Faculty Of Engineering
Computer and Systems Engineering Department

Acceleration of Numerical Solutions of Differential Equations Using FPGA-Based Emulation Technology

Authors:
Mohamed Tarek Ibn Ziad
Mohamed Hossam ElDin
Mohamed Ahmad AbdElAziz
Mohamed Nagy Eliwa
Hesham Ahmad Adel

Supervisors:
Dr. Yousra Alkabani
Dr. Watheq El-Kharashi

A thesis submitted in Partial Fulfillment of the Requirement for the B.Sc. Degree

Computer and Systems Engineering Department, Faculty of Engineering, Ain Shams University, Cairo, Egypt

July 2014

Ain Shams University

Abstract

Faculty Of Engineering
Computer and Systems Engineering Department

B.Sc.

Acceleration of Numerical Solutions of Differential Equations Using FPGA-Based Emulation Technology

by Mohamed Tarek Ibn Ziad
Mohamed Hossam ElDin
Mohamed Ahmad AbdElAziz
Mohamed Nagy Eliwa
Hesham Ahmad Adel

Numerical analysis is the study of algorithms that use numerical approximation (as opposed to general symbolic manipulations) for the problems of mathematical analysis (as distinguished from discrete mathematics). Typical applications of numerical PDE problems include but are not limited to simulation and modeling problems.

In typical numerical methods such as Finite Element Methods (FEM), PDEs are discretized in space to bring them into finite-dimensional subspace and solved by standard linear algebra subroutines. Spatial discretization could easily result in millions, even billions, of discrete grid points, which correspond to large linear system equations with the same number of unknowns. If the problem was time-dependent, in order to simulate the transient behavior of the problem, we may need to solve the linear system equations for hundreds to thousands of discrete time steps. So, implementation of numerical solutions using software is very time consuming.

Field Programmable Gate Arrays (FPGAs) raises as a suitable solution for this problem. However, area constraints still represent a major obstacle that faces FPGA designs. So, a commercial HW emulation platform was used instead to implement two different hardware-based solvers. In addition, a qualitative comparison with pure software programs such as Matlab, the best industerial tool for matrix operations and ALGLIB, a numerical analysis and data processing library are provided to highlight the merits of implementation of such algorithms on Emulation as the size of FPGAs does not fit the huge size of these algorithms.

Acknowledgements

We would like to thank Dr. Yousra Alkabani for her guidance and motivation during the project.

We would like to thank Dr. Mohamed AbdElsalam and Dr. Khaled mohamed from Mentor Graphics Egypt for their advice and encouragement. We have learnt a lot from them.

Finally, we would like to thank Eng. Mohamed H. Sakr for taking the time to help us whenever we needed.

Contents

List of Figures

List of Tables

Abbreviations

EM	**E**lectro**m**agnetic
TDR	**T**ime **D**omain **R**eflectometer
PDE	**P**artial **D**ifference **E**quation
ADE	**A**uxiliary **D**ifferential **E**quation
ASIC	**A**pplication **S**pecific **I**ntegral **C**ircuit
FPGA	**F**ield **P**rogramable **G**ate **A**rray
FEM	**F**inite **E**lement **M**ethod
MOM	**M**ethod **O**f **M**oment
BEM	**B**oundary **E**lement **M**ethod
FDTM	**F**inite **D**ifference **T**ime **D**omain
GPU	**G**raphics **P**rocessing **U**nit
CG	**C**onjugate **G**radient
PCG	**P**reconditioned **C**onjugate **G**radient
PLRC	**P**iecewise **L**inear **R**ecursive **C**onvolution
PE	**P**rocessing **E**lement
CR	**C**herenkov **R**adiation
SLS	**S**parse **L**inear **S**ystems
DPI	**D**irect **P**rogramming **I**nterface
SoC	**S**ystem **o**n **C**hip
PLI	**P**rogramming **L**anguage **I**nterface
VPI	**V**erilog **P**rocedural **I**nterface
TBX	**T**est**B**ench-**X**press

Chapter 1

Introduction

1.1 State of the art

Simulation of electromagnetic fields (in short: EM simulation) is growing in importance as a planning tool for high frequency systems. All available optimization opportunities can already be exploited in the design phase of, for example, an antenna or radar system, allowing for the performance of the final product to be maximized. The precise modeling of the relevant physical interactions in a simulation environment constitutes a large part of this optimization process and ensures that the developed system is optimally adapted to blend into its environment.

The increasing number of radar and communication systems in modern vehicles often results, however, in mutual interference between the various systems. All of those systems' parameters along with the immediate environment must also be considered in an EM simulation to produce precise results.

The main objective of an EM Simulation Process is to find an approximate solution to Maxwell's equations that satisfies given boundary conditions and a set of initial conditions.

The key steps are:

- **Creation of the Physical Model:**
  Drawing/Importing Layout Geometry, Assigning Materials, etc...

- **EM Simulation Setup:**

  Defining Boundary Conditions, Ports, Simulation Settings, etc...

- **Perform the EM Simulation:**

  1 - Discretizing the physical model into mesh cells and approximating the field-/current using a local function (Expansion/Basis function)

  2 - Adjusting the function coefficients until the boundary conditions are satisfied.

- **Post-processing:**

  Calculating S-parameters, TDR Response, Antenna Far Field Patterns, etc...

The third step is the most important, as it depends on the type of solver used. There is a variety of numerical techniques for solving Maxwell's equations and Partial Differential Equations (PDEs),in general. For example, the finite difference method [1], finite element method [2], finite volume method [3], boundary element method [4], meshfree method [5], and the spectral method [6]. The finite element method and finite volume method are widely used in engineering to model problems with complicated geometries; the finite difference method is often regarded as the simplest method [7]; the meshfree method is used to facilitate accurate and stable numerical solutions for PDEs without using a mesh.

Most of the mentioned techniques result in a system of linear equations that needs to be solved simultaneously. A great part of the simulation time is consumed in solving these equations. As a result, Software-based electromagnetic solvers are often too slow. Solving such problems may easily take traditional CPUs a couple of days, even months. An alternative is to build specific computer systems using ASICs, but this approach requires a long development period, and is inflexible and costly. GPUs, provide a new approach and are based around a large number of simplified CPU-units. Unfortunately, GPUs are not suitable for problems that are data-intensive due to the long latency of memory. Based on the discussion above, HW facilities were used to accelerate these solvers.

Efforts were exerted in previous work to implement efficient solutions for solving the resultant system of linear equations on Field Programmable Gate Array (FPGA). The first FPGA was invented by Ross Freeman in 1985. FPGAs are arrays of reconfigurable

logic blocks connected by reconfigurable wiring. These form the underlying flexible fabric that can be used to build any required function. In the past decade, the capabilities of FPGAs have been greatly improved; modern FPGAs also contain highly optimized dedicated functional blocks, e.g. hardware multipliers, memory blocks, and even embedded microprocessors. From their origins as simple glue-logic to the modern day basis of a huge range of reprogrammable systems, FPGAs have now reached sufficient speed and logic density to implement highly complex systems. However, area constraints still represent a major problem that faces FPGA designs. So, we aimed to use a commercial HW emulation platform [8] instead. In general, HW emulation platforms when compared with FPGAs allow large SoC designs to be modeled in hardware which can run at several MHz speed with advanced debug visibility and run-time control. Efficiency and use of this platform is described by a case study solving for Electric field (E) and magnetic field (H) in metamaterials using Finite Element Method (FEM). Results were compared to MATLAB, the best reference benchmark for matrix calculations.

1.2 Contributions

- We motivate our solution by demonstrating the importance of the electromagnetic simulation timing problem.
- A novel idea was introduced. It is the first time to use Emulation technology to accelerate the solver of an Electromagnetic field simulator (EM simulator). As a proof of concept, A time-domain problem of solving Maxwell's equation in metamaterials using Finite Element Method (FEM) is used. Preprocessing and postprocessing FEM calculations are done on MATLAB and the solver part, that is reposible for solving sparse system of linear equations, is accelerated using Emulator.
- Two different parallel hardware architectures were presented to solve the sparse system of linear equations resultant from the FEM. The first architecture is based on a direct method (Gaussian elimination) and the second architecture is based on an iterative method (Jacobi).
- Compared to MATLAB special operator for solving equations on a 2.00GHz Core i7-2630QM CPU, a speed-up of 7.86 was achieved for solving 60,900 equations using a Gaussian design with 32-bit floating-point arithmetic on the Mentor Graphics CairoTrioR emulator with 2 AVBs (Advanced Verification Boards), whereas, a speed-up of 11.41 was achieved for solution of the same number of equations using a Jacobi design with 32-bit floating-point arithmetic.
- Compared to ALGLIB, a numerical analysis and data processing library, using C++ programming language under linux, a speed-up of 2 was achieved for solving 60,900 equations using a Gaussian design with 32-bit floating-point arithmetic on a 2.00GHz Core i7-2630QM CPU, whereas, a speed-up of 3 was achieved for solution of the same number of equations using a Jacobi design with 32-bit floating-point arithmetic.

1.3 Thesis Organization

Chapter 2 presents a brief review of prior work in the domain of using FPGA to accelerate methods used for solving systems of linear equations.

Chapter 3 introduces one basic approache that was implemented in order to solve our problem. It is based on Cramer's method. Constraints and limitations of these approaches are illustrated in detail.

Chapters 4 and 5 describes our 2 final proposed hardware designs that were implemented and tested on the emulator. The first is based on a direct method for solving system of linear equations (Gaussian elimination).The second is a more complex implementation based on an iterative method (Jacobi). It uses 64-bit floating-point arithmetic in order to get higher resolution and it is deeply pipelined.

Chapter 6 introduces a complete case study on coding the two-dimensional edge element for solving Maxwell's equations for metamaterial using MATLAB. The whole programming process is illustrated including mesh generation, calculation of the element matrices, assembly process, postprocessing of numerical solutions and how we use the proposed solutions to accelerate it. Further details about Metamaterials and their applications could be found in A.

In Chapter 7, the hardware implementations are compared with software implementations in terms of speed-up and numerical precision. Furthermore, the performance of several hardware implementations is compared based on logic requirements and clock rates.

Chapter 8 discusses the conclusions of the study, and gives some recommendations for possible future work.

Appendix A gives more details about the origins of metamaterials, their basic electromagnetic properties and potential applications. Also a brief overview of the related mathematical problems is provided by introducing the governing equations used to model the wave propagation in metamaterials.

Appendix B introduces the most widely used general solution technique of PDEs: the finite element method (FEM).

In appendix C, we give a detailed description of the system of linear equations and the mathematical methods used to solve it.

Floating point standard and the characteristics of the used modules are discussed in Appendix D.

Appendix E discusses the concept of SystemVerilog Direct Programming Interface (DPI), which is used to interface SystemVerilog with foreign languages such as C and C++. That is used in our project to update the memory contents and catch the output during run time easily.

Chapter 2

Related work

This chapter surveys some prior work in solving systems of linear equations on FPGA hardware and highlights some previous trials to deal with metamaterials.

For iterative solvers, the bottleneck is a matrix-vector multiplication as in each refinement of the guess vector it requires $O(N^2)$ operations. The work in [9] and [10] implemented double precision sparse matrix-vector multiplication for FPGA. A sparse matrix is a matrix that has many zero elements and a dense matrix is a matrix with few zero elements.

In [9], the matrix and vector are broken up into blocks so that the matrix and vector can be of any size and are not limited by on-chip memory. A block matrix-vector multiplication method is performed until all the blocks have been multiplied. A matrix-vector multiplication can be broken up into one large dot-product operation per row of the matrix. Each such operation can be further broken into smaller dot products with additional logic needed to accumulate the results. The design in [9] has one dot-product operator with accumulation logic so that it can effectively perform a larger dot-product operation for the entire row of the matrix block. Each row of the matrix is computed in this manner until all the rows in the matrix have been used. Without the accumulation logic, the dot-product operator has to have an input size equal to the matrix block size. With the use of accumulation logic, the design can use a small input size dot-product operator, which uses less resources at the cost of longer latency than a large dot-product. Also, the block size of the matrix-vector multiplication can be independent of the input size of the dot-product operator.

The work in [11] implemented a conjugate gradient (CG) and a Jacobi iterative solver for sparse matrices on FPGA in double precision. This work used a SRC-6 Workstation which has dual 2.8GHz Xeon processors with a 5120KB cache and 1GB of RAM and two Xilinx Virtex II 6000 FPGAs. For the CG solver, only the matrix-vector multiplication was implemented on the FPGA while the remainder of the algorithm was executed in software from the SPARSKIT library [12], which has routines for sparse matrix computation. For the Jacobi iterative solver, the whole algorithm was implemented on the FPGA. Both designs need to perform a matrix-vector multiplication, which is the bottleneck of the algorithm. To perform the multiplication, both designs use a dot-product operator and additional logic to accumulate the results similar to the work in [9]. The entire matrix is stored in on-chip memory on the FPGA and so there is a limit on the matrix size. The matrix size is limited to 4,096 for the CG solver and 2,048 for the Jacobi iterative solver. By having the whole matrix in on-chip memory, the memory bandwidth requirement of the design is reduced as the data only has to be loaded once when performing the entire algorithm and so performance is not memory bandwidth limited. This work reported a speed up of 2.4 for the CG using the Virtex II over running the entire CG algorithm from SPARSKIT on the 2.8 GHz Xeon processor. The Jacobi iterative solver achieved a speed-up of 2.2 using the Virtex II over software from SPARSKIT running on the 2.8 GHz Xeon processor.

The work in [13] implemented a complete conjugate gradient (CG) for dense matrices on the FPGA in single precision. To perform the matrix-vector multiplication, the design implemented one dot-product operator, similar to the one in Figure 2.4, that operates on a whole row of the matrix. The input size of the dot-product operator is the size of the matrix, N. Each row of the matrix is sequentially operated in turn to complete the matrix-vector multiplication. Thus, a large amount of resources is dedicated to creating such a large dot-product operator and the input size of the dot-product operator limits the matrix size. All the data necessary to perform the CG algorithm is stored in on-chip memory. The design is fully pipelined and has a throughput of one iteration of the CG algorithm per cycle. However, if there is only one matrix to solve, most of time the components in the design are waiting for the next iteration. To maximize performance, multiple problem matrices with the same matrix size have to be solved at the same time to fully utilize the components in the design. On a Xilinx Virtex II 6000 with multiple matrices, the performance achieved is 5 GFLOPS in single precision and the matrix is

limited to a size of 16. For a Virtex 5 LX330, the performance achieved is 35 GFLOPS in single precision and the matrix is limited to a size of 58.

In all these prior works for iterative solvers except for [9], a limit is imposed on the matrix size based on the on-chip memory capacity of the FPGA. For [9], there is no reuse of matrix data. For each iteration of the solver, the matrix has to be loaded from off-chip memory onto the FPGA to perform one matrix-vector multiplication. Thus, the memory bandwidth is $O(N^2)$ while the computation complexity is also $O(N^2)$; therefore, a large memory bandwidth is needed in order to obtain high performance. The performance is likely to be memory-bandwidth limited. For matrices of limited size, the entire matrix can be stored on-chip and reused for each iteration of the solver. Since the matrix only has to be loaded once, the memory bandwidth requirement can be amortized across all the iterations of the algorithm. Thus, for the same amount of memory bandwidth, more computations can be performed. Thus the performance is limited by the computations and not by memory bandwidth. *It seems that for iterative solvers, either the matrix size is limited or the performance will be bandwidth limited.*

For direct solvers, the computation complexity order, $O(N^3)$, is higher than iterative solvers for the same matrix. Because the computation complexity is higher than the memory bandwidth required to load the matrix, performance will be limited by computation and not by memory bandwidth. Direct solvers are commonly used to solve dense matrices. Iterative solvers either require less computation but do not guarantee convergence or require the same order of computation as direct solvers to guarantee convergence. Therefore, direct solvers are still very useful today in scientific applications.

The work in [14] implemented a direct solver using the same LU factorization method employed in our work. It used a circular linear array of processing elements (PEs) in double precision as shown in Figure 2.1. One of the PEs (PE_0) has a divider while the other PEs have a multiplier and adder. Matrix elements are passed from PE to PE starting with PE_0. PE_0 performs the normalization of the column elements while the other PEs perform the matrix multiplication and subtraction operations. The implemented design stores all the matrix elements in on-chip memory and so it imposed a limit on the matrix size. A blocking version to remove the matrix size limit was proposed, but was not implemented, in [15]. This design achieved 4 GFLOPS in double precision on a Virtex-II Pro XC2VP100. Compared to software from an AMD optimized library called

ACML [16] running on a 2.2 GHz Opteron with a L2 cache of 1MB, this work reported a speed-up of about 1.2 in double precision using a Virtex-II Pro over software.

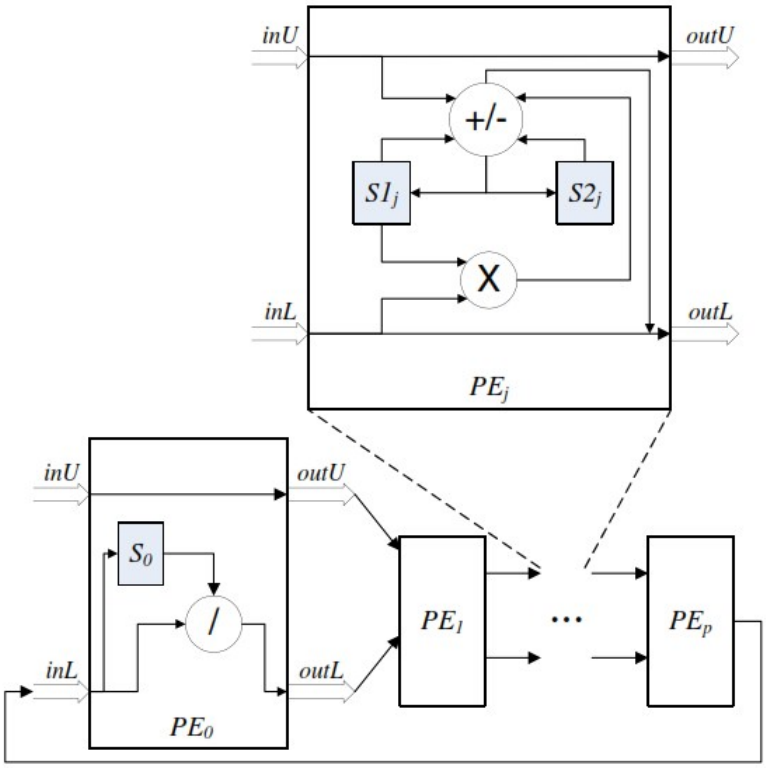

FIGURE 2.1: LU Factorization Design Block Diagram.

