OPTIMIZATION OF VLSI ARCHITECTURE FOR MATRIX-VECTOR MULTIPLICATION AND PARK TRANSFORMATION WITH APPLICATION TO POWER SYSTEM SIMULATION

A Thesis

by

NAGA SHIVA SAI PAVAN KUMAR DEVARASETTI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2021

Major Subject: Computer Engineering

ABSTRACT

Contemporary Power systems with renewable generators, power electronics possess new challenges to the system operators. The significant increase of variable energy resources in the power grid leads to a stressed grid with much higher variabilities at the operational stage. Such variabilities together with today's lack of accurate simulation capabilities lead to significant uncertainties in predicting the dynamics across the grid.

Hence, Fast simulation tools are required for transient analysis such as the electro-magnetic transient program (EMTP). Electromagnetic transient (EMT) simulation is the most powerful tool which could handle various detailed device models and capture high-speed dynamic behavior in the power system. However, due to the enormous complexity of the power systems, these simulators tend to be slow in simulating real-time data. In order to accelerate these computations, we can take the advantage of GPUs and FPGAs. But still, the 'real-time' simulation is not fast enough to simulate the true real-time of a large system. To overcome this limitation, a custom chip (ASIC) for simulating power systems is necessary.

Matrix Multiplication is one of the most fundamental basic operations in the current complex digital circuits and is vastly used in the image, signal processing applications[1]. In EMTP simulation, Matrix-Vector Multiplication is used in solving network equations and the input to these are current(I) from different buses of a large system. These vectors tend to be periodic with a fundamental frequency of 60 Hz along with other higher-order harmonics. When sampled with a relatively higher frequency rate, the consecutive samples of the inputs tend to be close and it is inefficient to compute the matrix operation again and again at every time step. Instead, In this thesis, we aim to propose an architecture based on the higher-order difference which takes variation with respect to the previous sample to compute the actual result.

Park Transformation and Inverse Park Transformation are the interfaces between non-network and network part. They convert state variables from DQ0 coordinate to three-phase coordinate and vice versa. Linear interpolation is a method of curve fitting using polynomials to construct new data points within the range of a discrete set of known data points. In this thesis, we propose a ROM architecture combined with linear interpolation technique to compute the high precision sinusoid values.

DEDICATION

To my parents, younger brother and grandparents for their unconditional love, support and sacrifice.

ACKNOWLEDGMENTS

I would like to express my gratitude to my thesis advisor and committee chair Dr. Weiping Shi for his continuous guidance, untiring support and immense knowledge and for keeping me motivated throughout the last two years. I appreciate the trust he had in me before accepting me to his research group. The many discussions I have had with him have proven to be instrumental in completing my research work.

I would like to thank Dr. P. R. Kumar and Dr. Vivek Sarin for serving on my committee.

I would like to thank Zhang Lu and Zhixing Li, PhD scholars for taking their time in conducting meetings, collaborating with me in making progress.

I would also like to thank my friend Ram Chander Bhaskara, RA from Aerospace Department for lending his Zybo Z7-20 board on which a part of this thesis work has been carried out.

Last but not least, I would like to thank my friends here at College Station who always supported me and kept me smiling during the time of distress. Without their unconditional support, this wouldn't have been possible.

Special thanks to Dr. Vipin Kizheppatt, Assistant Professor BITS Goa, India for his playlist on Reconfigurable Embedded Systems with Xilinx Zynq APSoC in YouTube and Dr Christopher Batten, Associate Professor, Cornell University for his free tutorial on Synopsys/Cadence ASIC Tools using NanGate 45nm process.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

All work conducted for the thesis (or) dissertation was completed by the student, under the advisement of Weiping Shi of the Department of Electrical and Computer Engineering.

Funding Sources

Graduate study was partly supported through a scholarship from the ECE department at Texas AM University.

NOMENCLATURE

EMT	Electromagnetic Transients
DAE	Differential- Algebraic Equation
ODE	Ordinary-Differential Equation
CPU	Central Processing Unit
GPU	Graphic Processing Unit
FPGA	Field - Programmable Gate Array
ASIC	Application - Specific Integrated Circuit
SVD	Singular Value Decomposition
HiLap	Hierarchical Low-rank Approximation
RTL	Register Transfer Level
HDL	Hardware Description Language
ARM	Advanced RISC Machine
ROM	Read Only Memory
SoC	System on Chip
DSP	Digital Signal Processing
DRC	Design Rule Check
EDA	Electronic Design Automation
VLSI	Very Large Scale Integration
IC	Integrated Circuits

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Introduction

In power system, EMT is a powerful tool that can perform high speed electromagnetic transients and various issues ranging from nanoseconds to seconds. Initially, EMT has been used for transmission line switching studies. But EMT simulation has become more and more powerful and essential as the power systems became complicated.

In the current state, EMT has developed to handle extremely variety of models of generators, transmission line, transformers and other elements in the power system. It can solve any network consisting of interconnections of resistance, inductance , capacitance and other elements. Due to its fast simulation, it is highly demanded in many designs and studies like HVDC converter control, lightning overvoltage computation etc.,

EMT simulation consists of 10 steps as follows:

1. Initialization.
2. Flux to Voltage.
3. Pre – calculate generator stage.
4. Solve electrical.
5. Park(D2A).
6. Network Solution.
7. Update $I_{history}$.
8. Inverse Park(A2D).
9. Update Current.
10. Solve Mechanical and Control Elements.[2]

In our thesis we aim to present an optimized VLSI architecture to implement matrix - vector multiplication as part of solving network equations, Park(D2A) and Inverse Park(A2D) transformation.

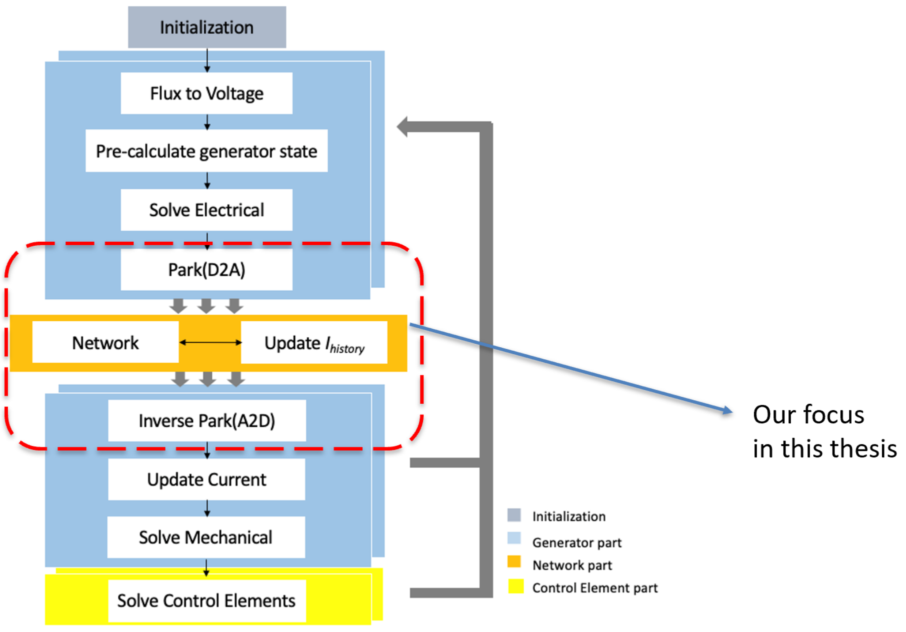


Figure 1.1: EMT steps

1.2 Network Equations

In EMT simulations, the network equations has to be solved at every time step and that causes huge computation burden on the large – scale systems. There are three general approaches to solve it.

1. Dense inversion and matrix – vector products.
2. Direct sparse solver.
3. Direct dense solver.

These methods are inherently sequential and are not a good option for hardware implementation. The paper [3] [4] describes a highly parallel approach to solve the linear system equations based on hierarchial low – rank approximation.

The proposed approach in the papers, [3] [4] first partitions the network hierarchically, calculates the inverse of G (sparse network conductance matrix) offline and approximates the result using low rank approximation by truncated SVD (singular value decomposition). After this, the values are loaded into the system and matrix – vector multiplication is performed online to compute the nodal voltage vector v(t). This method takes O(N logN) time when compared to the traditional matrix – vector multiplication, which takes O(N^2) time.

In our thesis, we implement the architecture on FPGA and ASIC both to solve low rank approximated matrix - vector multiplication as part of solving the network equations [3].

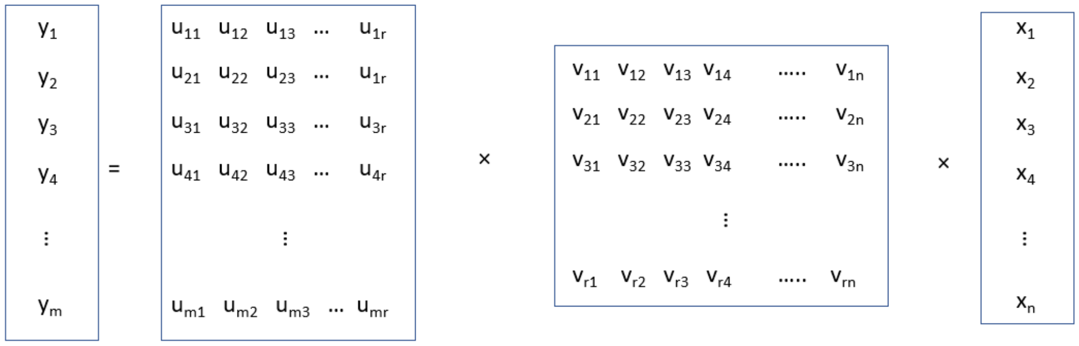

y_1 y_2 y_3 y_4 ⋮ y_m
=
u_{11} u_{12} u_{13} … u_{1r}
u_{21} u_{22} u_{23} … u_{1r}
u_{31} u_{32} u_{33} … u_{3r}
u_{41} u_{42} u_{43} … u_{4r}
⋮
u_{m1} u_{m2} u_{m3} … u_{mr}
×
v_{11} v_{12} v_{13} v_{14} ….. v_{1n}
v_{21} v_{22} v_{23} v_{24} ….. v_{2n}
v_{31} v_{32} v_{33} v_{34} ….. v_{3n}
⋮
v_{r1} v_{r2} v_{r3} v_{r4} ….. v_{rn}
×
x_1 x_2 x_3 x_4 ⋮ x_n

Figure 1.2: Matrix vector multiplication

1.3 Park Transformation and Inverse Park Transformation

Park Transformation and Inverse Park Transformation are the interface between non-network and network part.

Park transformation converts the state variables from DQ0 coordinate in the generator side to the three phase coordinates in the network, whereas inverse park transformation reverts the conversion back to the DQ0 coordinates through matrix – vector multiplication as shown below.

Park Transformation:

$$\begin{pmatrix} I_a(t) \\ I_b(t) \\ I_c(t) \end{pmatrix} = \begin{pmatrix} sin(\theta + \frac{1}{2}\pi) & -sin(\theta) & 1 \\ sin(\theta - \frac{1}{6}\pi) & sin(\theta + \frac{1}{3}\pi) & 1 \\ sin(\theta + \frac{1}{6}\pi) & sin(\theta - \frac{1}{3}\pi) & 1 \end{pmatrix} \begin{pmatrix} I_d(t) \\ I_q(t) \\ 0 \end{pmatrix} \tag{1.1}$$

Inverse Park Transformation:

$$\begin{pmatrix} e_d(t) \\ e_q(t) \\ \epsilon \end{pmatrix} = \begin{pmatrix} sin(\theta + \frac{1}{2}\pi) & sin(\theta - \frac{1}{6}\pi) & -sin(\theta - \frac{1}{6}\pi) \\ -sin(\theta) & sin(\theta + \frac{1}{3} & sin(\theta - \frac{1}{3} \\ 0.5 & 0.5 & 0.5 \end{pmatrix} \begin{pmatrix} V_a(t) \\ V_b(t) \\ V_c(t) \end{pmatrix} \tag{1.2}$$

1.4 Outline

The rest of the thesis is organized as follows. Chapter 2 aims to discuss about previous implementations of the required subject. Chapter 2 talks about representation of numbers using Fixed Point and Floating Point. Chapter 3 focuses on explaining the higher order difference technique to reduce the hardware resources for matrix vector multiplication. Chapter 4 discusses about implementation of park and inverse park transformation and finally in chapter 5, we produce the results.

2. PREVIOUS WORK

2.1 Matrix Vector Multiplication

The algebra routine of Matrix-Vector Multiplication is essential in numerous scientific usages such as linear system solvers and LU Factorization. It is essential not only in theory, but also in applied sciences and engineering. In literature, many researchers have worked on this routine by exploiting different platforms.

In the work [5], new methods were explored to accomplish high performance on a floating-point DSP. This device possessed two elements capable to hurry the matrix multiplication a software-managed memory hierarchy, and a direct memory access (DMA) which brings block copies from central memory to into the memory hierarchy. Whereas, Ž. Jovanoviü and V. Milutinoviü [6] proposed an implementation of FPGA architecture for Floating-Point Matrix Multiplication. It uses the block matrix multiplication algorithm that sends backs the product blocks to the CPU instantaneously to facilitate the placement and routing on the chip. The objective was to achieve maximium clock frequency and minimize the resource utilization. In [7] authors have proposed an architecture which is based on streaming approach which exploits the parallelism available in FPGA. In stream computing, the data is arranged into streams which are a group of data like arrays. This method maps well to the FPGA logic. In addition to these, Distributed Arithmetic and systolic architecture based approach have been explored and implemented in [8].

Our approach is this thesis is based on implementation explored in [9]. In this approach, in order to optimize the FPGA architecture resource usage, the data from input matrices U and V have been read from memory for exactly once. By simultaneously reading one column of matrix U and one row of matrix V, and performing all multiply operations based on those values , optimal data re-use is achieved. Data read in this sequence allows one partial product term of every element to be computed in output matrix C per clock cycle.

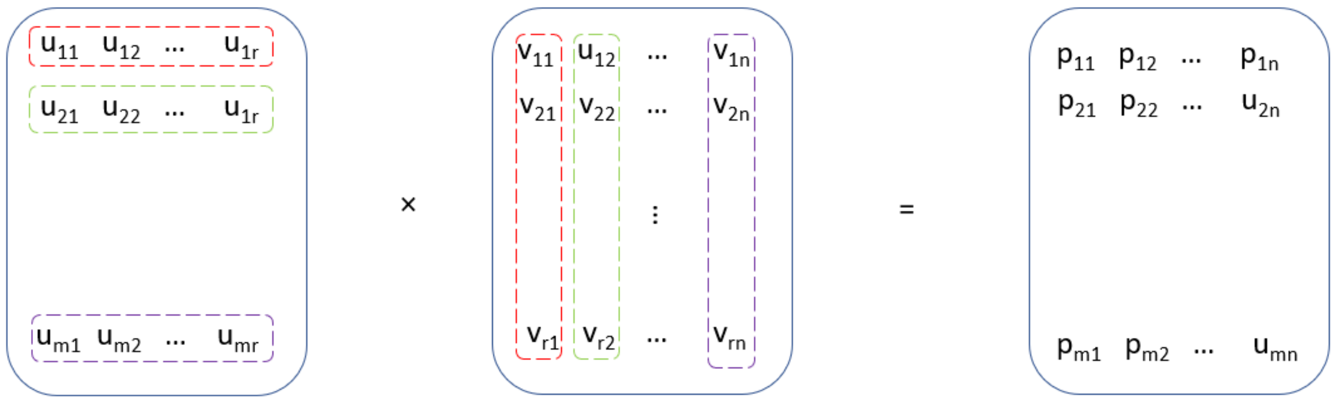

u_{11} u_{12} ... u_{1r}
u_{21} u_{22} ... u_{1r}
u_{m1} u_{m2} ... u_{mr}
×
v_{11} u_{12} ... v_{1n}
v_{21} v_{22} ... v_{2n}
⋮
v_{r1} v_{r2} ... v_{rn}
=
p_{11} p_{12} ... p_{1n}
p_{21} p_{22} ... u_{2n}
p_{m1} p_{m2} ... u_{mn}

Figure 2.1: Matrix Multiplication

2.2 Park and Inverse park transformation

In the paper [10], a phase to sinusoid amplitude converter based on first order polynomial approximation of the sine function has been proposed. In [11], a non-uniform based piecewise linear approximation with one bit error correction has been proposed to compute sinusoid values. We discussed the advantages and drawbacks of these methods more in the section 5.

Embedded read-only-memory (ROM) is required in various situations when designing ASIC [12]. The on-chip ROM is usually applied to store fixed information such as micro-code etc. The micro-code ROM is a key component in the microprocessors, which is in generates the control signals for CPU execution. Our Implementation is based on the approach described in [13].

There are two basic ROM cell structures using silicon-gate transistors for ROM design, they are the NOR-gate type or parallel ROM cell structure, and the NAND-gate type or serial ROM cell structure [14] [15]. The parallel ROM cell structure consists of MOS transistors in parallel in which ROM data are generally fixed by a contact mask. The cell structure has the advantage of high-speed operation, but comes at a cost of low bit density. The serial ROM cell structure however on the other hand consists of serial MOS transistors and each memory cell state is determined by an enhancement or depletion mode MOS transistor. The cell structure has the disadvantage of relatively low-speed operation but the high packing density [16]. In our design, we chose to go for

NOR structure as speed is our top priority.

The design of the address decoders has a huge impact on the speed and power consumption of the memory. In our design, we have both row and column decoders which can be designed using either standard cells, NOR structure or NAND structure. Another component in our design is column mux, which is implemented as tree decoder that uses a binary reduction scheme [17].

3. NUMBER REPRESENTATION

In this chapter, we talk about conventional number format in computer systems. Differences between Fixed Point, Floating Point [18] and tradeoffs when it comes to VLSI implementation.

3.1 Radix Number System

A conventional radix number P can be represented as follows

$$(b_{n-1}b_{n-2}...b_1b_0)_r \tag{3.1}$$

with r being radix. Hence, P $= b_{n-1}.w_{n-1} + b_{n-2}.w_{n-2}... + b_0.w_0$ with w_i being the weight of position of i. If r is a fixed radix number,

$$P = b_{n-1}.r_{n-1} + b_{n-2}.r_{n-2}... + b_0.r_0 = \sum_{i=0}^{n-1} b_i.r_i. \tag{3.2}$$

To include fractional part, let us take '.' As radix point and include integer part in the left, fractional part in the right

$$(b_{n-1}b_{n-2}...b_1b_0b_{-1}...b_{-m})_r \tag{3.3}$$

Now,

$$P = \sum_{i=-m}^{n-1} b_i.r^i. \tag{3.4}$$

For example, in decimal system, r = 10 and b$_i$ ϵ 0 ... 9

P = 73.210

= b$_1$. r^1 + b$_0$. r^0 + b$_{-1}$. r^{-1}

= 7 x 101 + 3 x 100 + 2 x 10-1

= 73.2

In Hexadecimal system, r = 16 and b$_i$ ϵ 0...15

P = 2A16

= $b_1 . r^1 + b_0 . r^0$

= 2 x 16 + 10 x 1

= 42

Similarly, in binary system, r = 2 and $b_i \epsilon$ 0,1

P = 1010.112 = $b_1 . r^1 + b_0 . r^0 + b_{-1} . r^{-1}$

= $1 \text{ x } 2^3 + 0 \text{ x } 2^2 + 1 \text{ x } 2^1 + 0 \text{ x } 2^0 + 1 \text{ x } 2^{-1} + 1 \text{ x } 2{-}2$

= 8 + 2 + 0.5 + 0.25

= 10.75

In Computer Systems, we use 1s and 0s to represent all the arithmetic numbers and all our discussions in the upcoming chapters is based on binary system.

3.2 Signed Number Representation

The above discussion on the number system only talks about representing positive numbers. However, all the scientific calculations don't necessarily be confined to positive numbers. We need a notation to describe negative integers. The general convention is leftmost bit(MSB) describes sign of a number in which bit 0 is for positive and 1 is for negative.

However, the user has to determine, if a number is signed or unsigned. For a signed number, then left most bit represents the sign and rest of the binary string represent the actual magnitude of a number. Whereas for an unsigned number, all the string makeup the actual magnitude of a number.

For example, a string of bits 01010 can be considered as 10(unsigned) or +10(signed) binary and a string of bits 11010 can be considered as 26(unsigned) or -10(signed) binary.

Thus, with 5 bits of a string, we can represent from 0 to 31 by declaring a number unsigned. Whereas using signed keyword, we can represent from -15 to + 15 only.

Although this is a simple method to represent numbers, computer systems adopt a different system to represent negative numbers called signed-complement system.

Two methods in signed-complement system[19].

(1) 1's complement

(2) 2's complement

As an example, consider we have eight bits to represent a number.

Binary equivalent of +10 is 00001010. However, to represent -10, we can employ three methods.

Signed magnitude form: 10001010

Signed 1's – complement form: 11110101

Signed 2's – complement form: 11110110

1's complement for a negative number is obtained by logically reversing all the bits of its positive equivalent. +10 is 00001010 and reversing all the bits yield 11110101.

2's complement is obtained by adding 1 to 1's complement.

In computer arithmetic, 1's complement is seldom used for arithmetic operations.

Adding and subtracting numbers in 2's complement is much easier and doesn't require additional hardware circuitry to take care of sign bit.

3.3 Fixed Point Representation

As discussed in section Radix number system, we use a fixed radix point to distinguish between integer and fractional point.

Consider a system in which 7 bits are used to represent integer part and 8 bits are for fractional part, MSB 1 bit for sign.

Now, 15.375 in fixed point 2's complement number is as follows:

P = 15.375_{10}

= 00001111.01100000_2

-15.375 in 2's complement can be obtained by taking 1's complement of 15.375, reversing the bits and adding 1 to the result.

P = -15.375_{10}

= 11110000.10100000_2

3.4 Floating Point Representation

The floating-point notation is more flexible. Any number, P can be written as following:

$$P = \pm(1.b_1b_2...b_n)_2x2^m \tag{3.5}$$

is normalized representation of P. This representation is achieved by choosing the exponent ‘m’ such a way that the binary points float to the position after the first non-zero bit. This is simply the binary format of scientific notation.

To store a number in 32-bit number, we can reserve 1 bit for sign, 8 bits for the exponent and 23 bits for the fractional part of the number. The leading digit 1 is not stored and it is often called as ‘hidden bit’.

The fractional part is also referred with another name called ‘Mantissa’.

The exponent term has something called bias term, which is 127 for 8-bit exponent field. This bias ensures both positive and negative exponent values can be fit in the field.

Hence, the range for exponent field is -126 $\leq$ m $\leq$ 127.

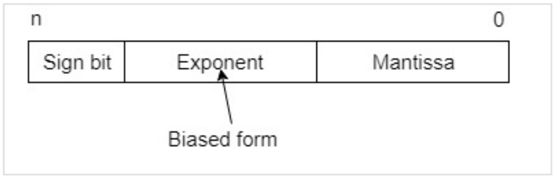


Figure 3.1: Floating point notation

This 32-bit representation of a decimal number is standardized as IEEE 754 notation.

For example, consider a number 263.3 and we need to represent this in 32-bit floating point representation.

Integer part 263: 100000111

Fractional part 0.3: 01001100110011...

Hence 263.3 is 100000111. 01001100110011...

Now, we need to shift the radix point to first '1' i.e., left shift the number by 8.

263.3 becomes 1.0000011101001100110011... x 2^8 which is scientific notation.

Now, sign bit is 0, because the number is a positive number.

Mantissa field is 23 bits: 0000011101001100110011

Exponent field is 8 bits: bias + 8 = 127 + 8 = 135: 10000111

So, 263.3 in IEEE 754 floating point representation is - 1 10000111 0000011101001100110011.

3.5 Fixed vs Floating Point format

Choosing one among Fixed and Floating point comes with its own benefits and tradeoffs. In general, floating point format is preferred when precision is of utmost importance whereas fixed point is chosen when area and timing in a VLSI chip is critical.

The exponentiation inherent in floating point format assures a much higher precision and larger dynamic range. Thus, both largest and smallest numbers can be represented using this format which is very essential and important for scientific computation.

The largest number that can be represented using IEEE 754 floating point notation is $(1 - 2^{-24})$ x 2^{128} and smallest number is 1.0 x 2^{-126}.

Whereas 32 - bit fixed point notation with 16 - bit for fractional , the largest number that can be represented is $(1 - 2^{-16})$ x 2^{16} and smallest number is 2^{-16}.

Thus, floating point format can fit a given number with much higher precision. However, the disadvantage for a floating-point number is that arithmetic operations are more costly and often takes more time. The reasons are as follows:

(1) When addition or subtraction operation needs to be performed, all the exponents have to be into account. Aligning the bits to common radix point is very costly requiring more hardware resources.

(2) During multiplication operation of two floating numbers, in addition to multiplying mantissas, exponents have to be added too.

In our application, we chose to implement our circuit in fixed point as we target an architecture with lowest area and power consumption possible at the cost of precision.

4. MATRIX VECTOR MULTIPLICATION

4.1 Efficient Hardware for computing Network Solution

With increased number of buses on a network, the matrix – vector operation becomes a big bottleneck with increased time complexity. In order to meet the latency requirements, it is required to instantiate multiple arithmetic units to perform all the computations simultaneously resulting in increased die area size. Since our application is targeted towards custom VLSI architecture with smallest chip area possible, it is essential to come up with efficient algorithms[20] to solve the problem. Below sections will discuss the proposed approaches to compute the matrix operation effectively.

4.1.1 Higher Order Difference

Multiplication operation is one of the most timing critical and area consuming component in a matrix vector computation. The more the number of bits it takes to represent a number, the larger the area it consumes to compute the multiplication operation. Thus, it is important to cut down the number of input bits the MAC(multiply and accumulate) module consumes. Reducing the bits in the integer part of a fixed point number reduces the upper bound whereas trimming down the fractional part compromises the precision. This higher order difference approach that we are about to discuss can solve this problem effectively without much sacrifice in the precision.
The algorithm is as follows:

Let us assume x_a,x_b, ... be a series of input vectors each of dimension n x 1.

Also, let y_a,y_b, ... be a series of output vectors each of dimension m x 1, where y_i = M. x_i

Here, M is a matrix of dimension m x n.

Now, Iteration 1:

$$y_a = M.x_a \tag{4.1}$$

In this step, store both y_a and x_a.

$$\begin{aligned} y_b &= M.x_b \\ &= y_a + M.(x_b - x_a) \\ &= y_a + p_a \end{aligned} \tag{4.2}$$

where, $p_a = M.(x_b - x_a)$

compute $M.(x_b - x_a)$ and use the previous stored result y_a to calculate y_b.

In this step, delete y_a, x_a and store y_b, p_a, $x_b - x_a$ and x_b.

Iteration 3:

$$\begin{aligned} y_c &= M.x_c \\ &= M.x_b + M.(x_c - x_b) \\ &= M.x_b + M.(x_b - x_a) + M.((x_c - x_b) - (x_b - x_a)) \\ &= y_b + p_a + M.((x_c - x_b) - (x_b - x_a)) \\ &= y_b + p_b \end{aligned} \tag{4.3}$$

where $p_b = M.(x_c - x_b)$

Now at this stage, we have already computed $M.x_b$ as y_b and $M.(x_b - x_a)$ as p_a.

All we need to do is calculate $M.((x_c - x_b) - (x_b - x_a))$

In this step, delete y_b, x_b and store y_c, p_b, $x_c - x_b$ and x_c.

