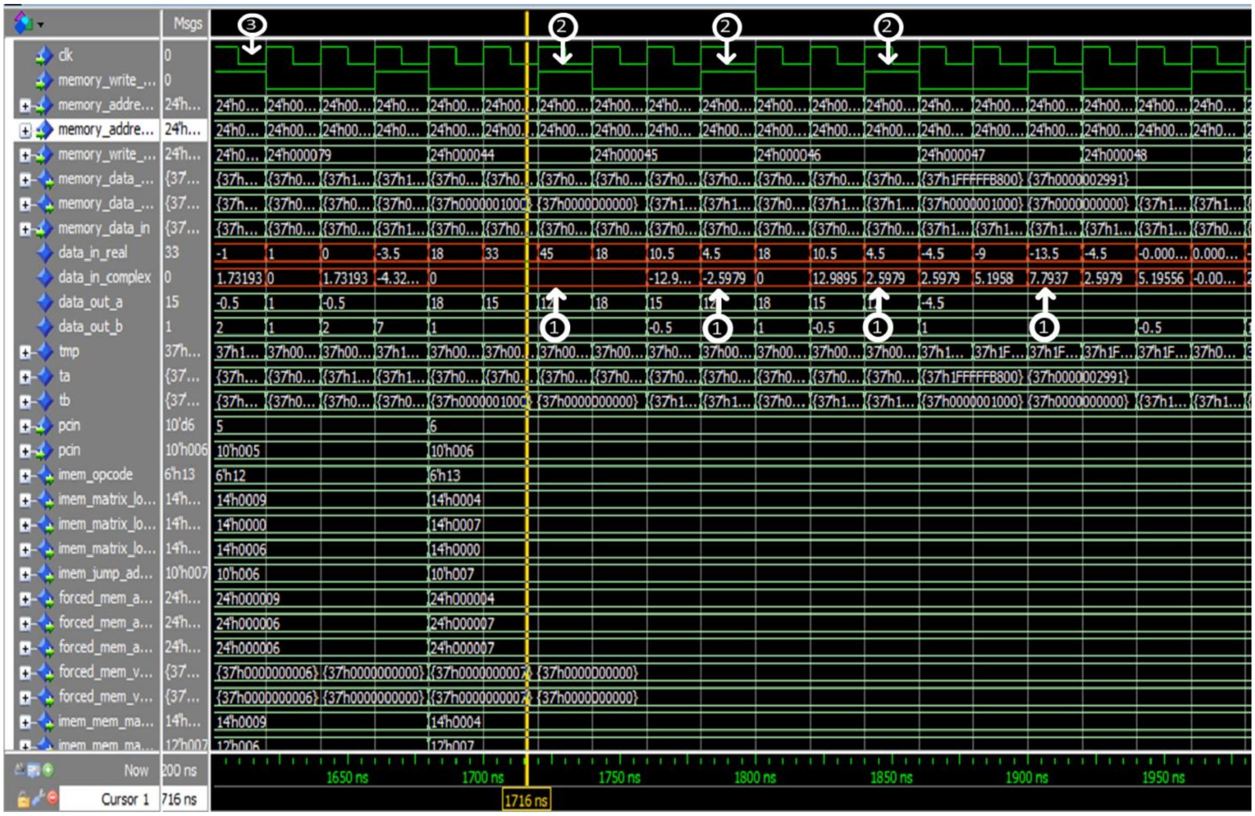


Figure 6-2 : Wave/Timing Diagram while calculating the DTFT

Matrix Multiplication

Matrix Multiplication is also a challenging part, rather bottle neck in the whole architecture and that the instruction with the most complexity. The matrix Multiplication consumes N x N cycles. For the testing purpose two matrices A and B are stored in the main memory. They are random matrices which are fed to the processor for multiplication via an instruction.

$$A = \begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{bmatrix}, B = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 9 & 8 & 7 \end{bmatrix}$$

The matrix Multiplication results obtained with the help of MATLAB are given as follows

$$A * B = \begin{bmatrix} 16 & 15 & 14 \\ 52 & 48 & 44 \\ 88 & 81 & 74 \end{bmatrix}$$

The wave diagram showing the timing of the outputs is as explained and given below, the identifier pointer (2) Indicates the Write signals to the main memory, the main memory is only written on a write signal. The other marker (1) Indicate the results produced by the ALU, which are then forwarded to the input of the main memory to be written to the specific locations. The single index generation mechanism will automatically generate the exact memory address of the source, target and destination operands. The Matrix A and B had Real numbers with no imaginary part, so the resultant will also be real. Images in digital forms are expressed by matrices. Each element represents pixel intensity .This representation of pixel intensity is actually in the forms of real numbers. Multiplying matrices which have elements as integer the resultant will be in integer. The multiply instruction will also consume multiple clock cycles. The matrix multiplication in this case will take N^3 cycles to complete a multiplication of matrices with size (N x N).In the case of (3 x 3) matrix it will consume 27 clock cycles to complete this instruction.