The work presented in this thesis can solve systems of linear equations of any size up to the capacity of the emulator memory, which is an important feature as it is the largest matrices which most need accelerated solutions. many previous works are impractical when dealing with larger matrices due to memory-bandwidth limits.

Now, let us have a look on EM simulation and metamaterials. In 1968, Veselago[17] theoretically stated that the double negative (DNG) mediums, whose permittivity and permeability are both of negative values, would have distinct electromagnetic characteristics. These mediums have attracted much research interest recently. Among numerical methods, the Finite-Difference Time-Domain method (FDTD), as a popular algorithm, has been successfully used in modeling the interactions of electromagnetic waves with double negative (DNG) materials [18, 19]. In the time-domain algorithm, frequency-dependent problems can be solved using the piecewise linear recursive convolution (PLRC) method [20], the auxiliary differential equation (ADE) method [20], and

the z-transform method. In [4], the z-transform was adapted and used for simulating the transient evolution of wave propagation in structures with DNG materials.

Although the FDTD method has proved to be very successful, extremely large computer memory space and a long computational time for large data sets are usually a burden even with a supercomputer[21]. For above reasons, authors in [4] examined how to improve FDTD method by using the massively parallel architecture of GPGPU cards.

Chapter 3

First approach: Cramer Method

In this chapter, we will discuss our 1st hardware approach, which is based on Cramer's method for solving systems of linear equations. We will give a brief overview about Cramer's rule. After that, we will illustrate our hardware implementation. Finally, advantages and disadvantages of this approach will be mentioned.

3.1 Overview

In linear algebra, Cramer's rule is an explicit formula for the solution of systems of linear equations with as many equations as unknowns, valid whenever the system has a unique solution. It expresses the solution in terms of the determinants of the (square) coefficient matrix and of matrices obtained from it by replacing one column by the vector of right hand sides of the equations.

Cramer's Rule is an efficient way of solving a system of linear equations via hardware. Let's consider the following example.

$$A_1x + B_1y + C_1z = D_1$$

$$A_2x + B_2y + C_2z = D_2$$

$$A_3x + B_3y + C_3z = D_3$$

The determinant T is

$$T = \begin{vmatrix} A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \end{vmatrix}, T \neq 0$$

The unique solution will be

$$x = \frac{\begin{vmatrix} D_1 & B_1 & C_1 \\ D_2 & B_2 & C_2 \\ D_3 & B_3 & C_3 \end{vmatrix}}{T}$$

$$y = \frac{\begin{vmatrix} A_1 & D_1 & C_1 \\ A_2 & D_2 & C_2 \\ A_3 & D_3 & C_3 \end{vmatrix}}{T}$$

$$z = \frac{\begin{vmatrix} A_1 & B_1 & D_1 \\ A_2 & B_2 & D_2 \\ A_3 & B_3 & D_3 \end{vmatrix}}{T}$$

In this way, we need to calculate the relative determinants of the system of linear equations then all the variables will be computed easily. We must mention that the value of T should be checked because if $T = 0$, the equations wouldn't have a unique solution.

3.2 Design modules

As an example, we will explain the design that solves 3 equations simultaneously. Designs dealing with higher numbers of equations would be based on the same hierarchy. Figure 3.1 shows the design hierarchy for the first approach in case of 3 equations.

3.2.1 Top Module

This module basically encapsulates the design. It contains one instance of eme_3*3_opt.

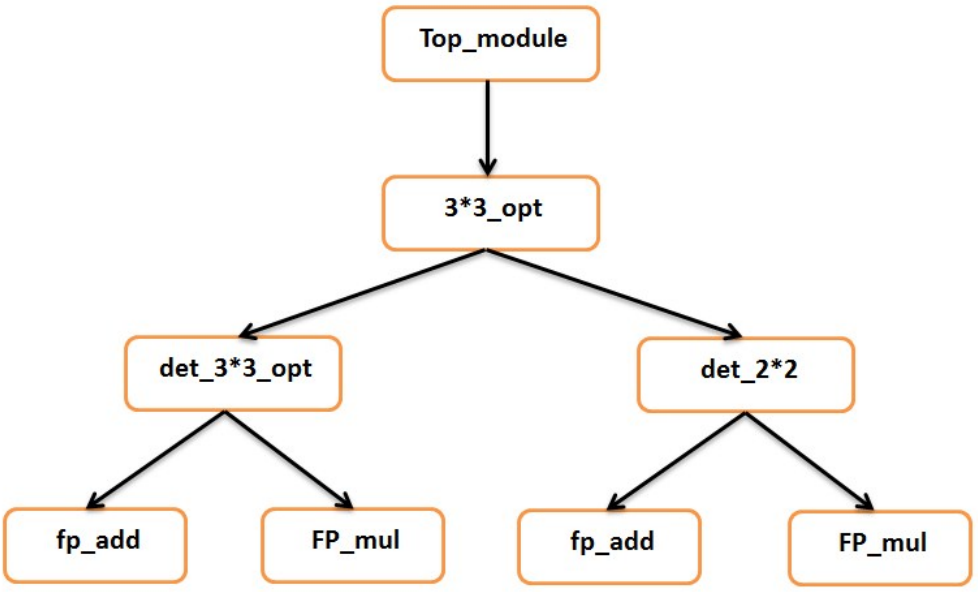

FIGURE 3.1: Design hierarchy for the first approach in case of solving 3 equations.

3.2.2 eme_3*3_opt

This module contains four instances of modules det_3*3_opt and seven instances of module det_2*2. They represent the number of unrepeated units in the method tree. The module inputs are the matrix's values and its output is the value of the determinants before and after substitution with the b's values. It calculates the value of the matrix (the main matrices 3*3 and the small matrices 2*2)

3.2.3 det_3*3_opt

This module calculates the value of the determinant 3*3. Its inputs are elements values and the sub matrices value. Module output is the value of that determinant.

Mainly, it consists of FP_mul modules to multiply the floating-point values and fp_add to add or subtract the floating-point values following the cramer's rule.

3.2.4 det_2*2

this module calculates the value of the det_2*2,and its inputs are elements values and the sub matrices value . and its output is the value of that determinant. It consists of

FP_mul modules to multiply the floating-point values and fpadd to add or subtract the floating-point values following the cramer's rule.

3.2.5 FP_mul

This module multiplies two 32 bit floating-point numbers and outputs another 32 bit floating-point value. For more information on the floating-point modules used, please refer to appendix D.

3.2.6 fp_add

This module performs floating point addition/subtraction. It uses the following sub-blocks:

- fpalign - aligns the mantissae of the inputs according to their exponents and prepends leading 1.
- special - generates signals for special inputs (infinity, NaN, signaling NaN).
- mantadd - adds (subtracts) the aligned mantissae, determines presticky bit (or of discarded mantissa).
- normlize - finds the leading one and shifts it to the front producing a normalized sum. Determines if result is denormal and does proper denorm shifting. Also calulates round, and sticky bits as well as whether the sum is 0 or inexact.
- rounder - rounds normalized mantissa according to selected rounding mode and calculates the final exponent.
- final - this assembles all the pieces and uses special case information to determine correct final result. Also determines all exception flags.

3.3 Pros and Cons

The main advantage of this design is that it is very fast compared to other methods. It needs fewer number of clock cycles to calculate the result. However, the first approach has one major disadvantage. It consumes a lot of logic resources; the number

of consumed LUTs increases dramatically as the number of equations increases. This is discussed in detail in section 7.2.1.1. As a result, it is not scalable.

That is the main reason which guided us to search for the two other approaches described later on the coming chapters.

Chapter 4

Second approach: Gaussian Elimination Method

4.1 Introduction

The main idea behind this approach is using Gauss Elimination as a method of solving linear systems of equations. The most time-consuming operations were implemented on hardware, while the rest of the operations were implemented on software to maximize the utilization of the available hardware resources.

4.2 Gauss elimination hardware algorithm

The hardware algorithm described here is slightly different from the basic one mentioned in appendix C. Minor modifications were made to improve performance, while maintaining the same results gained from the basic algorithm.

In gauss elimination, we use the matrix notation of linear equations

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{4.1}$$

First, forward elimination is used to eliminate all the elements in the lower triangle of the matrix.

$$\begin{pmatrix} a'_{1,1} & a'_{1,2} & \cdots & a'_{1,n} \\ 0 & a'_{2,2} & \cdots & a'_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a'_{n,n} \end{pmatrix} \begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{pmatrix} \tag{4.2}$$

Next, backward elimination is used to eliminate all the elements in the upper triangle of the matrix, leaving only the main diagonal.

$$\begin{pmatrix} a"_{1,1} & 0 & \cdots & 0 \\ 0 & a"_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a"_{n,n} \end{pmatrix} \begin{pmatrix} b"_1 \\ b"_2 \\ \vdots \\ b"_n \end{pmatrix} \tag{4.3}$$

All the previous operations are implemented on hardware while the last operation involving division of all elements in the right-hand-side vector by the corresponding elements in the main diagonal is implemented on software.

$$\begin{pmatrix} \frac{b"_1}{a"_{1,1}} \\ \frac{b"_2}{a"_{2,2}} \\ \vdots \\ \frac{b"_n}{a"_{n,n}} \end{pmatrix} \tag{4.4}$$

4.3 Basic design

4.3.1 Overview

This is the basic design which implements gauss elimination without any optimization. It contains three main modules memory, ALU and control unit. The memory is used for storing all the matrix elements as well as the right hand vector in little endian

architecture. The ALU is based on vector operations which are performed on the data received from the memory. Finally, the control unit used for synchronizing the ALU and the memory together, and controlling memory addresses and memory reading and writing operations.

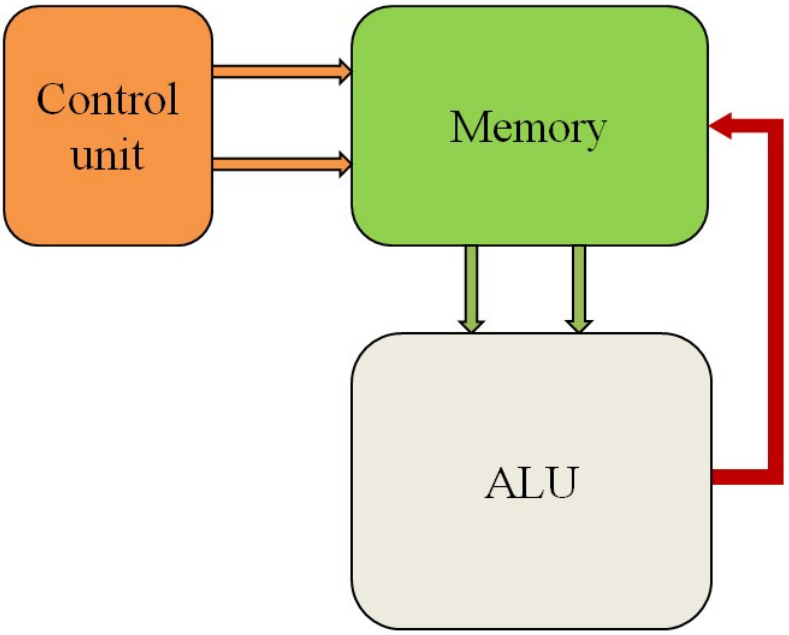

FIGURE 4.1: The architecture for Basic design of second approach.

4.3.2 Design modules

4.3.2.1 Memory

The memory contains the matrix and right hand side vector stored in the IEEE 754 floating point representation with single precision. Each memory entry represents one row from the matrix concatenated with the corresponding element from the right hand side vector, so the word size of the memory equals 32*(number of matrix columns + 1).

The memory has two addresses for reading, and one for writing. Consequently, it's possible to read two memory rows and write one memory row during the same clock cycle.

4.3.2.2 ALU

The ALU is designed to support vector floating point of single precision, which means that the operations are done vector by vector and not element by element which is the main advantage of using hardware-solutions as they allow higher degrees of parallelism than software-based solutions.

The ALU input is two rows from the memory and address from the control unit and output is one row to be saved in the memory. The ALU operates on the two rows and uses the address from the control unit to know which element to be eliminated according to gauss elimination operations.

The ALU consists of:

1. Division module which divides one element from the input row by an element from the other row.
2. Two multiplexers are used to choose the elements from the two input rows to enter the division module according to the address from the control unit.
3. Vector multiplication module which consists of number of multiplication modules equal to the number of the elements of a row, each one multiplies one element of the first input row by the output of the division operation.
4. Vector subtraction module which consists of a number of subtraction modules equal to the number of the elements of a row, each one subtract one output of the multiplication modules from the corresponding element in the second input row.

4.3.2.3 Control unit

The main purpose of the control unit is synchronizing the ALU and the memory, and it consists of a number of signals used to fulfill its mission:

1. Memory addresses, the control unit controls the two memory addresses for both reading and writing.
2. Read and write signal which control when data is read from or writting into the memory.

3. Halt signal this signal is used to indicate that the equations have been solved.

The control unit consists mainly of three counters:

1. The first counter is a simple up counter used to control the read and write signals as it counts from 0 to 7, as long as the counter holds a value between 0 and 6 the signal represents a read operation, when the counter's value becomes 7 the signal is write.

2. The second counter is an up-down counter and can be loaded with an initial value to start counting from, it is used to control one of the memory addresses as the counter value is the value of the address corresponding to the row for which elimination takes place. The counter inputs:

   (a) Enable signal: used to allow the counter to count. It is controlled by the first counter.

   (b) Forward-backward signal: this signal indicates whether the system is working in forward or backward elimination, and it is controlled by the third counter. The second counter is counting up or down according to forward-backward signal.

   (c) Load value: used to initialize the counter to begin counting from this value. It is equal to third counter's value + 1 in forward elimination and third counter's value -1 in backward elimination.

   (d) Set signal: used to initialize the counter with the load value. This signal is controlled according to the value of the first counter and the value of the second counter itself.

3. The third counter is also an up-down counter but unlike the second counter it does not have a load value for initialization. The third counter is mainly used to control the second address of the memory as its counter value is the value of the address. It also controls the halt signal as the halt signal is high when the counter value is zero during backward elimination. The third counter has an enable signal like the second which works like the set signal of the second counter.

4.3.3 Design disadvantages

1. Slow design as the time complexity is $O(n^2)$ and a lot of time is wasted in eliminating elements which already equal zero.
2. A lot of hardware resources are needed as the ALU size increases linearly with the number of equations due to the use of vector modules.
3. Large memory is used.

4.4 Gaussain with skip

4.4.1 Overview

In this design we aim to solve one of the disadvantages of the basic design which is the time wasted on eliminating elements which are already equal to zero. After studying the output of the finite element method thoroughly, we found out that most of the matrix elements are zeros which means that a lot of time is wasted on eliminating zero elements.

In this design we proposed a solution of checking elements before sending them to the ALU, skipping any element whose value is equal to zero and advancing to the next row. In the previous design eliminating one element consumes 8 clock cycles, one cycle for reading the row of data, 6 cycles for ALU operations until the result is available, and one cycle for writing the data into the memory. Now if the element is zero it consumes two cycles before it is skipped, one cycle for reading the data from memory and one cycle for checking its value. However, if the element's value is not zero, it consumes the usual 8 clock cycles.

4.4.2 Design modules

4.4.2.1 Memory

The same as the previous design.

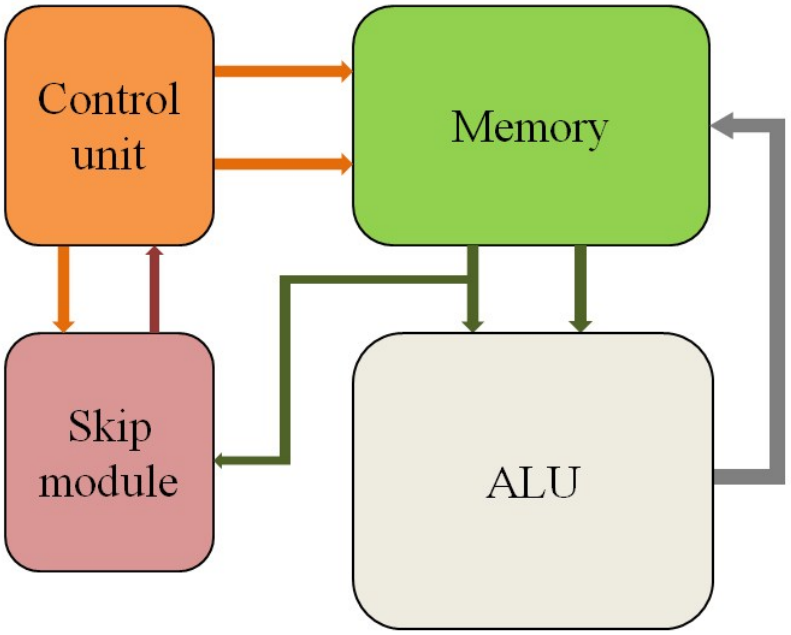

FIGURE 4.2: The architecture for Gaussian with skip.

4.4.2.2 ALU

The same as the previous design.

4.4.2.3 Control unit

Most of the control unit design is the same as the previous design except for some minor changes, a new input named skip signal which results from the skip checking module that makes the control unit skip the current row of data and go for the next one, and a new output named check signal used by the checking module as an enable signal.

As discussed in the previous design the control unit mainly depends on three counters having the same design and hierarchy used here, but for the first counter there is a new input, the skip signal, that forces the counter to reset. Also the second counter's enable signal depends on the skip signal besides the value of the first counter.

4.4.2.4 Skip checking module

The main purpose of this module is to check if the element to be eliminated is already zero or not and sends a skip signal to the control unit to skip the current row of data

and advance to the next row if the element to be eliminated has a value of zero. Skip checking module has three inputs:

1. Input data which is one of the rows entering the ALU which has the element to be eliminated.
2. Address, this address is needed to know which element in the row is to be eliminated in order to perform the skip check on that element, this can be done as the module contains a multiplexer that chooses which element in the current row is to be checked according to this address.
3. Check signal this signal acts as enable to the module to specify the time at which checking occurs, and this signal is controlled by the control unit.

The only output of the module is the skip signal which lets the control unit skip this row of data and advance to the next one, this signal is simply controlled by an NOR gate that operates on the element's bits. If all of them are zeroes, skip signal is high otherwise skip signal is low.