Iteration 4:

$$\begin{aligned} y_d &= M.x_d \\ &= M.x_c + M.(x_d - x_c) \\ &= M.x_c + M.(x_c - x_b) + M.((x_d - x_c) - (x_c - x_b)) \\ &= y_c + p_b + M.((x_d - x_c) - (x_c - x_b)) \\ &= y_c + p_c \end{aligned} \tag{4.4}$$

where p_c = M.($x_d - x_c$)

The above sequence can be written as $y_i = y_{i-1} + p_{i-1}$ + M.(($x_i - x_{i-1}$) – ($x_{i-1} - x_{i-2}$)), where p_{i-1} = M.($x_{i-1} - x_{i-2}$).

Thus,

$$\begin{aligned} y_i &= y_{i-1} + \Delta y_{i-1} + M.((x_i - x_{i-1}) - (x_{i-1} - x_{i-2})) \\ &= y_{i-1} + \Delta y_{i-1} + M.\Delta^2 x_i \\ &= y_{old} + \Delta y_{old} + \Delta^2 y_i \end{aligned} \tag{4.5}$$

Hence, for every iteration, instead of computing the actual result, we can compute M. $\Delta^2 x_i$ and add the product to the previous stored results. All we need is initial values, y_0 and Δy_0 which can be loaded as part of initialization sequence.

The Equation 4.5, can be generalized to any order of differentiation as follows:

$$y_i = y_{i-1} + (\sum_{j=1}^{k-1} \Delta^j y_{i-1}) + \Delta^k y_i. \tag{4.6}$$

$$y_i = y_{i-1} + (\sum_{j=1}^{k-1} \Delta^j y_{i-1}) + M.\Delta^k x_i. \tag{4.7}$$

where k is the order of differentiation.

Now, why does this approach work to our application and how does it help in reducing the hardware resources?

The key here is that the input vectors in the network are current(I) with fundamental frequency 60Hz along with harmonics of small magnitude. As we are simulating a power system with high frequency sampling rate, the variation in the consecutive samples is minuscule and we can use this advantage to reduce the multiplication area size and thus number of standard cells required.

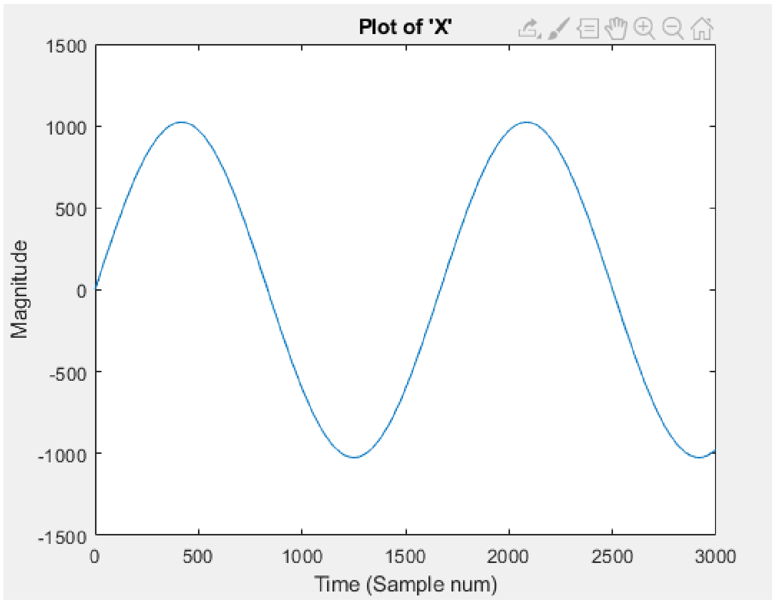


Figure 4.1: ’X’ waveform

For example, Fig 4.1 shows the waveform of a sample sine wave with magnitude 1000.

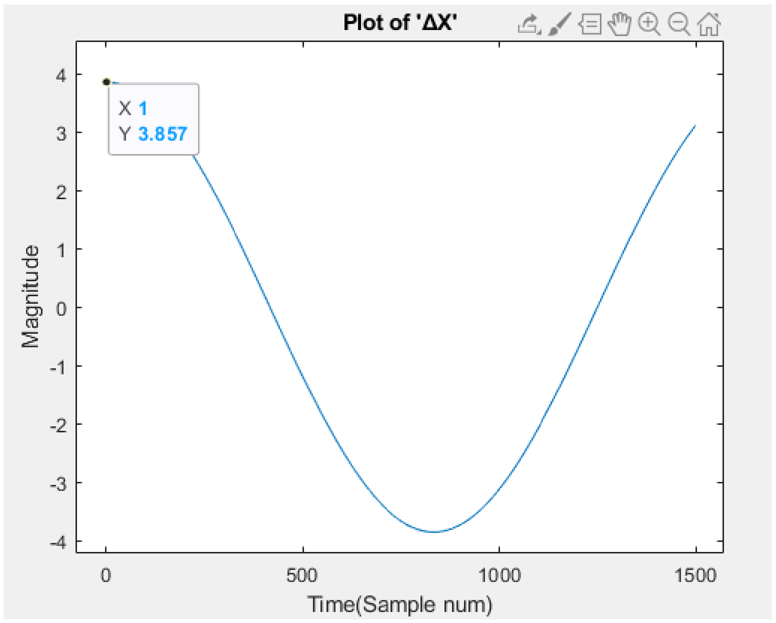


Figure 4.2: ’ΔX’ waveform

Fig 4.2., shows the magnitude of Δx.

If we need 10 bits (integer) + 32 (fractional) + 1 (sign) = 43 bits to store 'X'. We need only 2bits (integer) + 32 (fractional) + 1 (sign) = 35 bits to store Δx.

Another question, how do we know the value of k (In Eq 4.6), the order of the differentiation?

It depends on factors like amplitude of a signal, the number of bits we dedicate to represent the fractional part, frequency of the signal and sampling frequency etc.,

The next sub section will talk about this in detail on how to decide the required differentiation order.

4.1.2 Bit reduction scheme

In this sub-section, we derive a general formula to calculate the number of bits that can be reduced using the above stated approach.

For this, consider a sine wave, with frequency f , with max possible amplitude A, sampled at frequency f$_s$. Let p be the number of bits, we use to represent the fractional part.

Any two consecutive samples of the signal can be written as:

$$\begin{aligned} x(n) &= Asin(2\pi n\frac{f}{f_s}) \\ x(n+1) &= Asin(2\pi(n+1)\frac{f}{f_s}) \end{aligned} \tag{4.8}$$

x lies within the interval, [-A, A] and can be represented using p + log$_2$A bits

Now,

$$\begin{aligned} x(n+1) - x(n) &= Asin(2\pi(n+1)\frac{f}{f_s}) - Asin(2\pi n\frac{f}{f_s}) \\ \Delta x(n) &= (Acos(2\pi\frac{f}{f_s})sin(2\pi n\frac{f}{f_s}) + Asin(2\pi\frac{f}{f_s})cos(2\pi n\frac{f}{f_s})) - Asin(2\pi n\frac{f}{f_s}) \\ &= A(cos(2\pi\frac{f}{f_s}) - 1)sin(2\pi n\frac{f}{f_s}) + Asin(2\pi\frac{f}{f_s})cos(2\pi n\frac{f}{f_s}) \\ &= A2\pi\frac{f}{f_s}cos(2\pi n\frac{f}{f_s}) \end{aligned} \tag{4.9}$$

(when $f_s \gg f$, cos ($2\pi \frac{f}{f_s}$) $\approx$ 1 and sin($2\pi \frac{f}{f_s}$) $\approx 2\pi \frac{f}{f_s}$)

Δx lies within the interval, [-A $2\pi \frac{f}{f_s}$, A $2\pi \frac{f}{f_s}$] and can be represented using p + $\log_2$(A $2\pi \frac{f}{f_s}$) = $\log_2$(A) + p + $\log_2$($2\pi \frac{f}{f_s}$) bits.

Similarly, Δ^2x, the double differentiation of x lies within the interval [-A ($2\pi \frac{f}{f_s})^2$, A ($2\pi \frac{f}{f_s})^2$] and can be represented using $\log_2$(A) + p + 2. $\log_2$($2\pi \frac{f}{f_s}$) bits.

Thus, given a signal x(n) of frequency 'f' with sampling frequency f_s, its kth order difference signal Δ^kx lies within the interval [-A ($2\pi \frac{f}{f_s})^k$, A ($2\pi \frac{f}{f_s})^k$] can be represented using $\log_2$(A) + p + k. $\log_2$($2\pi \frac{f}{f_s}$) bits, ($\log_2$($2\pi \frac{f}{f_s}$) is a negative number).

From the above derivation, we can conclude that as the order of the differentiation is increased, the range of the number becomes smaller and it needs fewer bits to represent.

let us assume that m is a real number whose value is greater than 1.

Going back to Eq 4.7,

$$y_i = y_{i-1} + (\sum_{j=1}^{k-1} \Delta^j y_{i-1}) + m.\Delta^k x_i. \tag{4.10}$$

The value of the y_i changes, only when $|m.\Delta^k x_i| > 0$.

We know that, the smallest positive fractional number that can be represented using p bits is 2^{-p}.

Hence,

$$m.\Delta^k x_i. \geq 2^{-p} \tag{4.11}$$

$$m.A(2\pi\frac{f}{f_s})^k \geq 2^{-p} \tag{4.12}$$

If we choose m > 1, Eq 4.12 becomes

$$A(2\pi\frac{f}{f_s})^k \geq 2^{-p} \tag{4.13}$$

$$k \leq \frac{p + log_2 A}{log_2(2\pi \frac{f_s}{f})} \tag{4.14}$$

By choosing the order of differentiation k, as described in the Eq 4.13, we can say for sure that by applying the higher order difference technique to y = m.x, the relative error of calculating output y(n) doesn't get worsen.

Extending this hypothesis to the matrix vector multiplication y = M.x, let us assume we have a vector x whose elements are current(I) with Amplitudes A_1,A_1 ... A_n with frequency f.

Now, the order of the differentiation k has to be chosen as described in Eq 4.15

$$k \leq min\{\frac{p + log_2 A_1}{log_2(2\pi \frac{f_s}{f})}, \frac{p + log_2 A_2}{log_2(2\pi \frac{f_s}{f})}, ... \frac{p + log_2 A_n}{log_2(2\pi \frac{f_s}{f})}\} \tag{4.15}$$

The above derivation is applicable only to a pure sinusoid repeating signal, but the theory can be extended to the power systems which consists of complex waveforms. As we all know, input variables current(I) in a power system are of fundamental frequency 60 Hz and can be approximated into linear combination of sine and cosines of increasing harmonics using Fourier series.

So, let us assume that x is current(I) vector and each element 'v' has fundamental frequency f_{v1} with Amplitude A_{v1} and higher order harmonic frequencies f_{v2},$f_{v3}\ldots f_{vk}$ with amplitudes A_{v2},$A_{v3}\ldots A_{vk}$ respectively. Using Fourier series, the signal be can approximated as follows.

$$x(m) = A_0 + \sum_{n=1}^{n} A_k cos(\frac{2\pi f_n m}{f_s} + \theta_n) \tag{4.16}$$

Extending 4.12 here, we get

$$\sum_{n=1}^{N} (A_{1n}(2\pi \frac{f_{1n}}{f_s})^{k_1}) \geq 2^{-p} \tag{4.17}$$

$$\sum_{n=1}^{N} (A_{2n}(2\pi \frac{f_{2n}}{f_s})^{k_2}) \geq 2^{-p} \tag{4.18}$$

$$\sum_{n=1}^{N} (A_{vn}(2\pi \frac{f_{vn}}{f_s})^{k_v}) \geq 2^{-p} \tag{4.19}$$

solve for k_1, k_2 ... k_v in the above equations and the order of differentiation is

$$k = min\{k_1, k_2...k_v\} \tag{4.20}$$

4.2 Matrix vector multiplication

To give some context on the matrices x, y, U, and V, the column vector 'x' represents a series of samples of current (I) from different 'n' buses of a large power system. whereas 'y' represents voltage (v) at different 'm' buses of the same power system due to current (I). U and V are the low rank approximated matrices which are derived from conductance matrix G^{-1} [3].

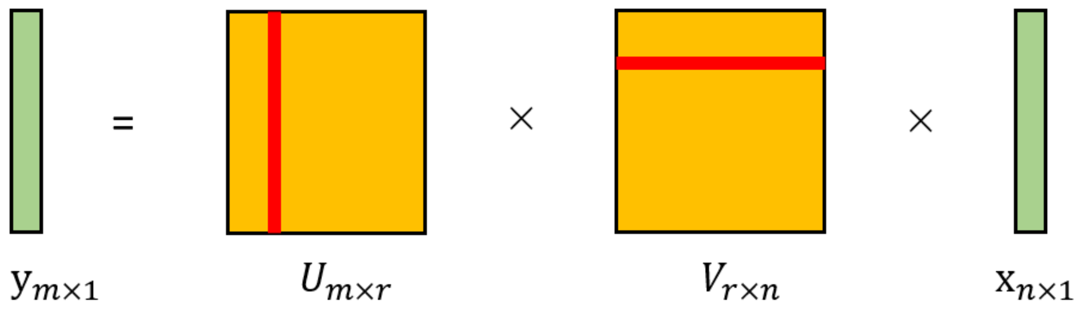


Figure 4.3: Matrix Vector Multiplication

The implementation example shown in the following section assumes 'm' value to be 12 (dimension of y vector), 'n' value 12 (dimension of x vector), r value as 6, time step 20us and the order of differentiation to be 2.

In the simulation, the time step has chosen to be 20us i.e., sampling rate is 50KHz. Simulation results have shown that we need 1 bit for sign, 16 bits to represent integer part and 32 bits to represent the fractional part of input signal current(I).

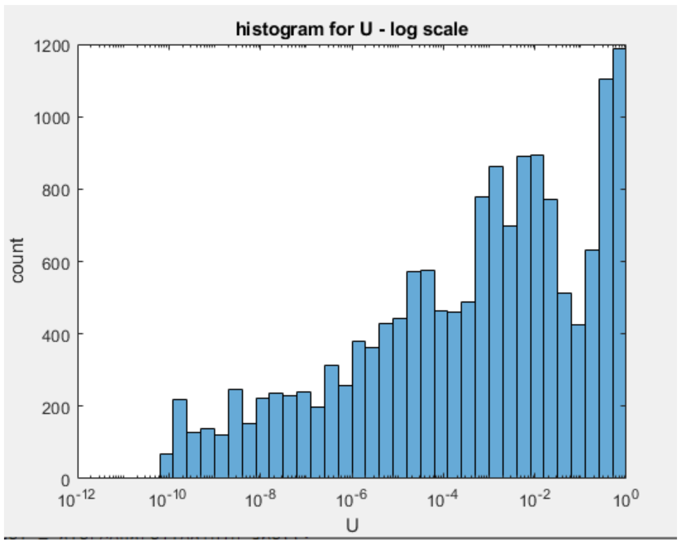


Figure 4.4: Histogram for U matrix elements

Also, all the values of matrix U and V are found to have magnitude <1. Hence, we take 32 bits to represent. 31 bits for fractional and 1 bit for sign.

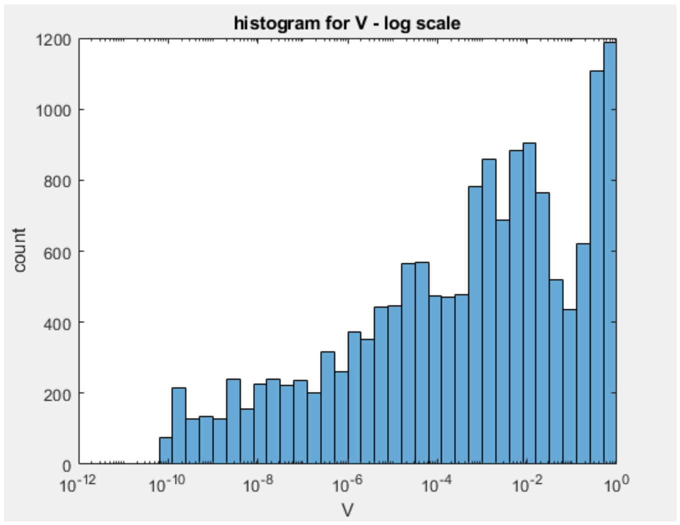


Figure 4.5: Histogram for V matrix elements

Current(I) at 12 different buses as shown in the Figure 4.6 has been taken as an example for rest of the thesis.

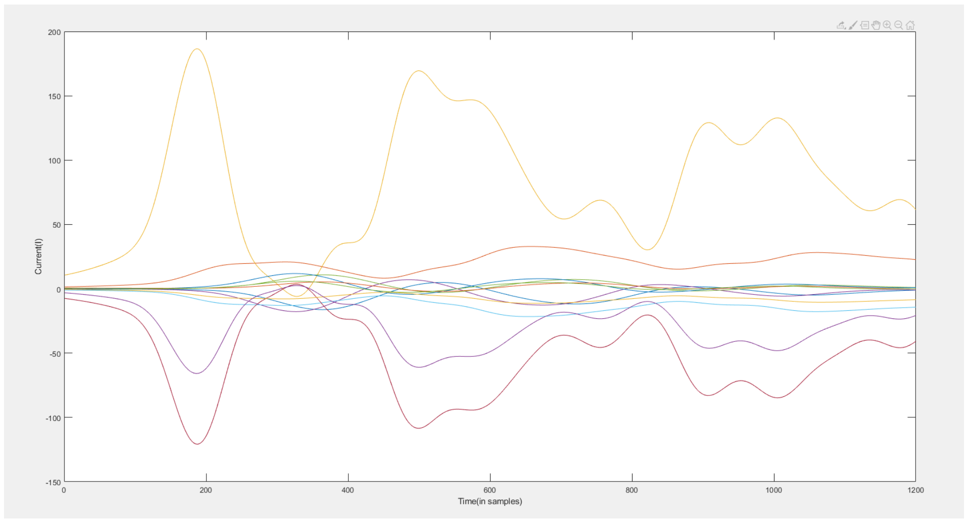


Figure 4.6: Current(I) at 12 different buses

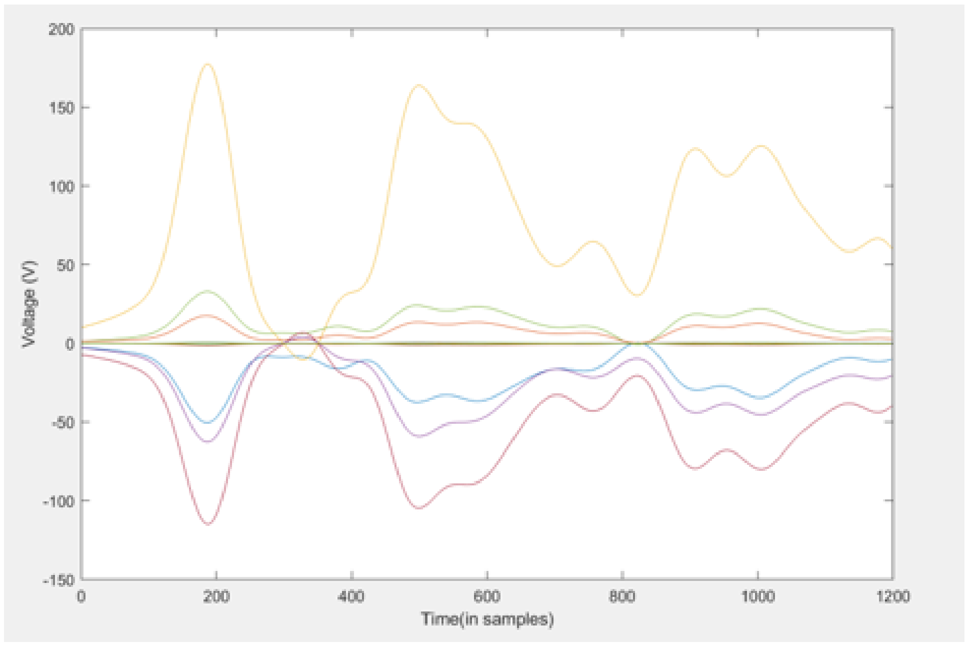
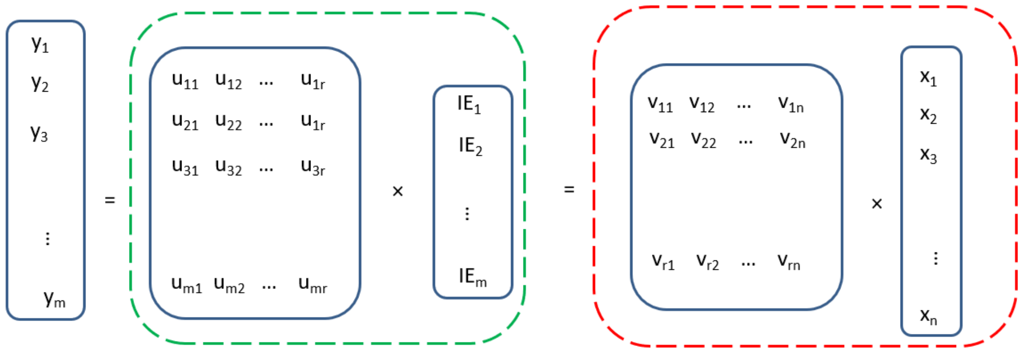


Figure 4.7: Voltage(V) at 12 different buses

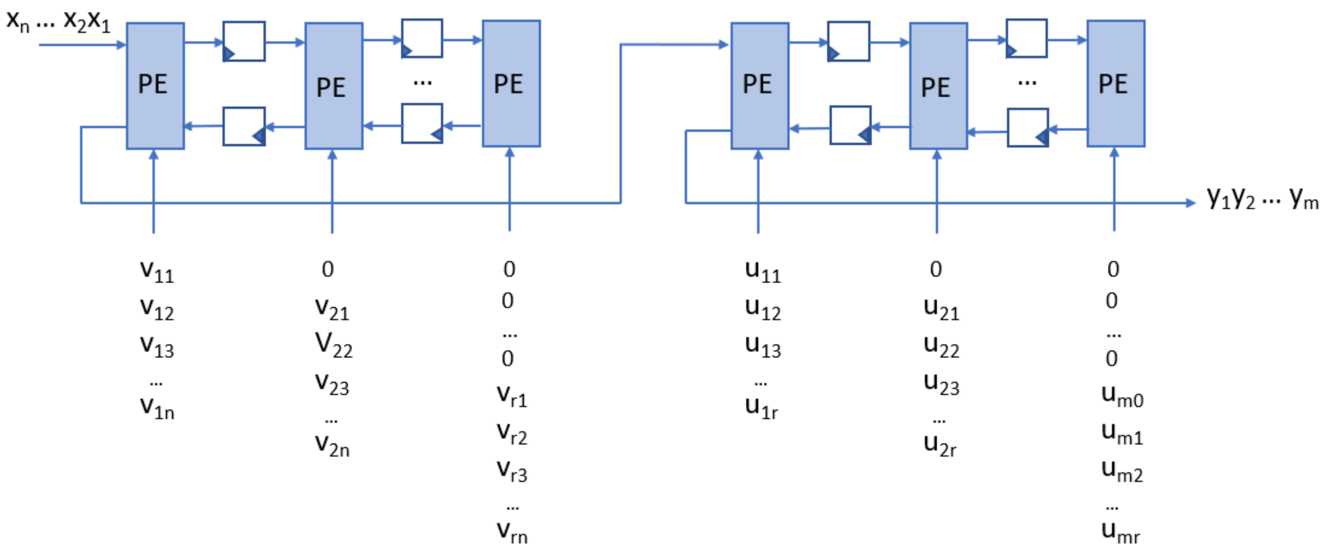
4.2.1 Previous Work

In the approach mentioned in [21], the matrix - vector multiplication is achieved through systolic array architecture as shown in the below figures.



y_1 y_2 y_3 ⋮ y_m
=
u_{11} u_{12} … u_{1r}
u_{21} u_{22} … u_{1r}
u_{31} u_{32} … u_{3r}
u_{m1} u_{m2} … u_{mr}
×
IE_1 IE_2 ⋮ IE_m
=
v_{11} v_{12} … v_{1n}
v_{21} v_{22} … v_{2n}
v_{r1} v_{r2} … v_{rn}
×
x_1 x_2 x_3 ⋮ x_n

Figure 4.8: Matrix Vector Multiplication using Karra's approach



$x_n \ldots x_2 x_1$
PE
PE
…
PE
PE
PE
…
PE
$y_1 y_2 \ldots y_m$
v_{11} v_{12} v_{13} … v_{1n}
0 v_{21} v_{22} v_{23} … v_{2n}
0 0 … 0 v_{r1} v_{r2} v_{r3} … v_{rn}
u_{11} u_{12} u_{13} … u_{1r}
0 u_{21} u_{22} u_{23} … u_{2r}
0 0 … 0 u_{m0} u_{m1} u_{m2} … u_{mr}

Figure 4.9: Circuit Design for Matrix Vector Multiplication using Karra's approach

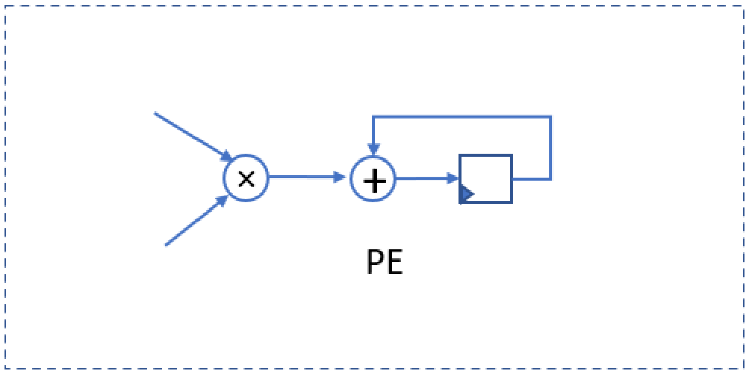


Figure 4.10: Processing Element

In the approach mentioned in [9], to optimize the the data from input matrices U and V have been read from memory for exactly once. By simultaneously reading one column of matrix U and one row of matrix V, and performing all multiply operations based on those values , optimal data re-use is achieved. Data read in this sequence allows one partial product term of every element to be computed in output matrix C per clock cycle.