Table 6-2 Matrix Multiplication Results comparison with MATLAB

Matrix SUB Index	Processor Results	MATLAB Results
(1,1)	16	16
(1,2)	15	15
(1,3)	14	14
(2,1)	52	52
(2,2)	48	48
(2,3)	44	44
(3,1)	88	88
(3,2)	81	81
(3,3)	74	74

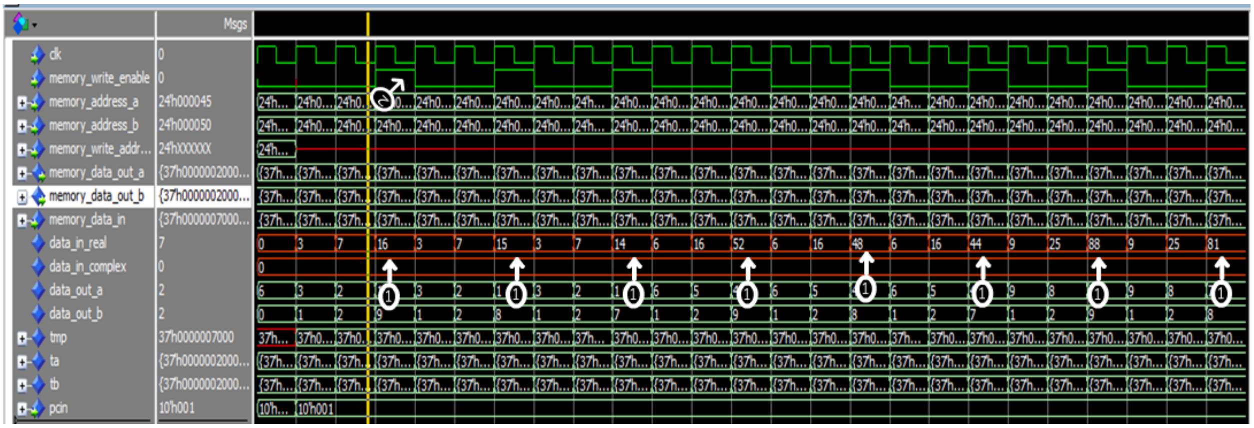


Figure 6-3 : Wave Diagram representing Matrix Multiplication

Matrix Arithmetic

Matrix Addition and subtraction operations are completed in N^2 Cycles .The arithmetic result of each corresponding element is produced in a single cycle. Again the addition of two matrices A and B is performed on this processor.

$$A = \begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{bmatrix}, B = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 9 & 8 & 7 \end{bmatrix}$$

The actual results for addition are as follows

$$A + B = \begin{bmatrix} 4 & 3 & 2 \\ 8 & 7 & 6 \\ 18 & 16 & 14 \end{bmatrix}$$

The Addresses will be generated automatically. For all other matrix arithmetic operations other than Multiplication the results are output in column oriented manner. The following table shows the actual results of the matrix addition, and next is the diagram of waveform showing the view at the input of the memory. The inputs are the resultants in the column oriented form. The marker (2) indicates the write signal to the memory, the indication marker (1) is pointing out the results of addition and other arithmetic operations are carried out in the same way .The only difference occurs in the OPcode that is sent to the ALU for example the data paths and selections would be the same, the signal (OPcode) will determine whether to subtract, add etc.

Table 6-3 Matrix Multiplication Addition Results comparison with MATLAB

Matrix SUB Index	Processor Results	MATLAB Results
(1,1)	4	4
(1,2)	3	3
(1,3)	2	2
(2,1)	8	8
(2,2)	7	7
(2,3)	6	6
(3,1)	18	18
(3,2)	16	16
(3,3)	14	14