4.4.3 Design disadvantages

1. This design is much faster than the previous design but still not fast enough because the skipped elements still waste a reasonable amount of time.
2. A lot of hardware resources are needed as the ALU size increases linearly with number of equations due to the use of vector modules.
3. Large memory is used.

4.5 Gaussian with full skipping

4.5.1 Overview

In this design we try to improve the performance of the previous design by reducing the time wasted in skipping. To achieve this goal, full skipping is used instead of regular skipping, which means that the zero checking is done on a column by column basis

instead of element by element. After checking, the addresses of the rows for which elimination is performed are obtained. Sounds simple but it is a fact the main challenge is how to design the memory, so that we could read a column of the matrix for zero checking and read a row of the matrix to enter the ALU and all read operations should take only one cycle.

To design a memory with those characteristics we chose the distributed memory model, so we use a number of memory segments instead of one big memory, each memory segment stores two rows of the matrix and has two reading ports so in only one clock cycle we could read all the memory contents. After reading all memory contents we use multiplexers to get the desired columns for zero checking and the desired rows to enter the ALU.

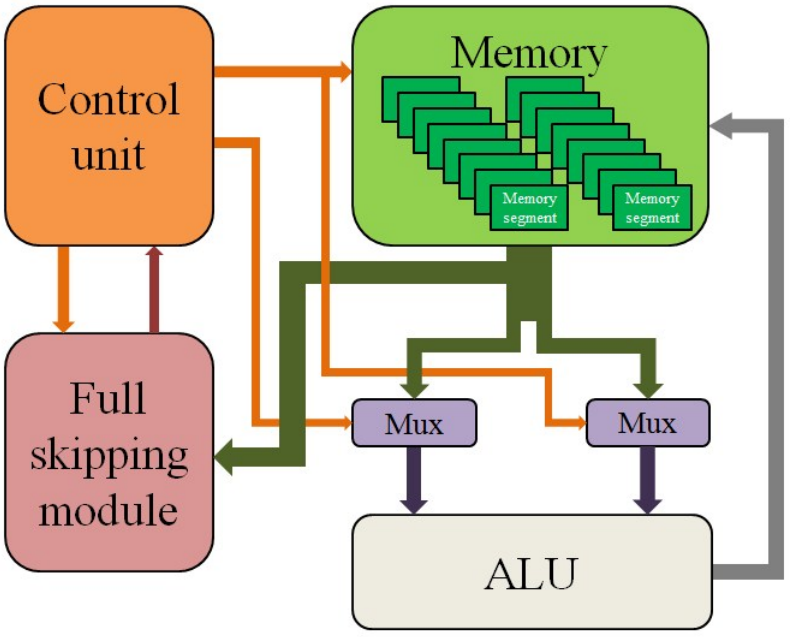

FIGURE 4.3: The architecture for Gaussian with full skipping.

4.5.2 Design modules

4.5.2.1 Memory

As discussed before the memory follows the distributed memory model, the memory doesn't have read addresses as all the memory contents are read all the time. Memory has three main inputs, write address as we write in one row only at a time, write enable signal to enable writing on the write address and input data which is the data to be

written. Memory has only one output, the data output, which carries all the data contents of the memory. Memory consists of:

1. Memory segments: the memory contains many memory segments equal to the number of the matrix rows divided by two as each memory segment stores two rows of the matrix. Memory segments have the same inputs and outputs of the memory, all the data outputs of the segments are concatenated to form the memory data output.

2. Decoder: the main purpose of the decoder is to indicate which memory segment will be enabled for writing according to the writing address and also indicates the address of the segment (0 or 1) on which the writing occurs. Decoder input is the write address and the output is a vector of which segment will be enabled and the address inside the segment.

4.5.2.2 ALU

The same as the previous design.

4.5.2.3 Control unit

This control unit is different from the previous designs as it has the first and the third counters only, the second counter is not suitable for this design as it does not read from memory row by row as before, instead it decides the row to be read from memory according to the full skipper module and the control unit has an encoder to fulfill this task and it will be discussed in details. Control unit has a new input from the full skipper module which is the skip vector instead of the skip signal in the previous design; the length of the skip vector is equal to the length of the matrix column and each bit of this vector represents which rows will be skipped and which rows will enter the ALU. Control unit outputs:

1. Write enable signal: this signal is used to enable writing on the memory and is controlled by the first counter like before.

2. Halt signal: used to indicate that the equations have been solved according to the value of the third counter.

3. Forward - backward signal: used to indicate on which mode the system currently runs; forward elimination mode or backward elimination mode and this signal is used by the full skipper module.

4. Memory addresses: there are two memory addresses, the writing address which is controlled using the skip vector and the encoder, the other address is not for the memory as the memory has one address only. The other is used by the multiplexers to choose a row of data from the memory output data, and this address is controlled by the third counter.

Control unit consists of:

1. First counter: has the same design and role as the previous design.

2. Third counter: has the same design and role as the previous design except for its enable signal which is controlled by the encoder.

3. Encoder: the main role of the encoder is to encode the skip vector and output the memory writing address, it is done as the skip vector has several bits that are equal to one. The encoder outputs the address corresponding to the least significant bit then after the ALU operates on the row corresponding to this address and writes in the memory with the eliminated element the full skipper will find it and change the value of the skip vector by clearing the bit corresponding to the previous row so the encoder outputs the next address and so on. After clearing all the bits of the skip vector the encoder sets the enable signal of the third counter.

   The challenge in designing the encoder is that it needs to be a parameterized module to have its size proportional to the length of the skip vector, to achieve this goal we designed the modules using multiplexers of fixed size and having a number equal to the length of the skip vector, and connected as shown in Figure .4.4.

4.5.2.4 Full skipper

The main purpose of this module is to drive the skip vector which is used by the control unit, and perform the zero checking on each element in the matrix column. Full skipper inputs:

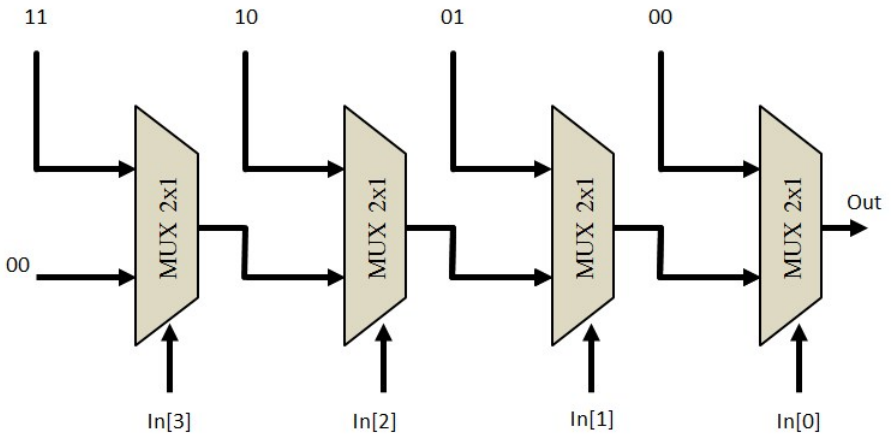

FIGURE 4.4: Internal design of the encoder based on 2x1 Muxs.

1. Data input: data input it is the whole data that comes from the memory which is, in fact, the whole memory's contents

2. Address: it is the address coming from the control unit and it is used to decide on the column of the matrix that will be subject to the zero checking process

3. Forward-backward signal: this signal is received from the control unit and used to determine the mode of elimination (forward or backward), this signal and the address are used in the full skipper to form a mask that will control zero checking process.

The full skipper contains a number of multiplexers equal to the number of the rows in the matrix, each multiplexer chooses an element according to the input address from its corresponding row, then all the choses elements subject to zero checking which is a simple OR gate operateing on the element bits, outputting zero for zero value and one otherwise. A mask is formed according to the input address and the forward-backward signal to mask the rows with lower addresses than the input address from the zero checking in the forward mode, while in the backward mode mask the rows of addresses greater than the input address.

4.5.2.5 Multiplexers

The design contains two multiplexers. Their purpose is to choose which rows of the output data of the memory will enter the ALU, and these rows are chosen according to the addresses generated by the control unit.

4.5.3 Design disadvantages

1. A lot of hardware resources are needed as the ALU size increases linearly with number of equations due to the use of vector modules.
2. Large memory is used.

4.6 Gaussian using sparse model

4.6.1 Overview

In this design we tried to solve most of the problems of the previous design related to performance and resource utilization. This design uses a new concept in storing the matrix in the memory known as the sparse model, the sparse model is usually used for matrices whose elements' values are mostly zeroes, so it stores only the non-zero elements and their locations (indices) in the matrix, this model has a huge impact on the memory size which is far smaller than the other designs.

After studying the output of the finite element method, we found that it has the following properties:

1. All elements in the main diagonal of the matrix are non-zero elements.
2. Output matrix is symmetric.
3. Maximum number of non-zero elements in one row is three.

We tried to exploit those properties in this design to improve performance and hardware utilization. The design contains ALU, control unit and two types of memories. The concept of zero skipping is not used in this design since that problem has been solved by using a sparse memory model.

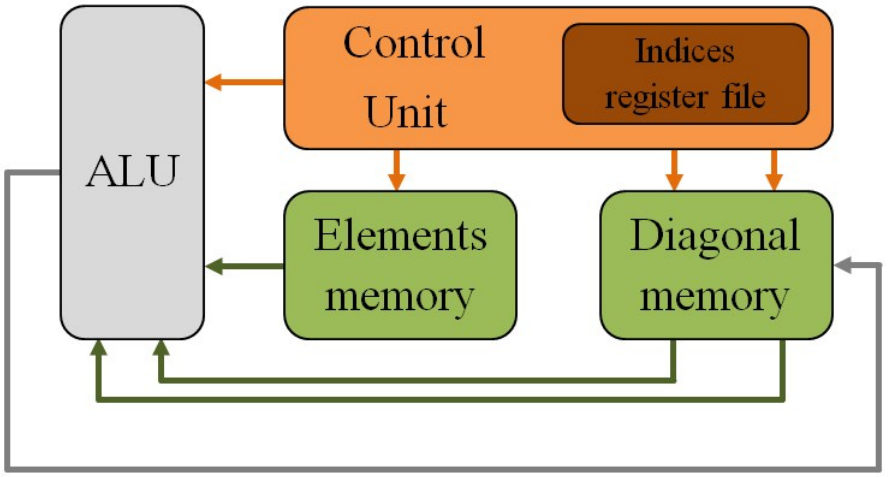

FIGURE 4.5: The architecture for Gaussian using sparse model.

4.6.2 Design modules

4.6.2.1 Diagonal memory

This memory is used to store the diagonal elements in the matrix concatenated with the elements of the right hand side elements. The word size of this memory is 64 bits; 32 bits for the diagonal elements and 32 bits for the right hand side matrix, the memory length equals the number of equations of the system. Memory inputs are:

1. Memory addresses: it has two addresses, one for reading only, and one for both reading and writing. These addresses are controlled by the control unit.
2. Read-write signal: used to control the read and write from the memory, when the signal is cleared the memory goes for writing otherwise goes for reading.
3. Halt signal: this signal is set after the equations are solved. When this signal is high, no further memory writing operations are performed.
4. Input data: the data that will be written into the memory at one of the memory addresses when both the read-write and halt signals are low.

4.6.2.2 Elements memory

This memory is used to store the non-zero non-diagonal elements, which is a read only memory, there no need to write on those elements. The memory word size is 32 bits

which is the size of the element, the length of the memory equals the number of non-zero non-diagonal elements.

The memory simply has one address for reading controlled by the control unit and one output for the data output.

4.6.2.3 ALU

This ALU is different from the other designs as it has a fixed size as we take advantage of the third property of the matrix which states that the maximum number of non-zero elements in a row is three. The inputs of the ALU:

1. Two rows from the diagonal memory.
2. One element from the elements memory.
3. Forward-backward signal.

The new ALU consists of:

1. One division module: divides the non-diagonal element by the diagonal elements from the second input row.
2. Two multiplication modules:
   (a) First module multiplies the division result by the right hand side element of the second row.
   (b) Second module has two cases:
      i. In forward elimination, it multiplies the result of division by the same element from the elements memory due to symmetry of the matrix
      ii. In backward elimination, it multiplies the result of division by zero
3. Two subtraction modules:
   (a) First module subtracts the result of the first multiplication module from the right hand side element of the first row.
   (b) Second module subtracts the result of the second multiplication module from the diagonal element of the first row.

The result of those multiplication modules are written into the diagonal memory, and overwritten at the location of the first input row in the diagonal memory.

4.6.2.4 Control unit

This control unit is different from the previous as it has a register file that stores the indices of each element in the elements memory, the size of this register file equals the length of elements memory and each entry consists of the element row index concatenated with the element column index. Control unit output:

1. Read-write signal: signal used by the diagonal memory to control its read and write operations.
2. Halt signal: used to indicate that the equations have been solved. Also used by the diagonal memory to control write operations
3. Forward-backward signal: indicates the mode of the system, forward or backward elimination, this signal is used to control ALU operations
4. Diagonal memory addresses: the read addresses of the diagonal memory. The first address is used for writing as well.
5. Elements memory address: the read address of the elements memory.

The control unit consists of two counters and register file:

1. First counter is used to control the read-write signal and drive the enable signal of the second counter
2. Second counter is used to control the address of the elements memory and drive the halt and forward-backward signals
3. The register file stores the indices of the elements of elements memory, those indices are used as diagonal memory addresses, the indices are fetched from the register file according to the second counter's value.

4.6.3 Design disadvantages

This design achieves its main goals which are improving performance, memory size, and hardware size. However, there are some disadvantages regarding the hardware resources utilization as the hardware size is fixed regardless of number of equations solved due to the fixed size of the ALU, which is a waste of the available resources.

4.7 Gaussian using sparse and clusters model

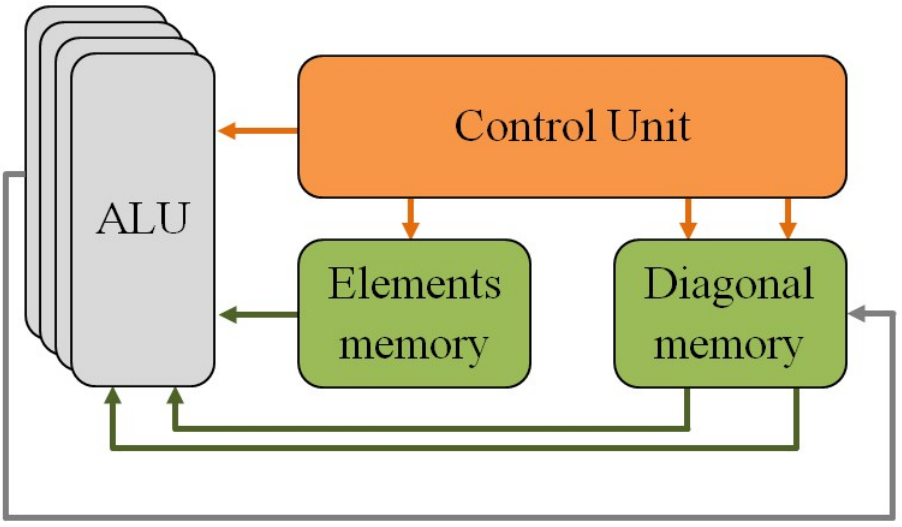

FIGURE 4.6: The architecture for Gaussian using sparse and clusters model.

4.7.1 Overview

In this design we try to solve the disadvantage of the previous design, which is the bad utilization of the available hardware resources, by improving the hardware utilization to achieve better performance levels.

The goal of this design mainly depends on the use of the clusters concept, which is discussed in details in section 6.2.2 to minimize the dependencies between equations and allow more higher levels of paralellism by using multiple ALUs in parallel.

The main advantages gained from clusters:

1. Separate equations into clusters.

2. Each cluster has fixed shape as shown in Figure 4.7, so location of the non-zero elements in each row is predictable.

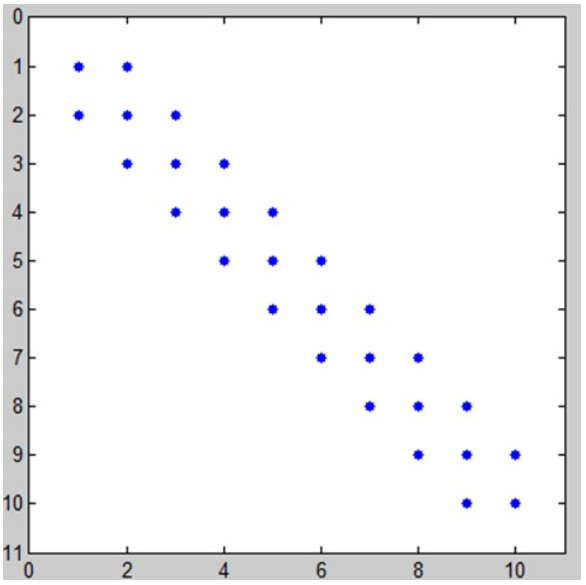

FIGURE 4.7: The fixed shape of clusters(three diagonals).

4.7.2 Design modules

4.7.2.1 Diagonal memory

Has the same features as the previous design. The only difference between them is the length and the word size. Each entry of the memory stores the diagonal element and the right hand side element concatenated with those of the other clusters, so the memory word size is 64 bit * number of clusters. The length of the diagonal memory equals the number of equations per cluster.

4.7.2.2 Elements memory

No changes occurred to the Elements memory either except for the length and word size like the diagonal memory. The memory word size is 32 * number of clusters and its length is equal to the number of non-zero non-diagonal elements per cluster.

4.7.2.3 ALU

The same ALU design as before but the system contains more than one ALU. Their numbers are equal to the number of clusters. Each ALU operates on a separate cluster and each input of data is its corresponding data portion from the memories' outputs.

4.7.2.4 Control unit

It has the same design as the previous design except that the register file has been removed due to the usage of clusters. The elements' indices are predictable and function of the value of the second counter, so the diagonal memory addresses are controlled by the second memory.

4.8 Summary

The second approach passes through many designs from the basic design which implements the Gaussian elimination techniques on hardware to achieve higher performance levels than software, By using vector floating point ALU to operate on row by row. Basic design wastes a lot of time on zero elements and the design with skip comes to solve this problem by adding skipping module which performs zero checking on each row before entering the ALU and skiping it if the element to be eliminated is already equal to zero.

The design with full skipping comes to add more improvement in performance by extending the concept of skipping, such that the full skipping module performs the zero checking on the whole column of the matrix and according to the result the control unit decides which rows will enter the ALU. To achieve these goals, the memory had to be redesigned according to the distributed memory model.