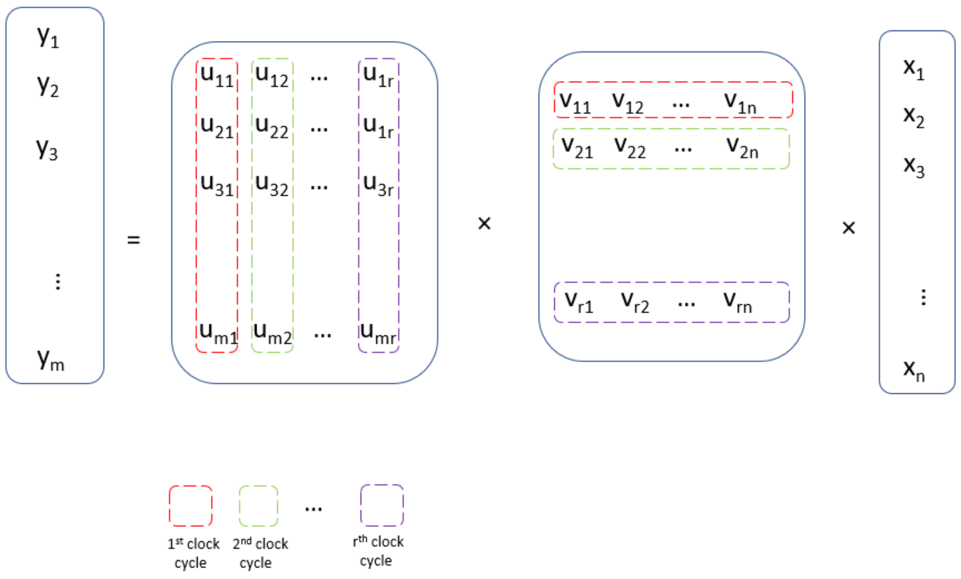
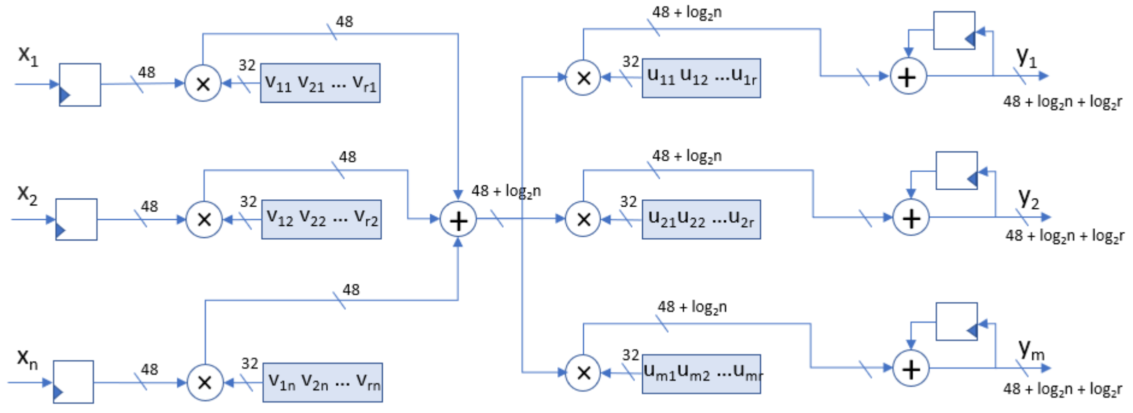


Figure 4.11: Matrix vector multiplication using Khatri's approach



x_1
x_2
x_n
48
32
$v_{11}\ v_{21} \dots v_{r1}$
$v_{12}\ v_{22} \dots v_{r2}$
$v_{1n}\ v_{2n} \dots v_{rn}$
48 + log₂n
$u_{11}\ u_{12} \dots u_{1r}$
$u_{21}u_{22} \dots u_{2r}$
$u_{m1}u_{m2} \dots u_{mr}$
y_1
y_2
y_m
48 + log₂n + log₂r

Figure 4.12: Circuit Design using Khatri's approach

4.2.2 Comparison between two architectures for matrix vector multiplication

Let q be the number of bits we use to represent U and V.

Let p be the number of bits we use to represent X.

Compute y = U(ΣV.x)	Multipliers	Adders	SRAM cells	Flipflops	Latency
Karra's method	(m+r)	(m+r)	q*r*(m+n)	p*(m+r)	(m + n + r)
Khatri's method	(m+n)	(m+n)	q*r*(m+n)	p*(m+n)	r

For our network solution, we have chosen Khatri's method and improved upon its architecture as it offers low latency.

4.2.3 Matrix vector multiplication using higher order difference

$y_a = \sum_{i=1}^{r} u_i(v_i x_a)$

$y_b = \sum_{i=1}^{r} u_i(v_i x_b)$

Subtracting the above two equations, we get

$y_b - y_a = \sum_{i=1}^{r} u_i(v_i(x_b - x_a))$

y_b, $y_b - y_a$, x_b, $x_b - x_a$ vectors are stored as initial conditions in the flipflops.

Now Eq 4.26, can be written as follows

$y_k = y_{k-1} + (y_{k-1} - y_{k-2}) + \sum_{i=1}^{r} u_i(v_i((x_k - x_{k-1}) - (x_{k-1} - x_{k-2})))$

$$y = y_{old} + \Delta y_{old} + \Delta^2 y_k \tag{4.21}$$

$$\Delta^2 y_k = \sum_{i=1}^{r} u_i(u_i x_k) \tag{4.22}$$

The above two equations can be implemented as follows:

We divide the hardware design into four parts:

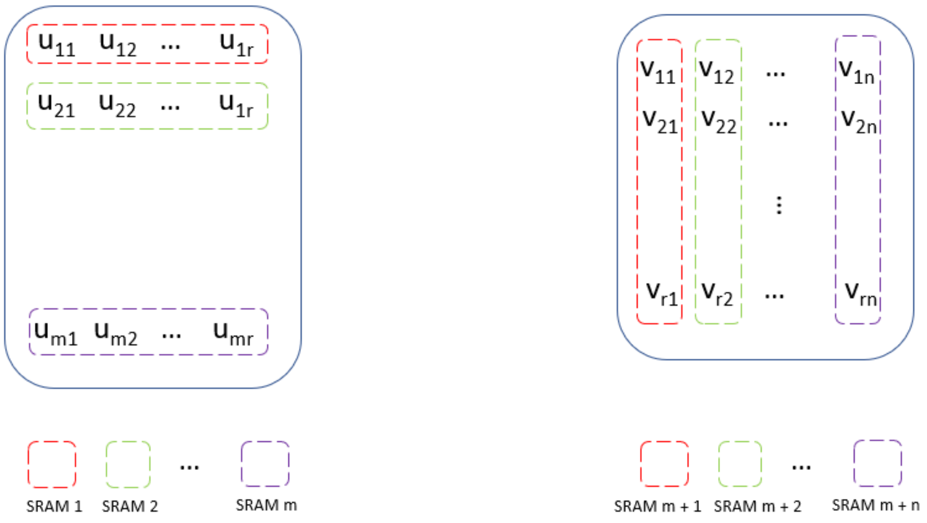
(i) Storing U and V in SRAM memory.

(ii) computing $\Delta^2 x_k$ using subtractors.

(iii) computing v$_i$ $\Delta^2 x_k$ and then $\sum_{i=1}^{r} u_i.(v_i \Delta^2 x_k)$ using multipliers and adders.

(iv) computing y$_k$ using adders.

4.2.4 Storing U and V



u_{11} u_{12} ... u_{1r}
u_{21} u_{22} ... u_{1r}
u_{m1} u_{m2} ... u_{mr}
v_{11} v_{12} ... v_{1n}
v_{21} v_{22} ... v_{2n}
v_{r1} v_{r2} ... v_{rn}
SRAM 1
SRAM 2
SRAM m
SRAM m + 1
SRAM m + 2
SRAM m + n

Figure 4.13: Storing U and V

To store U and V, we need m+n SRAM memory modules. m of them stores each row of U matrix and n of them store columns of V.

Each SRAM module stores ‘r’ values each of 32 – bit. The following diagrams shows the cell arrangements.

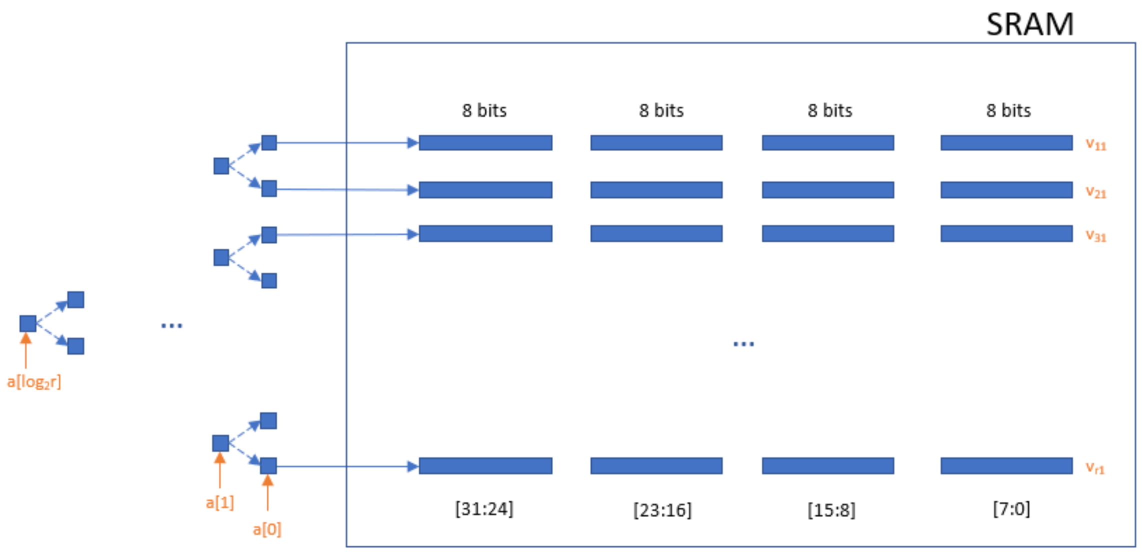


Figure 4.14: SRAM to store U and V

4.2.5 Computing $\Delta^2 x_k$ using subtractors

$\Delta^2 x_k = (x_k - x_{k-1}) - (x_{k-1} - xk - 2)$

For this computation, we need

n, 48- bit flipflops to store x vector and n, 41 – bit flipflops to store Δx vector. n, 48- bit subtractors to compute (x$_k$ – x$_{k-1}$) and n, 41- bit subtractors to compute (x$_k$ – x$_{k-1}$) – (x$_{k-1}$ – x$_{k-2}$).

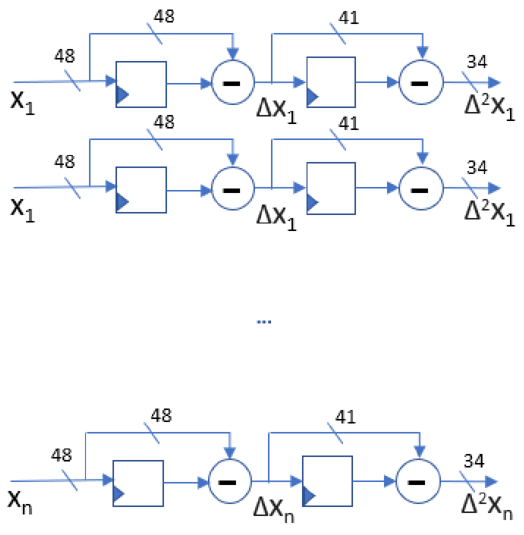


Figure 4.15: Computing $\Delta^2 x_k$ using subtractors

4.2.6 Computing v$_i.\Delta^2 x_k$ and $\sum_{i=1}^{r} u_i.(v_i \Delta^2 x_k)$ then using multipliers and adders

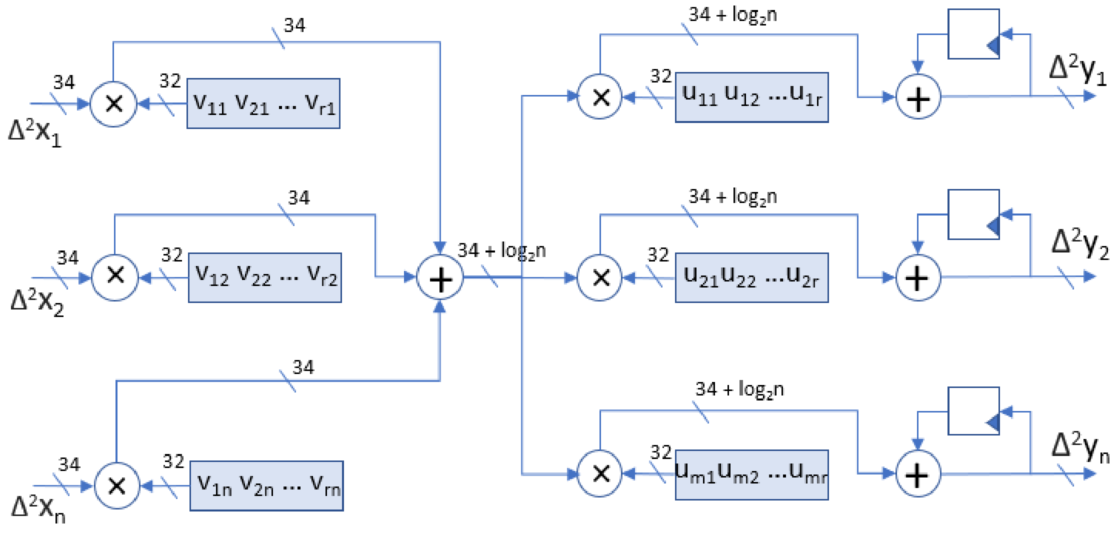


Figure 4.16: Computing v$_i \Delta^2 x_k$ and $\sum_{i=1}^{r} u_i.(v_i \Delta^2 x_k)$ then using multipliers and adders

First, a row in V matrix and Δ^2x vector are multiplied and added resulting in an intermediate element. This intermediate element is multiplied with all the elements in the column of U matrix to get the partial sum of $\Delta^2 y$.

$$I.E = \sum_{i=1}^{r} (v_i x) \tag{4.23}$$

$$\Delta^2_{partial} = u_i.I.E \tag{4.24}$$

$$\Delta^2 y_k = \sum_{i=1}^{r} \Delta^2 y_{partial} \tag{4.25}$$

4.2.7 Computing $\Delta^2 y_k$ using adders

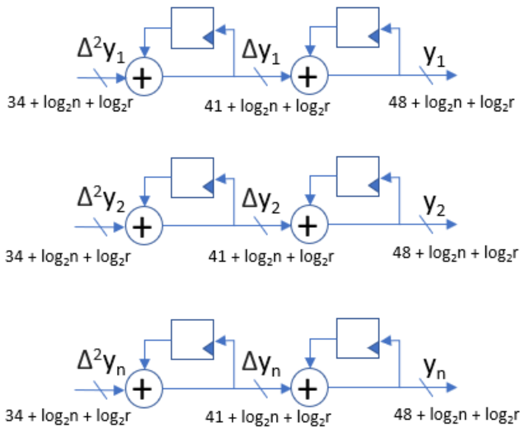


Figure 4.17: Computing $\Delta^2 y_k$ using adders

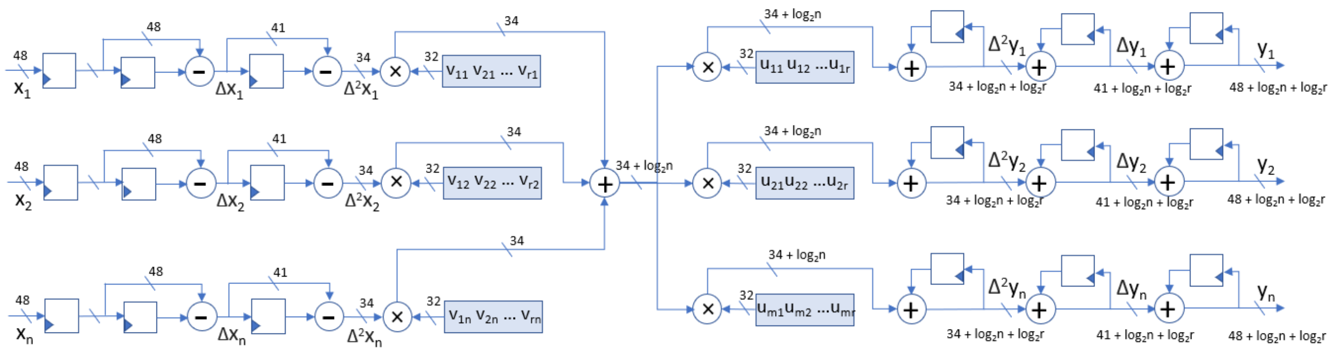
y = $y_{old} + \Delta y_{old} + \Delta^2 y_k$

For computation we need

m, 48 + $\log_2$n + $\log_2$r - bit flipflops to store y_{old} vector and n, 41 + $\log_2$n + $\log_2$r bit flipflops to store Δy_{old} vector.

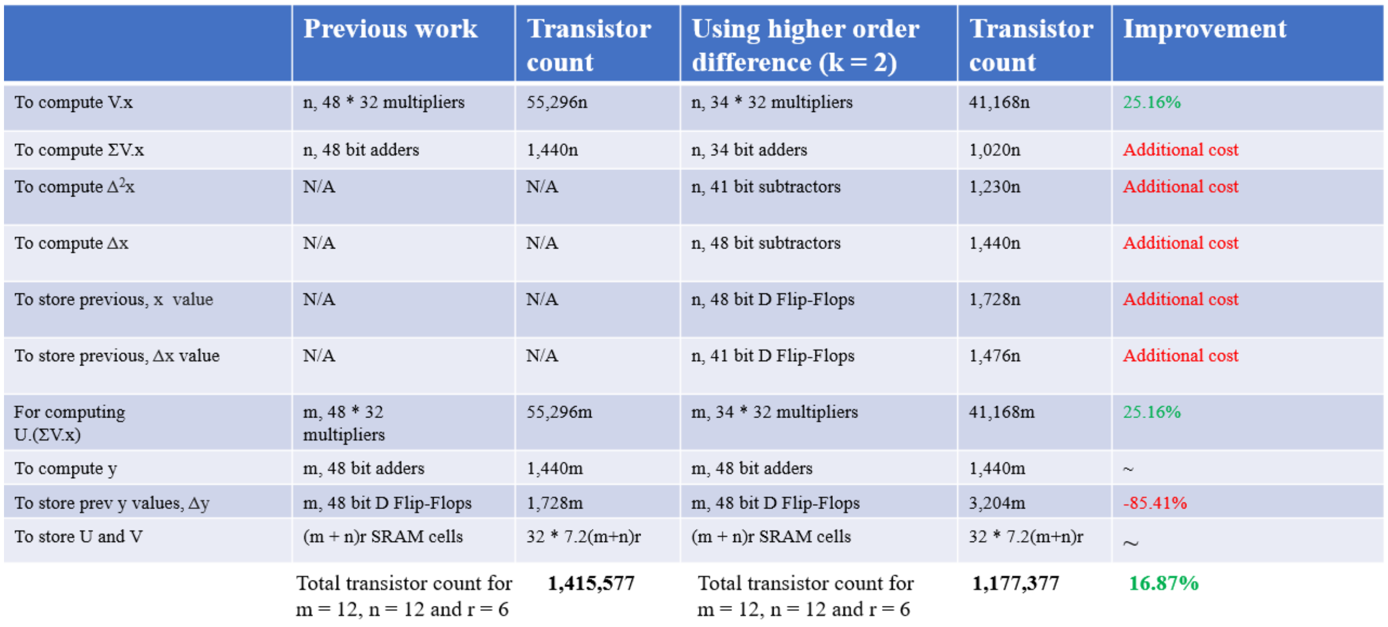
m, 41 + $\log_2$n + $\log_2$r bit adder to compute $\Delta y_{old} + \Delta^2 y_k$ and n, 48 + $\log_2$n + $\log_2$r bit adders to compute y.

4.2.8 Final Circuit



48
48
41
34
32
34
x_1
Δx_1
$\Delta^2 x_1$
$v_{11}\ v_{21} \dots v_{r1}$
x_2
Δx_2
$\Delta^2 x_2$
$v_{12}\ v_{22} \dots v_{r2}$
x_n
Δx_n
$\Delta^2 x_n$
$v_{1n}\ v_{2n} \dots v_{rn}$
34 + $\log_2 n$
$u_{11}\ u_{12} \dots u_{1r}$
$u_{21} u_{22} \dots u_{2r}$
$u_{m1} u_{m2} \dots u_{mr}$
$\Delta^2 y_1$
Δy_1
y_1
$\Delta^2 y_2$
Δy_2
y_2
$\Delta^2 y_n$
Δy_n
y_n
34 + $\log_2 n$ + $\log_2 r$
41 + $\log_2 n$ + $\log_2 r$
48 + $\log_2 n$ + $\log_2 r$

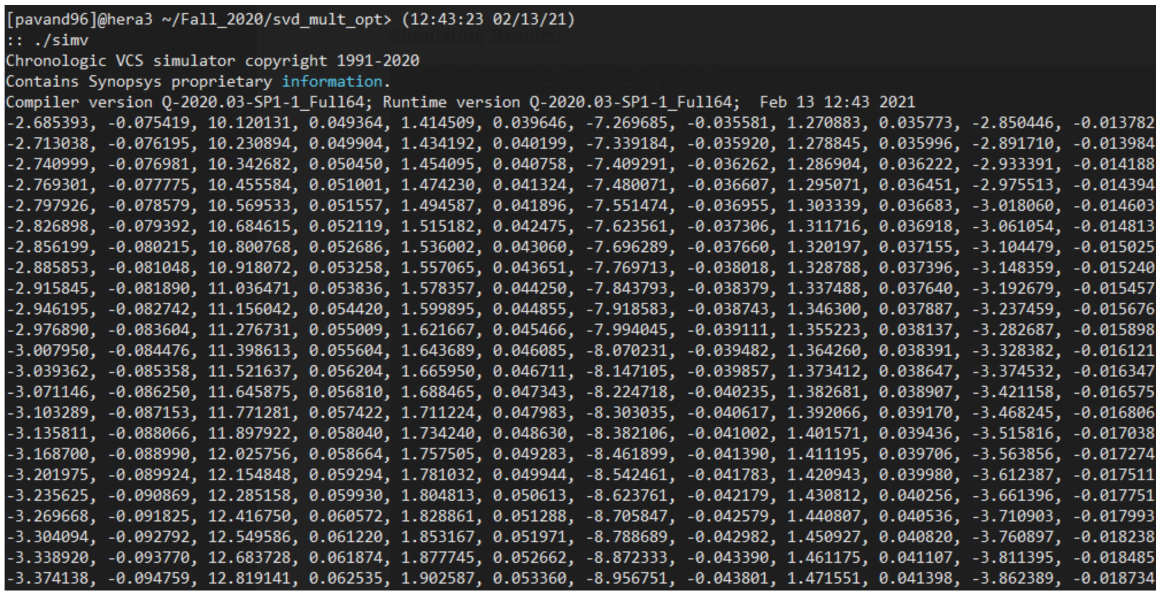
Figure 4.18: Final Circuit



	Previous work	Transistor count	Using higher order difference (k = 2)	Transistor count	Improvement
To compute V.x	n, 48 * 32 multipliers	55,296n	n, 34 * 32 multipliers	41,168n	25.16%
To compute ΣV.x	n, 48 bit adders	1,440n	n, 34 bit adders	1,020n	Additional cost
To compute Δ^2x	N/A	N/A	n, 41 bit subtractors	1,230n	Additional cost
To compute Δx	N/A	N/A	n, 48 bit subtractors	1,440n	Additional cost
To store previous, x value	N/A	N/A	n, 48 bit D Flip-Flops	1,728n	Additional cost
To store previous, Δx value	N/A	N/A	n, 41 bit D Flip-Flops	1,476n	Additional cost
For computing U.(ΣV.x)	m, 48 * 32 multipliers	55,296m	m, 34 * 32 multipliers	41,168m	25.16%
To compute y	m, 48 bit adders	1,440m	m, 48 bit adders	1,440m	~
To store prev y values, Δy	m, 48 bit D Flip-Flops	1,728m	m, 48 bit D Flip-Flops	3,204m	-85.41%
To store U and V	(m + n)r SRAM cells	32 * 7.2(m+n)r	(m + n)r SRAM cells	32 * 7.2(m+n)r	~
	Total transistor count for m = 12, n = 12 and r = 6	**1,415,577**	Total transistor count for m = 12, n = 12 and r = 6	**1,177,377**	**16.87%**

Figure 4.19: Comparison of hardware resources

4.2.9 RTL Simulation Results



```
[pavand96]@hera3 ~/Fall_2020/svd_mult_opt> (12:43:23 02/13/21)
:: ./simv
Chronologic VCS simulator copyright 1991-2020
Contains Synopsys proprietary information.
Compiler version Q-2020.03-SP1-1_Full64; Runtime version Q-2020.03-SP1-1_Full64;  Feb 13 12:43 2021
-2.685393, -0.075419, 10.120131, 0.049364, 1.414509, 0.039646, -7.269685, -0.035581, 1.270883, 0.035773, -2.850446, -0.013782
-2.713038, -0.076195, 10.230894, 0.049904, 1.434192, 0.040199, -7.339184, -0.035920, 1.278845, 0.035996, -2.891710, -0.013984
-2.740999, -0.076981, 10.342682, 0.050450, 1.454095, 0.040758, -7.409291, -0.036262, 1.286904, 0.036222, -2.933391, -0.014188
-2.769301, -0.077775, 10.455584, 0.051001, 1.474230, 0.041324, -7.480071, -0.036607, 1.295071, 0.036451, -2.975513, -0.014394
-2.797926, -0.078579, 10.569533, 0.051557, 1.494587, 0.041896, -7.551474, -0.036955, 1.303339, 0.036683, -3.018060, -0.014603
-2.826898, -0.079392, 10.684615, 0.052119, 1.515182, 0.042475, -7.623561, -0.037306, 1.311716, 0.036918, -3.061054, -0.014813
-2.856199, -0.080215, 10.800768, 0.052686, 1.536002, 0.043060, -7.696289, -0.037660, 1.320197, 0.037155, -3.104479, -0.015025
-2.885853, -0.081048, 10.918072, 0.053258, 1.557065, 0.043651, -7.769713, -0.038018, 1.328788, 0.037396, -3.148359, -0.015240
-2.915845, -0.081890, 11.036471, 0.053836, 1.578357, 0.044250, -7.843793, -0.038379, 1.337488, 0.037640, -3.192679, -0.015457
-2.946195, -0.082742, 11.156042, 0.054420, 1.599895, 0.044855, -7.918583, -0.038743, 1.346300, 0.037887, -3.237459, -0.015676
-2.976890, -0.083604, 11.276731, 0.055009, 1.621667, 0.045466, -7.994045, -0.039111, 1.355223, 0.038137, -3.282687, -0.015898
-3.007950, -0.084476, 11.398613, 0.055604, 1.643689, 0.046085, -8.070231, -0.039482, 1.364260, 0.038391, -3.328382, -0.016121
-3.039362, -0.085358, 11.521637, 0.056204, 1.665950, 0.046711, -8.147105, -0.039857, 1.373412, 0.038647, -3.374532, -0.016347
-3.071146, -0.086250, 11.645875, 0.056810, 1.688465, 0.047343, -8.224718, -0.040235, 1.382681, 0.038907, -3.421158, -0.016575
-3.103289, -0.087153, 11.771281, 0.057422, 1.711224, 0.047983, -8.303035, -0.040617, 1.392066, 0.039170, -3.468245, -0.016806
-3.135811, -0.088066, 11.897922, 0.058040, 1.734240, 0.048630, -8.382106, -0.041002, 1.401571, 0.039436, -3.515816, -0.017038
-3.168700, -0.088990, 12.025756, 0.058664, 1.757505, 0.049283, -8.461899, -0.041390, 1.411195, 0.039706, -3.563856, -0.017274
-3.201975, -0.089924, 12.154848, 0.059294, 1.781032, 0.049944, -8.542461, -0.041783, 1.420943, 0.039980, -3.612387, -0.017511
-3.235625, -0.090869, 12.285158, 0.059930, 1.804813, 0.050613, -8.623761, -0.042179, 1.430812, 0.040256, -3.661396, -0.017751
-3.269668, -0.091825, 12.416750, 0.060572, 1.828861, 0.051288, -8.705847, -0.042579, 1.440807, 0.040536, -3.710903, -0.017993
-3.304094, -0.092792, 12.549586, 0.061220, 1.853167, 0.051971, -8.788689, -0.042982, 1.450927, 0.040820, -3.760897, -0.018238
-3.338920, -0.093770, 12.683728, 0.061874, 1.877745, 0.052662, -8.872333, -0.043390, 1.461175, 0.041107, -3.811395, -0.018485
-3.374138, -0.094759, 12.819141, 0.062535, 1.902587, 0.053360, -8.956751, -0.043801, 1.471551, 0.041398, -3.862389, -0.018734
```

Figure 4.20: RTL Simulation Results

4.2.10 Comparison between Floating and Fixed Point implementation

The following plots are Voltage (V), Absolute Error and Relative Error for a particular bus during our simulation.