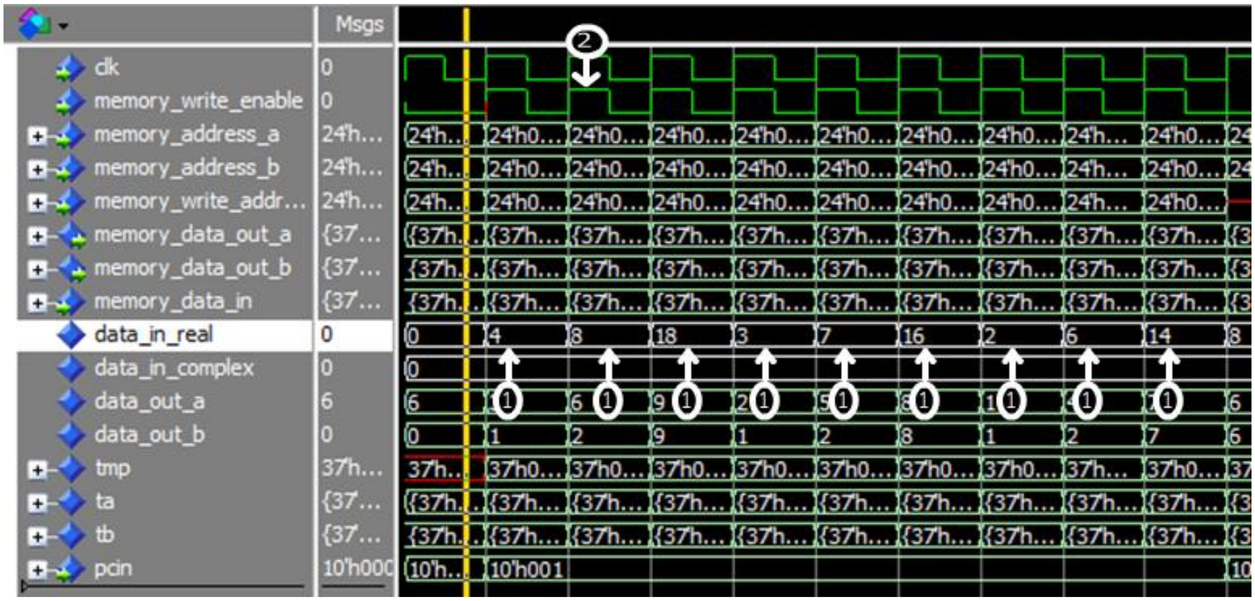


Figure 6-4 : Wave Diagram representing Matrix Addition

Matrix Transpose

The design as mentioned is capable of performing the transpose operation of a matrix. The transpose instruction was run on the following matrix this is the same random matrix.

$$A = \begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{bmatrix}$$

The matrix a was initially stored on the linear memory with the base address 68 i.e. the first element of the matrix can be found on the address 68 in the liner memory, the second element on the 69^{th} location vice versa. The transpose of the matrix is mapped on a new location, The following two figures illustrates the orientation of the two locations the locations on the right side indicates the source matrix addresses and the right i.e. 95-103 represents the destination addresses.

68	69	70	71	72	73	74	75	76	…	95	96	97	98	99	100	101	102	103

Figure 6-5 : Actual memory Addresses in liner memory

68	69	70
71	72	73
74	75	76

95	96	97
98	99	100
101	102	103

Figure 6-6 : Main Memory linear addresses in matrix form

The transpose of a matrix can be calculated by reversing the row index and column index .this way the transpose can be found, the next diagram indicates the data of the matrix a mapped on the new address. The data on the 68^{th} location which is 3 will be mapped on the memory address location 95, similarly the data on location 69 (i.e. 2) will be mapped to the 98^{th} location vice versa.

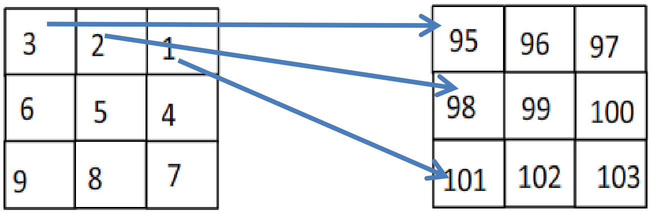


Figure 6-7 : Mapping of Matrix values to new locations

The results of the matrix transpose carried out on the processor are shown below in the wave diagram captured from the results. The marker indicator (2) represents the Memory write signal, in the case of transpose, the memory is written on every clock cycle as one result has to be stored to new location. The marker indicator (4) indicates the read address which is in column orientation .Also the indicator (3) represents the write address .The data which is written on the memory are tagged by marker (1). This clearly indicate that the values are read form the specified address and written to a new location accordingly.

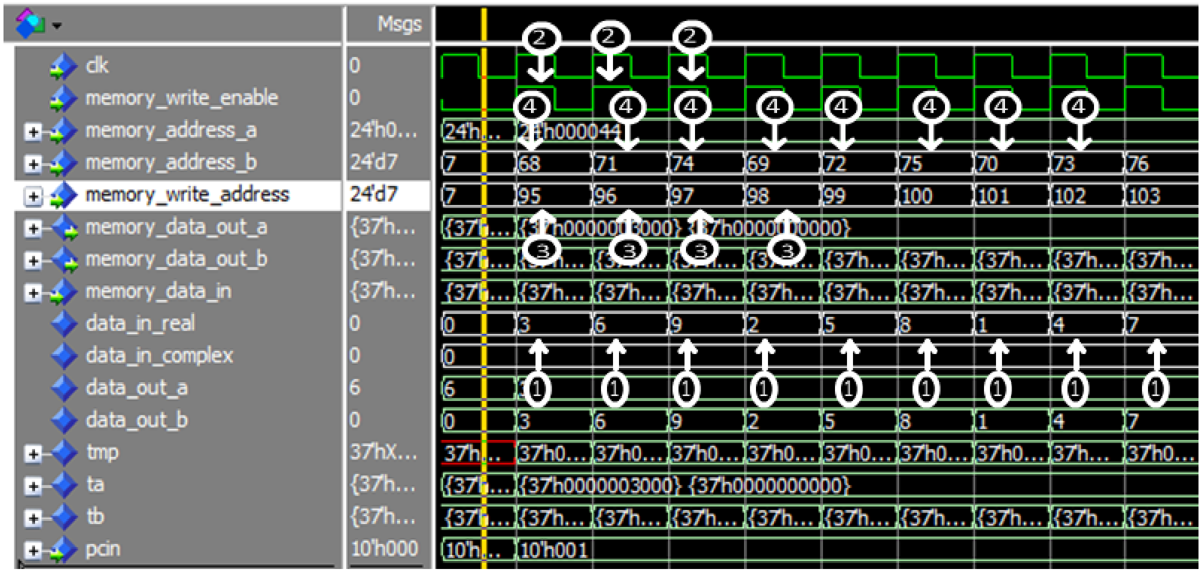


Figure 6-8 : Mapping of Matrix Transpose

Branching

The processor also provides branching facility, the results of an unconditional jump is shown in the wave diagram below. On the second location of the instruction memory and unconditional branch instruction is used .The second instruction (i.e. on location (1) of instruction memory) jumps to the 55^{th} instruction, the Identifier pcin marked by marker (2) clearly indicates that the flow of instruction branched from second instruction to 55^{th} instruction and started execution in the normal manner again.