Further improvements were added when introducing the concept of sparse matrix in the design using sparse, which led to smaller hardware and memory sizes and reasonable performance, but also led to bad hardware resources utilization. The design using sparse and cluster solves the bad utilization problem by using clusters to allow further parallelism and add more ALUs that work in parallel, leading to the best timing and hardware utilization among the other designs.

Chapter 5

Third approach: Jacobi Method

In this chapter, we will give a complete explanation about our third hardware approach which is based on Jacobi method. Jacobi as an iterative method for solving system of linear equations is discussed in appendix C. Here, we will present the 4 major designs implemented in this approach and illustrate their advantages and disadvantages. The last design is the best one in terms of performance and scalability.

5.1 Basic Jacobi

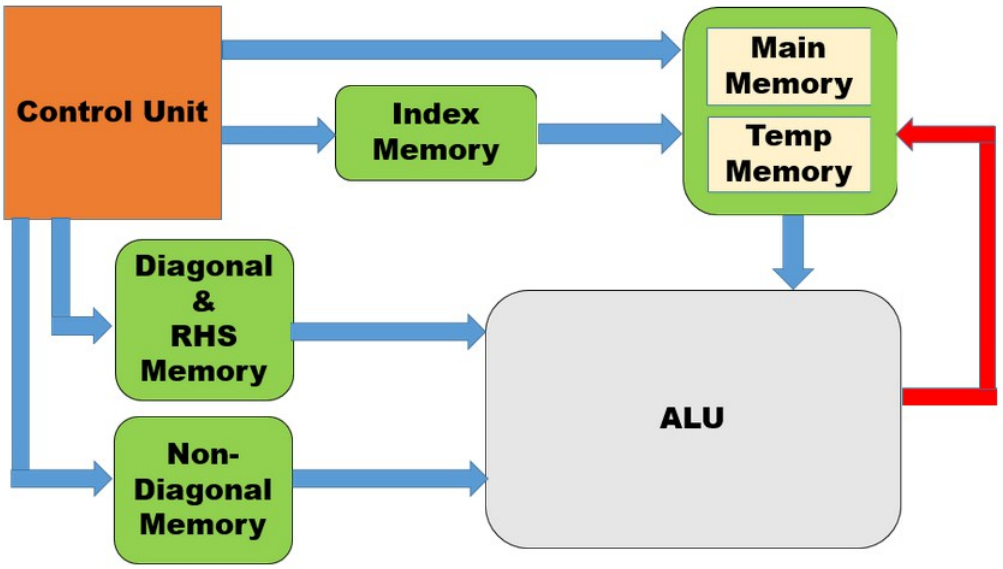

FIGURE 5.1: The architecture for Basic Jacobi.

5.1.1 Overview

This architecture shown in Figure 5.1 is the simplest design used. It models the Jacobi iterative method without optimizations. This design consists of a control unit, main memory, replica of main memory, diagonal and right-hand-side elements' memory, non-diagonal elements' memory, index memory, and only one ALU.

The control unit sends control signals to all memories to control data flowing out of memories and into the ALU. Upon receiving the appropriate addresses, memories load the corresponding data onto the data lines which connect them to the ALU, which in turn operates on that data and sends the new data back to the memory to save it.

5.1.2 Design modules

5.1.2.1 Top module

This module basically encapsulates all of the design and contains all of the main units, it also connects all these units together. Table 5.1 shows the parameterized components of the design.

TABLE 5.1: Parameterized components of *Basic Jacobi*.

Parameter	Description
number_of_equations	Represents the number of rows in each memory. It is used to define memory depth
address_width	Represents the number of bits needed for the address lines of each memory

5.1.2.2 Control unit

The control unit consists mainly of 3 counters, namely, pipeline stage counter, row counter, and iteration counter which work as follows:

- Pipeline Stage Counter:

  This counter shows the current step which is being executed in the design. That is, loading data from memory, executing arithmetic operations, or storing data in memory. It is also used as a driver for the row counter, since the row counter is incremented after each row has been operated on.

- Row Counter:

  The row counter represents the current row being operated on of the matrix, it is used for loading data that corresponds to its value from all memories (all the data needed by the ALU). It is also used as a driver for the iteration counter, since the iteration counter is incremented when all rows have been operated on during the current iteration.

- Iteration Counter:

  The iteration counter represents the current iteration (since Jacobi is an iterative method) and it is compared with a pre-defined value given by the user to check whether or not calculations have been finished. This counter is also used to generate signals that enable either the main or replica memory to load or store data depending on the current iteration number.

The control unit also generates the halt signal when the calculations have been finished, this signal is used as an indicator that can be used to stop simulation, write data into files, or do some post-processing.

5.1.2.3 Memory

1. Main Memory (Main Result Memory) - Replica of Main Memory (Temp Result Memory):

   These two memories contain the required solutions. Initially, they are loaded with an assumption of the result, which is chosen to be zero in our case.

   During each iteration, one of these memories loads data to be operated on, while the other memory stores the results of the calculations. This is to ensure that all data operated on is the data from the last iteration and not new data.

   After each iteration, data is stored in one of these memories depending on the current iteration number. This data is then operated on during the next iteration until the pre-defined number of iterations is reached.

   Each row of these memories contains 1 floating-point element. Memory depth is equal to the matrix dimension (the number of rows of the matrix).

2. Diagonal and right-hand-side memory:

   This is a read-only memory. For each row of the matrix, it contains the diagonal and right-hand-side elements.

   Depending on the input address, this memory loads these two elements of the corresponding row onto the data lines for further processing in the ALU.

   Each row of this memory contains 2 floating-point elements. Memory depth is equal to the matrix dimension (the number of rows of the matrix).

3. Non-diagonal memory:

   This is a read-only memory. For each row of the matrix, it contains the non-diagonal elements. Only the non-zero elements of each row are stored in this memory (since this is a sparse matrix).

   Since the application under research generates matrices that contain only 2 non-diagonal elements of non-zero values per row, each row of this memory contains only 2 floating-point elements.

   Depending on the input address, this memory loads these two elements of the corresponding row onto the data lines for further processing in the ALU.

   Each row of this memory contains 2 floating-point elements. Memory depth is equal to the matrix dimension (the number of rows of the matrix).

4. Index memory:

   This is a read-only memory. For each row of the matrix, it contains the indices of the non-diagonal elements of non-zero values. These indices are then used to choose the corresponding elements in the result memories to be multiplied by the non-diagonal elements.

   Depending on the input address, this memory loads these two elements of the corresponding row onto the data lines for further processing in the ALU.

   Each row of this memory contains 2 integer values. Memory depth is equal to the matrix dimension (the number of rows of the matrix).

5.1.2.4 ALU

The ALU is responsible for all arithmetic operations performed on data. It models the operations used in the Jacobi iterative method and works as follows:

1. Multiplies the values of the non-diagonal memory with their corresponding values from the main or replica memory depending on the current iteration number (whose indices are obtained from the index memory).

2. Adds the results of the multiplication in step 1 together.

3. Subtracts the addition result from the right-hand-side value (obtained from the diagonal and right-hand-side memory) which corresponds to the current row being operated on.

4. Divides the subtraction result by the diagonal value (obtained from the diagonal memory) which corresponds to the current row being operated on.

5.1.3 Pros and Cons

The main advantage of this design is that, it is very simple and does not need much hardware resources to be implemented. However, it is very slow since this design is not pipelined and does not utilize the available hardware resources for parallel execution (Only a single ALU is used).

5.2 Parallel Jacobi

5.2.1 Overview

This is the most complex, and least scalable architecture. It targets a great level of parallelism in exchange for an exponential increase in cost in terms of hardware resources.

Figure 5.2 shows the design architecture. The ALU works just like Basic Jacobi's except for the fact that this design uses multiple ALU cores that operate on multiple data at the same time for parallelism. This design's complexity lies in its ability to operate on all of the matrix rows in parallel, at the expense of a complex memory architecture that limits this design's scalability. This is explained in more depth in each section below.

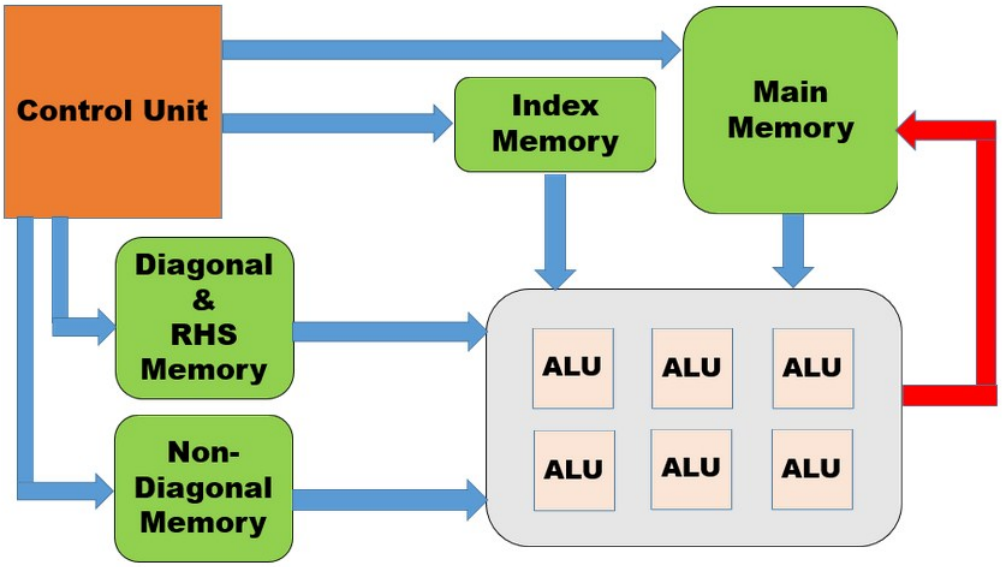

FIGURE 5.2: The architecture for Parallel Jacobi.

5.2.2 Design modules

5.2.2.1 Top module

This module encapsulates the design. It contains the main modules of the design and connects them together. It is also responsible for instantiating the required number of ALUs. Table 5.2 shows the parameterized components of the design.

TABLE 5.2: Parameterized components of *Parallel Jacobi*.

Parameter	Description
number_of_equations number_of_ALUs	Both are used to determine the number of ALUs to be instantiated and also to determine the number of block memories required
address_width	Represents the number of bits required to access any element of the data memory. Used in Index Memory to determine the size of each element.
number_of_columns	Represents the number of elements per row.
number_of_memories	Represents the number of block memories needed
element_width	Represents the number of bits used for each floating-point element (32 or 64).

5.2.2.2 Control unit

Refer to Basic Jacobi's control unit. One difference is that this design does not contain a row counter. This is because this design's main target was to do all calculations in parallel, meaning that all rows are operated on at the same time, and this means that the row counter is no longer needed.

5.2.2.3 Memory

1. Main Memory:

   The architecture of this memory is completely different from that used in Basic Jacobi. Due to the restrictions of on-chip memories, this memory instantiates small memory blocks of 2 rows each. This is due to the fact the on-chip memories usually have a small number of ports (usually 2). This memory instantiates as many memories of these small memory elements as needed to hold all of the matrix data. **Using this technique, all of the memory elements can be accessed at the same time, at the expensive of limited scalability**. These memory elements are then sent to all of the ALUs to be operated on. For more details on the contents of the Main Memory, refer to Basic Jacobi's Main Memory section.

2. Diagonal and right-hand-side Memory:

   This memory uses the same technique discussed in the Main Memory section above. The contents of this memory are the same as Basic Jacobi's Diagonal and right-hand-side Memory described before.

3. Non-diagonal Memory:

   This memory uses the same technique discussed in the Main Memory section above. The contents of this memory are the same as Basic Jacobi's Non-diagonal Memory shown before.

4. Index Memory:

   This memory uses the same technique discussed in the Main Memory section above The contents of this memory are the same as Basic Jacobi's Index Memory described before.

5.2.2.4 ALU

This ALU operates in the same way as Basic Jacobi's ALU with one extra step. The choice of which data elements from the result memory to operate on is done in each ALU on its own, since on-chip memories are limited in terms of flexibility and usually do not have enough ports to serve all of the ALUs in this design. And this is the main reason why this design is not scalable, since each ALU needs its own multiplexer which chooses 2 elements from the data memory. This accounts for a number of multiplexers equal to number of ALUs multiplied by 2, and each of these multiplexers needs to be large enough to choose between a huge number of elements which are all of the Main Memory's elements. This results in an exponential increase in hardware resources required as the number of matrix rows increases due to the presence of 2 factors, the number of ALUs needed for full parallelism, and consequently the number of multiplexers needed to server all of these ALUs.

5.2.3 Pros and Cons

This design is extremely fast, operations are performed simultaneously on all the RHS matrix rows. Total number of clock cycles needed for the design is independent of matrix size. However, it has some disadvantages. It is a very complex design. This also leads to less satisfactory synthesis results and lower clock frequency rates. The amount of hardware resources needed increases extremely fast when matrix size increases. As a result, it is not scalable. Software pre-processing is needed because initial memory contents need to be split among several files first.

5.3 Pipelined Jacobi

5.3.1 Overview

The architecture shown in Figure 5.3 is an improved version of Basic Jacobi. It makes a trade-off between design simplicity and high performance at low cost of resources.

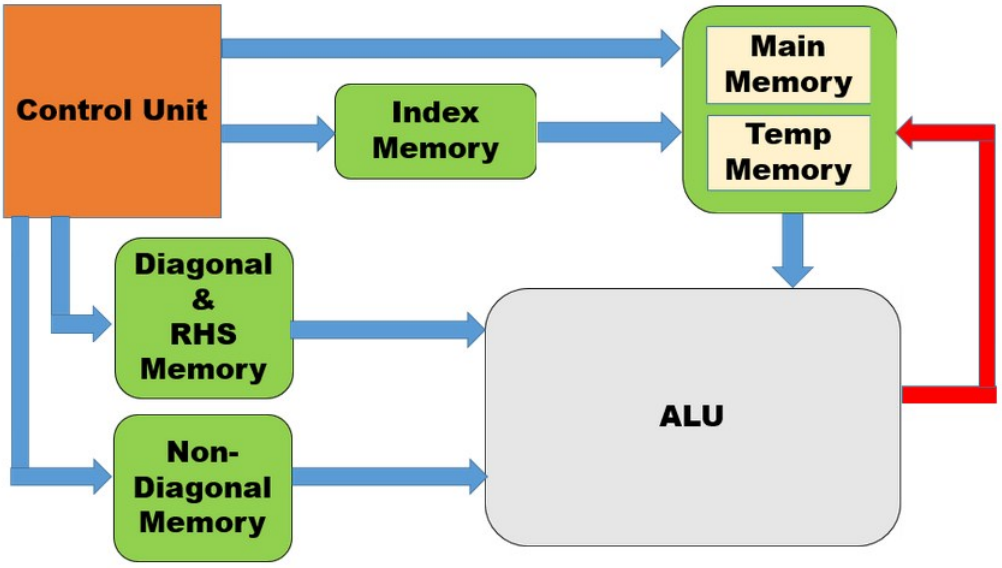

FIGURE 5.3: The architecture for Pipelined Jacobi.

Pipelined Jacobi consists of the same main components as Basic Jacobi. A control unit, main memory, replica of main memory, diagonal and right-hand-side elements' memory, non-diagonal elements' memory, index memory, and an ALU.

When a pre-defined number of iterations are reached, the halt signal is generated and that indicates that the required computations have been finished.

Data flow for this design works as follows:

1. Control unit sends address to read data from the Index Memory.
2. Data read from the Index Memory is then sent to the Main Memory or Replica of Main Memory to fetch data which needs to be operated on.
3. Control unit sends address to read data from the Non-diagonal Memory.
4. The ALU multiplies the data obtained from steps 2 and 3.
5. The ALU adds the results of the multiplication operations in step 4 together.
6. After step 5 is done, the control unit sends an address to read data from the Diagonal and right-hand-side Memory.
7. The result obtained from step 5 is subtracted from the right-hand-side element obtained from step 6. Meanwhile, the diagonal element obtained from step 6 is delayed by ALU registers to be used later.

8. The result of the subtraction operation in step 7 is divided by the diagonal element saved in the ALU registers.

9. The control unit sends the appropriate address to the Main and Replica memories and the appropriate signals which enable one of them for writing data. Meanwhile, the ALU output it sent to those 2 memories as well to be stored.

5.3.2 Design modules

5.3.2.1 Top module

Refer to Basic Jacobi's Top Module section. Table 5.3 shows the parameterized components of the design.

TABLE 5.3: Parameterized components of *Pipelined Jacobi*.

Parameter	Description
number_of_equations	Represents the number of rows in the matrix. It is used for defining the depth of each memory.
memories_address_width	Represents the number of bits needed for the address lines of each memory
number_of_iterations	Represents the required number of iterations to be executed. When this iterations to be executed. When this
iteration_number_register_width	Represents the number of bits needed for the iteration number counter

5.3.2.2 Control unit

Due to the pipelined nature of this design, the control unit is more complex than its counterpart in Basic Jacobi. Each memory had to have its own counter that generates an address that corresponds to the current element required for each ALU operation. Jacobi method requires many arithmetic operations, 2 multiplications, 1 addition, 1 subtraction, and 1 division operation. Each of them needs different operands. That is why the control unit is responsible for giving correct addresses to each memory to fetch the appropriate operands for each operation.

The following counters are the main components of the control unit:

1. memory_index_nonzero_elements_read_address:
   This counter's value is sent to the Index Memory to fetch the required row.

2. memory_nondiagonal_values_read_address:
   This counter's value is sent to the Non-diagonal Memory to fetch the required row. It is delayed by 1 cycle after the memory_index_nonzero_elements_read_address to account for the time needed by the ALU operations.

3. memory_diagonal_rhs_read_address:
   This counter's value is sent to the Diagonal and right-hand-side Memory to fetch the required row. It is delayed by 5 cycle after the memory_index_nonzero_elements read_address to account for the time needed by the ALU operations.

4. write_address_both_result_memories:
   This counter's value is sent to the Main Memory and Replica of Main Memory along with the signals controlling data writing operations to write the data newly generated by the ALU. It is delayed by 10 cycle after the memory_index_nonzero elements_read_address to account for the time needed by the ALU operations.