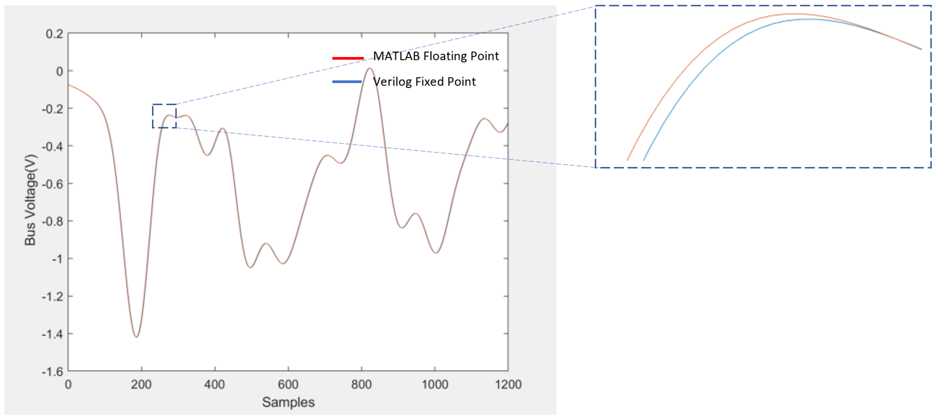


Figure 4.21: Comparison between Matlab and Verilog

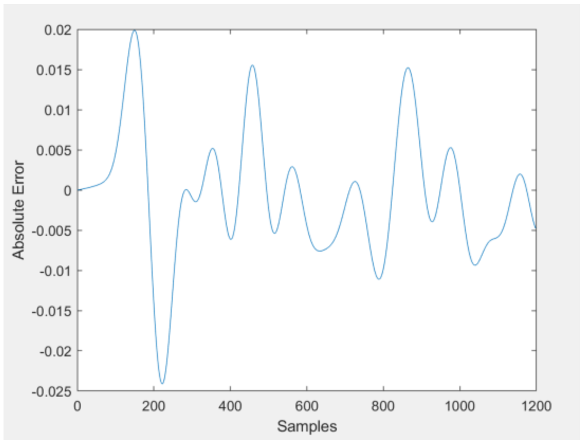


Figure 4.22: Absolute Error between Floating and Fixed Point implementation

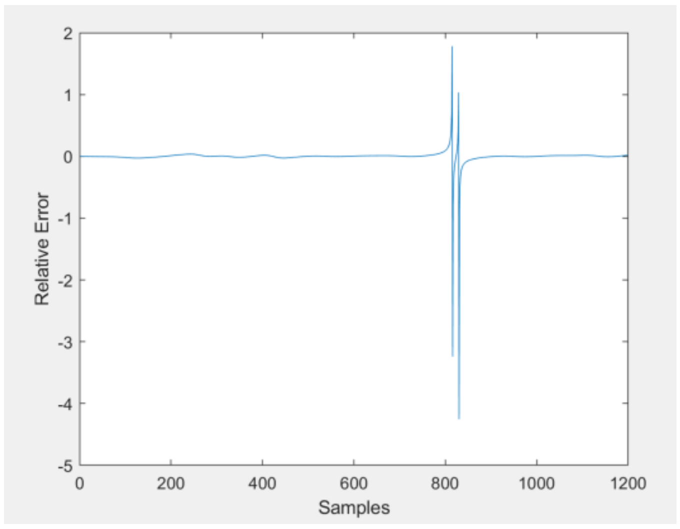


Figure 4.23: Relative Error between Floating and Fixed Point implementation

Relative error spikes up, when the Voltage(V) approaches 0, which is expected.

4.3 FPGA Implementation

4.3.1 Integrating and Simulating the circuit on FPGA

The FPGA prototyping of the hardware has been done on the Zynq platform[22], Zybo Z7-20 board[23].

The defining features of Zynq:

1. Processing System (PS): Dual – core ARM cortex – A9 CPU

2. Programmable logic (PL) – Consists of LUTs, BRAM and Flipflops.

3. Advanced Extensible Interface (AXI) : Low latency and High bandwidth between PS and PL.

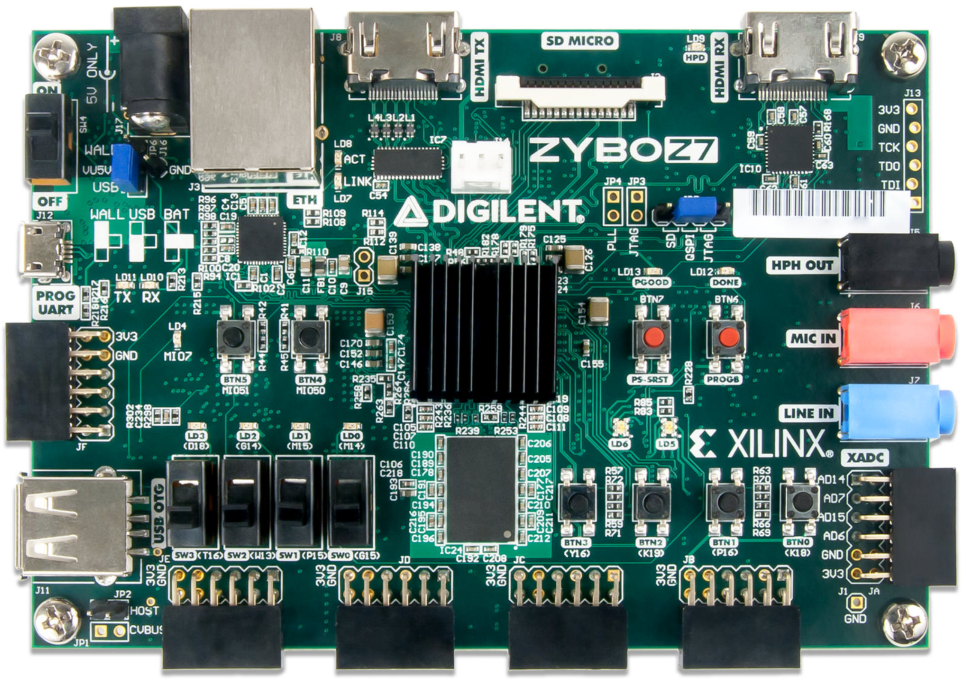


Figure 4.24: Zynq Zybo Z7-20 board

The Xilinx FPGA supports three types of AXI interfaces

(i) AXI – Lite interface

(ii) AXI – full interface

(iii) AXI – stream interface

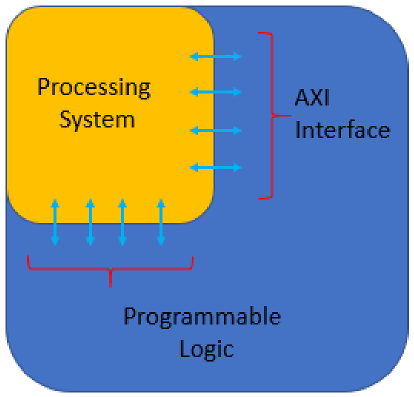


Figure 4.25: SOC - Zybo Z7-20 board

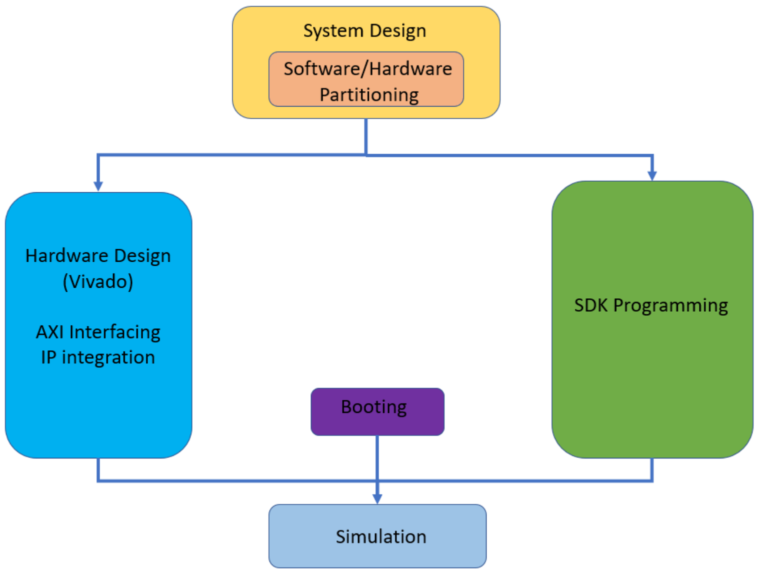


Figure 4.26: FPGA Implementation flow

Prototyping on FPGA[24] can be divided into two parts

IP block design (Xilinx Vivado)

- IP block creation
- AXI interfacing.
- IP integration
- HDL wrapper
- Generate Bitstream

ARM programming (Xilinx SDK)

- ARM programming
- Launching on hardware

For our thesis, we perform the simulation in four steps

1. First, we store the instructions and data in the form of micro codes in the DDR3 memory available on the FPGA board.

2. With the help of Zynq Processor, we configure the DMA controller, the starting address and length of the transfer.

3. we take back the results to the DDR3 memory.

4. Print the results on the terminal.

4.3.2 System Architecture

In the following section, we discuss in detail different types of system architectures we explored for this thesis and which one suits the best for faster operation[25].

In the System Architecture 1 as shown in fig 5.4, the processor takes the control of the system bus, requests the data from the Memory and sends the data to the peripheral.

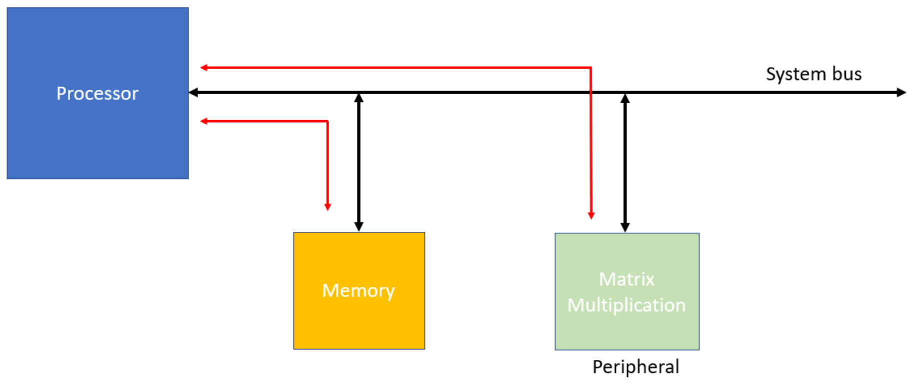


Figure 4.27: System Architecture 1 – Processor intervention is needed

As a whole, two memory transfer operations are required in this architecture -

- Memory to the Processor (Processor is the master and memory is the slave)
- Processor to the peripheral. (Processor is the master and peripheral is the slave)

There are two disadvantages in this architecture:

- It requires the intervention of the processor for every memory access.
- Both memory transfer operations cannot happen at the same time as we share the system bus.

In the second architecture 2, we add DMA controller (DMA stands for direct memory access) to take control of the memory transfers.

With DMA controller at our disposal, the CPU initiates the transaction and the length of the transfer. The controller then sends the data to the peripheral accordingly without any further intervention from the CPU.

Two memory operations are required -

- Memory to the DMA controller (Memory is the slave and DMA controller is the master)
- DMA controller to the peripheral. (DMA controller is the master and peripheral is the slave)

Since all the transfers happens through the system bus, either one of the memory operations must happen at time, limiting the throughput. This resource allocation of the system bus to memory and the processor is usually done by inbuilt 'Arbitration logic'.

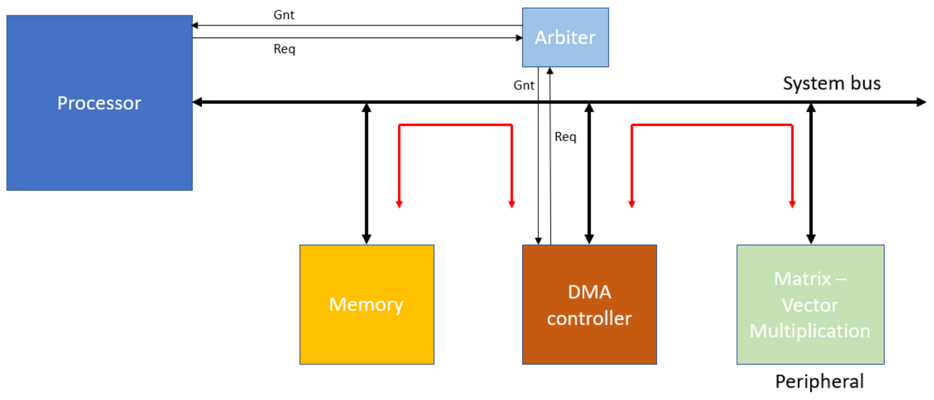


Figure 4.28: System Architecture 2 – Only one operation at a time

Although this architecture eliminates the processor intervention for every transfer, it still limits the throughput as arbitration logic for the data transfer is necessary.

In the System Architecture 3, the communication between the memory and DMA controller happens through the system bus. But the transfer between the DMA controller and peripheral happens through AXI stream protocol which is direct point to point based maximizing the throughput.

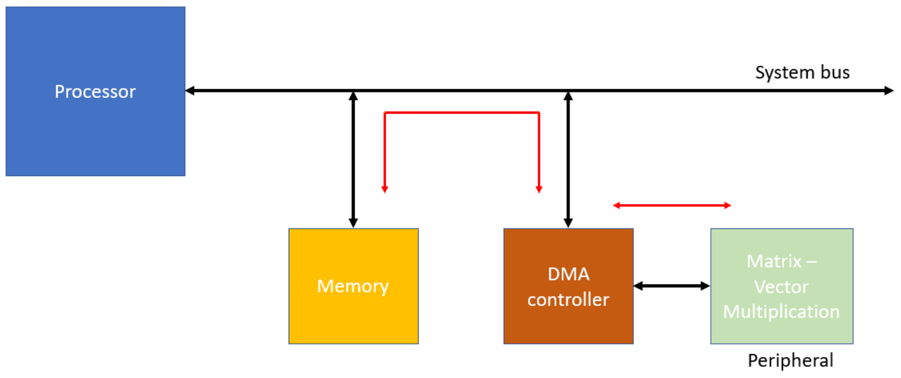


Figure 4.29: System Architecture 3 – Two operations happen concurrently

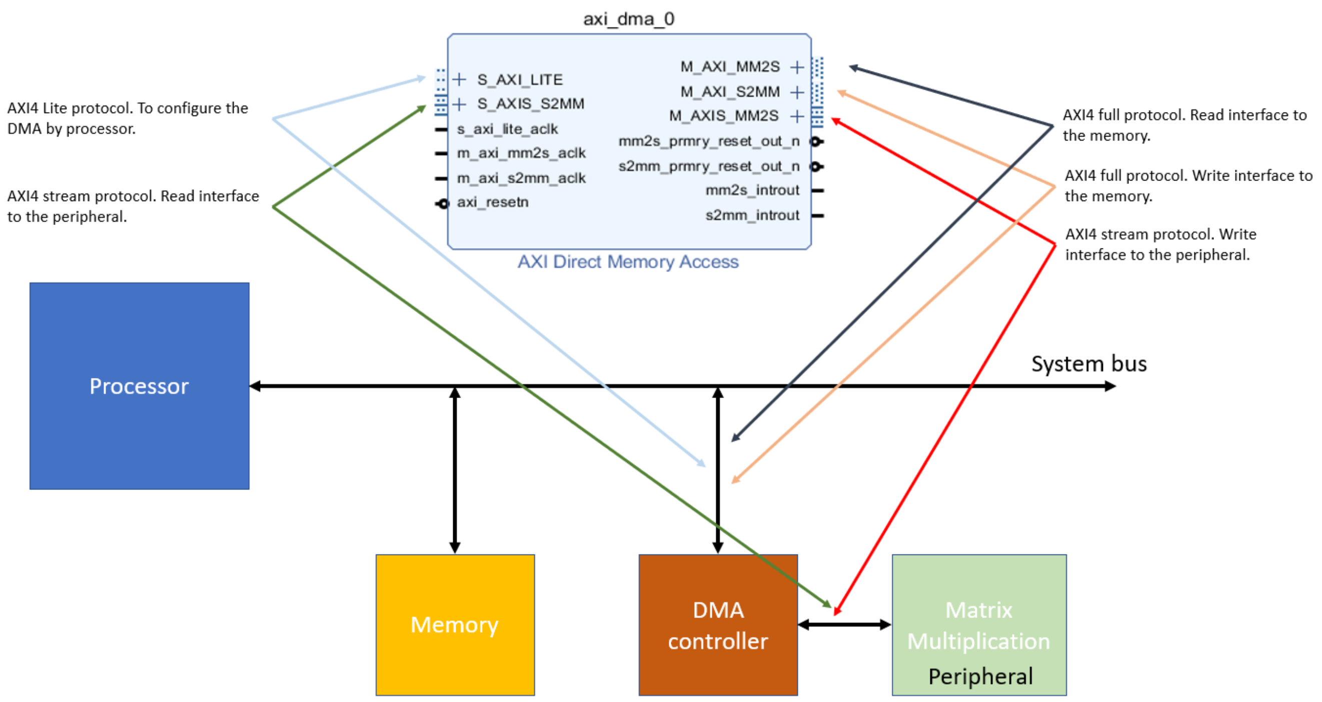

axi_dma_0
S_AXI_LITE
S_AXIS_S2MM
s_axi_lite_aclk
m_axi_mm2s_aclk
m_axi_s2mm_aclk
axi_resetn
M_AXI_MM2S
M_AXI_S2MM
M_AXIS_MM2S
mm2s_prmry_reset_out_n
s2mm_prmry_reset_out_n
mm2s_introut
s2mm_introut
AXI Direct Memory Access
AXI4 Lite protocol. To configure the DMA by processor.
AXI4 stream protocol. Read interface to the peripheral.
AXI4 full protocol. Read interface to the memory.
AXI4 full protocol. Write interface to the memory.
AXI4 stream protocol. Write interface to the peripheral.
System bus
Processor
Memory
DMA controller
Matrix Multiplication Peripheral

Figure 4.30: Communication through DMA

4.3.3 Matrix Multiplication Peripheral

Since our FPGA is a 32-bit machine and has very little PL fabric logic resources, we used only 32-bit bus to load the inputs into our peripheral from the processing system (PS).

As discussed in the circuit diagram of the multiplication peripheral in the previous section, we have taken M = 12, N = 12 and r = 6 for this example. Hence, we have to load current(I) for 12 different buses of 48 bits each.

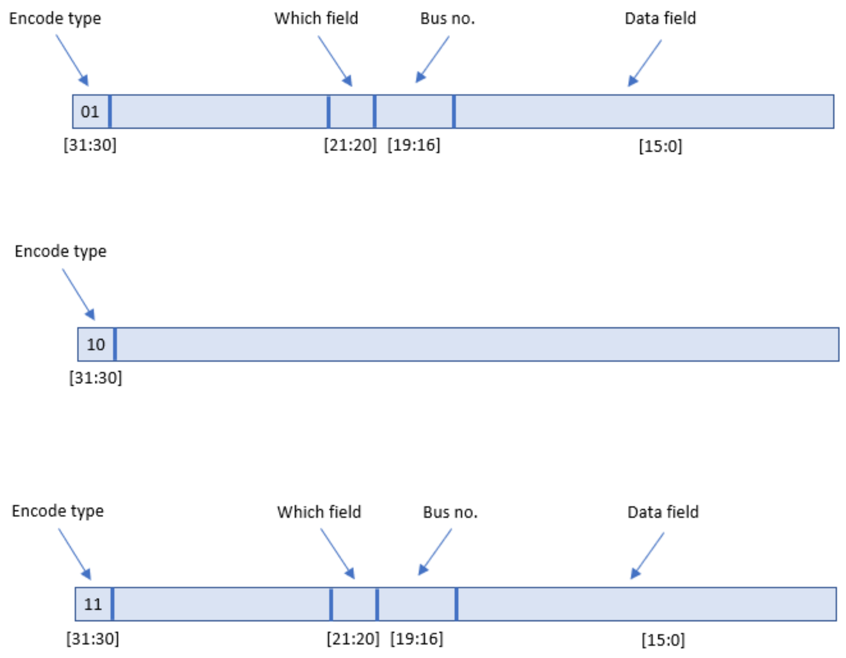


Figure 4.31: Encoding

- When the databus[31:30] is encoded 00, we do nothing.
- When the databus[31:30] is encoded 01, we use it to load the data input.
- When the databus[31:30] is encoded 10, we execute the matrix multiplication.
- When the databus [31:30] is encoded 11, we pass the voltage(v) to the processing system.

As the current (I) input is of 48 bits, we use which field [21:20] to describe which part of the data we are loading.

Databus [19:16] describe the bus no. from 1 to 12.

4.3.4 Block Diagram

After custom matrix vector multiplication IP is generated, we connect the entire system like a block diagram as shown in the Fig 5.9. CPU , Global reset, DMA and required interconnect IPs are provided by the Xilinx in the Vivado tool itself.

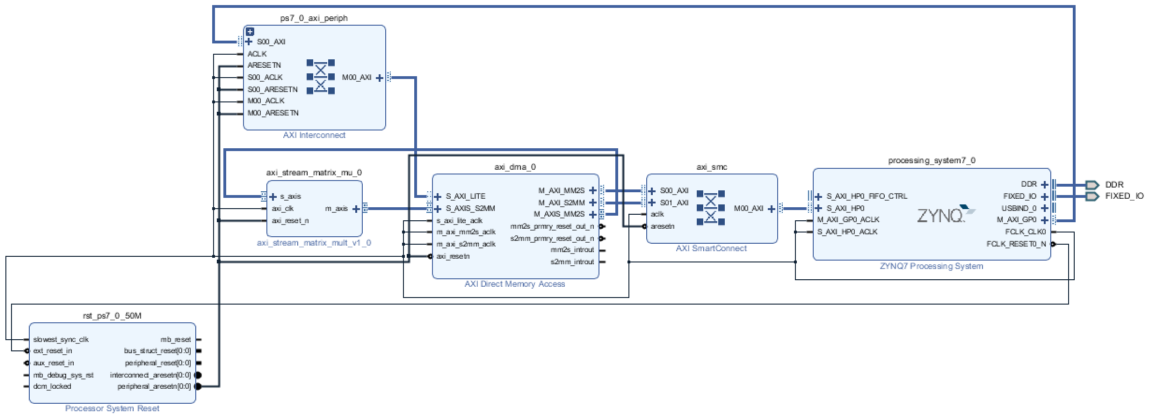
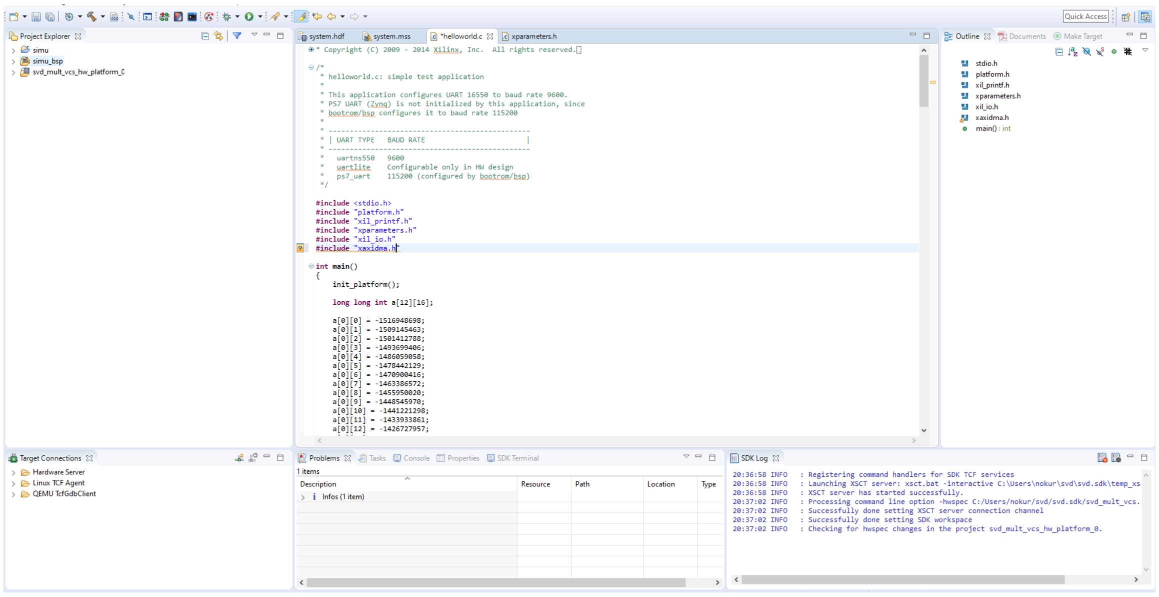


Figure 4.32: Block Diagram

4.3.5 ARM Programming



```
* Copyright (C) 2009 - 2014 Xilinx, Inc.  All rights reserved.

/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xaxidma.h"

int main()
{
    init_platform();

    long long int a[12][16];

    a[0][0] = -1516948698;
    a[0][1] = -1509145463;
    a[0][2] = -1501412788;
    a[0][3] = -1493699406;
    a[0][4] = -1486059058;
    a[0][5] = -1478442129;
    a[0][6] = -1470900416;
    a[0][7] = -1463386572;
    a[0][8] = -1455950020;
    a[0][9] = -1448545970;
    a[0][10] = -1441221298;
    a[0][11] = -1433933861;
    a[0][12] = -1426727957;
```

```
20:36:58 INFO   : Registering command handlers for SDK TCF services
20:36:58 INFO   : Launching XSCT server: xsct.bat -interactive C:\Users\nokur\svd\svd.sdk\temp_xs
20:36:58 INFO   : XSCT server has started successfully.
20:37:02 INFO   : Processing command line option -hwspec C:/Users/nokur/svd/svd.sdk/svd_mult_vcs.
20:37:02 INFO   : Successfully done setting XSCT server connection channel
20:37:02 INFO   : Successfully done setting SDK workspace
20:37:02 INFO   : Checking for hwspec changes in the project svd_mult_vcs_hw_platform_0.
```

Figure 4.33: ARM programming

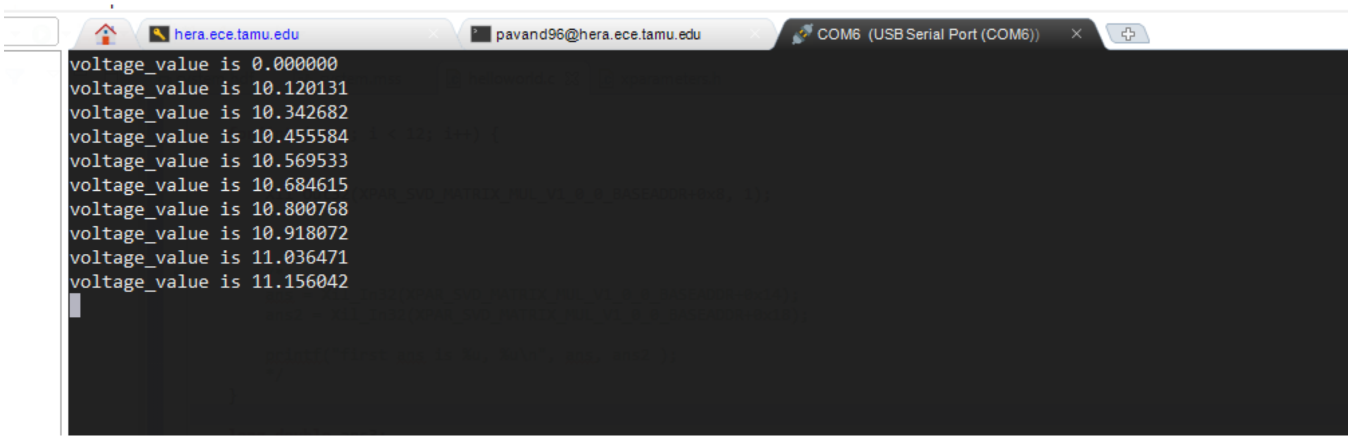
In the software part, we do the following steps

- We store all the instructions and the data in an array.

- Initiate two different DMA transfers one for sending the data and the other for receiving the data.
- Print the obtained results on the terminal via printf.