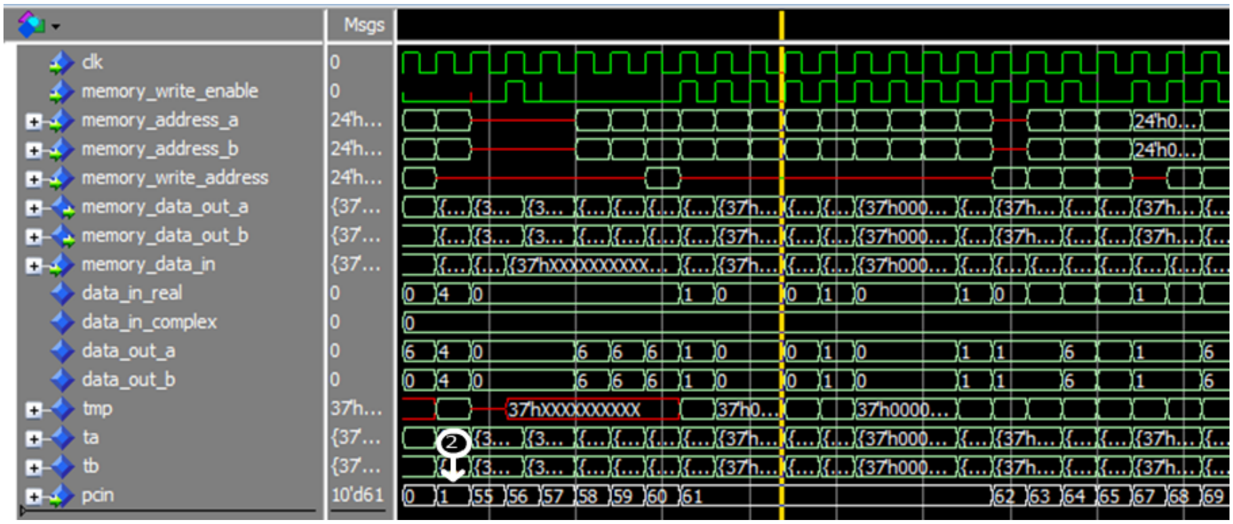


Figure 6-9 : Simulation results for branch

Scalar Arithmetic

Scalar at thematic is one fundamental part of the processor, the results of the following instruction written in text format represents the binaries of the instruction. The instruction LOD basically writes a specified value on the specified location .The first instruction load a fixed point value 2 on the 0 address of the memory, similarly the fixed point value 2 is also written on the address 1 of the memory.

The third instruction adds the contents of the location 0, 1 of the memory and store to the third location. The marker (1) indicates the result which is 4.the 4th instruction Subtracts the data on the specified two locations, the result is marked by (3) in the wave timing diagram below. Fifth instruction multiplies the data available on the two memory locations and stores on the memory location number 3 of the memory.