5.3.2.3 Memory

1. Main Memory:
   Refer to Basic Jacobi's Main Memory section.

2. Diagonal and right-hand-side Memory:
   Refer to Basic Jacobi's Diagonal and right-hand-side memory section.

3. Non-diagonal Memory:
   Refer to Basic Jacobi's Non-diagonal Memory section.

4. Index Memory:
   Refer to Basic Jacobi's Index Memory section.

5.3.2.4 ALU

There is one difference between this ALU and Basic Jacobi's ALU. Since this is a pipelined design, a new row is fetched at each new clock edge. This caused a minor

problem because the diagonal and right-hand-side memory contained 2 different elements that are required at different time steps. To solve this, some registers were added in the ALU to delay the diagonal element by the required number of cycles. This problem was solved in a later design by splitting the diagonal and right-hand-side memory into 2 separate entities to avoid using extra hardware resources.

5.3.3 Pros and Cons

This design is faster than Basic Jacobi. It does not need much hardware resources to be implemented. However, The pipeline needs to be flushed after every iteration, (cannot just run continuously because all the new data needs to be stored first). Morever, the design does not utilize the available hardware resources which can be used to turn this design into a multi-core design for much more productivity.

5.4 Clustered Jacobi

5.4.1 Overview

Figure 5.4 shows the architecture for Clustered Jacobi. This design is quite different from all of the previous versions. It is different in terms of memory architecture, control unit architecture and ALU architecture.

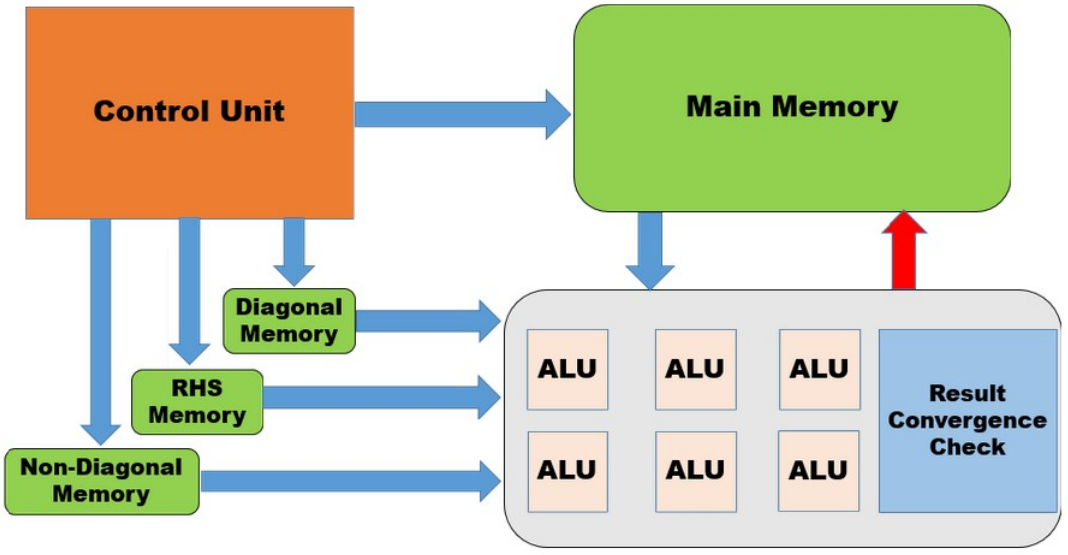

FIGURE 5.4: The architecture for Clustered Jacobi.

It performs the same arithmetic operations of Jacobi Method except that the final division operations is turned into multiplication operations with the second operand being the inverse of the diagonal element (calculated using software) to save some hardware resources for more cores. Division operations are not very time-consuming when done on software. For the order of these arithmetic operations, refer to the ALU section in Basic Jacobi.

This design splits the matrix into clusters, where each cluster contains a number of rows. Each cluster is independent on the others, so all of them can be operated on in parallel. Each cluster takes a single step to be operated on in this design's pipeline, making this design very powerful.

Memory architecture is different in that, unlike Basic Jacobi and Pipelined Jacobi, each memory row contains a whole cluster of data instead of just 1 row of the matrix. This makes up for memory constraints and allows memories to serve more arithmetic cores simultaneously. These clusters are also pre-processed on software to minimize any extra operations needed to get the required data - like multiplexing which costs quite a lot of hardware resources when used on a large scale - and each memory row is self-contained such that the older Index Memory used in Basic Jacobi and Pipelined Jacobi is no longer needed. Each row has its corresponding operands ready in the other memories. Temporary result memory is no longer needed since each row of the matrix is self-contained and can - without any conflicts - be overwritten after being operated on without affecting the other rows, since the other rows are also self-contained and have their own independent data elements.

5.4.2 Design modules

5.4.2.1 Top module

It encapsulates all of the design modules and simple connects them together. It has no other functionalities unlike Basic Jacobi and Pipelined Jacobi. Table 5.4 shows the parameterized components of the design.

Table 5.4: Parameterized components of *Clustered Jacobi*.

Parameter	Description
number_of_clusters	Represents the number of clusters in the matrix. Used to define the number of rows in memories.
number_of_equations_per_cluster	Represents the number of rows per cluster. Used to define the number of elements in each row of each memory.
element_width	Represents the number of bits used for each floating-point element (32 or 64).
memories_address_width	Represents the number of bits needed for each address counter in the control unit.

5.4.2.2 Control unit

This design contains a control unit that is somewhat more complex than its counterpart in Pipelined Jacobi, but still a lot less complex than its counterpart in Parallel Jacobi.

The control unit controls data flowing out of and into memories, which means that this unit controls data flowing into and out of the ALU as well. It simply controls the whole design.

The following counters are the main components of the control unit:

1. result_memory_read_address:

   This counter controls which row is read from the result memory at the current cycle.

2. nondiagonal_memory_read_address:

   This counter controls which row is read from the non-diagonal memory at the current cycle, it is completely synchronized with result_memory_read_address since we no longer need to get the indices of the non-diagonal elements of non-zero values because each row is self-contained.

3. diagonal_memory_read_address:

   This counter controls which row is read from the diagonal memory at the current cycle. It is delayed after result_memory_read_address by 6 cycles to account for the order in which ALU operations are done.

4. rhs_memory_read_address:

   This counter controls which row is read from the right-hand-side memory at the

current cycle. It is delayed after result_memory_read_address by 10 cycles to account for the order in which ALU operations are done.

5. result_memory_write_address:
   This counter controls which row is written into the result memory at the current cycle. It is delayed after the result_memory_read_address by 13 cycles to account for the order in which ALU operations are done.

5.4.2.3 Memory

1. Main Memory:
   Refer to Basic Jacobi's Main Memory section. The only difference is that each row in this memory contains a whole cluster instead of a single element. This memory's depth is equal to the matrix dimension (the number of rows in the matrix) divided by the number of rows per cluster.

2. Diagonal and right-hand-side Memory:
   There is some difference from Basic Jacobi concerning this module. This design uses 2 separate memories instead of just 1 memory for both the diagonal and right-hand-side elements. This allows much more flexibility and also removes the need to use extra registers to delay data as illustrated in Pipelined Jacobi's ALU section. Each row in each of these memories contains a whole cluster of elements. For further details on these memories' contents, refer to Basic Jacobi's Diagonal and right-hand-side Memory section.

   This memory's depth is equal to the matrix dimension (the number of rows in the matrix) divided by the number of rows per cluster. Each row of each of these memories contains a number of floating-point elements equal to the number of rows per cluster.

3. Non-diagonal Memory:
   Refer to Basic Jacobi's Non-diagonal Memory section. The only difference is that each row of this memory contains a whole cluster's worth of data instead of just a single element. This memory's depth is equal to the matrix dimension (the number of rows in the matrix) divided by the number of rows per cluster.

5.4.2.4 ALU

This ALU is different from the ones implemented in the previous designs in 2 ways. First, the final division operation is turned into a multiplication by inverse operation to minimize the ALU core cost in terms of hardware resources and to allow more cores to be implemented. Second, unlike the previous designs, which used a pre-defined value as the number of required iterations, this design uses a stand-alone module called Result Convergence Check which is sophisticated enough to decide when to stop operating on data. For more details on the Result Convergence Check module, refer to Result Convergence Check section below.

5.4.3 Result Convergence Check

This is a new module introduced in this design. It is responsible for deciding when to stop operating on data. It is given a pre-defined value representing the error tolerance between iterations (the difference in value between two consecutive iterations) and whenever the current tolerance is below that pre-defined tolerance value, this module generates a halt signal indicating end of operations.

5.4.4 Pros and Cons

This design has many advantages. It has the least execution time among other designs. It is a very flexible multi-core system that can be configured for a broad range of matrix sizes with a broad range of ALU cores for increased productivity (depends on available hardware resources). Unlike Pipelined Jacobi, the pipeline does not need to be flushed after each iteration since the memory architecture allows each row to be self-contained. On the other hand, extra software-based pre-processing is needed to put the matrix in a form compatible with this design (Cluster form).

5.5 Summary

The Jacobi approach went through many optimizations depending on the changing requirements and dealing the different drawbacks of each new design.

The first design used was Basic Jacobi, which is a simple design modeling the Jacobi iterative method in a simple manner without any optimizations. There were two available tracks to optimize this design. One of them was by implementing a pipelined design with a little more hardware resources in order to improve performance greatly. This led to Pipelined Jacobi. The other track was to utilize the vast hardware resources available on the emulator by implementing a multi-core design. This led to a prototype, Parallel Jacobi, which proved to be impractical.

Further improvements occurred which utilize both the advantages of the pipelined design and the multi-core design. These resulted in an extremely powerful design. After improving that new design's pipeline even further, Clustered Jacobi was introduced, which is by far the most powerful iterative design mentioned here.

Chapter 6

Case study

As a proof of concept, we applied our proposed hardware solutions on a complete case study by using time-domain finite element methods for solving metamaterial model equations. We choose Metamaterials because they attracted great attention of researchers since their successful construction in 2000.

There is currently an enormous effort in the electrical engineering, material science, physics, and optics communities to come up with various ways of constructing efficient metamaterials and using them for potentially revolutionary applications in antenna and radar design, subwavelength imaging, and invisibility cloak design. Hence, simulation of electromagnetic phenomena in metamaterials becomes a very important issue. For a detailed discussion about metamaterials and their mathematical models, please refer to appendix A.

In [22], Jichun Li and Yunqing Huang presents a detailed discussion on how to code the two-dimensional edge element for solving metamaterial Maxwell's equations. They cover the whole programming process including mesh generation, calculation of the element matrices, assembly process, and postprocessing of numerical solutions. They constructed a complete MATLAB source codes of a mixed finite element method (FEM) for a 2-D Drude metamaterial model. We modifed the codes to fit with our case study and our complete program is discussed below.

Our programs starts with a GUI interface to choose between software or hardware solution as shown in Figure 6.1.

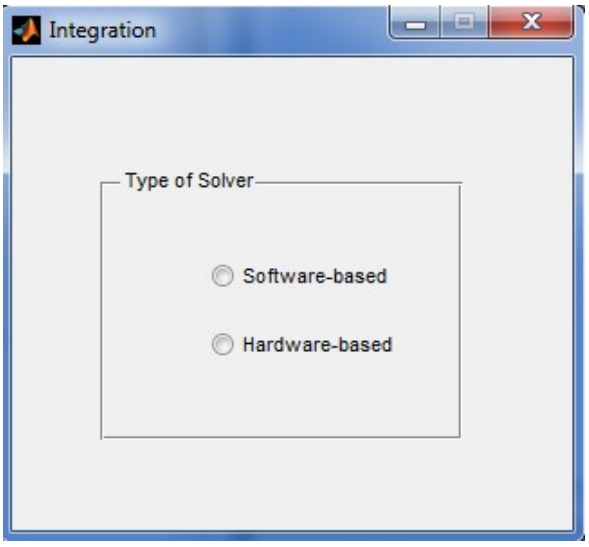

FIGURE 6.1: Main GUI application screen.

In the Software solution the whole calculations are performed using MATLAB and the solver part is based on MATLAB special operator [23]. For the hardware solution, only pre and post processing calculations are performed on MATLAB while the solver part is moved to the emulator environment.

6.1 Software solution

If we choose the software solution, another GUI screen would appear as shown in Figure 6.2 to take the inputs from the user as following:

- X_Low:

  It represents the X co-ordinate of the lower edge in the two-dimentional element.

- Y_Low:

  It represents the Y co-ordinate of the lower edge in the two-dimentional element.

- X_High:

  It represents the X co-ordinate of the higher edge in the two-dimentional element.

- Y_High:

  It represents the Y co-ordinate of the higher edge in the two-dimentional element.

- X_Elements:

  It represents the number of elements in which the user needs to divide the horizontal part into meshes in X-direction.

- Y_Elements:

  It represents the number of elements in which the user needs to divide the vertical part into meshes in Y-direction.

- Damping freq.:

  It represents the damping frequency of the metamaterial.A simple case for achieving negative refraction index is to choose Damping frequency to be 0 as illustrated in appendix A.

- Plasma freq.:

  It represnets the plasma frequency of the metamaterial (the frequency at which it becomes transparent). Any metal below its plasma frequency yields negative values of the permittivity.

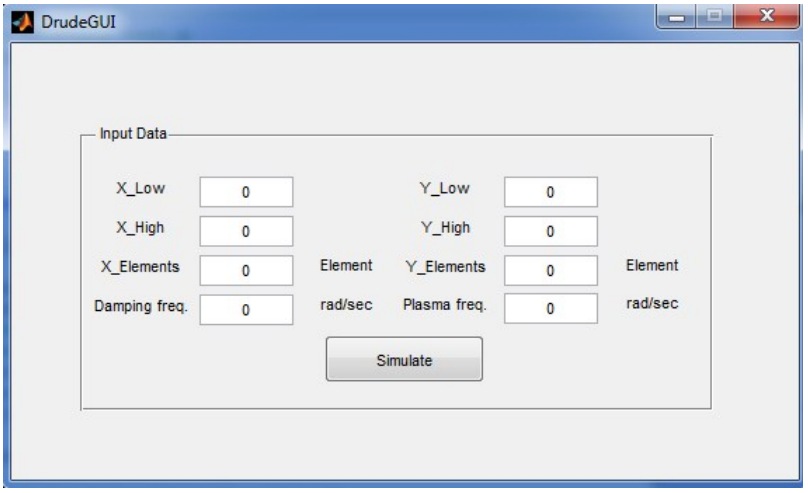

FIGURE 6.2: Software-based solver screen.

The GUI button **Simulate** calls the main function **Drude_cn** which is divided into three main parts: **Preprocessing**, **Solver** and **Postprocessing**. These parts are discussed in the next sections. The final output of this function is the graphs of Electric and Magnetic fields.

6.1.1 Preprocessing

This part of code is responsible of generating regular rectangular meshes from the given two-dimentional element whose parameters are predifined by the user.

The first step is defining the global variables and vectors which are used in important calculations in the program. The main global variables are "gauss, psiJ, etaJ" which represent the Gaussian quadrature points, "rk4a, rk4b, rk4c" which represent Low storage Runge-Kutta coefficients [24], "Ex,Ey,Jx,Jy", which represent the exact electric field and electric polarization, "HH,KK" which represent the exact magnetic field and magnetic polarization, and "f1RHS,f2RHS,gRHS" which represent the exact RHS.

The electric polarization is a slight relative shift of positive and negative electric charge in opposite directions within an insulator, or dielectric, induced by an external electric field. Polarization occurs when an electric field distorts the negative cloud of electrons around positive atomic nuclei in a direction opposite the field. This slight separation of charge makes one side of the atom somewhat positive and the opposite side somewhat negative. Polarization P in its quantitative meaning is the amount of dipole moment p per unit volume V of a polarized material, $P = p/V$.

Magnetic polarization effects are similar to electric polarization effects. Just as electric dipole moments control the behavior of dielectric materials, magnetic dipole moments control the behavior of magnetic materials.This polarization produces a new field that adds to the applied field. However, there are important differences:

1. While dielectric materials reduce an applied electric field, typical magnetic materials enhance an applied magnetic field. These materials are said to be paramagnetic. Aluminum and sodium are paramagnetic at room temperatures, as is iron oxide. Iron oxide will be our representative paramagnet.

2. No magnetic charge has ever been isolated. Thus, while electric dipole moments are caused by a pair of electric charges with opposite sign, this does not seem to be the mechanism for creating magnetic dipole moments. For all cases in which the origin of a magnetic dipole moment is understood classically, the moment is caused by a circulating current. (This omits intrinsic magnetic moments such as that possessed by the electron, which is apparently a point object.)

The most important similarity between dielectric and paramagnetic phenomena is the relation between the applied field and the resulting polarization (measured by the dipole moment per unit volume).

There is another important variable, id_bc, which will be discussed later.

For creating a mesh, we start by defining the size of each mech through the following equation:

$$\mathbf{dx} = \frac{highx - lowx}{nelex} \tag{6.1}$$

$$\mathbf{dy} = \frac{highy - lowy}{neley} \tag{6.2}$$

Where dx represents the horizontal length of a mech in X-direction, dy represents the vertical length of a mech in Y-direction, nelex simulates the user input "X_Elements" represents the number of elements in which the user needs to divide the horizontal part into meshes in X-direction and neley is the same as nelex except it represents the number of elements in Y-direction. The resulted shape of meshes is similar to that in Figure 6.3, and number of meshes differs according to the given value by the user.

In order to save the coordinates of meshes' nodes , we created a two-dimentional array , no2xy, to do this job, where the first row holds the x-coordinate of all nodes and the second row holds the y-coordinate.

Similar to the classical nodal based finite element method [25], we need to build up a connectivity matrix "el2no" which holds the global values of all nodes as shown in Figure 6.4.

el2no is a two-dimentional matrix where each column represents the nodes of a mesh ordered counterclockwise. For the lowest-order rectangular edge element, el2no(i,j) denotes the global label of the i-th node of the j-th element, where i = 1,2,3,4, j = 1,....,numel, and numel denotes the total number of elements.