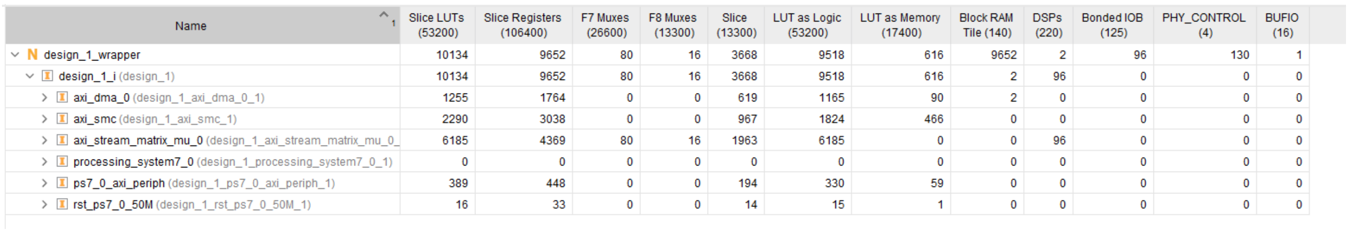
4.3.6 Results

The following figure shows voltage (V) for a particular bus during our FPGA simulation.



```
voltage_value is 0.000000
voltage_value is 10.120131
voltage_value is 10.342682
voltage_value is 10.455584
voltage_value is 10.569533
voltage_value is 10.684615
voltage_value is 10.800768
voltage_value is 10.918072
voltage_value is 11.036471
voltage_value is 11.156042
```

Figure 4.34: FPGA Simulation Result



Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	DSPs (220)	Bonded IOB (125)	PHY_CONTROL (4)	BUFIO (16)
design_1_wrapper	10134	9652	80	16	3668	9518	616	9652	2	96	130	1
design_1_i (design_1)	10134	9652	80	16	3668	9518	616	2	96	0	0	0
axi_dma_0 (design_1_axi_dma_0_1)	1255	1764	0	0	619	1165	90	2	0	0	0	0
axi_smc (design_1_axi_smc_1)	2290	3038	0	0	967	1824	466	0	0	0	0	0
axi_stream_matrix_mu_0 (design_1_axi_stream_matrix_mu_0_	6185	4369	80	16	1963	6185	0	0	96	0	0	0
processing_system7_0 (design_1_processing_system7_0_1)	0	0	0	0	0	0	0	0	0	0	0	0
ps7_0_axi_periph (design_1_ps7_0_axi_periph_1)	389	448	0	0	194	330	59	0	0	0	0	0
rst_ps7_0_50M (design_1_rst_ps7_0_50M_1)	16	33	0	0	14	15	1	0	0	0	0	0

Figure 4.35: FPGA utilization Results

Our multiplication IP peripheral consume 6185 LUTs, 4369 FFs and 96 DSPs.

Each DSP does (25 x 18 multiplication operation).

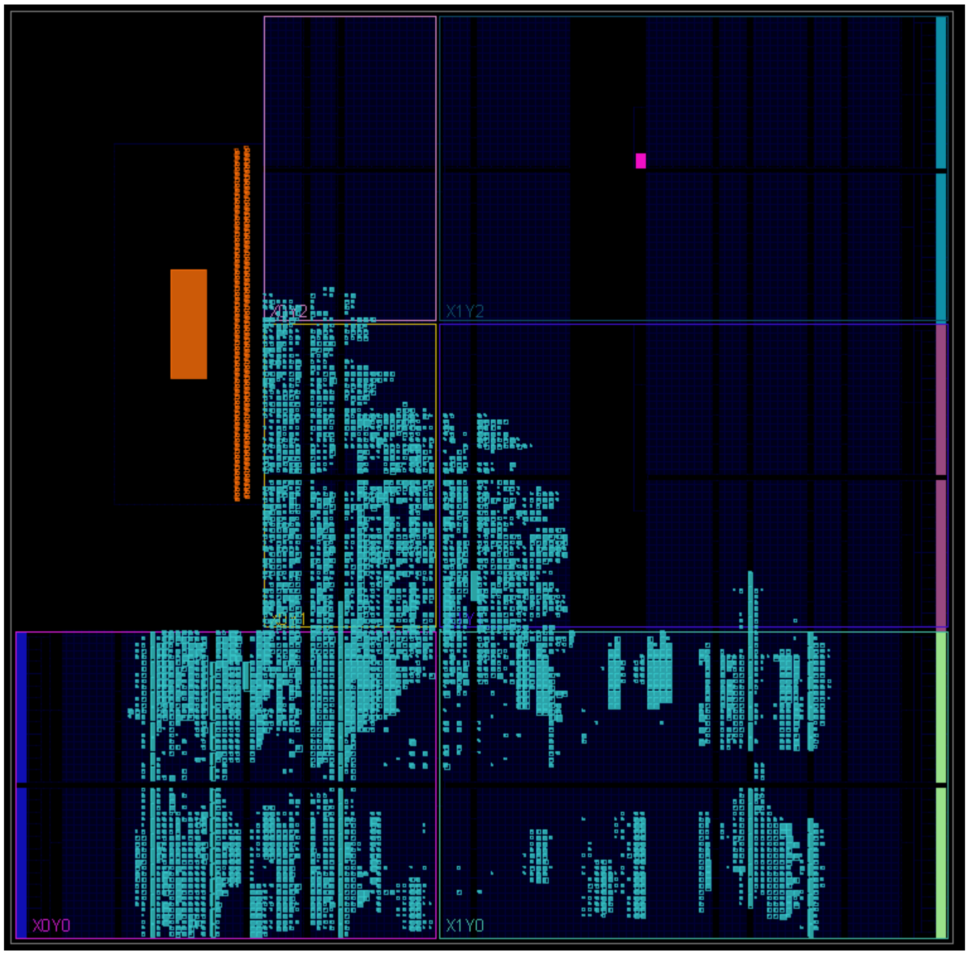
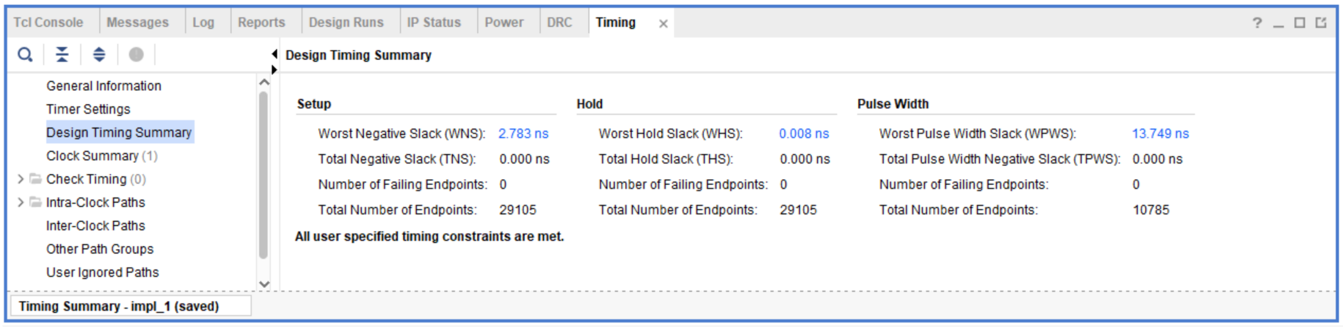


Figure 4.36: FPGA routing results

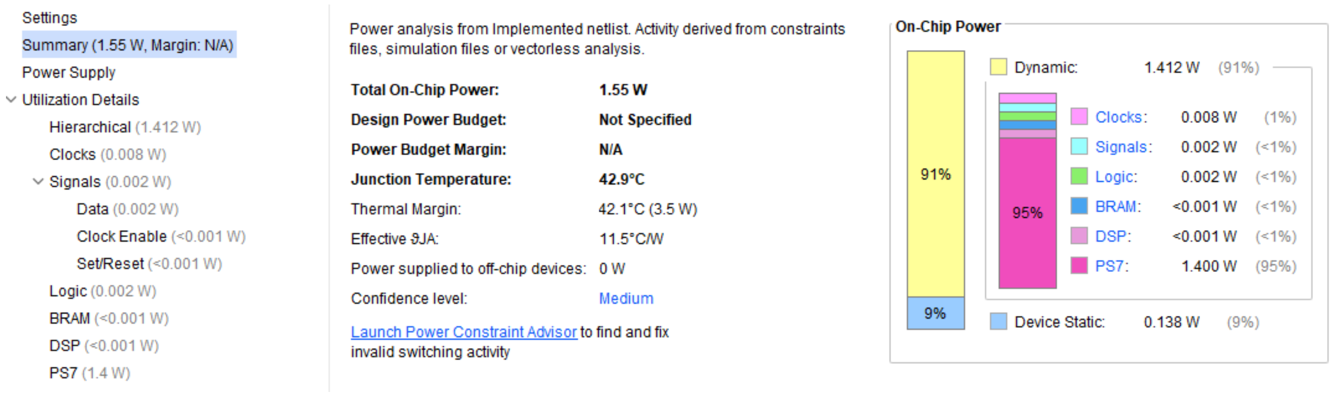


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2.783 ns	Worst Hold Slack (WHS):	0.008 ns	Worst Pulse Width Slack (WPWS):	13.749 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	29105	Total Number of Endpoints:	29105	Total Number of Endpoints:	10785

All user specified timing constraints are met.

Figure 4.37: FPGA timing results



Settings
Summary (1.55 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (1.412 W)
Clocks (0.008 W)
Signals (0.002 W)
Data (0.002 W)
Clock Enable (<0.001 W)
Set/Reset (<0.001 W)
Logic (0.002 W)
BRAM (<0.001 W)
DSP (<0.001 W)
PS7 (1.4 W)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.55 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**42.9°C**
Thermal Margin:	42.1°C (3.5 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:		1.412 W	(91%)
	Clocks:	0.008 W	(1%)
	Signals:	0.002 W	(<1%)
	Logic:	0.002 W	(<1%)
	BRAM:	<0.001 W	(<1%)
	DSP:	<0.001 W	(<1%)
	PS7:	1.400 W	(95%)
Device Static:		0.138 W	(9%)

Figure 4.38: FPGA Power consumption results

4.4 ASIC Implementation

The simulation and ASIC implementation[26] of the circuit is divided into three stages.

1. RTL Design using hardware description language(Verilog[27]).

2. Synthesizing the RTL code to netlist using Synopsys Design Compiler.

3. Place and Route using Cadence Innovus.

4.4.1 Synthesizing the RTL code to netlist using Synopsys Design Compiler

NanGate FreePDK standard library and Synopsys tools have been used to perform the design flow. The details are as follows:

1. Synopsys Library Compiler[28]: Library files(.lib) contain abstract logical and timing views of standard cells necessary for synthesis. Library Compiler converts a library file (.lib) format file to database format (.db) which is suitable for synopsys tools.

2. Synopsys Design Compiler[29] [30]: Design Compiler first converts a synthesizable RTL code to technology independent format called GTECH. Upon providing with standard cells, it takes the GTECH format and gives gate level design optimized for the specific technology node in terms of area, power and delay.

NanGate FreePDK 45[31] [32] is an open-source standard library kit held by US based multinational corporations for exploring EDA flows. It contains tech files, design rules for 45nm process.

The OCL kit provides three different models as follows:

1. NLDM (Non - Linear Delay Model): This is a voltage source-based model which characterizes input – output delay and output transition times with sensitivity to input transition time, side input states and output load.

2. ECSM (Effective Current Source Model): This is a current based model which considers the non – linear transistor switching behaviors, allowing the accurate modelling of interconnections.

3. CCS (Composite Current Source): A model similar to ECSM model in which dynamic current is calculated using transient analysis and leakage power is the leakage current measured using simple DC analysis.

These models are characterized into five different corners as follows [33]:

- Typical
- Slow
- Fast
- Low Temperature
- Worst case low temperature

In the .lib[34] file, the following attributes are present

- Voltage unit
- Current unit
- Leakage power unit
- Leakage power unit
- Leakage power unit
- Input threshold at rise and fall time.

- Output threshold for rise and fall time.
- Slew rate.

For each cell (AND, NAND, OR etc.,) following attributes are defined:

- Voltage unit
- Current unit
- Leakage power unit
- Leakage power unit
- Leakage power unit
- Input threshold at rise and fall time.
- Output threshold for rise and fall time.
- Slew rate.
- Area of cell
- Capacitance
- Rise and fall capacitance
- Direction of the pin etc.,

Synthesis steps:

1. Loading the Design: The design is provided in the form of High-level Description language - Verilog or VHDL file. The analyze command reads the HDL file checks for syntax and synthesizable logic. The elaborate command converts the design to a synopsys internal generic library called gtech.

2. Loading Technology Libraries: Next step is to load technology libraries(.lib) from Nangate FreePDK and Synopsys designware libraries. Synopsys designware library is a building block IP and collection of reusable arithmetic modules tightly integrated to Synopsys synthesis environment.

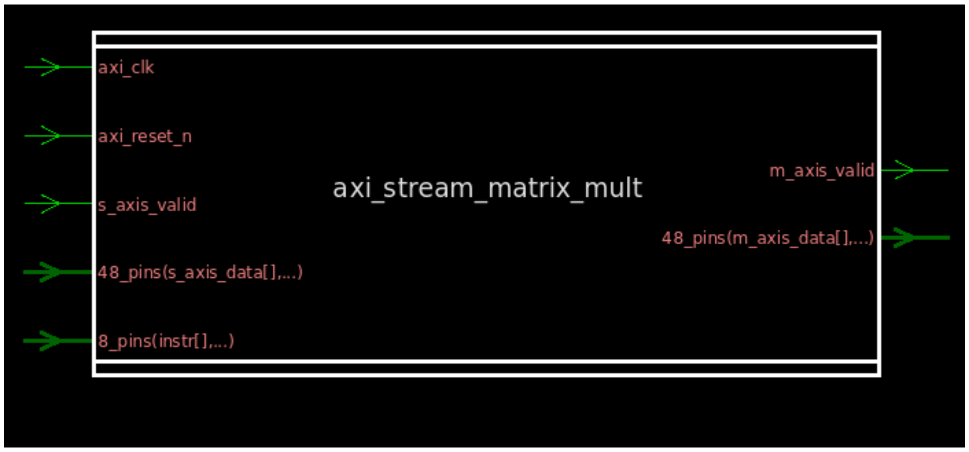


Figure 4.39: Block diagram of the entire design in the Design Compiler

3. Creating virtual clock: create_clock clk -name ideal_clock1 -period 6. This command creates a virtual clock that will be connected to all the flip flops required for our design.

4. Setting the operating conditions: Operating conditions are the process, voltage and temperature variations on which the design operates. WCCOM(class) is the condition window on which our design operates.

5. Ungrouping the cells: It is important to flatten the design by eliminating hierarchical boundaries. The tool will be able to perform better and achieve better PPA results.

6. Check Design: check_design command checks if there are no obvious errors in RTL code.

7. Compile Command: This command performs the actual optimization by converting the RTL to technology optimized netlist.

8. Writing down the output: We output the result in two different file formats: .ddc and Verilog.

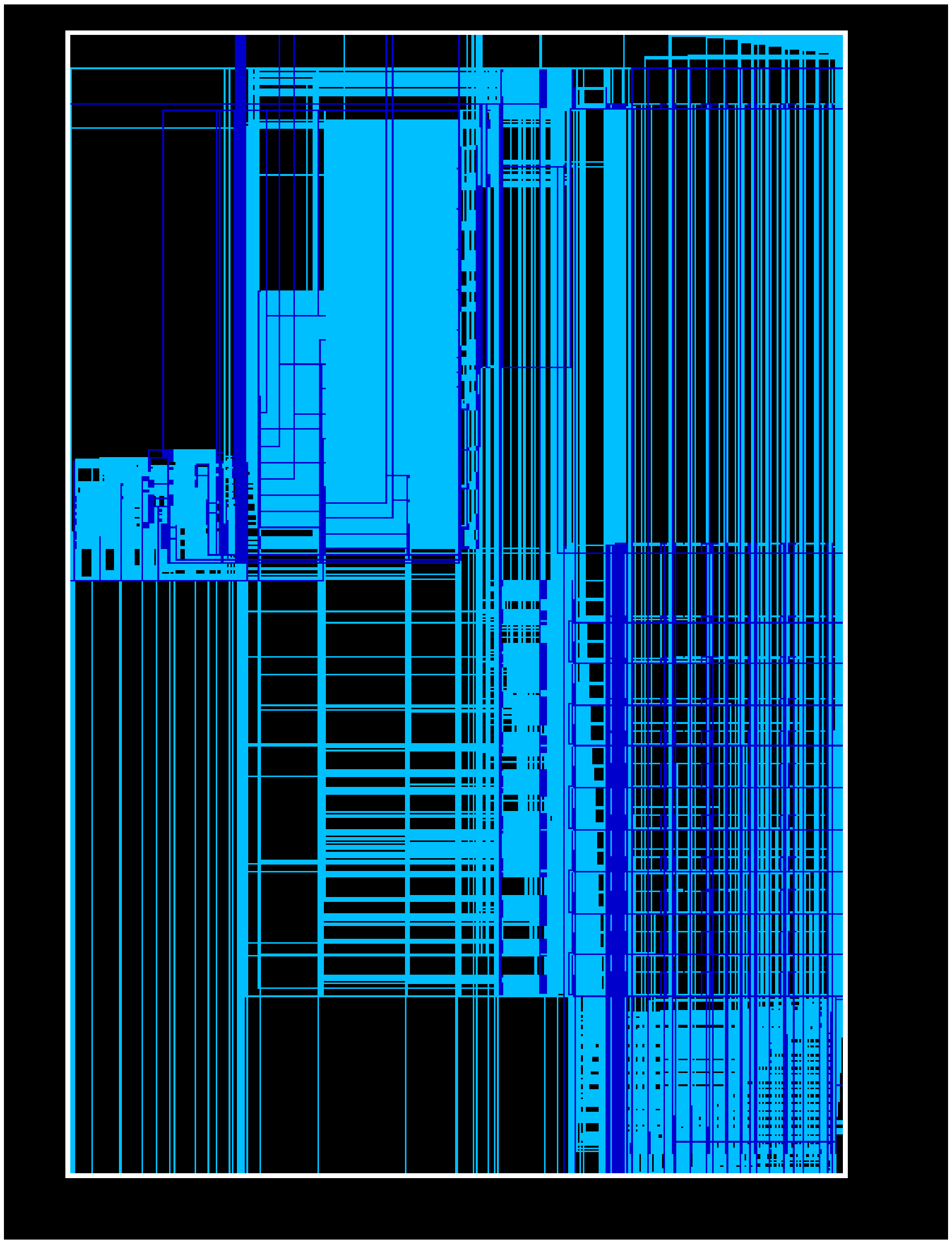


Figure 4.40: Detailed circuit diagram for the entire design in the Design Compiler

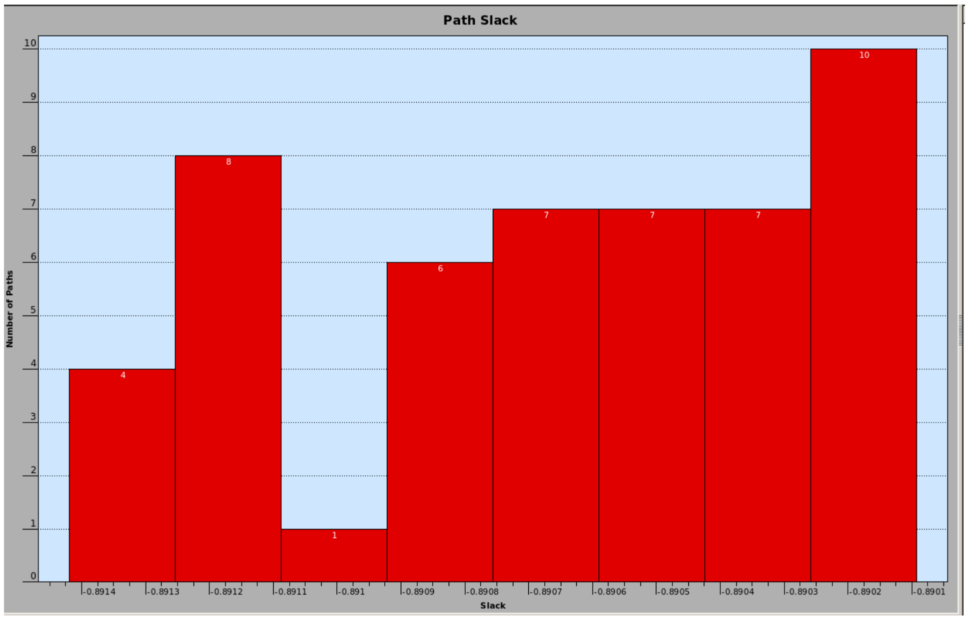


Figure 4.41: Slack paths when clock period is set to 5ns.

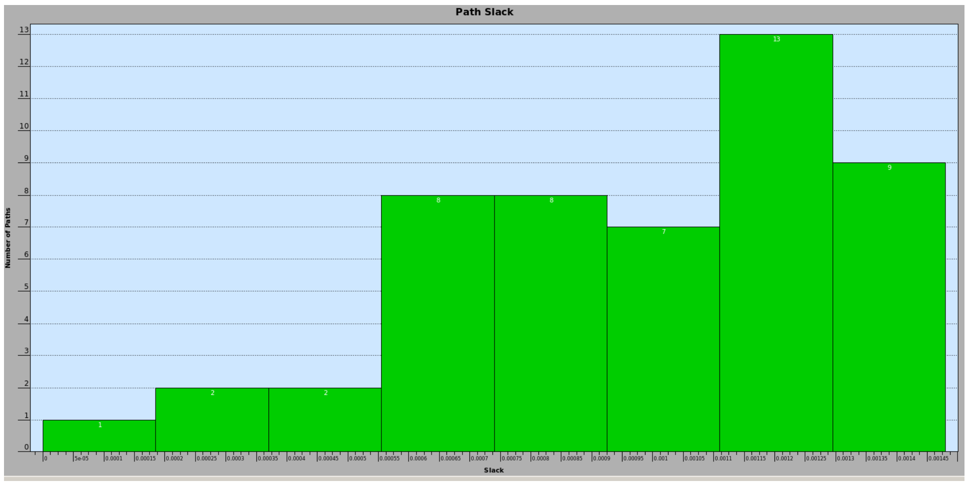
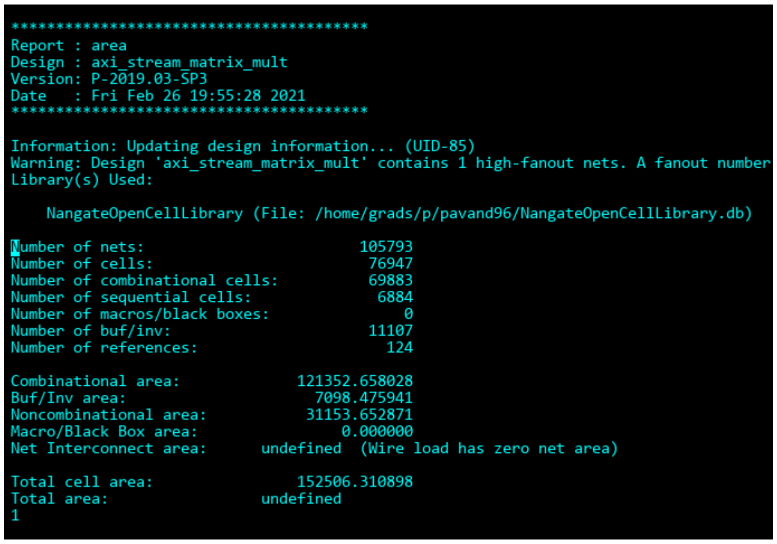


Figure 4.42: Slack paths when clock period is set to 7ns.

This matrix multiplication peripheral has been synthesized for clock of time period 7.2ns.



```
****************************************
Report : area
Design : axi_stream_matrix_mult
Version: P-2019.03-SP3
Date   : Fri Feb 26 19:55:28 2021
****************************************

Information: Updating design information... (UID-85)
Warning: Design 'axi_stream_matrix_mult' contains 1 high-fanout nets. A fanout number
Library(s) Used:

    NangateOpenCellLibrary (File: /home/grads/p/pavand96/NangateOpenCellLibrary.db)

Number of nets:                        105793
Number of cells:                        76947
Number of combinational cells:          69883
Number of sequential cells:              6884
Number of macros/black boxes:               0
Number of buf/inv:                      11107
Number of references:                     124

Combinational area:             121352.658028
Buf/Inv area:                     7098.475941
Noncombinational area:           31153.652871
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                152506.310898
Total area:                 undefined
1
```

Figure 4.43: Area and Standard cell consumption before Place and Route

4.4.2 Comparison between different order of differentiation after implementation

Method	Area(in um^2)	Improvement	clock time(in ns)	latency (in clock cycles)
Previous work	178,200	-	7.7	6
Higher Order Difference with k = 1	164,566	7.65%	7.45	7
Higher Order Difference with k = 2	152,708	14.30%	7.2	8

Area is in proportion to the standard cell count.

4.4.3 Place and Route using Cadence Innovus

In this section, we will explain the detailed steps to perform the backend placing and routing using cadence Innovus[35], an industry leading tool for place and route.

Needed files:

1. Gate level netlist which is generated using Design compiler.

2. NanGate FreePDK LEF files necessary for place and route.

3. Multi-mode multi-corner(MMMC) file for timing analysis.

The LEF [34] file is the abstract view of standard cells. It gives the idea about the pin position, PR boundary and metal layer information of the cell.

Multi-mode multi-corner file specified what corner to use for timing analysis. A corner is a characterization of standard cell with specific assumptions about process, temperature and voltage(V). For this placement and routing, we choose typical corner(average PVT).

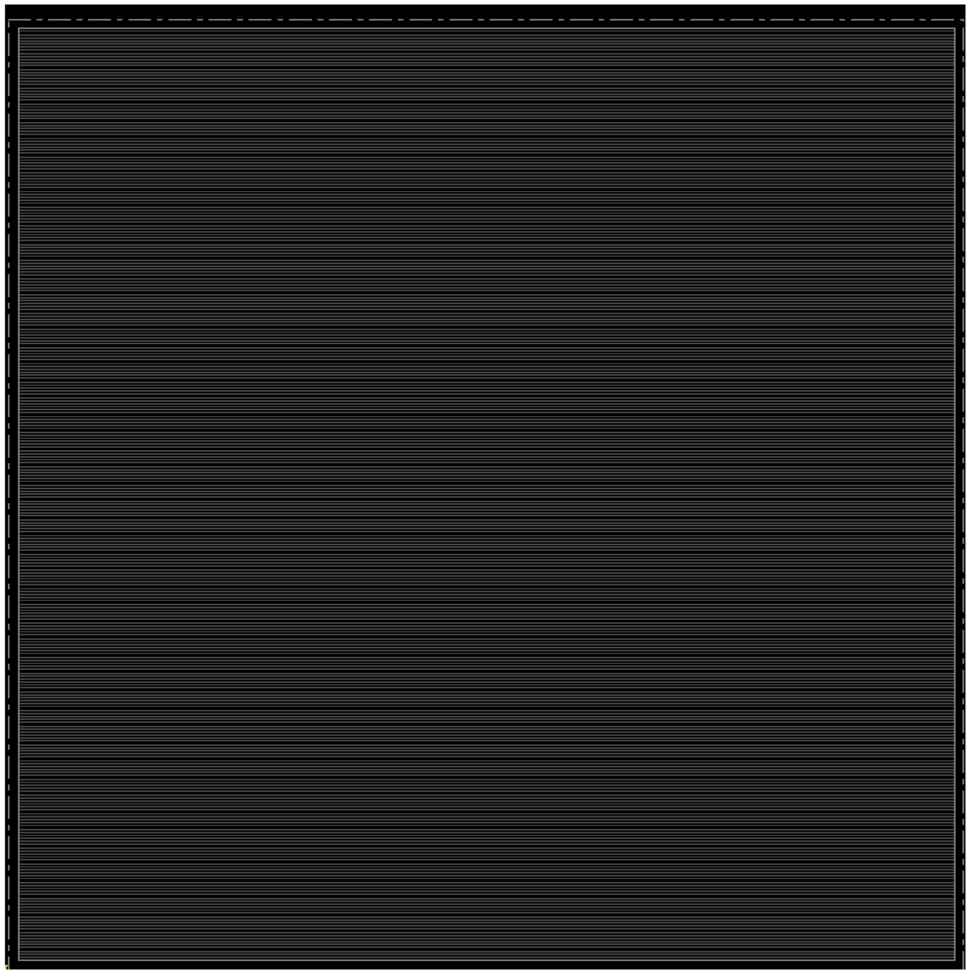


Figure 4.44: Initial floor planning

Sub steps: 1. The first step is to input the tool with netlist, setting up the design name, timing analysis, reading the technology .lef for layer information and standard cell.