1. LOD & R0 & RF2
2. LOD & R1 & RF2
3. ADD & R3 & R0 & R1;
4. SUBB & R3 & R0 & R1;
5. MUL & R3 & R0 & R1;

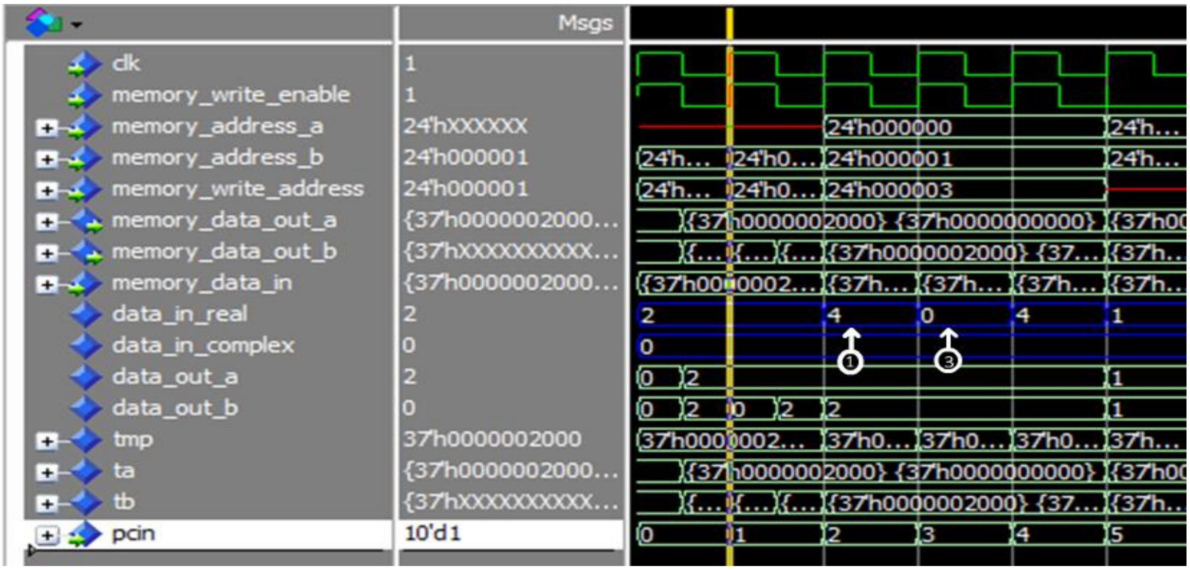
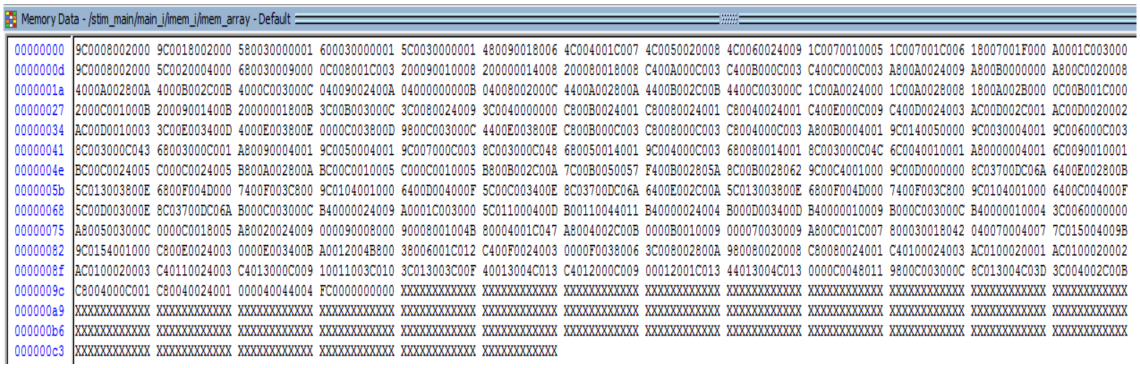


Figure 6-10 : Wave diagrams for results

Instruction Memory Contents View



```
Memory Data - /stim_main/main_i/imem_i/imem_array - Default
00000000 9C0008002000 9C0018002000 580030000001 600030000001 5C0030000001 480090018006 4C004001C007 4C0050020008 4C0060024009 1C0070010005 1C007001C006 18007001F000 A0001C003000
0000000d 9C0008002000 5C0020004000 680030009000 0C008001C003 200090010008 200000014008 200080018008 C400A000C003 C400B000C003 C400C000C003 A800A0024009 A800B0000000 A800C0020008
0000001a 4000A002800A 4000B002C00B 4000C003000C 04009002400A 04000000000B 04008002000C 4400A002800A 4400B002C00B 4400C003000C 1C00A0024000 1C00A0028008 1800A002B000 0C00B001C000
00000027 2000C001000B 20009001400B 20000001800B 3C00B003000C 3C0080024009 3C0040000000 C800B0024001 C80080024001 C80040024001 C400E000C009 C400D0024003 AC00D002C001 AC00D0020002
00000034 AC00D0010003 3C00E003400D 4000E003800E 0000C003800D 9800C003000C 4400E003800E C800B000C003 C8008000C003 C8004000C003 A800B0004001 9C0140050000 9C0030004001 9C006000C003
00000041 8C003000C043 68003000C001 A80090004001 9C0050004001 9C007000C003 8C003000C048 680050014001 9C004000C003 680080014001 8C003000C04C 6C0040010001 A80000004001 6C0090010001
0000004e BC00C0024005 C000C0024005 B800A002800A BC00C0010005 C000C0010005 B800B002C00A 7C00B0050057 F400B002805A 8C00B0028062 9C00C4001000 9C00D0000000 8C03700DC06A 6400E002800B
0000005b 5C013003800E 6800F004D000 7400F003C800 9C0104001000 6400D004000F 5C00C003400E 8C03700DC06A 6400E002C00A 5C013003800E 6800F004D000 7400F003C800 9C0104001000 6400C004000F
00000068 5C00D003000E 8C03700DC06A B000C003000C B40000024009 A0001C003000 5C011000400D B00110044011 B40000024004 B000D003400D B40000010009 B000C003000C B40000010004 3C0060000000
00000075 A8005003000C 0000C0018005 A80020024009 000090008000 90008001004B 80004001C047 A8004002C00B 0000B0010009 000070030009 A800C001C007 800030018042 040070004007 7C015004009B
00000082 9C0154001000 C800E0024003 0000E003400B A0012004B800 38006001C012 C400F0024003 0000F0038006 3C008002800A 980080020008 C80080024001 C40100024003 AC0100020001 AC0100020002
0000008f AC0100020003 C40110024003 C4013000C009 10011003C010 3C013003C00F 40013004C013 C4012000C009 00012001C013 44013004C013 0000C0048011 9800C003000C 8C013004C03D 3C004002C00B
0000009c C8004000C001 C80040024001 000040044004 FC0000000000 XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX
000000a9 XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX
000000b6 XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX
000000c3 XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX XXXXXXXXXXXX
```