Since unknowns in edge element space are associated with edges in the mesh, we need to number the edges and associate an orientation direction with each edge. To do this, we assume that each edge is defined by its start and end points, and each edge is assigned a

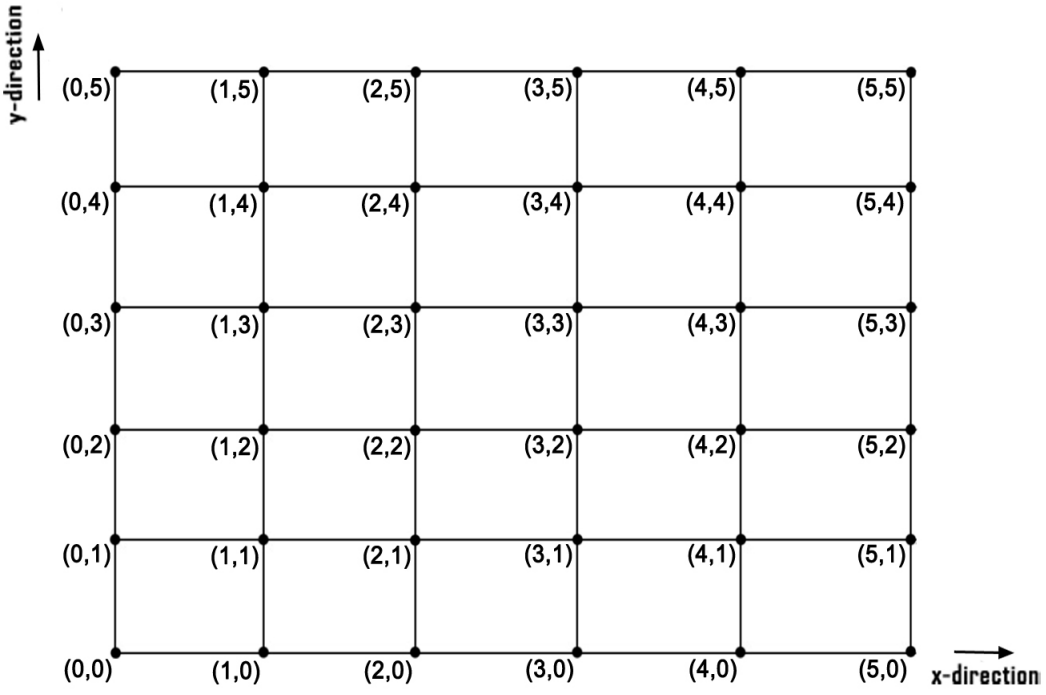

FIGURE 6.3: Mesh generation.

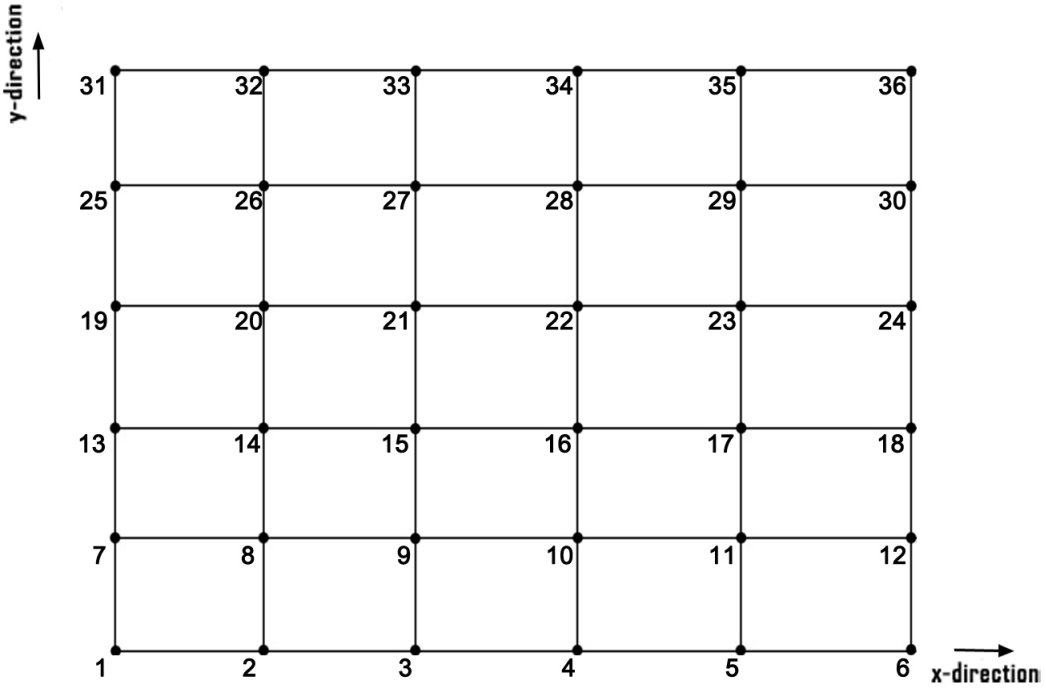

FIGURE 6.4: Mesh donation using el2no.

global edge number. This task can be done by creating a two-dimentional array , edges, which holds the start and end points of all edges of all meshes, this array is sorted to make sure that all edges are in the increasing direction of X and Y directions. After that, the function "Unique" is used to remove all repeated edges from the array (interior edges between adjacent meshes).The next step, a two-dimentional array el2ed(i,j), is created to store the global number of i-th edge of the j-th element, where i = 1,...,4, and j = 1,...,numel as shown in Figure 6.5.

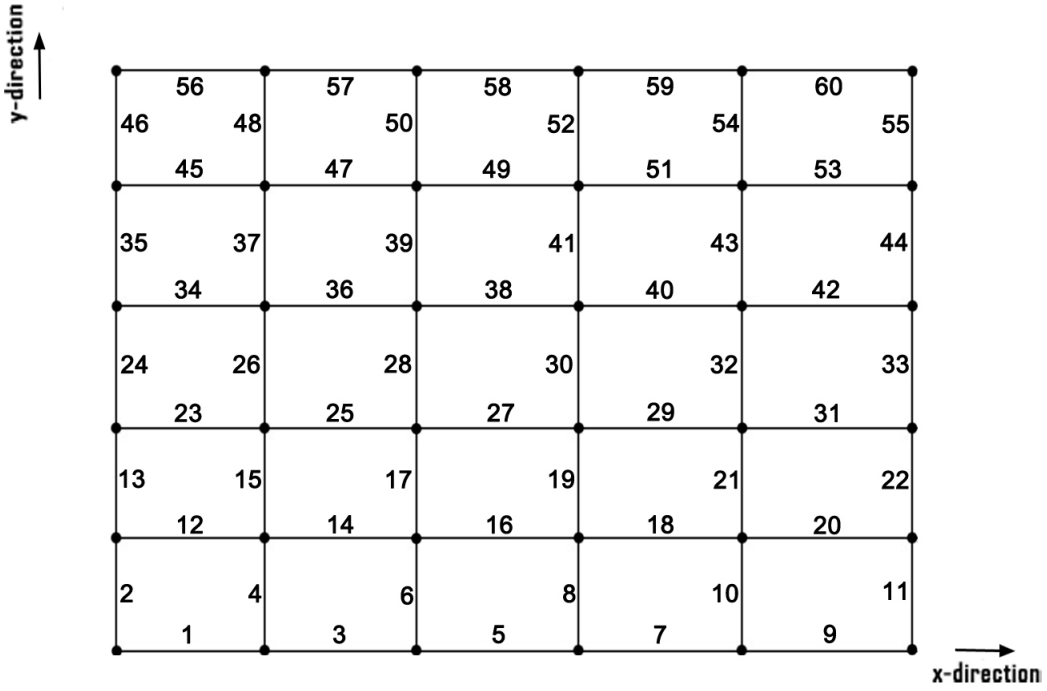

FIGURE 6.5: Final shape of meshes.

Depending on the type of boundary condition of the problem, we set the value of the variable "id_bc". As our test case is dealing with Dirichlet boundary condition(Partial differential equation boundary conditions which give the value of the function on a surface), we set the value of id_bc to be 1 to deal only with the interior edges of the mesh grid as shown in Figure 6.6.

The last step in preprocessing, two arrays are created, the first one, eint, holds the label of all interior edges only in the grid, and the other array, edori, is a two-dimentional array which holds a notation to the orientation of every edge in the grid. If the edge is oriented clockwise, its value in this array will be one, and if it is oriented counterclockwise, its value will be negative one.

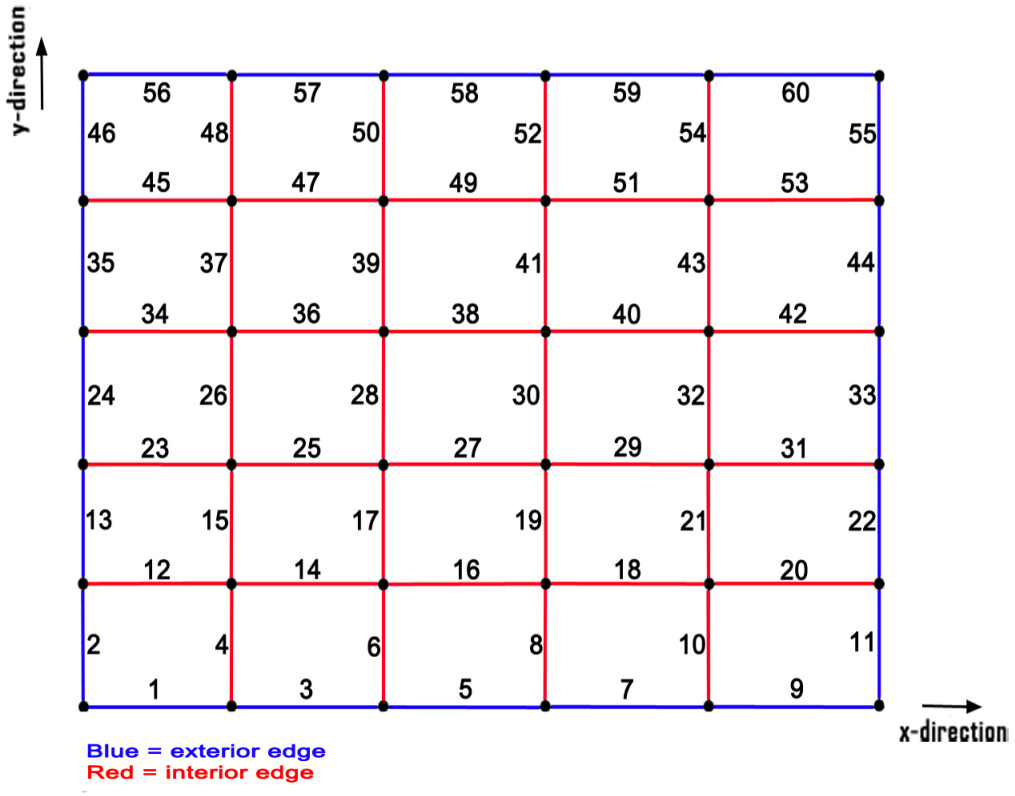

FIGURE 6.6: Using interior edges only from mesh grid.

6.1.2 Solver

The aim of this part of the program is to solve a system of linear equations represented by metamaterial mesh generated(discussed in the previous section). The main matrix and the RHS vector are both used to produce electric and magnetic fields. In this software solution, we use matlab special operator (mldivide, \). $x = A\backslash B$ solves the system of linear equations $A * x = B$ where matrices A and B must have the same number of rows.

6.1.3 Postprocessing

Calling the function **"Drude_cn_post"**, our program starts to perform postprocessing. In this function we draw the resulted electric field and magnetic field of the given material as shown in Figures 6.7 and 6.8 after solving the system of equations.These figues shows the

By increasing the number of mesh elements in x-direction and y-direction, the resolution of calculated electric field and magnetic field will increase too.

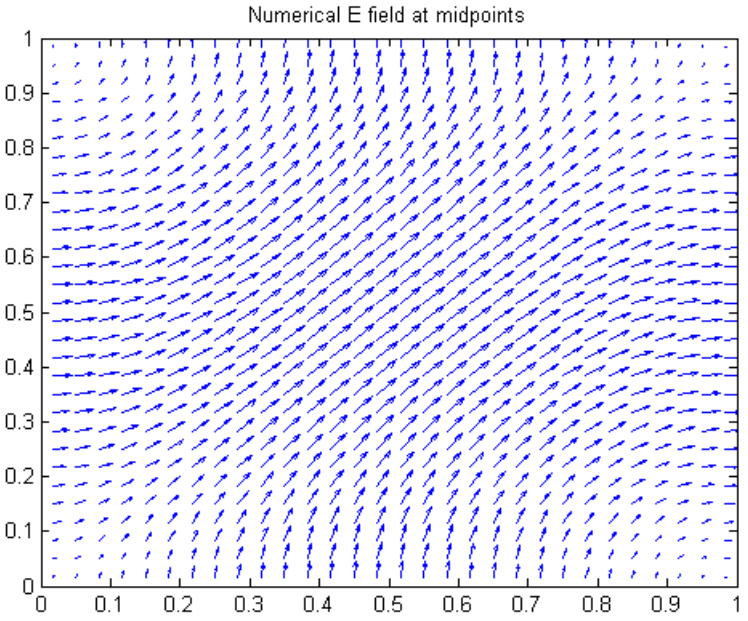

FIGURE 6.7: Electric field.

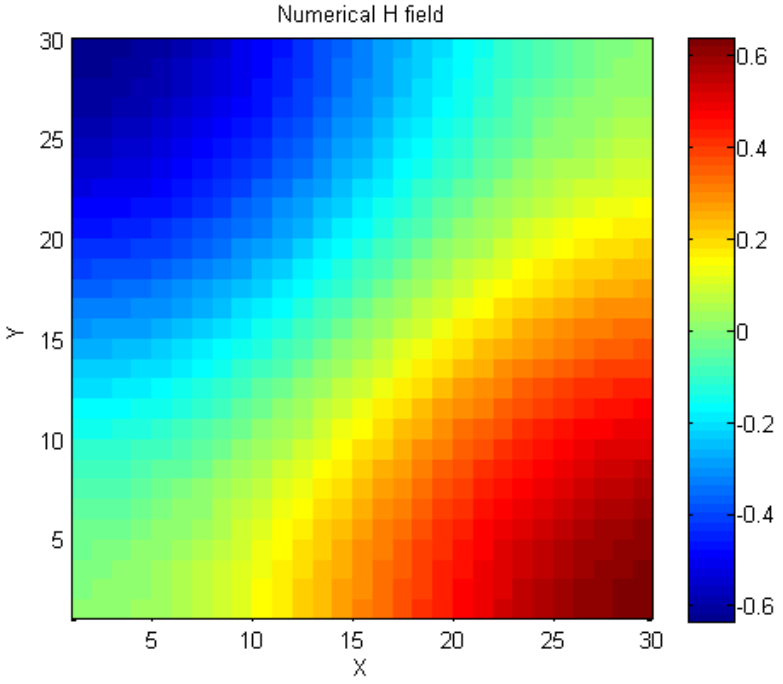

FIGURE 6.8: Magnetic field.

6.2 Hardware solution

If the user chooses the hardware-based solution from the main GUI screen, another window appears as shown in Figure 6.9, which consists of the same input parameters used in software solution, and another four simple GUI buttons which are **Generate(Jacobi)**, **Post-process** (for the Jacobian solution technique), **Generate(Gaussian)** and **Post-process**(for the Gaussian solution technique).

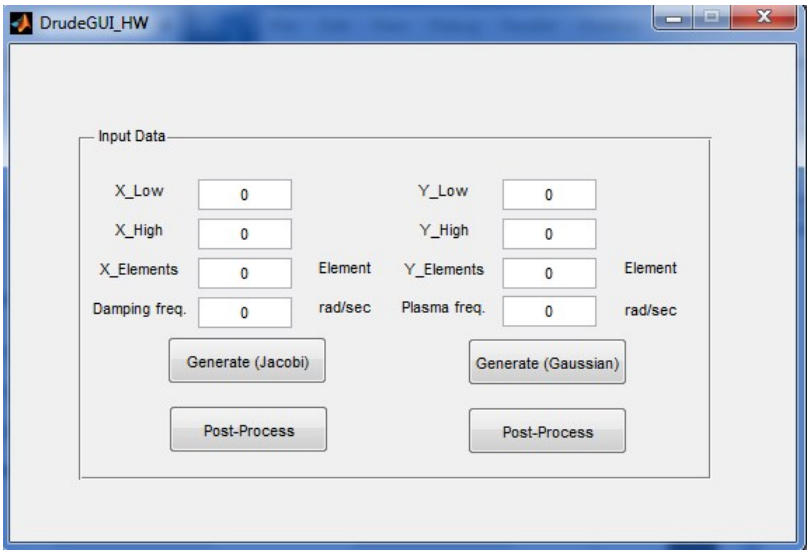

FIGURE 6.9: Hardware-based solver screen.

The hardware solution is divided into three parts like the software one, preprocessing, solver and postprocessing. But, the difference is that in hardware solution, pre and postprocessing parts are executed using MATLAB script, while the solver is executed on an emulator machine.

6.2.1 Preprocessing

The two GUI buttons, Generate(Jacobi) and Generate(Gaussian), are used to perform this task in the same way as the software solution preprocessing, which is discussed in an earlier section. For each button, two Matlab functions are called. The first one, **Drude_cn_fast_cluster_generate** is the same for both buttons. It is responsible for general preprocessing and cluster preparation that will be discussed later. The second function varies between the two buttons. The Generate(Jacobi) button calls **script_cluster_separatedMem** and the other button, Generate(Gaussian), calls

two_point_six_writefiles. Both are responsible for generating memory files needed by the hardware solution. The generated files depend on a new concept called, Clustering, which is discussed in the coming section.

6.2.2 Clustering

In order to gain a high performance on the hardware for solving system of linear equations, we need to solve them in parallel. We couldn't solve linear equation in parallel as long as there are high dependencies between them, so we need a way to reduce those dependencies between equations. There is a property in the output matrix of the finite element method that can be used to reduce those dependencies named Clustering.

This property appears in the output matrix as the finite element method divides the test element into square meshes and calculations takes place on the edges of the mesh. Vertical edges in one row and horizontal edges in one column each contribute in the main output matrix with equations with high dependencies between them and low dependencies with other equations in the matrix we named each group of those equations a cluster. Figure 6.10 shows the edges contribution in each cluster, edges of the same cluster has the same color.

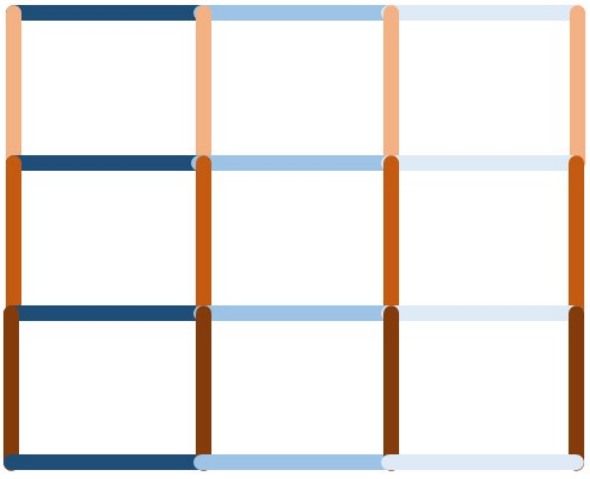

FIGURE 6.10: Edges contribution in each cluster for 3x3 meshes.