2. Second step is to do floor planning. In this design, we chose aspect ratio to be 1.0 and cell utilization to be 70%. Aspect ratio is the ratio of width and height of the entire chip and cell utilization is the percentage of the actual chip used for standard cells.

3. The next step is power routing. We create a grid of power and ground wires on top metal layers and connect this grid down to M1 power rails in each row.

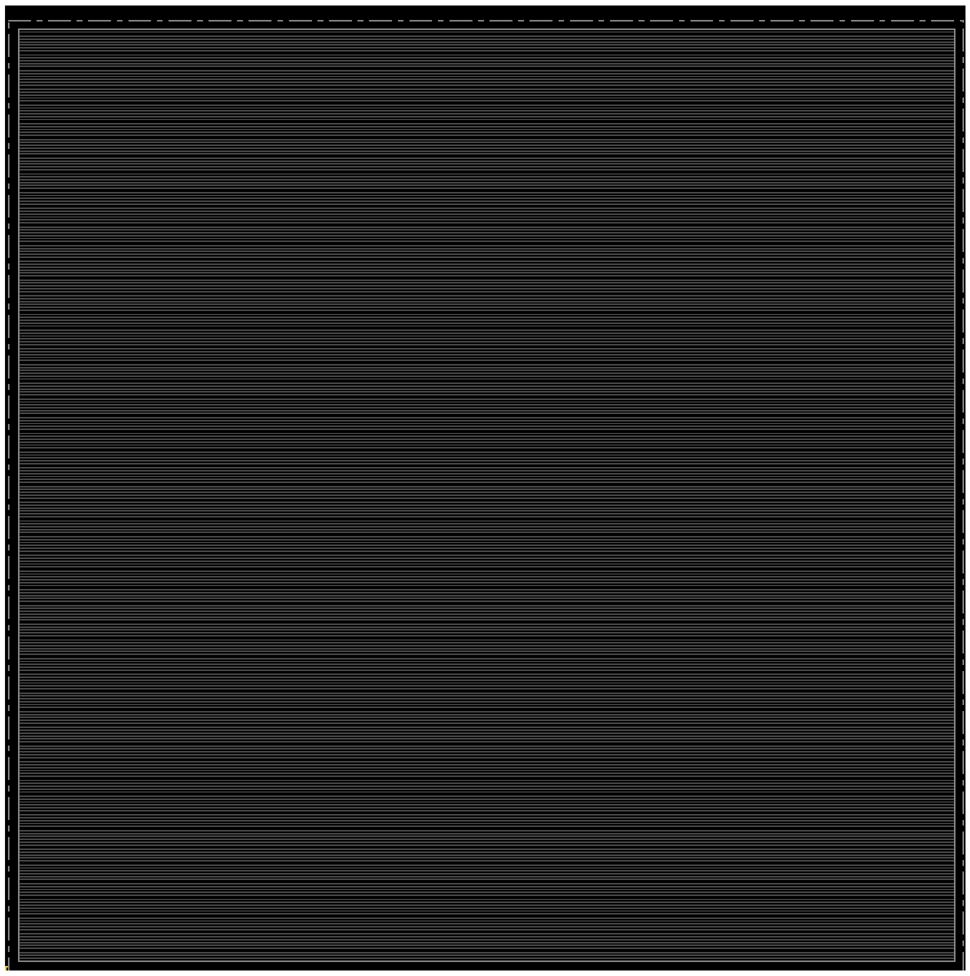


Figure 4.45: Power planning

4. Fourth step is to create a power ring around the chip. This ensures that all the standard cell get power and ground easily. We will use M6 and M7 to put the power ring across the chip.

5. The next step is to place standard cells using place_design command.

6. Sixth step is to optimize the clock tree routing using ccopt_design command.

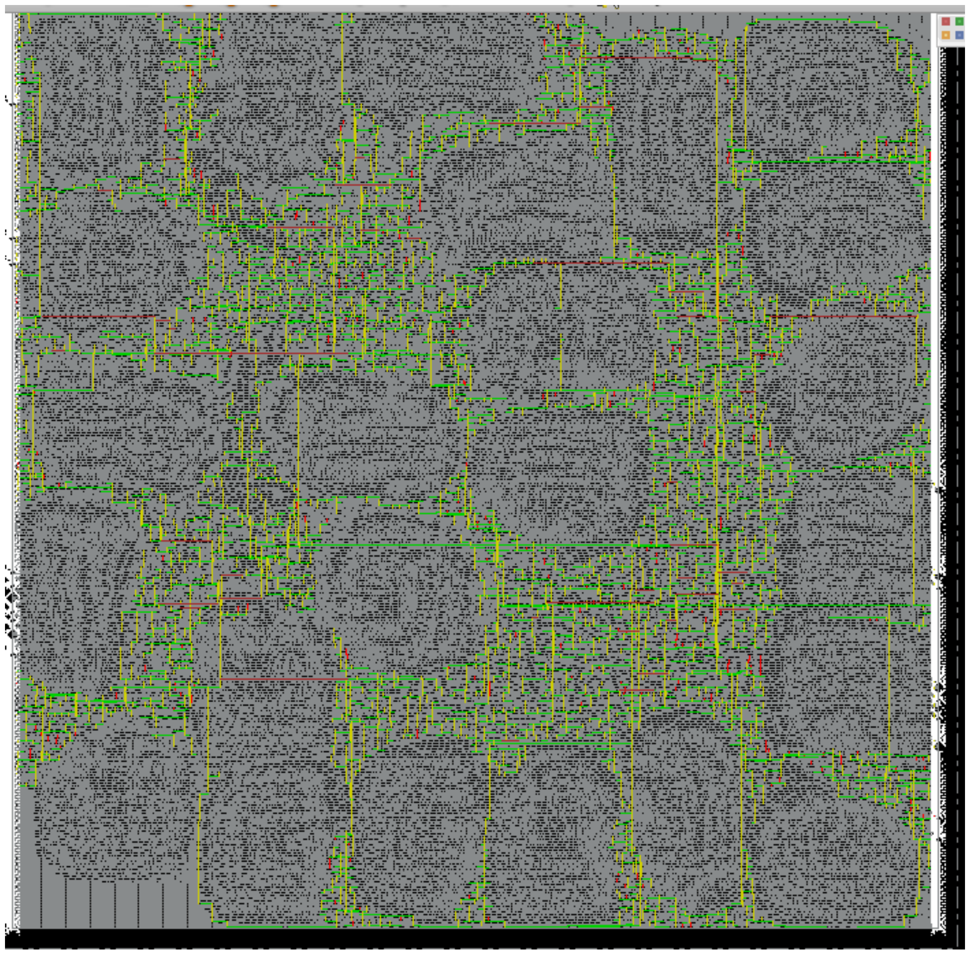


Figure 4.46: Clock tree planning

7. Final step is signal routing.

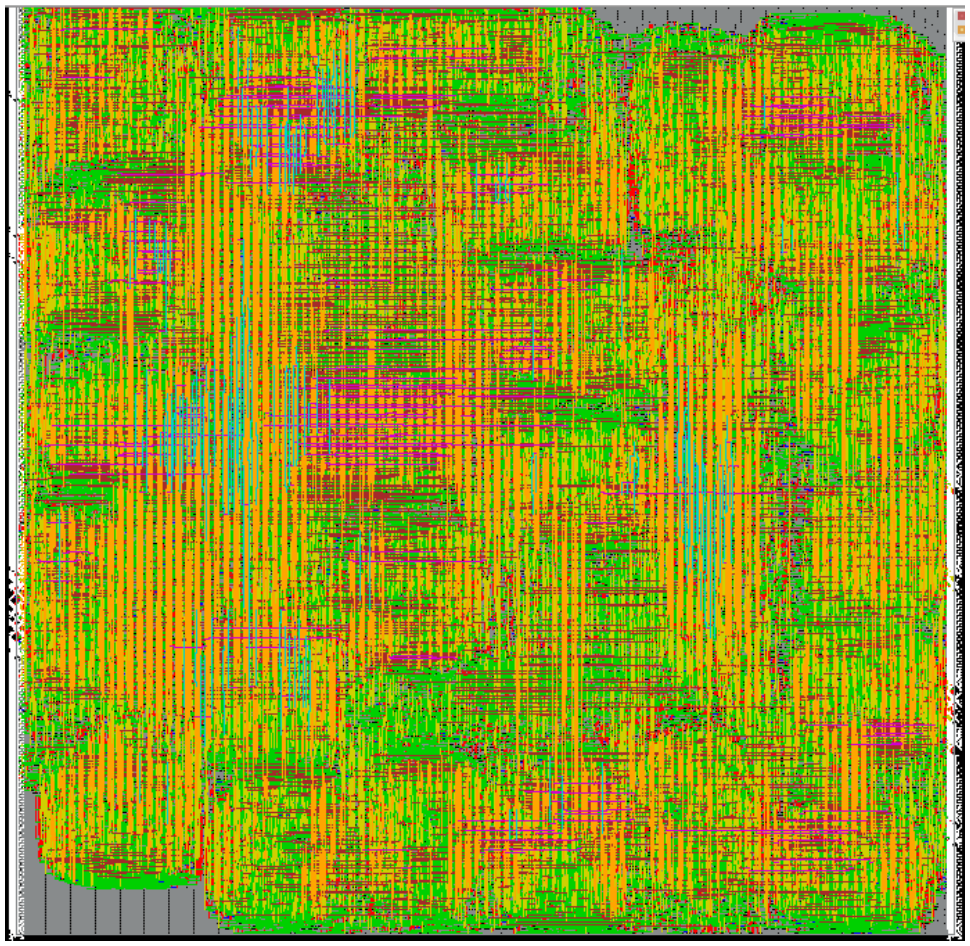


Figure 4.47: Standard cell placement

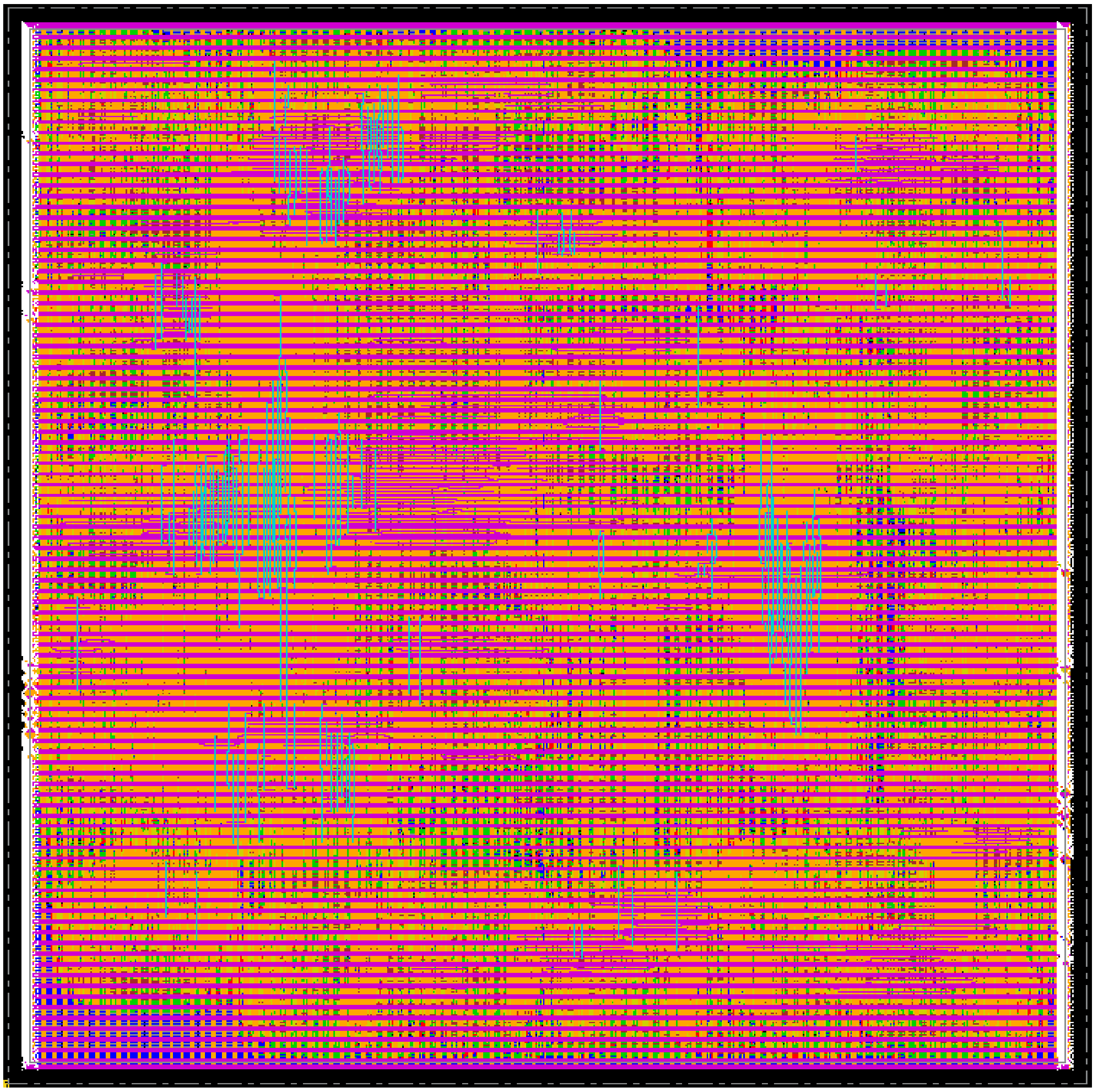
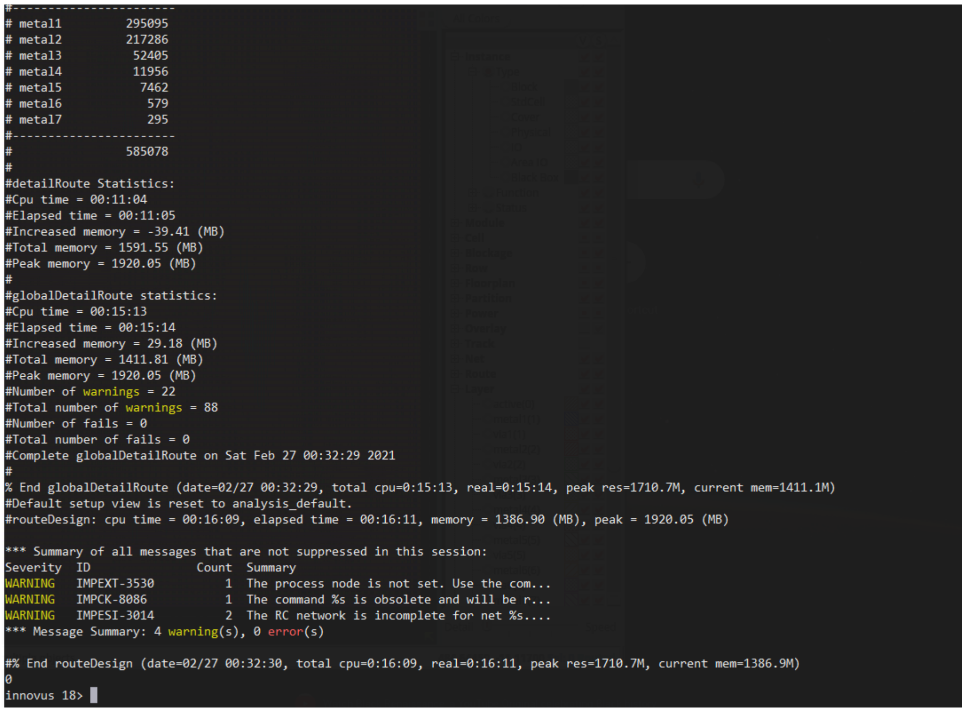
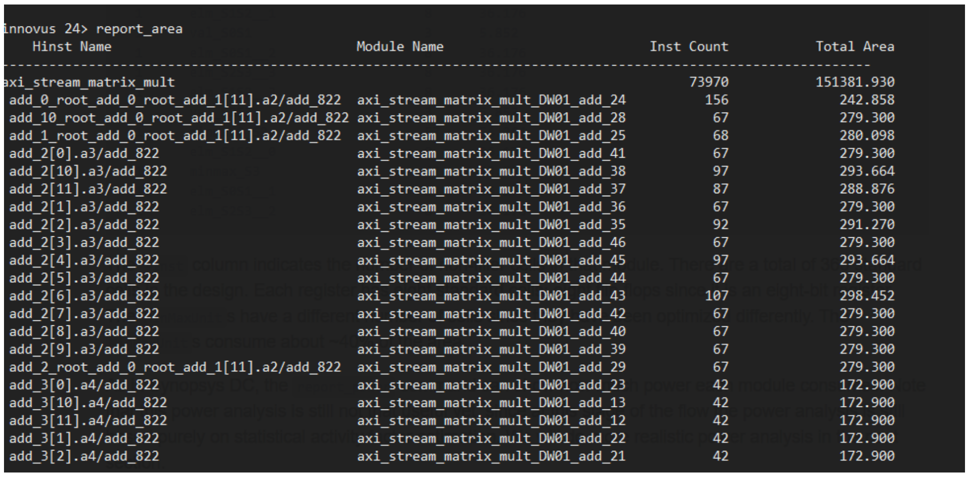


Figure 4.48: Place and Route



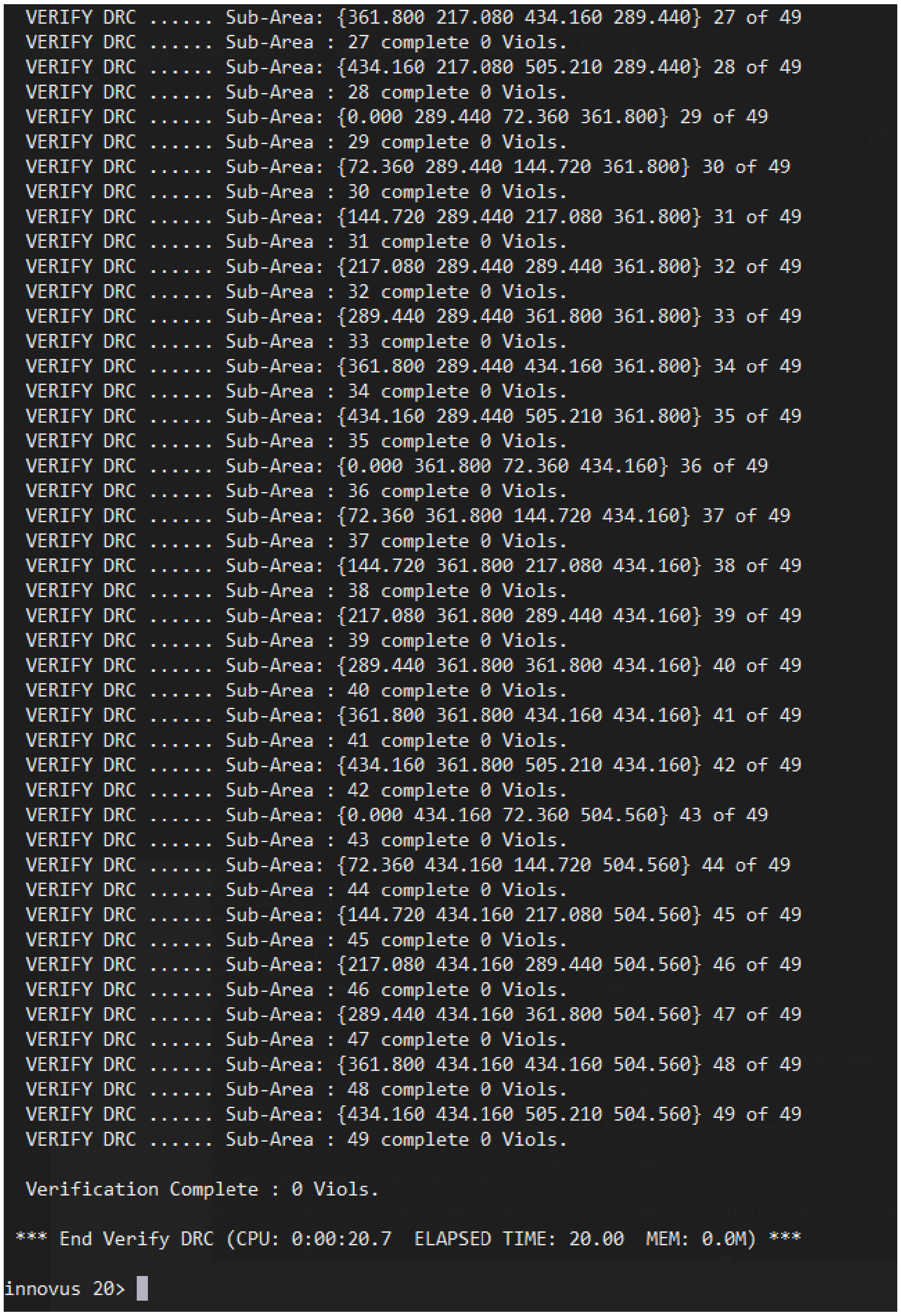
```
#----------------------
# metal1        295095
# metal2        217286
# metal3         52405
# metal4         11956
# metal5          7462
# metal6           579
# metal7           295
#----------------------
#               585078
#
#detailRoute Statistics:
#Cpu time = 00:11:04
#Elapsed time = 00:11:05
#Increased memory = -39.41 (MB)
#Total memory = 1591.55 (MB)
#Peak memory = 1920.05 (MB)
#
#globalDetailRoute statistics:
#Cpu time = 00:15:13
#Elapsed time = 00:15:14
#Increased memory = 29.18 (MB)
#Total memory = 1411.81 (MB)
#Peak memory = 1920.05 (MB)
#Number of warnings = 22
#Total number of warnings = 88
#Number of fails = 0
#Total number of fails = 0
#Complete globalDetailRoute on Sat Feb 27 00:32:29 2021
#
% End globalDetailRoute (date=02/27 00:32:29, total cpu=0:15:13, real=0:15:14, peak res=1710.7M, current mem=1411.1M)
#Default setup view is reset to analysis_default.
#routeDesign: cpu time = 00:16:09, elapsed time = 00:16:11, memory = 1386.90 (MB), peak = 1920.05 (MB)

*** Summary of all messages that are not suppressed in this session:
Severity  ID                 Count  Summary
WARNING   IMPEXT-3530            1  The process node is not set. Use the com...
WARNING   IMPCK-8086             1  The command %s is obsolete and will be r...
WARNING   IMPESI-3014            2  The RC network is incomplete for net %s....
*** Message Summary: 4 warning(s), 0 error(s)

#% End routeDesign (date=02/27 00:32:30, total cpu=0:16:09, real=0:16:11, peak res=1710.7M, current mem=1386.9M)
0
innovus 18> 
```

Figure 4.49: Place and Route terminal result



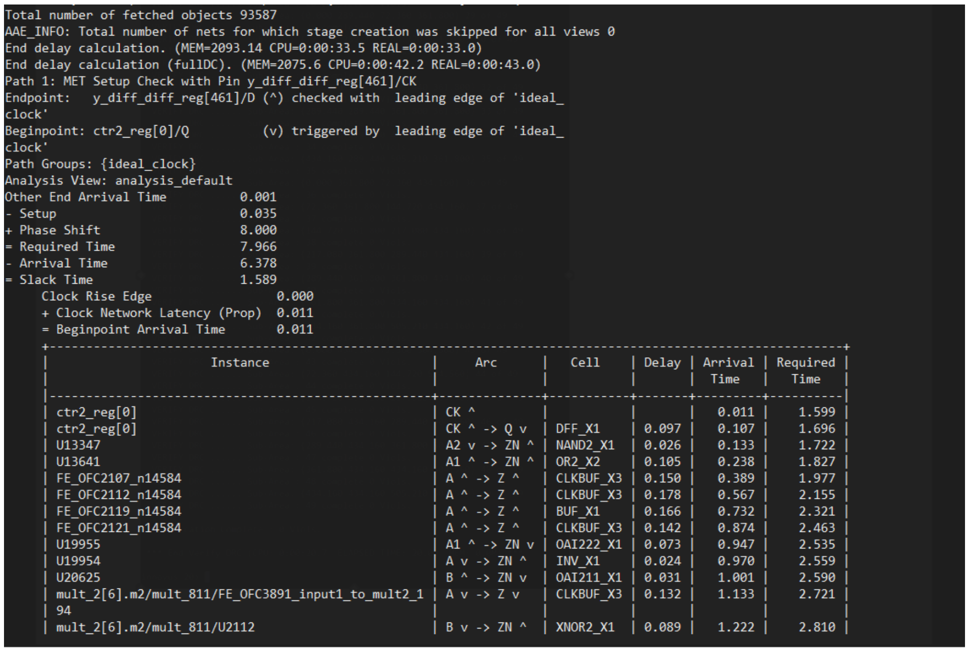
```
innovus 24> report_area
   Hinst Name                                Module Name                           Inst Count         Total Area
-------------------------------------------------------------------------------------------------------------------
axi_stream_matrix_mult                                                                  73970         151381.930
 add_0_root_add_0_root_add_1[11].a2/add_822  axi_stream_matrix_mult_DW01_add_24           156            242.858
 add_10_root_add_0_root_add_1[11].a2/add_822 axi_stream_matrix_mult_DW01_add_28            67            279.300
 add_1_root_add_0_root_add_1[11].a2/add_822  axi_stream_matrix_mult_DW01_add_25            68            280.098
 add_2[0].a3/add_822                         axi_stream_matrix_mult_DW01_add_41            67            279.300
 add_2[10].a3/add_822                        axi_stream_matrix_mult_DW01_add_38            97            293.664
 add_2[11].a3/add_822                        axi_stream_matrix_mult_DW01_add_37            87            288.876
 add_2[1].a3/add_822                         axi_stream_matrix_mult_DW01_add_36            67            279.300
 add_2[2].a3/add_822                         axi_stream_matrix_mult_DW01_add_35            92            291.270
 add_2[3].a3/add_822                         axi_stream_matrix_mult_DW01_add_46            67            279.300
 add_2[4].a3/add_822                         axi_stream_matrix_mult_DW01_add_45            97            293.664
 add_2[5].a3/add_822                         axi_stream_matrix_mult_DW01_add_44            67            279.300
 add_2[6].a3/add_822                         axi_stream_matrix_mult_DW01_add_43           107            298.452
 add_2[7].a3/add_822                         axi_stream_matrix_mult_DW01_add_42            67            279.300
 add_2[8].a3/add_822                         axi_stream_matrix_mult_DW01_add_40            67            279.300
 add_2[9].a3/add_822                         axi_stream_matrix_mult_DW01_add_39            67            279.300
 add_2_root_add_0_root_add_1[11].a2/add_822  axi_stream_matrix_mult_DW01_add_29            67            279.300
 add_3[0].a4/add_822                         axi_stream_matrix_mult_DW01_add_23            42            172.900
 add_3[10].a4/add_822                        axi_stream_matrix_mult_DW01_add_13            42            172.900
 add_3[11].a4/add_822                        axi_stream_matrix_mult_DW01_add_12            42            172.900
 add_3[1].a4/add_822                         axi_stream_matrix_mult_DW01_add_22            42            172.900
 add_3[2].a4/add_822                         axi_stream_matrix_mult_DW01_add_21            42            172.900
```