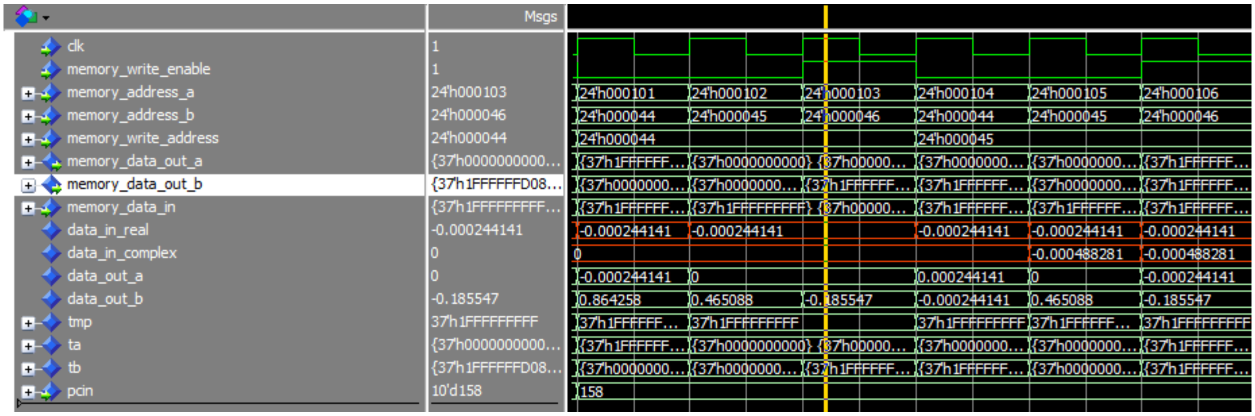
Figure 6-11 : View of the instruction memory in HEX format

EMACH Filter Calculation results

A sample program was written for calculation of EMACH filter coefficients with three Training sample (images) and the results obtained by MATLAB and the simulation are presented below.

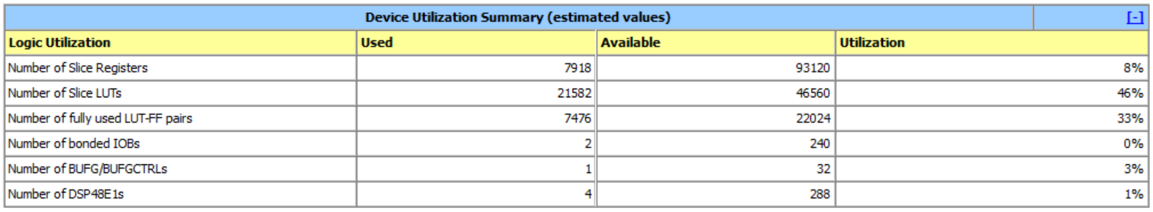
Table 6-4 Filter Results comparison with MATLAB

SUB Index	Processor Results	MATLAB Results
(1,1)	-0.000244141+0i	-4.9179e-05 + 0i
(1,2)	-0.000244141-0.000488281i	0.00024707 - 0.00040732i
(1,3)	0.000244141+0.000488281i	0.00024707 + 0.00040732i



	Msgs
clk	1
memory_write_enable	1
memory_address_a	24'h000103
memory_address_b	24'h000046
memory_write_address	24'h000044
memory_data_out_a	{37'h0000000000...
memory_data_out_b	{37'h1FFFFFFD08...
memory_data_in	{37'h1FFFFFFFFF...
data_in_real	-0.000244141
data_in_complex	0
data_out_a	0
data_out_b	-0.185547
tmp	37'h1FFFFFFFFF
ta	{37'h0000000000...
tb	{37'h1FFFFFFD08...
pcin	10'd158

Figure 6-12 : Filter generation results



Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	7918	93120	8%
Number of Slice LUTs	21582	46560	46%
Number of fully used LUT-FF pairs	7476	22024	33%
Number of bonded IOBs	2	240	0%
Number of BUFG/BUFGCTRLs	1	32	3%
Number of DSP48E1s	4	288	1%

Figure 6-13 : Synthesis summary

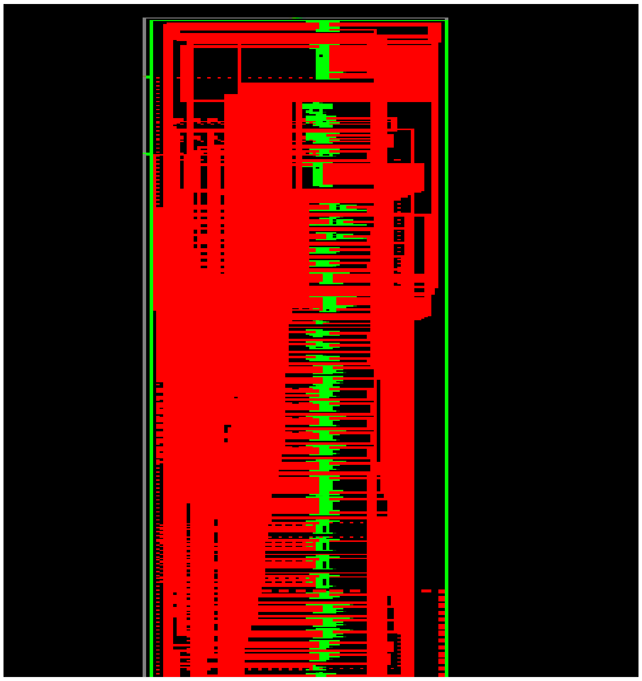


Figure 6-14: RTL Diagram

Device utilization summary:

Selected Device : 6vlx75tff484-3

Slice Logic Utilization:

Number of Slice Registers: 7918 out of 93120 8%

Number of Slice LUTs: 21582 out of 46560 46%

Number used as Logic: 21578 out of 46560 46%

Number used as Memory: 4 out of 16720 0%

Number used as RAM: 4

Slice Logic Distribution:

Number of LUT Flip Flop pairs used: 22024

Number with an unused Flip Flop: 14106 out of 22024 64%

Number with an unused LUT: 442 out of 22024 2%

Number of fully used LUT-FF pairs: 7476 out of 22024 33%

Number of unique control sets: 109

Chapter 7 Future Work and Conclusion

Over the last decades the use of the microprocessors has increased tremendously. The current microprocessor market is focusing more on embedded microprocessors as compared to common desktop Computers as they are power efficient, smaller in size and weight (in many cases). Various products in the market sector are dependent on these embedded processors. Many Features of a modern device can be controlled with one or more embedded processor, providing reliability, accessibility and efficiency to the user. There is a considerable scope for research in microprocessors as they provide greater functionality with optimal speeds at lower costs.

The mobile device sector has a great trend towards embedded systems, these devices provide all at one place like video/audio play back and recording, Video games, Internet, Communication Facilities Taking Photos and so much more. To perform this device must have a processing element that can perform everything at real time, keeping the cost in view. Many of the techniques used in general processor (i.e. Desktop Computers) are not suitable for dedicated processors due to higher power and space requirements to so alternate techniques are required.

The cost of the hardware and performance are directly proportional to each other, as the performance increase the cost of the hardware will also increase and when the hardware has to decrease the performance will decrease. The main goal is to increase hardware to some extent to provide flexibility and optimal throughput. CPI (clock per instruction) has been a scale to compare different machines/designs. This design belongs to the category of dedicated processor and is capable of performing arithmetic operations on Matrices, scalars and logical instructions with the assistance of jumps and branches (Loops), these instructions will allow the processor to many task related to image processing and pattern recognition any piece of can be run able directly or indirectly (may require some logical/syntax changes).The design and implementation of Matrix processor is complex and time consuming job. The engineering design was an iterative process of specification, Analysis, Implementation and synthesis.

The processor can be used in dedicated and standalone devices which will provide more flexibility off use. In this thesis a digital design of a correlation pattern recognition processor is presented in brief detail. This processor is capable of computing the filter using the training and test image. After the computations of filter for a specific class of training image, the digital design is also capable of correlating the test image to the filter. The correlation filtering is usually a carried out in frequency domain, it is basically an optical technique which is digitized. This conversing can also be carried out on other general computing machines. But the aim was to provide a faster and standalone machine capable of carrying out the application of correlation filters. The micro code structure can provide custom flexibility, i.e. like other general processor its usage can be modified according to the need. Suppose that the specific tasks or scenario don't fit good using a specific filter, another filter derivation algorithm can be brought on the processor, the processor is specifically designed to handle matrices of different size with providing the power of representing fractional numbers (fixed Points).

A general computing environment provides reduced instruction and sometimes a mixture of complex instructions .Using these instructions and the branching capability of a processor, Matrices can also be maintained on the software side, this methodology of providing the can save the cost in terms of hardware but will defiantly cost more in terms of consumption of clock cycles. The presented digital design saves comparison cycles by providing on chip memory management for matrices, besides saving cycles this will also provide a great deal of flexibility and ease of use to the programmer with higher computation speeds up to real time application.

Also the design of the processor is a base design, and can be used for educational purpose for understanding and practicing the core digital components. The processer can be custom re programed for changing the size the images, an AutoDetect mechanism can also be embedded for new size of matrices .This design basically provides the unitary building blocks for a universal use of unlimited applications. Besides Image process and machine learning the design can also be configured for other uses to, and data that can be represented in Matrices forms and requires processing in the form of matrices this design can provide the best functionality and flexibility with very high speed. Except for the memory rest of the design has very low cost but very high functionality.

Future Work

The currently presented design is capable of performing matrices arithmetic operations along with scalar handling and branching mechanism. The processor is currently running in a single cycle, sum of all the delay of components along with some other delays will determine the clock cycle time, A pipelined version can be introduced to achieve implicit parallel ism and higher speed of processing. The first goal was to provide such a processor capable of performing matrix operations. The next step is to optimize the processor over the time.

Pipeline

The proposed Design has provided the basic architecture that is capable of performing Matrix and scalar operations along with branching. A Pipelined version is currently under design consideration that will definitely speed up the throughput of the system.

Embedding FFT

The current design is handling Fourier transform in along with the multiplication data path , a design is proposed to which will be capable of computing FFT using almost the same components available in the design but will be very beneficial as the complexity of the FFT is lesser than DFT computations.

The switching mechanism can be simply modified to compute the FFT of the Matrices using the existing hardware.