The generation of the clusters depends on a matrix generated for the preparation of the generation of the output matrix in the Matlab code, the matrix name el2ed (element to edge) and it size is 4 * number of meshes in the test element. Each column of el2ed refers to a mesh and the four rows refer to the four edges in each mesh, the values stored in

el2ed represent the location of the contribution of each edge in the main output matrix. By using the el2ed matrix we could extract each cluster from the main matrix, and then reform them to smaller sizes as shown in Figure 6.11 and Figure 6.12. Figure 6.11 shows the main output matrix and the location of some clusters in it. While Figure 6.12 shows the shape of any cluster extracted from the main output matrix after reformation.

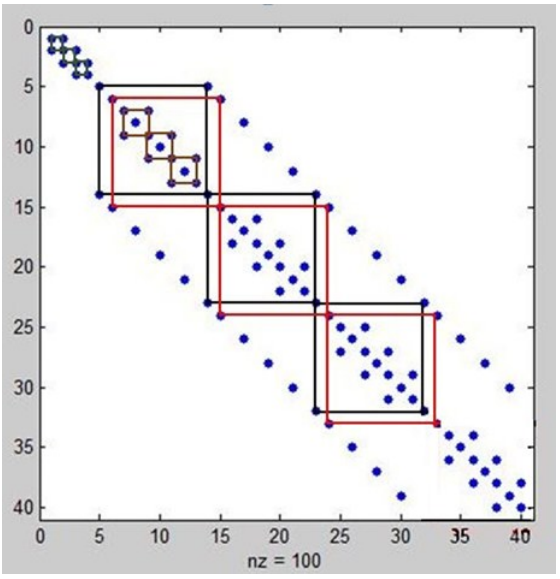

FIGURE 6.11: Clusters in 40x40 matrix where each colored square refers to a cluster, blue dots refer to non-zero elements, and white spaces refer to zero elements.

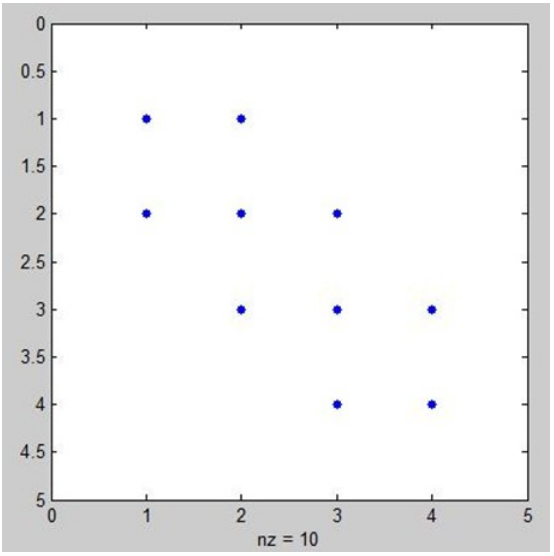

FIGURE 6.12: Cluster after reformation.

Number of clusters in the main output matrix is related to number of meshes in x-direction and in y-direction, which is equal to number of mesh in x-direction plus number of mesh in y-direction.

There is no doubt that using of the clusters would have a huge impact on the speed of solving linear system of equations. For example, if we have 40x40 matrix this matrix would have ten 4x4 clusters so if we design a system that have 10 ALUs to solve all the clusters in parallel, therefore the time needed to solve 40 linear equation will be equal to the of solving 4 linear equations which is a huge drop in solving time.

6.2.3 Solver

This part is the most consuming time and as a result it is moved to hardware-based platform (Emulator). Different hardware implementations were discussed in the previous chapters. Obtained results will be discussed in the next chapter.

6.2.4 Postprocessing

In this part, we use two different ways of postprocessing depending on the chosen solution technique (Gaussian or Jacobian). The difference comes from the 1st Matlab function to be called after the solver. For Jacobi, **read_out_Jacobi** takes the emulator output file as an input and co-operate with **declustering** to turns it into the regular form that Matlab can deal with(solution vector $n \times 1$). The function **two_point_six_readout** does the same operation but for Gaussian solution.

After that, the **Drude_cn_post** function is called. It does the same operations as illustrated before in the software solution section.

6.3 Another implementation for software solution using C++

n order to compare our solutions with software-based solvers other than MATLAB, we implemented a C++ code that performs the same software solution operations (preprocessing and solver) using different language and libraries discussed in the next sections.

Out program starts by calling the main function, which asks the user to provide the required input parameters used in different operations in the program sequence. As shown in Figure 6.13, all the input parameters are the same as that used in MATLAB

software-based solution, except the input variable **state** which allows the user to choose whether he wants to solve for interior edges only or to solve for the whole grid of meshes.

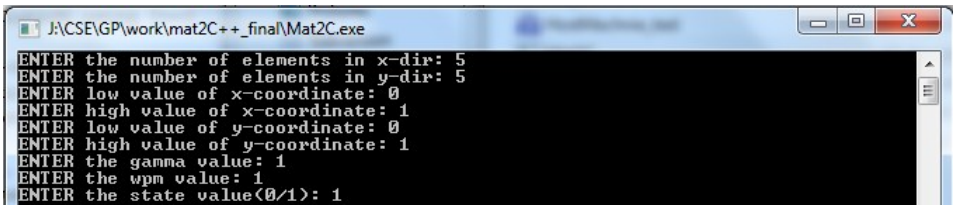

FIGURE 6.13: C++ software-based implementation main screen.

6.3.1 Preprocessing

After setting the input parameters, our program's contructor method starts to define all variables and vectors used through the program. The main method we used to begin our preprocessing is called **"startCalulations"**. In this function we define the same important matrices as MATLAB software-based solver (no2xy, el2no, edges and el2ed)using the same algorithms discussed earlier in the MATLAB software-based solver.

The "startCalulation" function starts to call extra implemented functions to perform another tasks as "getMin" and "getMax" which are used to get the minimum and maximum elements in a given vector. The function "otherCalc" is used to perform some electromagnetic calculations used in creating the system of equations, and the last implemented method is "getInterior" which is used to produce system of equations using only the interior edges of the mesh grid.

To implement so, we used an open source library called **ALGLIB**. ALGLIB is a cross-platform numerical analysis and data processing library. It supports several programming languages (C++, C#, Pascal, VBA) and several operating systems (Windows, Linux, Solaris). ALGLIB features include:

- Data analysis (classification/regression, including neural networks).
- Optimization and nonlinear solvers.
- Interpolation and linear/nonlinear least-squares fitting.
- Linear algebra (direct algorithms, EVD/SVD), direct and iterative linear solvers, Fast Fourier Transform.

- and many other algorithms (numerical integration, ODEs, statistics, special functions).

6.3.2 Solver

That is the most important part of this software program. It is performed by calling the function **solving()**. In this function we use a conditional statement to determine whether to solve the whole mesh grid's edges or the interior ones only according to the value os state variable (id_bc) provided by the user at the begining of this program. If the value of id_bc is 1, then the solving will concern only the interior edges, and if it is 0, it will solve for all edges. The solution process is performed using ALGLIB solver through these functions: *lincgcreate*, which initializes linear CG Solver. This solver is used to solve symmetric positive definite problems, *LinCGSolveSparse* function which procedure for solution of $A \times x = b$ with sparse matrix, and *lincgresults* which produces the result from the solving process.

Finally, we should mention that the C code described above runs on a 2.00GHz Core i7-2630QM CPU with 6GB of RAM. The optained results are given in chapter 7

Chapter 7

Experimental results

7.1 Experimental setup

Our architectures were modeled using Verilog, and Xilinx ISE Design suite 14.6 was used to check their functional correctness. Designs were then compiled and run on Mentor Graphics Emulator (egytrior) with 2 Advanced Verification Boards (AVB). That platfrom provides a total capacity of 32 Crystal chips with 1GB of memory. Crystal chips are equivalent to FPGAs but use different technology. A Crystal chip has around 500K of logic gates. Construction of one AVB is shown in Figure 7.1.

Figure 7.2 highlights the main steps included in the emulation design process. The Analyze step takes Verilog files as inputs and performs syntax checking. RTLC is responsible for generating structural Verilog netlist of Veloce primitives; LUT, tri-state, flip-flop, latch, and memory. VELSYN or Veloce synthesizer performs partitioning and ASIC netlists for each crystal chip where crystal chip is the programmable logic for emulation. After that, VELCC (Veloce Chip Compiler) does placing and routing. Finally, VELGS or Veloce Global scheduler performs final timing analysis and generates timing info for resources access, memories and IO access, emulator events and clocks.

7.2 Experimental results

This section evaluates the performance of the hardware implementations for the designs described in chapters 3, 4 and 5 in terms of resource utilization, speed-up over

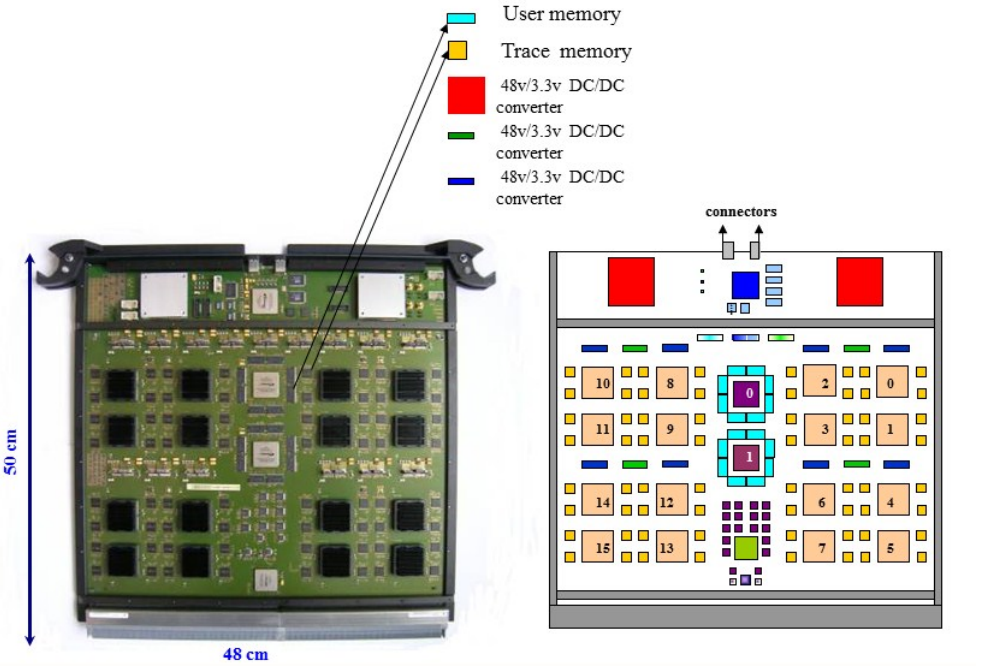

FIGURE 7.1: Veloce AVB.

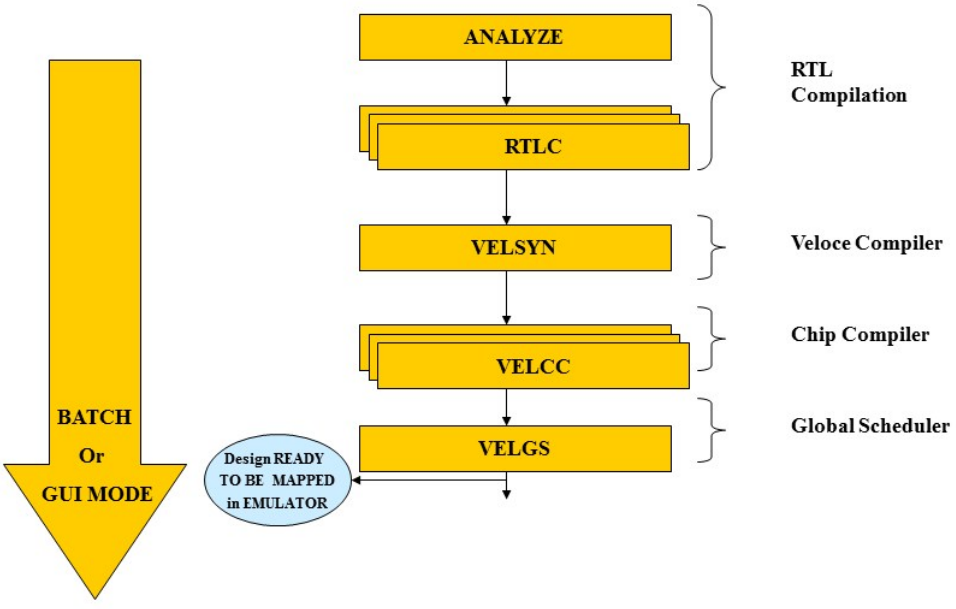

FIGURE 7.2: Emulation flow overview.

software solution and numerical precision. All test cases used are generated from the preprocessing part of the case study discussed in chapter 6.

7.2.1 Resource utilization

In this section, resource utilization for the various implementations is discussed. The measured resources include number of LUTs, number of flip-flops, amount of memory bytes and number of FPGAs used in each design. Also, the maximum frequency at which the design is allowed on the emulator is introduced.

7.2.1.1 Cramer architectures

Table 7.1 compares the logic utilization and operating frequency of the first hardware approach based on Cramer method for different test cases. The approach was discussed in detail in chapter 3. The obtained results show that the design is not scalable.

TABLE 7.1: Hardware resource utilization for first approach.

	Cramer			
Number of equations	2	3	4	5
Max frequency (kHz)	1851.9	1851.9	1851.9	1851.9
Number of LUTs	13,339	59,507	194,331	557,759
Number of flip-flops	288	512	800	1,152

7.2.1.2 Gaussian architectures

Table 7.2 compares the logic utilization, memory capacity and operating frequency of the first two Gaussian elimination architectures, second approach with skip and Full skip discussed in Sections 4.4 and 4.5.

TABLE 7.2: Hardware resource utilization for second approach With skip and Full skip.

	With skip	Full skip	
Number of equations	12	60	420
Max frequency (kHz)	469	432	497
Number of LUTs	84,298	154,500	1,057,067
Number of flip-flops	6,769	15,774	107,946
Memory bytes	624	15,616	862,208
Number of FPGAs in design	3	3	15

Table 7.3 compares the logic utilization, memory capacity and operating frequency of Parallel Jacobi discussed in Section 4.7 for different test cases.

TABLE 7.3: Hardware resource utilization for single floating point Clustered Gaussian using different test cases.

	second approach using sparse with clusters					
Number of equations	40	420	11,100	19,800	44,700	60,900
Number of ALUs	10	30	150	200	300	350
Max frequency (kHz)	411	406.5	387	396	347	370.4
Number of LUTs	65,178	195,472	977,195	1,302,918	1,954,331	2,280,075
Number of flip-flops	5,132	15,376	76,822	102,422	153,624	179,224
Memory bytes	480	5,760	230,400	307,200	921,600	1,075,200
Number of FPGAs in design	1	3	13	18	26	31

7.2.1.3 Jacobi architectures

Table 7.4 compares the logic utilization, memory capacity and operating frequency of basic and pipelined jacobi discussed in Sections 5.1 and 5.3.

TABLE 7.4: Hardware resource utilization for Basic and Pipelined Jacobi.

	Basic Jacobi	Pipelined Jacobi
Number of equations	24	24
Number of ALUs	1	1
Max frequency (kHz)	476.2	1851.9
Number of LUTs	6,634	6,863
Number of flip-flops	805	884
Memory bytes	832	832
Number of FPGAs in design	1	1

Table 7.5 compares the logic utilization, memory capacity and operating frequency of Parallel jacobi discussed in Section 5.2 for different test cases. The results show that the number of LUTs and flip-flops used increases exponentially and the operating frequency decreases as the number of solved equations increases. That occurs due to the problem of distributed memory illustrated in Section 5.2.2.3.

Table 7.6 and Table 7.7 show the logic utilization, memory capacity and operating frequency of Clustered jacobi discussed in Section 5.4 using single and double floating point precision for different test cases. The obtained results show that the operating frequency is almost stable for different test cases and equals the maximum available frequency for the used emulator. That guarantees linear running time as it will depend

TABLE 7.5: Hardware resource utilization for Parallel Jacobi using different test cases.

	Parallel Jacobi			
Number of equations	12	24	60	144
Max frequency (kHz)	450.5	387.6	411.5	352.4
Number of LUTs	85,401	185,149	586,797	2,101,129
Number of flip-flops	8,171	16,379	41,051	99,083
Memory bytes	252	516	1,290	3,168
Number of FPGAs in design	2	3	8	29

only on the number of clock cycles. The main source that introduces an increase in hardware resources consumption is the increase in the number of ALUs. The critical path is generally not affected.

TABLE 7.6: Hardware resource utilization for single floating point Clustered Jacobi using different test cases.

	Clustered Jacobi (single)					
Number of equations	420	1,200	4,900	11,100	44,700	179,400
Number of ALUs	14	24	49	74	149	299
Max frequency (kHz)	1666.7	1666.7	1851	1754.4	1587.3	1754.4
Number of LUTs	106,401	177,933	356,749	535,566	1,071,976	2,144,796
Number of flip-flops	18,752	29,878	57,684	85,490	168,896	335,702
Memory bytes	8,704	30,208	124,416	376,832	1,521,664	6,115,328
Number of FPGAs in design	2	3	5	8	15	29

TABLE 7.7: Hardware resource utilization for double floating point Clustered Jacobi using different test cases.

	Clustered Jacobi (double)		
Number of equations	420	1,200	3,120
Number of ALUs	14	24	39
Max frequency (kHz)	1282.1	1333.3	1149.4
Number of LUTs	716,902	1,236,864	2,016,786
Number of flip-flops	69,359	114,645	182,571
Memory bytes	17,408	60,416	197,632
Number of FPGAs in design	10	17	27

7.2.2 Speed-up

Speed-up is a very common criterion to evaluate the performance of a parallel system. In this section, the speed-up is evaluated using the hardware implementations presented in chapters 3, 4 and 5 against the best software algorithms; Matlab and ALGLiB.