Figure 4.50: Area consumption after place and route



```
  VERIFY DRC ...... Sub-Area: {361.800 217.080 434.160 289.440} 27 of 49
  VERIFY DRC ...... Sub-Area : 27 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {434.160 217.080 505.210 289.440} 28 of 49
  VERIFY DRC ...... Sub-Area : 28 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {0.000 289.440 72.360 361.800} 29 of 49
  VERIFY DRC ...... Sub-Area : 29 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {72.360 289.440 144.720 361.800} 30 of 49
  VERIFY DRC ...... Sub-Area : 30 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {144.720 289.440 217.080 361.800} 31 of 49
  VERIFY DRC ...... Sub-Area : 31 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {217.080 289.440 289.440 361.800} 32 of 49
  VERIFY DRC ...... Sub-Area : 32 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {289.440 289.440 361.800 361.800} 33 of 49
  VERIFY DRC ...... Sub-Area : 33 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {361.800 289.440 434.160 361.800} 34 of 49
  VERIFY DRC ...... Sub-Area : 34 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {434.160 289.440 505.210 361.800} 35 of 49
  VERIFY DRC ...... Sub-Area : 35 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {0.000 361.800 72.360 434.160} 36 of 49
  VERIFY DRC ...... Sub-Area : 36 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {72.360 361.800 144.720 434.160} 37 of 49
  VERIFY DRC ...... Sub-Area : 37 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {144.720 361.800 217.080 434.160} 38 of 49
  VERIFY DRC ...... Sub-Area : 38 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {217.080 361.800 289.440 434.160} 39 of 49
  VERIFY DRC ...... Sub-Area : 39 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {289.440 361.800 361.800 434.160} 40 of 49
  VERIFY DRC ...... Sub-Area : 40 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {361.800 361.800 434.160 434.160} 41 of 49
  VERIFY DRC ...... Sub-Area : 41 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {434.160 361.800 505.210 434.160} 42 of 49
  VERIFY DRC ...... Sub-Area : 42 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {0.000 434.160 72.360 504.560} 43 of 49
  VERIFY DRC ...... Sub-Area : 43 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {72.360 434.160 144.720 504.560} 44 of 49
  VERIFY DRC ...... Sub-Area : 44 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {144.720 434.160 217.080 504.560} 45 of 49
  VERIFY DRC ...... Sub-Area : 45 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {217.080 434.160 289.440 504.560} 46 of 49
  VERIFY DRC ...... Sub-Area : 46 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {289.440 434.160 361.800 504.560} 47 of 49
  VERIFY DRC ...... Sub-Area : 47 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {361.800 434.160 434.160 504.560} 48 of 49
  VERIFY DRC ...... Sub-Area : 48 complete 0 Viols.
  VERIFY DRC ...... Sub-Area: {434.160 434.160 505.210 504.560} 49 of 49
  VERIFY DRC ...... Sub-Area : 49 complete 0 Viols.

  Verification Complete : 0 Viols.

 *** End Verify DRC (CPU: 0:00:20.7  ELAPSED TIME: 20.00  MEM: 0.0M) ***

innovus 20>
```

Figure 4.51: DRC verification



```
Total number of fetched objects 93587
AAE_INFO: Total number of nets for which stage creation was skipped for all views 0
End delay calculation. (MEM=2093.14 CPU=0:00:33.5 REAL=0:00:33.0)
End delay calculation (fullDC). (MEM=2075.6 CPU=0:00:42.2 REAL=0:00:43.0)
Path 1: MET Setup Check with Pin y_diff_diff_reg[461]/CK
Endpoint:   y_diff_diff_reg[461]/D (^) checked with  leading edge of 'ideal_
clock'
Beginpoint: ctr2_reg[0]/Q          (v) triggered by  leading edge of 'ideal_
clock'
Path Groups: {ideal_clock}
Analysis View: analysis_default
Other End Arrival Time          0.001
- Setup                         0.035
+ Phase Shift                   8.000
= Required Time                 7.966
- Arrival Time                  6.378
= Slack Time                    1.589
     Clock Rise Edge                 0.000
     + Clock Network Latency (Prop)  0.011
     = Beginpoint Arrival Time       0.011
     +------------------------------------------------------------------------------------------------------------------+
     |                      Instance                     |     Arc      |   Cell    | Delay | Arrival | Required |
     |                                                   |              |           |       |  Time   |   Time   |
     |---------------------------------------------------+--------------+-----------+-------+---------+----------|
     | ctr2_reg[0]                                       | CK ^         |           |       |   0.011 |    1.599 |
     | ctr2_reg[0]                                       | CK ^ -> Q v  | DFF_X1    | 0.097 |   0.107 |    1.696 |
     | U13347                                            | A2 v -> ZN ^ | NAND2_X1  | 0.026 |   0.133 |    1.722 |
     | U13641                                            | A1 ^ -> ZN ^ | OR2_X2    | 0.105 |   0.238 |    1.827 |
     | FE_OFC2107_n14584                                 | A ^ -> Z ^   | CLKBUF_X3 | 0.150 |   0.389 |    1.977 |
     | FE_OFC2112_n14584                                 | A ^ -> Z ^   | CLKBUF_X3 | 0.178 |   0.567 |    2.155 |
     | FE_OFC2119_n14584                                 | A ^ -> Z ^   | BUF_X1    | 0.166 |   0.732 |    2.321 |
     | FE_OFC2121_n14584                                 | A ^ -> Z ^   | CLKBUF_X3 | 0.142 |   0.874 |    2.463 |
     | U19955                                            | A1 ^ -> ZN v | OAI222_X1 | 0.073 |   0.947 |    2.535 |
     | U19954                                            | A v -> ZN ^  | INV_X1    | 0.024 |   0.970 |    2.559 |
     | U20625                                            | B ^ -> ZN v  | OAI211_X1 | 0.031 |   1.001 |    2.590 |
     | mult_2[6].m2/mult_811/FE_OFC3891_input1_to_mult2_1 | A v -> Z v   | CLKBUF_X3 | 0.132 |   1.133 |    2.721 |
     | 94                                                |              |           |       |         |          |
     | mult_2[6].m2/mult_811/U2112                       | B v -> ZN ^  | XNOR2_X1  | 0.089 |   1.222 |    2.810 |
```

Figure 4.52: Timing check after place and route

5. PARK AND INVERSE PARK TRANSFORMATION

5.1 Interface between network and non - network part

As discussed in the introduction, Park transformation converts the state variables from DQ0 coordinate in the generator side to the three phase coordinates in the network, whereas inverse park transformation reverts the conversion back to the DQ0 coordinates.

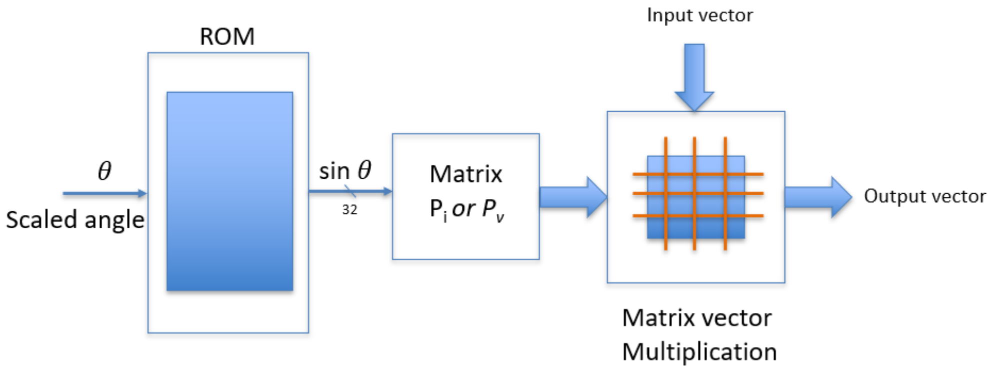


Figure 5.1: High level diagram

Park Transformation:

$$\begin{pmatrix} I_a(t) \\ I_b(t) \\ I_c(t) \end{pmatrix} = \begin{pmatrix} sin(\theta + \frac{1}{2}\pi) & -sin(\theta) & 1 \\ sin(\theta - \frac{1}{6}\pi) & sin(\theta + \frac{1}{3}\pi) & 1 \\ sin(\theta + \frac{1}{6}\pi) & sin(\theta - \frac{1}{3}\pi) & 1 \end{pmatrix} \begin{pmatrix} I_d(t) \\ I_q(t) \\ 0 \end{pmatrix} \tag{5.1}$$

Inverse Park Transformation:

$$\begin{pmatrix} e_d(t) \\ e_q(t) \\ \epsilon \end{pmatrix} = \begin{pmatrix} sin(\theta + \frac{1}{2}\pi) & sin(\theta - \frac{1}{6}\pi) & -sin(\theta - \frac{1}{6}\pi) \\ -sin(\theta) & sin(\theta + \frac{1}{3} & sin(\theta - \frac{1}{3} \\ 0.5 & 0.5 & 0.5 \end{pmatrix} \begin{pmatrix} V_a(t) \\ V_b(t) \\ V_c(t) \end{pmatrix} \tag{5.2}$$

As Eq 5.1 and 5.2 describe, the matrix is in terms of angle θ between a and q axes. This matrix has to be computed at every time step as Θ changes. The sinusoidal values of angles ranging from 0 to $\pi/2$ has to be stored in memory like a lookup table.

5.2 Linear Interpolation

Linear Interpolation is a special case of polynomial interpretation. Given two known points (x_0, y_0) and (x_1, y_1) on the xy plane, its linear interpolant is a straight line connecting the two points and the equation can be derived as follows:

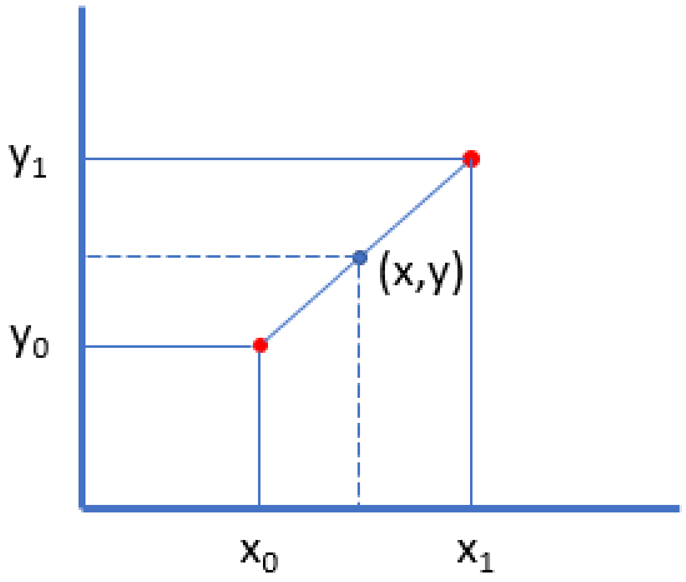


Figure 5.2: Linear Interpolation for two random points

Equation of slopes:

$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0} \tag{5.3}$$

solving for y, we get

$$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0} \tag{5.4}$$

5.3 Linear Interpolation for computing sinusoid values

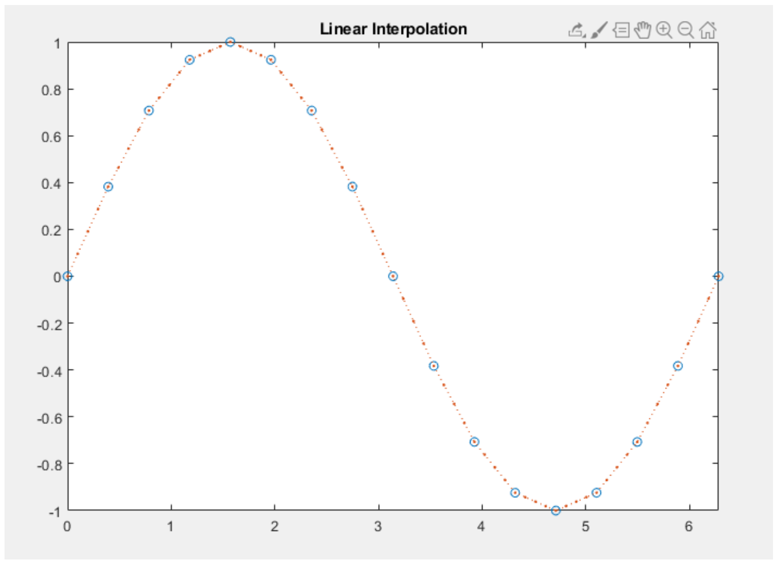


Figure 5.3: Linear Interpolation for a sinusoid waveform

The objective is to take 2^k, (where k is a whole number) number of sinusoid samples and linearly interpolate them (as shown in the figure 5.2) such that the difference between two adjacent interpolated values should be close to 2^{-32}. Upon simulating in matlab with different values of k, we concluded that k = 15 works best for our application.

In conclusion, we need 2^{15} = (32768) samples of sine wave (from angle 0 to $\pi/2$) with each value being 32 bits to meet the required accuracy for park transformation and its inverse.

5.3.1 Previous work

In the paper [11], authors have proposed a non-uniform based piecewise linear approximation with one bit error correction to compute sinsuoid values. ROM3 as shown in the below figure consumes large area when we extend this architecture for our application needs.

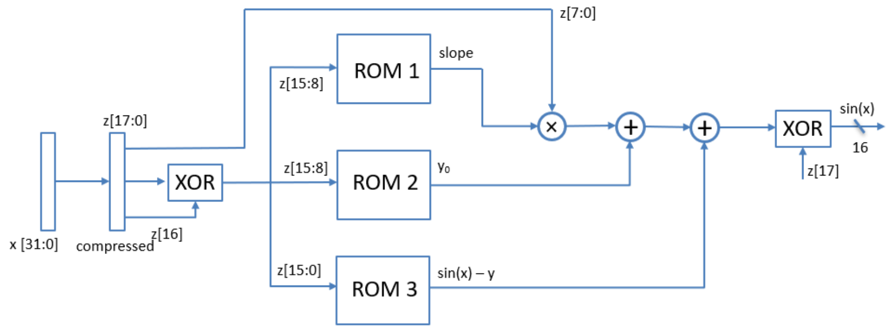


Figure 5.4: Linear Interpolation for computing sinusoid values using three ROM structure

In the paper [10], authors have proposed a phase to sinusoid amplitude converter based on first order polynomial approximation of the sine function.

In this architecture, they use two ROM structures, one for storing the sinusoid values sampled at 'N' points and one to store difference between two consecutive samples.

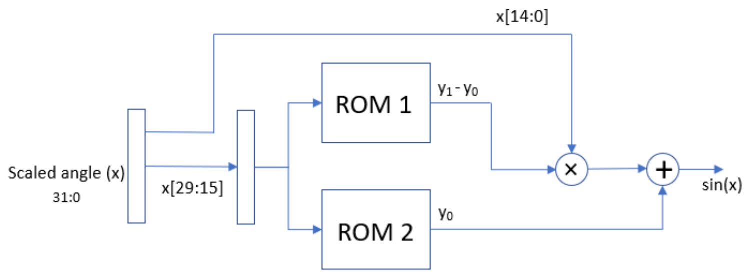


Figure 5.5: Linear Interpolation for computing sinusoid values using two ROM structure

For this method, we need 32768(2^{15}) rows in ROM1 each of width 32 bits , 32768(2^{15}) rows in ROM2 each of width 13 bits, one adder (32 - bit) and one multiplier(15 x 14).

5.3.2 Proposed method

Now, given an angle(x) in 32 bits, the two MSB bits x[31:30] can be used to represent the quadrant, x[29:15] to address the ROM and LSB 15 bits x[14:0] for linear interpolation.

Fig 5.2 shows the architecture we propose in this thesis.

x_0 = x[29:15]

y_0 = lookup table(x[29:15])

x_1 = x[29:15] + 1

y_1 = lookup table(x[29:15] + 1)

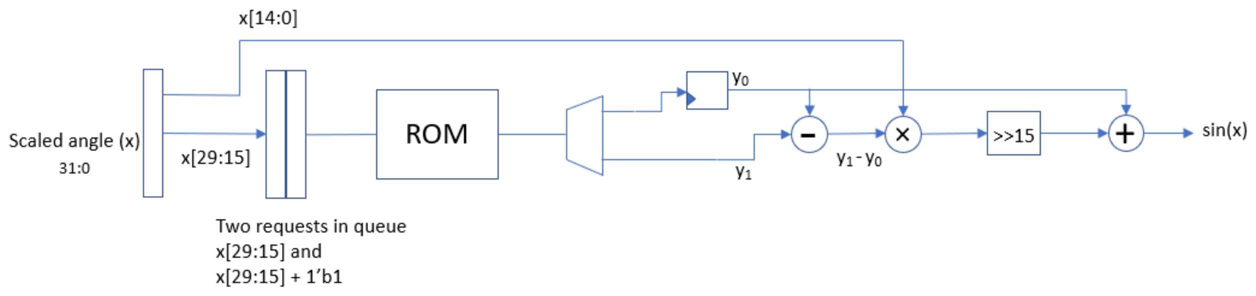


Figure 5.6: Proposed method

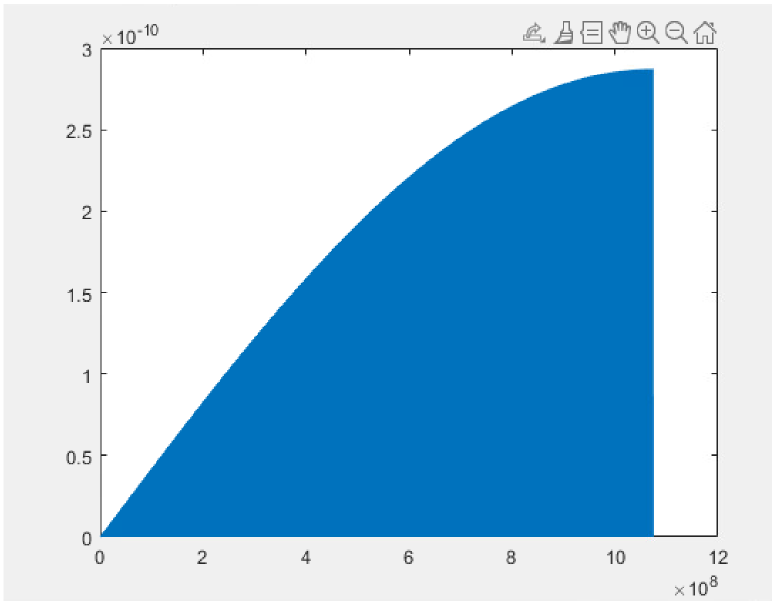


Figure 5.7: Absolute error b/w Original and interpolated signal

Thus, to compute the required value y, we need two accesses to the lookup table, one for y_0 and another for y_1, two 32-bit adders and one multiplier (15 x 14).

5.3.3 Skipping Rows

As shown in the Fig 5.6, the absolute error is a monotonically increasing function i.e., Error($\theta = \pi/2$) > Error(θ = 0). This is because the value of the sinusoid approaches to ‘1’ as θ approaches to π/2. The absolute error is directly proportional to the actual magnitude in this case. (Our emphasis here is to make absolute error as low as possible rather than the relative error).

As the absolute error between the actual value and interpolated value is less when θ is near 0, we can take advantage to skip few more rows in the beginning.

The total 32768 rows can be divided into two groups:

Group 1 : consists of 4096 (0 – 4095) values.

Group 2 : consists of remaining 28,672 (4096 – 32,768) values.

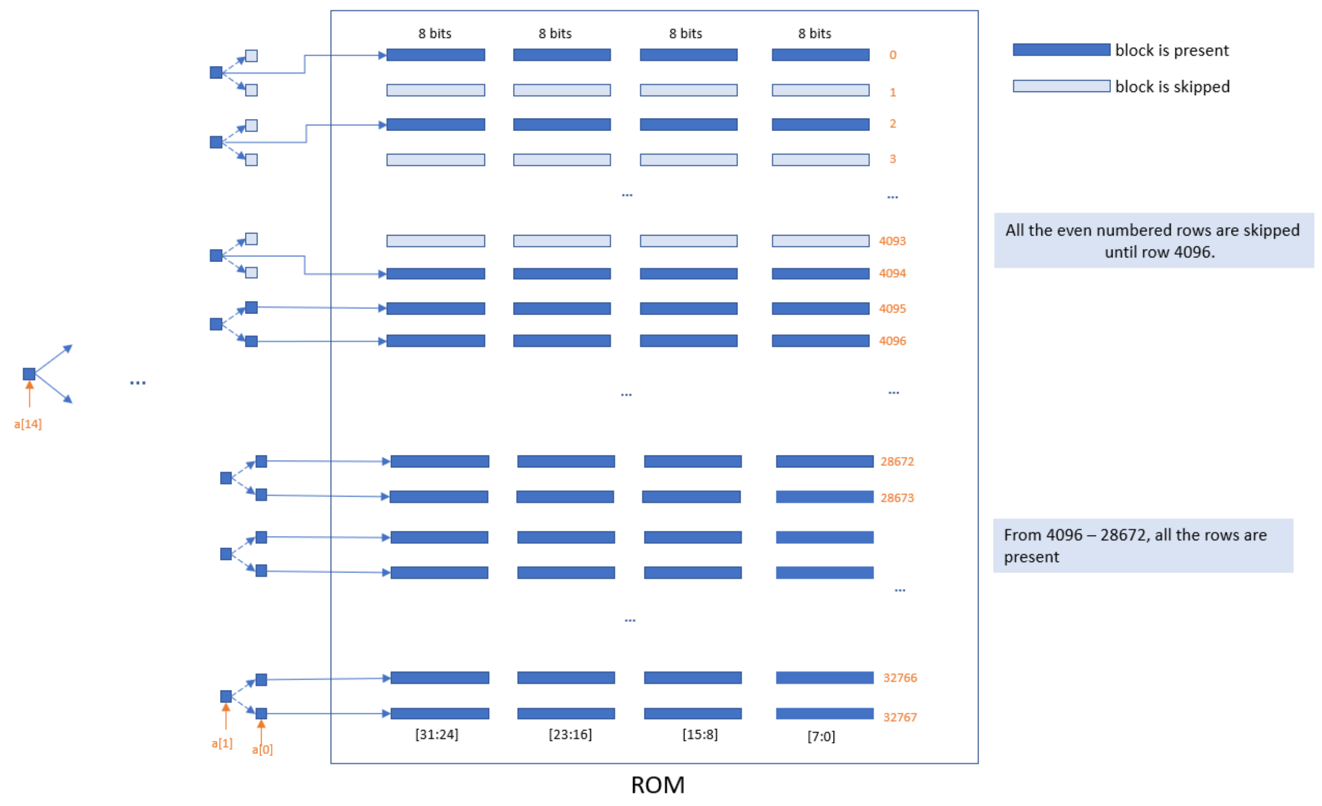


Figure 5.8: Skipped rows

In the Group 1, instead of storing all the 4096 values, we can store only half of them i.e., 2048 values by skipping the odd numbered rows(1,3 … 2047). Group 2 stays as it is.

In our memory, we store the values in the block of 8 bits as shown in the fig 5.6 However, this comes at a cost of complicated decoder, i.e., when ($x < 4094$ and x is even), row(x) and row($x + 1$) should have the same word_en signal.

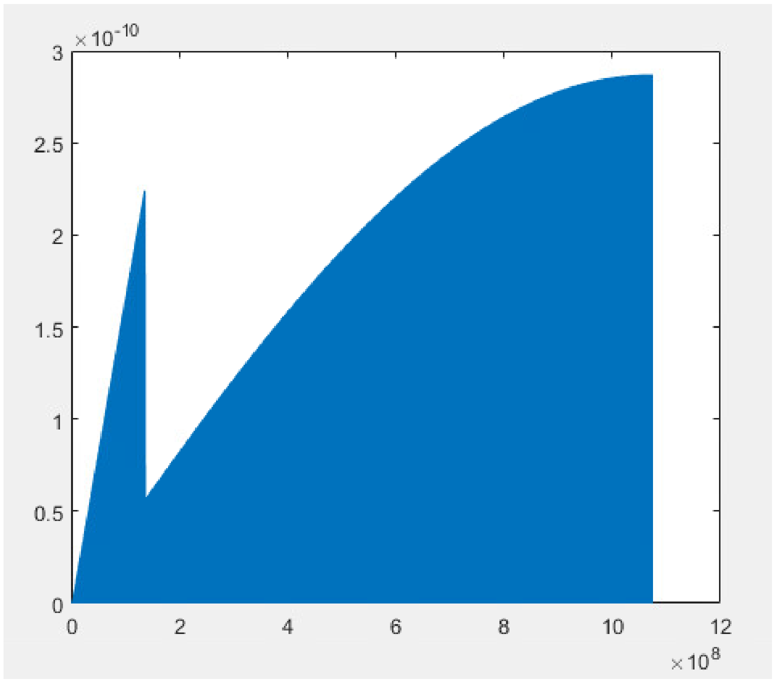
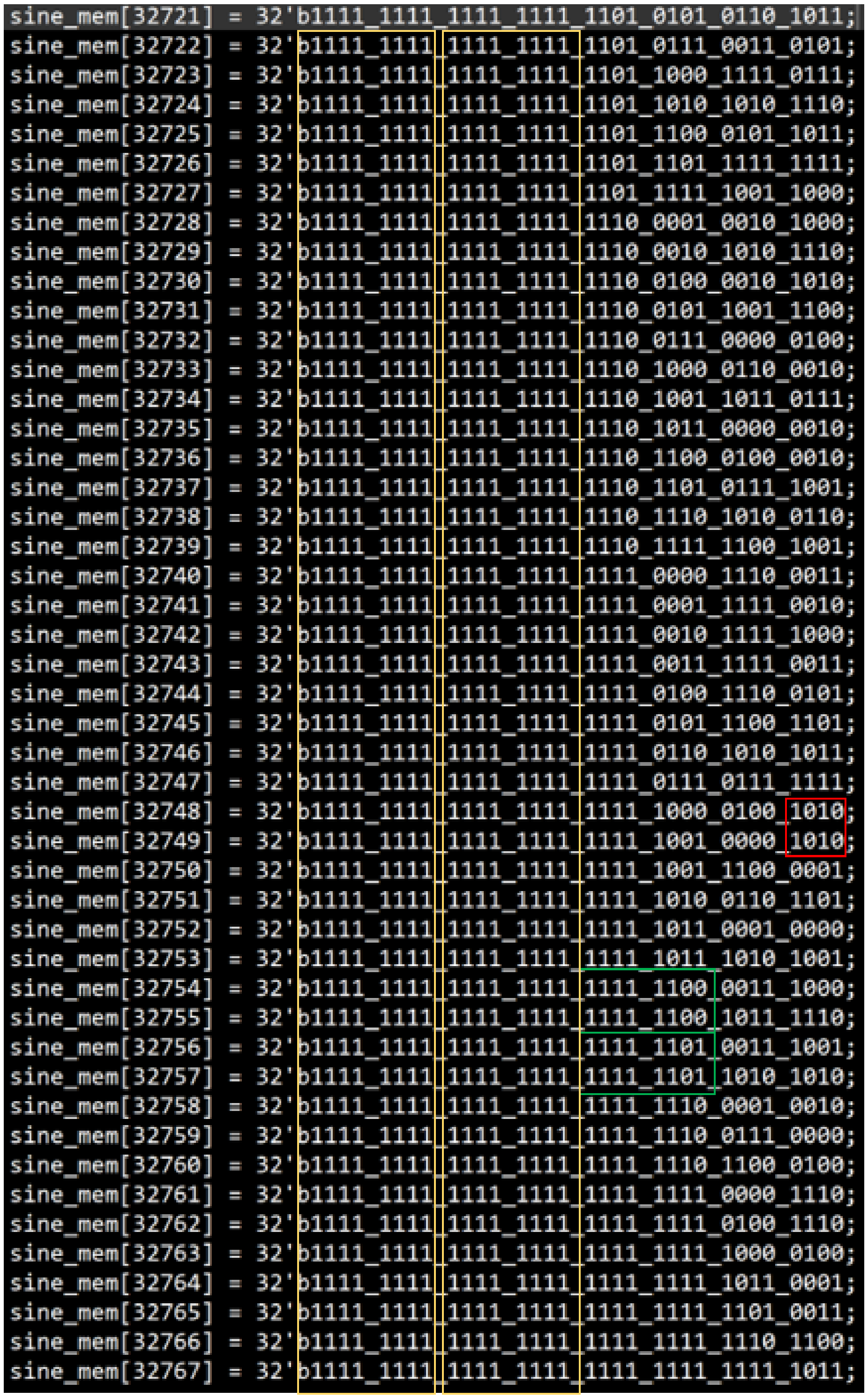


Figure 5.9: Absolute Error after skipping rows near θ=0

5.3.4 Skipping Columns

As shown in the figure 5.9, yellow rectangles capture all the ‘1111_1111’ pattern and green and red rectangles capture the repetitive patterns. Instead of storing these values separately, we can store one value and share it among all the rows.