System I/O design

The system was actually tested without inputs and outputs, by adding simple Inputs and Outputs the system can be either interfaced with a general computer for viewing results or either configuring the system.

References

1. B. V. K. Vijaya Kumar, A.M., Richard D. Juday "Correlation Pattern Recognition" Cambridge University Press ISBN-13 948-0-511-13320-6.
2. Duda, R.O.H., P.E.; Stork, D.G., Pattern classification, 2nd edn. and N.Y. John Wiley & Sons.
3. James, A.P.D., S., Inter-image outliers and their application, to image classification. Pattern Recogn. 43(12) (2010).
4. Gardezi, A.A., A.; Birch, P.; Young, R.; Chatwin, C., A space variant maximum average correlation height (MACH) filter for object recognition in real time thermal images for security applications. SPIE 7838, 78380N (2010).
5. Mahalanobis, A.K., B.V.K.; Sims, S.R.F.; Epperson, J.F., Unconstrained correlation filters. Appl. Opt. 33(17), and (1994).
6. Qureshi, W.S.A., A.B.N., Object tracking using MACH filter, a.o.f.c.s.a.v.l. conditions., and W.p. (2009).
7. Mohamed, A.K., V.B.V.K.; Abhijit, M., Improved clutter rejection in automatic target recognition (ATR) synthetic aperture radar (SAR) imagery using the extended maximum average correlation height (EMACH) filter. In: Zelnio, E.G. (ed.) Proceedings of the SPIE, vol. 4053, pp. 332–339. Algorithms for Synthetic Aperture Radar Imagery VII August (2000).
8. Vijaya Kumar, B.V.K.A., M.;Mahalanobis, A., Improving the false alarm capabilities of composite correlation filters. Opt. Eng. 39(5), 1133–1141 (2000).
9. David Paul Casasent, T.-H.C., Optical Pattern Recognition, Volume 17.
10. Kumar, B.V.K.V., Applied Optics, pp. 4773-4801, 1992.
11. Casasent, G.R.a.D.P., Noise and discrimination performance of the MINACE optical correlation filter Applied and p.-. Optics 31(11), April 1992.
12. Brigham, Prentice Hall. ISBN 0-13-307505-2.
13. Kamisetty Rao, D.N.K., Jae Jeong Hwang, Fast Fourier Transform - Algorithms and Applications: Algorithms
14. Sen-Maw Kuo, W.-S.G., "Digital signal processors: architectures, implementations, and applications " Pearson Prentice Hall, 2005.

15. Cook, S.M.M., "Raspberry Pi for dummies" Hoboken, NJ : John Wiley & Sons, ©2013.

16. Furber, S.B., "ARM System-on-chip Architecture" Addison-Wesley, 2000.

17. Choudhry, S., Project Management, Tata McGraw-Hill Education, 1988.

18. Russo, D.A., Aerospace and Electronic Systems, IEEE Transactions on (Volume:AES-3, Issue: 5) pp 779 - 783 Sept. 1967.

19. Gonzalez, R.C., Digital Image Processing 3rd Ed. (DIP/3e) by Gonzalez and Woods © 2008.

20. Strang, G.M.J.W.A.S.R.O.

21. Cooley, J.W.T., John W. (1965). "An algorithm for the machine calculation of complex Fourier series". Math. Comput. 19: 297–301.

22. Khan, S.A., Digital Design of Signal Processing Systems: A Practical Approach, John Wiley & Sons, 02-Feb-2011.

23. Hyde, R., Write Great Code: Volume I: Understanding the Machine November 2004.

24. Roger Woods, J.M., Dr. Ying Yi, Gaye Lightbody, FPGA-based Implementation of Signal Processing Systems

25. Navabi, Vhdl:Modular Design Tata McGraw-Hill Education, 2010.

26. Vahid, F., Digital Design with RTL Design, Verilog and VHDL John Wiley & Sons, 08-Mar-2010.

27. Robert Schilling, S.H., Fundamentals of Digital Signal Processing Using MATLAB Cengage Learning, 01-Jan-2011

28. Cragon, H.G., Computer Architecture and Implementation, Cambridge University Press, 2000.

29. Shoup., R., Parameterized Convolution Filtering in a Field Programmable Gate Array nterval. Technical Report, Palo Alto, California .1993.

30. F.G.Lorca, L.K.a.D.D., Efficient ASIC and FPGA implementation of IIR filters for Real time edge detection. In the International Conference on image processing (ICIP-97) Volume 2. Oct 1997.

31. Nelson., Implementation of Image Processing Algorithms on FPGA Hardware. Masters Thesis, Graduate School of Vanderbilt University, 2000.

32. Shinichi Hirai, M.Z., Tatsuhiko Tsuboi,, Implementing Image Processing Algorithms on FPGA-based Realtime Vision System, Proc. 11th Synthesis and System Integration of Mixed Information Technologies (SASIMI 2003), pp.378-385, Hiroshima, April, 2003.

33. Chen, Fahad.A., Real-time high performance Edge detector for computer vision applications. In the Proceedings of ASP-DAC, 1997, pp 671-672.

34. Peter Baran, R.B.a.J.H., Reduce Build Costs by Offloading DSP Functions to an FPGA. FPGA and Structured ASIC Journal.