This comes at a cost of another complication in the decoder design requiring OR gates to share the word_en signal for rows having the same pattern.



```
sine_mem[32721] = 32'b1111_1111_1111_1111_1101_0101_0110_1011;
sine_mem[32722] = 32'b1111_1111_1111_1111_1101_0111_0011_0101;
sine_mem[32723] = 32'b1111_1111_1111_1111_1101_1000_1111_0111;
sine_mem[32724] = 32'b1111_1111_1111_1111_1101_1010_1010_1110;
sine_mem[32725] = 32'b1111_1111_1111_1111_1101_1100_0101_1011;
sine_mem[32726] = 32'b1111_1111_1111_1111_1101_1101_1111_1111;
sine_mem[32727] = 32'b1111_1111_1111_1111_1101_1111_1001_1000;
sine_mem[32728] = 32'b1111_1111_1111_1111_1110_0001_0010_1000;
sine_mem[32729] = 32'b1111_1111_1111_1111_1110_0010_1010_1110;
sine_mem[32730] = 32'b1111_1111_1111_1111_1110_0100_0010_1010;
sine_mem[32731] = 32'b1111_1111_1111_1111_1110_0101_1001_1100;
sine_mem[32732] = 32'b1111_1111_1111_1111_1110_0111_0000_0100;
sine_mem[32733] = 32'b1111_1111_1111_1111_1110_1000_0110_0010;
sine_mem[32734] = 32'b1111_1111_1111_1111_1110_1001_1011_0111;
sine_mem[32735] = 32'b1111_1111_1111_1111_1110_1011_0000_0010;
sine_mem[32736] = 32'b1111_1111_1111_1111_1110_1100_0100_0010;
sine_mem[32737] = 32'b1111_1111_1111_1111_1110_1101_0111_1001;
sine_mem[32738] = 32'b1111_1111_1111_1111_1110_1110_1010_0110;
sine_mem[32739] = 32'b1111_1111_1111_1111_1110_1111_1100_1001;
sine_mem[32740] = 32'b1111_1111_1111_1111_1111_0000_1110_0011;
sine_mem[32741] = 32'b1111_1111_1111_1111_1111_0001_1111_0010;
sine_mem[32742] = 32'b1111_1111_1111_1111_1111_0010_1111_1000;
sine_mem[32743] = 32'b1111_1111_1111_1111_1111_0011_1111_0011;
sine_mem[32744] = 32'b1111_1111_1111_1111_1111_0100_1110_0101;
sine_mem[32745] = 32'b1111_1111_1111_1111_1111_0101_1100_1101;
sine_mem[32746] = 32'b1111_1111_1111_1111_1111_0110_1010_1011;
sine_mem[32747] = 32'b1111_1111_1111_1111_1111_0111_0111_1111;
sine_mem[32748] = 32'b1111_1111_1111_1111_1111_1000_0100_1010;
sine_mem[32749] = 32'b1111_1111_1111_1111_1111_1001_0000_1010;
sine_mem[32750] = 32'b1111_1111_1111_1111_1111_1001_1100_0001;
sine_mem[32751] = 32'b1111_1111_1111_1111_1111_1010_0110_1101;
sine_mem[32752] = 32'b1111_1111_1111_1111_1111_1011_0001_0000;
sine_mem[32753] = 32'b1111_1111_1111_1111_1111_1011_1010_1001;
sine_mem[32754] = 32'b1111_1111_1111_1111_1111_1100_0011_1000;
sine_mem[32755] = 32'b1111_1111_1111_1111_1111_1100_1011_1110;
sine_mem[32756] = 32'b1111_1111_1111_1111_1111_1101_0011_1001;
sine_mem[32757] = 32'b1111_1111_1111_1111_1111_1101_1010_1010;
sine_mem[32758] = 32'b1111_1111_1111_1111_1111_1110_0001_0010;
sine_mem[32759] = 32'b1111_1111_1111_1111_1111_1110_0111_0000;
sine_mem[32760] = 32'b1111_1111_1111_1111_1111_1110_1100_0100;
sine_mem[32761] = 32'b1111_1111_1111_1111_1111_1111_0000_1110;
sine_mem[32762] = 32'b1111_1111_1111_1111_1111_1111_0100_1110;
sine_mem[32763] = 32'b1111_1111_1111_1111_1111_1111_1000_0100;
sine_mem[32764] = 32'b1111_1111_1111_1111_1111_1111_1011_0001;
sine_mem[32765] = 32'b1111_1111_1111_1111_1111_1111_1101_0011;
sine_mem[32766] = 32'b1111_1111_1111_1111_1111_1111_1110_1100;
sine_mem[32767] = 32'b1111_1111_1111_1111_1111_1111_1111_1011;
```

Figure 5.10: Skipping columns

5.4 VLSI Implementation of ROM

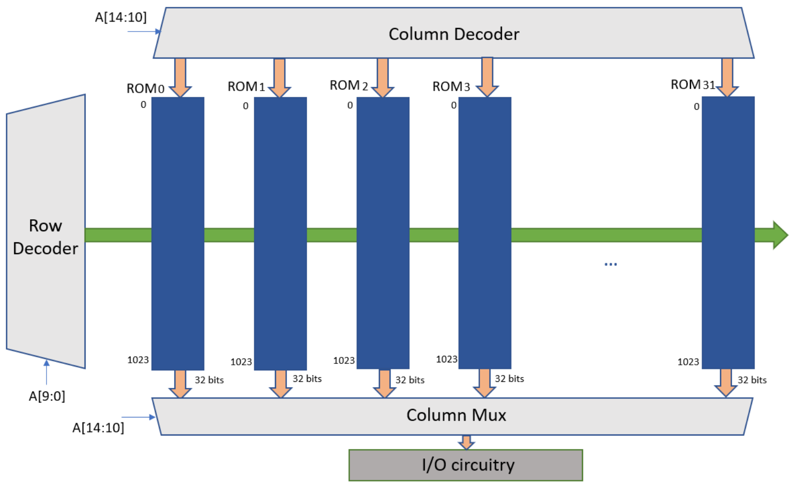


Figure 5.11: High-level implementation details of the ROM

As shown in the above figure, all the 32768 sinusoid values are divided into 32 blocks of ROM each containing 1023 rows. Each block contains all the information pertained to each bit of required 32-bit value.

This particular way of dividing the ROM into 1024 rows and 1024 columns ensures minimum area required and also capacitance of the wires will be kept lower.

Implementation type	**Bit cells for ROM (core logic)**	**Max. Absolute Error**	**Precision**	**Improvement (in bit cells)**	**Latency (in clock cycles)**	**Throughput (in samples/sec) ~clock 5ns**
Zhang's Approach	79,616 (each sine value is 16 bit only)	~$10^{-5}(2^{-16})$	~$10^{-5}(2^{-16})$	~	2	2 x 10^8
Langlois and D. Al-Khalili – Approach	1,474,560	~$10^{-10}(2^{-32})$	~$10^{-10}(2^{-32})$	~	2	2 x 10^8
Our Implementation	724,616	~$10^{-10}(2^{-32})$	~$10^{-10}(2^{-32})$	50.39%	3	10^8

Figure 5.12: Improvement

The improvement however comes at a cost as complicated decoder is required to design and also takes two clock cycles to get the required ROM data.

5.4.1 Block Diagram

The following diagrams Fig 5.10, Fig 5.11 and Fig 5.12 talk about cell arrangements in each block of the entire ROM.

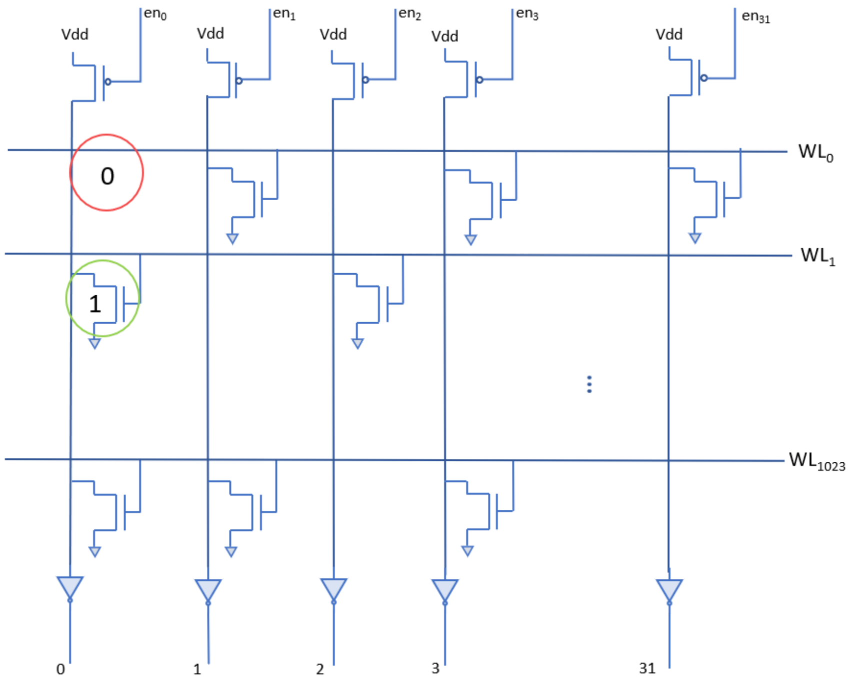


Figure 5.13: High-level implementation details of the ROM

The word lines WL_0, WL_1, … WL_{1023} come from the row decoder, whereas en_0, en_1 … en_{32} come from the column decoder.

The presence of a transistor represents a '1', and absence represents a '0'.

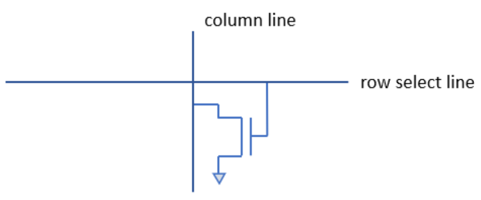


Figure 5.14: Represents a '1'

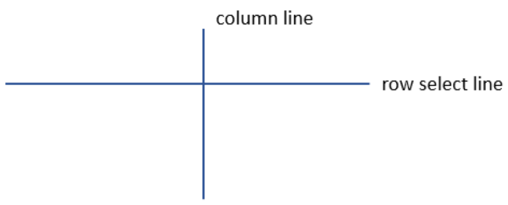


Figure 5.15: Represents a ’0’

5.4.2 Circuit Design - column and row decoders

The decoding logic can be divided into two parts – pre decoder and post decoder.

Since many gates share the exact same inputs and thus are redundant, the decoder area can be improved by factoring out the common gates. This technique is called pre - decoding[36].

Either of the two – designs, can be used for decoder. Decoders using standard cells consume lower dynamic power but more area whereas decoder using dynamic logic consume lower area but higher dynamic power[37].

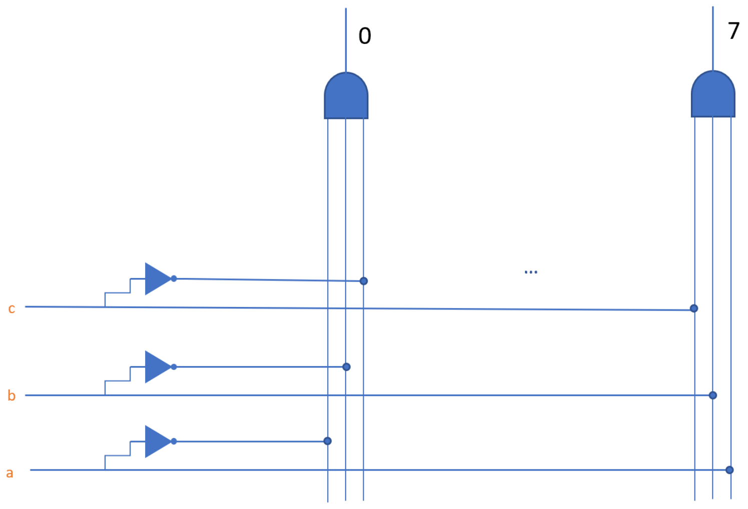


Figure 5.16: Pre - decoder using standard cells

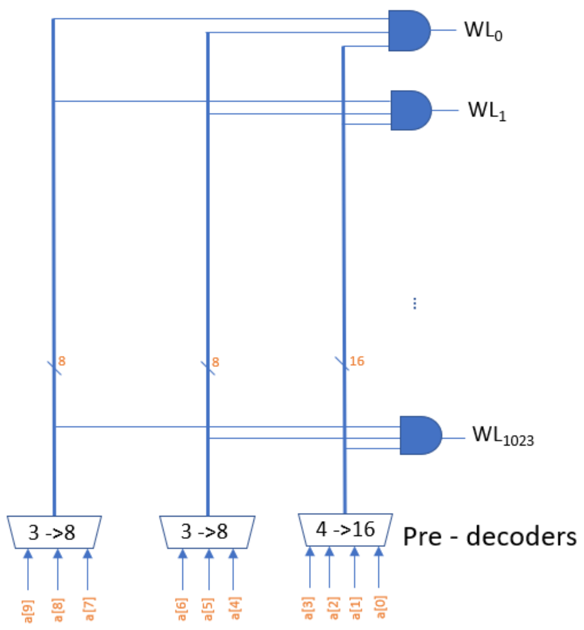


Figure 5.17: Decoder using standard cells

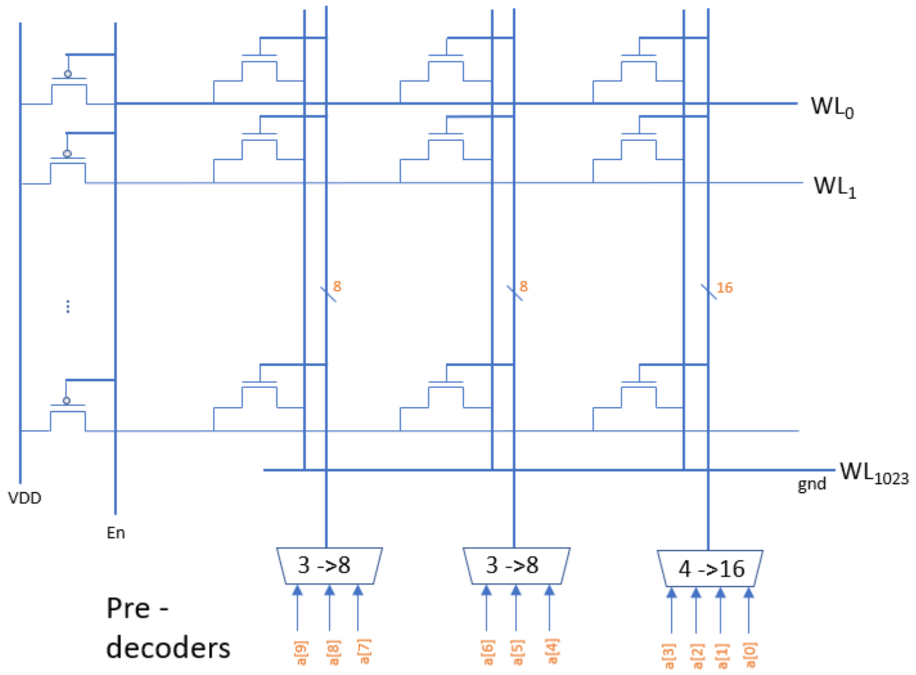


Figure 5.18: Decoder using dynamic logic

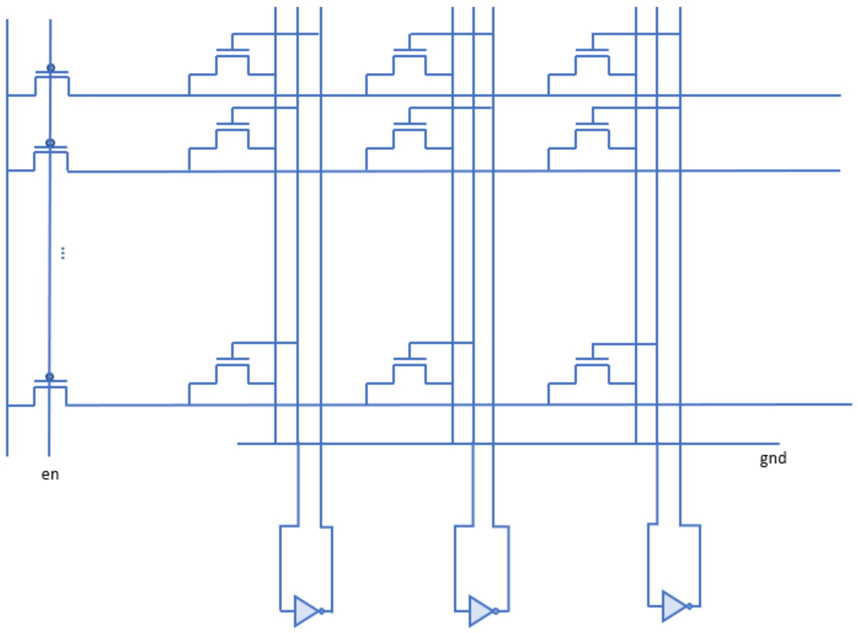


Figure 5.19: Pre - decoder using dynamic logic

5.4.3 Circuit Design - column mux

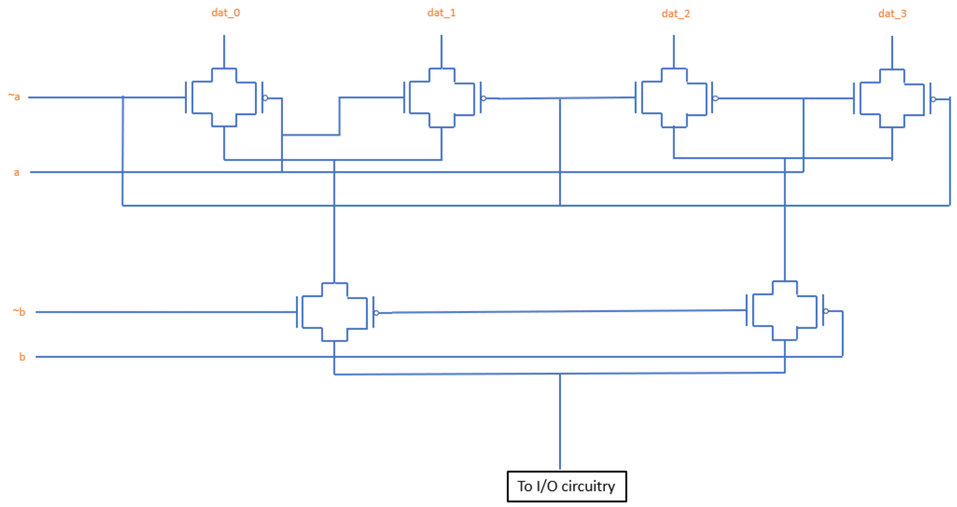


Figure 5.20: Column mux - Design 1

6. CONCLUSION

In the first problem statement of our thesis, we proposed optimized architecture to implement low rank based matrix vector multiplication for solving network equations. In a power system simulation, one of the input to the matrix vector operation is current(I) with a fundamental frequency 60 Hz. Through a mathematical proof and experimentation, we have shown when the current(I) is sampled at a high frequency, instead of computing the actual result at each time step we can use higher order differences to solve the problem using less hardware resources at a cost of slightly increased relative error. The entire flow has been prototyped on Zybo Z7-20 board using Xilinx hardware software co-design tool. Also, using the start-of-the art CAD tools combined with 45nm FREEPDK kit we implemented the low rank based matrix vector multiplication and presented the synthesized and routed results.

For the second problem statement, we proposed a custom ROM architecture along with linear interpolation technique to compute high precision sinusoid values as part of park transformation and inverse - park transformation design. We also presented two more techniques in the row and column dimension to further reduce the transistor count.

REFERENCES

[1] P. Kamranfar, S. A. Shahabi, G. Vazhbakht, and Z. Navabi, “Configurable systolic matrix multiplication,” in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pp. 336–341, 2014.

[2] L. Zhang, M. Wu, Z. Li, P. R. Kumar, L. Xie, and W. Shi, “Vlsi architecture for exciter, governor, and stabilizer in fast power system emt simulation,” in *2018 IEEE Texas Power and Energy Conference (TPEC)*, pp. 1–6, 2018.

[3] L. Zhang, B. Wang, X. Zheng, W. Shi, P. R. Kumar, and L. Xie, “A hierarchical low-rank approximation based network solver for emt simulation,” *IEEE Transactions on Power Delivery*, vol. 36, no. 1, pp. 280–288, 2021.

[4] L. Zhang, B. Wang, D. Wu, L. Xie, P. R. Kumar, and W. Shi, “Fast electromagnetic transient simulation based on hierarchical low-rank approximation,” in *2019 IEEE Power Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, pp. 1–5, 2019.

[5] R. Wunderlich, M. Püschel, and J. C. Hoe, “Accelerating blocked matrix-matrix multiplication using a software-managed memory hierarchy with dma,” in *High Performance Extreme Computing (HPEC)*, 2005.

[6] Ž. Jovanović and V. Milutinović, “Fpga accelerator for floating-point matrix multiplication,” *IET Computers & Digital Techniques*, vol. 6, no. 4, pp. 249–256, 2012.

[7] I. Sayahi, M. Machhout, and R. Tourki, “Fpga implementation of matrix-vector multiplication using xilinx system generator,” in *2018 International Conference on Advanced Systems and Electric Technologies* (*IC*$_{A}SET$)$, pp. 290--295, 2018.

[8] A. Amira and F. Bensaali, “An fpga based parameterizable system for matrix product implementation,” in *IEEE workshop on signal processing systems*, pp. 75–79, IEEE, 2002.

[9] S. J. Campbell and S. P. Khatri, “Resource and delay efficient matrix multiplication using newer fpga devices,” in *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pp. 308–311, 2006.

[10] J. Langlois and D. Al-Khalili, “Piecewise continuous linear interpolation of the sine function for direct digital frequency synthesis,” in *IEEE MTT-S International Microwave Symposium Digest, 2003*, vol. 1, pp. A65–A68, IEEE, 2003.

[11] R. Zhang, G. Chen, J. Zhang, G. Chen, and J. Yu, “A new ddfs based on unequal length piecewise linear approximation with one bit error correction,” in *2014 12th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pp. 1–3, IEEE, 2014.

[12] E. Bertagnolli, F. Hofmann, J. Willer, R. Mary, F. Lau, P. von Basse, M. Bollu, R. Thewes, U. Kollmer, U. Zimmermann, *et al.*, “Ros: An extremely high density mask rom technology based on vertical transistor cells,” in *1996 Symposium on VLSI Technology. Digest of Technical Papers*, pp. 58–59, IEEE, 1996.

[13] W. Cui and S. Wu, “Design of small area and low power consumption mask rom,” in *2007 IEEE International Conference on Integrated Circuit Design and Technology*, pp. 1–4, IEEE, 2007.

[14] C.-R. Chang, J.-S. Wang, and C.-H. Yang, “Low-power and high-speed rom modules for asic applications,” *IEEE Journal of solid-state circuits*, vol. 36, no. 10, pp. 1516–1523, 2001.

[15] C.-R. Chang and J.-S. Wang, “A new high-speed/low-power dynamic cmos logic and its application to the design of an aoi-type rom,” in *1999 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 1, pp. 254–257, IEEE, 1999.

[16] S. Kamuro, Y. Masaki, K. Sano, and S. Kimura, “A 256k rom fabricated using n-well cmos process technology,” *IEEE Journal of Solid-State Circuits*, vol. 17, no. 4, pp. 723–726, 1982.

[17] C. Melear, “Integrated memory elements on microcontroller devices,” in *Proceedings of WESCON’94*, pp. 507–514, IEEE, 1994.

[18] "Fixed point vs float point." `https://www.math.drexel.edu/~tolya/300_float.pdf`. Accessed: 2021-03-03.

[19] M. Lu *et al.*, *Arithmetic and logic in computer systems*, vol. 169. Wiley Online Library, 2004.

[20] K. K. Parhi, *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007.

[21] M. C. Karra, M. Bekakos, I. Milovanovic, and E. Milovanovic, "Fpga implementation of a unidirectional systolic array generator for matrix-vector multiplication," in *2007 IEEE International Conference on Signal Processing and Communications*, pp. 153–156, IEEE, 2007.

[22] "Zybo z7 reference manual." `https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/reference-manual`. Accessed: 2021-03-03.

[23] "Zybo z7 technical reference manual." `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`. Accessed: 2021-03-03.

[24] "Fpga tutorial." `http://www.cse.cuhk.edu.hk/~mcyang/ceng3430/2020S/Lec09%20Rapid%20Prototyping%20(I)%20-%20Integration%20of%20ARM%20and%20FPGA.pdf`. Accessed: 2021-03-03.

[25] "Dma system level design." `https://www.youtube.com/watch?v=5MCkjKhn1DM&list=PLXHMvqUANAFOviU0J8HSp0E91lLJInzX1&index=20`. Accessed: 2021-03-03.

[26] "Asic design tutorial." `https://cornell-ece5745.github.io/ece5745-tut5-asic-tools/`. Accessed: 2021-03-03.

[27] "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[28] "Library compiler tutorial." https://personal.utdallas.edu/~Xiangyu.Xu/lc/. Accessed: 2021-03-03.

[29] "Design vision." https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html. Accessed: 2021-03-03.

[30] "Prime time." https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/primetime-ds.pdf. Accessed: 2021-03-03.

[31] "Open cell library." https://si2.org/open-cell-library/. Accessed: 2021-03-03.

[32] "Freepdk45." https://www.eda.ncsu.edu/wiki/FreePDK45:Contents. Accessed: 2021-03-03.

[33] "Thesis reference." https://repository.library.fresnostate.edu/bitstream/handle/10211.3/203831/Umapathy_csu_6050N_10580.pdf?sequence=1. Accessed: 2021-03-03.

[34] "Lef def lib." http://www.signoffsemi.com/lef-def-lib/. Accessed: 2021-03-03.

[35] "Thesis reference." https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html. Accessed: 2021-03-03.

[36] N. H. Weste and K. Eshraghian, "Principles of cmos vlsi design: a systems perspective," *NASA STI/Recon Technical Report A*, vol. 85, p. 47028, 1985.

[37] "Cadence documentation." https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/circuit-design/virtuoso-analog-design-environment.html. Accessed: 2021-03-03.