We used Matlab's special operator for solving systems of linear equations on a 2.00GHz Core i7-2630QM CPU and ALGLiB, a numerical analysis and data processing library, using C++ programming language under linux on a 2.00GHz Core i7-2630QM CPU.

7.2.2.1 Gaussian architectures

Tables 7.8 and 7.9 show the obtained results from comparing the 32-bit Floating-point Clustered Gaussian Elimination Hardware Implementation against software solutions.

TABLE 7.8: 32 bit Floating-point Clustered Gaussian Hardware vs Software (MATLAB)

Number of equations	Matlab	Gaussian	speed-up
40	7.70E-05	7.79E-05	0.988
420	0.000203721	3.25E-04	0.627
11,100	0.004055954	0.001891473	2.1444
19,800	0.009724745	0.002479798	3.922
44,700	0.025323542	0.004270893	5.929
60,900	0.0367694	4.68E-03	7.8633

TABLE 7.9: 32 bit Floating-point Clustered Gaussian Hardware vs Software (ALGLIB)

Number of equations	ALGLIB	Gaussian	speed-up
40	7.59205E-06	7.79E-05	0.09752
420	4.09283E-05	3.25E-04	0.12604
11,100	0.001118666	0.001891473	0.591449
19,800	0.00211229	0.002479798	0.851832
44,700	0.00517497	0.004270893	1.21171
60,900	0.00948192	4.68E-03	2.0277844

Figure 7.3 shows a graphical representation of the results in Tables 7.8 and 7.9. Time (in seconds) is plotted against the number of equations solved for different test cases.

7.2.2.2 Jacobi architectures

Tables 7.10 and 7.11 give the obtained results from comparing the 32-bit Floating-point Clustered Jacobi Hardware Implementation against software solutions.

Figure 7.4 shows a graphical representation of the results in Tables 7.10 and 7.11. Time (in seconds) is plotted against the number of equations solved for different test cases. Both software and hardware solutions increase almost linearly. However the slope in

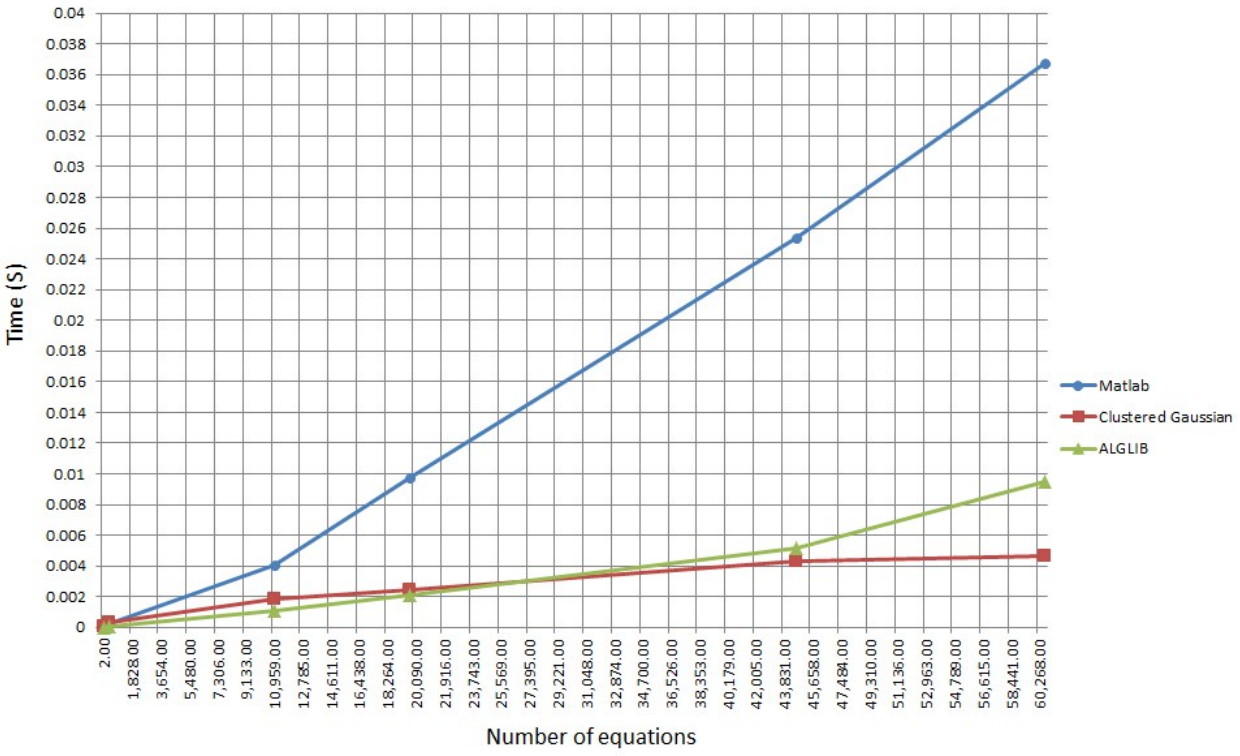

FIGURE 7.3: 32 bit Floating-point Clustered Gaussian Hardware vs Software

TABLE 7.10: 32 bit Floating-point Clustered Jacobi Hardware vs Software (MATLAB)

Number of equations	Matlab	Jacobi (32-bit)	speed-up
420	0.000203721	3.19E-04	0.639
1,200	0.000458758	6.72E-04	0.683
4,900	0.001747283	0.001091	1.6
11,100	0.004055954	0.001721	2.357
44,700	0.025323542	0.0037926	6.676
179,400	0.160800825	0.006167	26.07

TABLE 7.11: 32 bit Floating-point Clustered Jacobi Hardware vs Software (ALGLIB)

Number of equations	ALGLIB	Jacobi (32-bit)	speed-up
420	4.09283E-05	3.19E-04	0.128467
1,200	0.000107811	6.72E-04	0.160436
4,900	0.000513232	0.001091	0.470423
11,100	0.001118666	0.001721	0.650009
44,700	0.00517497	0.0037926	1.36449
179,400	0.03611	0.006167	5.85526

case of the hardware solution is less than the slope of software solutions. As a result, More speed-up is obtained at large number of equations.

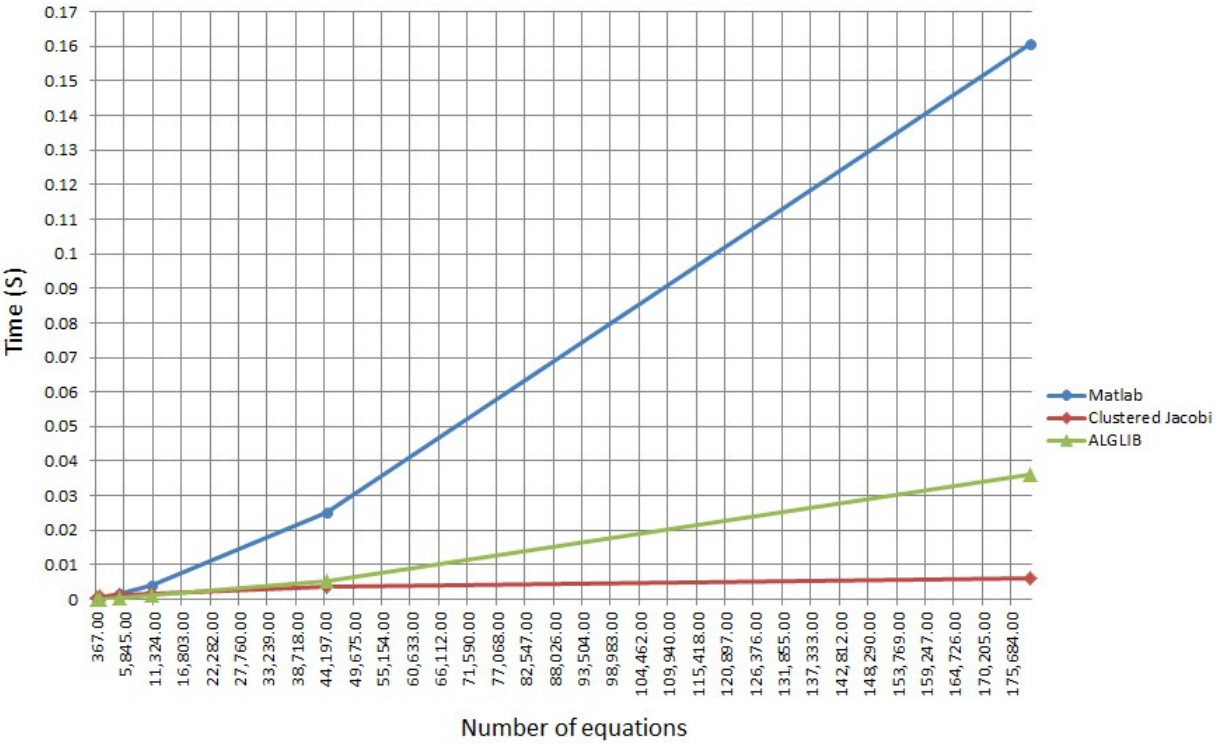

FIGURE 7.4: 32 bit Floating-point Clustered Jacobi Hardware vs Software

Tables 7.12 and 7.13 give the obtained results from comparing the 64-bit Floating-point Clustered Jacobi Hardware Implementation against software solutions.

TABLE 7.12: 64 bit Floating-point Clustered Jacobi Hardware vs Software (MATLAB)

Number of equations	Matlab	Jacobi (64-bit)	speed-up
420	0.000203721	9.57E-04	0.212
1,200	0.000458758	0.00152	0.302
3,120	0.001165882	0.0028	0.416

TABLE 7.13: 64 bit Floating-point Clustered Jacobi Hardware vs Software (ALGLIB)

Number of equations	ALGLIB	Jacobi (64-bit)	speed-up
420	4.09283E-05	9.57E-04	0.042766
1,200	0.000107811	0.00152	0.070928
3,120	0.000348027	0.0028	0.124295

7.2.3 Numerical precision

In this section, numerical precision is compared between hardware implementations and Matlab solution for our case study discussed in chapter 6. The maximum absolute error between Matlab solution and both hardware solutions (Clustered Jacobi and Clustered Gaussian) for different test cases are given in Table 7.14.

TABLE 7.14: Absolute error in the Clustered Jacobi and Clustered Gaussian with respect to software(Matlab)

Number of equations	Jacobi error	Gaussian error
420	6.17E-07	1.34E-07
11,100	9.88E-07	1.42E-07
44,700	1.03E-06	1.44E-07

Table 7.15 shows the effect of using 64 bit Floating-point modules instead of 32 bit Floating-point ones on accuracy. The absolute error decreased dramatically from 10^{-6} to 10^{-13}. That happens at the expense of the total number of LUTs and other hardware resources used to implement the design.

TABLE 7.15: Absolute error in the 64 bit Floating-point Clustered Jacobi with respect to software(Matlab)

Number of equations	Jacobi_d error
420	3.76E-13
1,200	6.65E-13
3,120	8.10E-13

Chapter 8

Conclusion and future work

8.1 Conclusion

This work tackled the electromagnetic fields simulation problem, which is one of the most widely studied problems in electrical engineering. The main target was to implement hardware solutions capable of minimizing the huge time requirements for simulating complex electromagnetic systems, which would consequently greatly minimize analysis time, debugging time, and time to market. Being one of the most recently introduced topics with a wide range of important applications, this work used Metamaterials as a case study.

Among the different available numerical methods for solving Maxwell's equations, the Finite Element Method was chosen as it is a highly versatile numerical method that has received considerable attention by scientists and researchers around the world after the latest technological advancements and computer revolution of the twentieth century.

Three different algorithms for solving systems of linear equations were implemented in Verilog as hardware-based solutions. Moreover, many variants of each of these algorithms were introduced depending on the growing need for more efficient solutions throughout this work. The introduced approaches included Cramer and Gaussian Elimination as direct methods, and Jacobi method as an example of iterative methods. These variants ranged from extremely fast designs which were not scalable such as Cramer and Parallel Jacobi, to considerably fast, scalable designs with a high degree of flexibility such as Clustered Sparse Gaussian Elimination and Clustered Jacobi, going through other

designs which were the bases of all these efficient algorithms such as Basic Jacobi and Gaussian Elimination. Other solutions were considered which were not as promising, and therefore, were not mentioned in this thesis.

Emulation technology was introduced in this work for the first time in electrical engineering research, which shed light on a new path of research in solutions to problems involving electromagnetic fields as a specific example, and other problems involving hardware platforms in general. Many test cases were considered and tested on the emulator hardware platform against software solutions including Matlab as a benchmark for matrix solvers, and ALGLiB as an example of standard C++ libraries. The results of these various test cases were documented in this thesis to show the difference in efficiency between software and hardware solutions.

8.2 Future work

Some suggestions and directions for future work are presented below:

1. The architecture of 64 bit floating-point Clustered Jacobi need to be enhanced to achieve higher speed up over software solutions.

2. There is a need to implement the GUI and the post-processing part of our program in C/C++. That will give us the opportunity to move all our work to the Emulator server machine.

3. Implementing hardware solutions for Maxwell's equations using methods other Finite Element Method would be another useful extension of this study, as the problem targeted in the thesis mainly depends on FEM.

4. The concept of DPI has only been demonstrated on one architecture. It would be useful to integrate the whole designs with DPI to have more flexiblity on the emulator environment and save compiling time.

5. Currently, A simple Electormagnetic fields case study is introduced as a proof of concept only. Complex designs need to be discussed to evaluate the performacne of the hardware-based implementations against large scale problems. Timing results need to be compared to EM simulators like HFSS and CST.

Appendix A

Metamaterials

Let us start this Appendix with a brief discussion on the origins of metamaterials, and their basic electromagnetic and optical properties. We then present some of metamaterial potential applications. After all these, a brief overview of the related mathematical problems is provided by introducing the governing equations used to model the wave propagation in metamaterials.

A.1 The Concept of Metamaterials

A metamaterial is a metallic or semiconductor substance whose properties depend on its inter-atomic structure rather than on the composition of the atoms themselves. Certain metamaterials bend visible light rays in the opposite sense from traditional refractive media. Some metamaterials also exhibit such behavior at infrared (IR) wavelengths.

A light beam passing from air or a vacuum into a common refractive medium such as glass, water or quartz is bent at the surface boundary so its path inside the material is more nearly perpendicular to the surface than its path outside the material Figure A.1. The extent of the bending depends on the angle at which the ray strikes the boundary and also on the index of refraction of the medium. All common transparent materials have positive indices of refraction. A metamaterial bends an incident ray so its internal direction is reversed Figure A.2.

More specifically, we are interested in a metamaterial with simultaneously negative electric permittivity ϵ and magnetic permeability μ. In general, both permittivity ϵ and

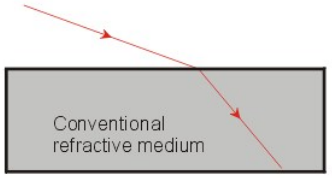

FIGURE A.1: Incident ray on conventional refractive medium.

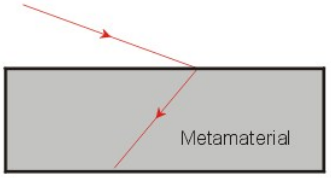

FIGURE A.2: Incident ray on Metamaterial.

permeability μ depend on the molecular and perhaps crystalline structure of the material, as well as bulk properties such as density and temperature.

A.2 Historical background

Back in 1968, Russian physicist Victor Veselago wrote a seminar paper [26] on metamaterials (he then called them left-handed materials). In that paper, he speculated that the strikingly unusual phenomena could be expected in a hypothetical lefthanded material in which the electric field E, the magnetic field H and the wave vector k form a left-handed system. The paper explicitly presented that to achieve such a left-handed material, the required material parameters should be simultaneously negative for both permittivity and permeability. However, due to the non-existence of such materials in nature, Veselago's paper did not make a big impact until the first successful construction of such a medium by Smith et al.

In 2000 [27], and the first experimental demonstration of the negative refractive index in 2001 [28]. Another catalyst was caused by Pendry's landmark work on perfect lens [29], which sparked the attempt to consider metamaterials for many potentially exciting applications. According to [30] p.317, these four seminar papers together made the birth of the subject of metamaterials. Since 2000, there has been a tremendous growing interest in the study of metamaterials and their potential applications in areas ranging from electronics, telecommunications to sensing, radar technology, sub-wavelength imaging, data storage, and design of invisiblity cloak.

A.3 Potential Applications

A.3.1 Antenna Applications

Recently, researchers have proposed some methods to obtain miniaturized antennas made of ideal homogenized metamaterials. The first design of a subwavelength antenna with metamaterials for the case of dipole and monopole radiators was proposed by Ziokowski's group [31]. The basic design consists of an electrically short electric dipole (or monopole) surrounded by a double-negative (DNG) or an epsilon-negative (ENG) spherical shell with an electrically short radius. The compact resonance arises at the interface between the DNG (or ENG) shell and the free space. Similar ideas have been used to design subwavelength patch antennas [32, 33] and leaky-wave antennas [34].

A.3.2 Cloaking

Invisibility has long been a dream of human beings. Cloaking devices are advanced stealth technologies still in development that can make objects partially or wholly invisible to some portions of the electromagnetic spectrum. Generally speaking, there are several major approaches to render objects invisible. But in May 2006, the first full wave numerical simulations on cylindrical cloaking was carried out by Cummer et al. [35]. A few months later, the first experiment of such a cloak at microwave frequencies was successfully demonstrated by Schurig et al. [36], where the cloak surrounding a 25-mm-radius Cu cylinder was measured. After 2006, numerous studies have been devoted to cloaking, mainly inspired by [35, 36].

A.3.3 Particle Detection and Biosensing

Other potential applications are Particle Detection and Biosensing. Particle Detection makes use of the negative refractive index of metamaterials , which results in the so-called reversed Cherenkov radiation (CR) [26], in improving the Cherenkov detectors inside nuclear reactors.

Conventional biosensors (such as those based on electro-mechanical transduction, fluorescence, nanomaterials, and surface plasmon resonance) often involve labor-intensive

sample preparation and very sophisticated equipment. In recent years, researchers have proposed to use metamaterials as candidates for detection of highly sensitive chemical, biochemical and biological analytes.

A.4 Governing Equations for Metamaterials

The Maxwell's equations are the fundamental equations for understanding most electromagnetic and optical phenomena. In time domain, the general Maxwell's equations can be written as

$$Faraday's law(1831) : \nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}. \tag{A.1}$$

$$Ampere's law(1820) : \nabla \times \mathbf{H} = \frac{\partial \mathbf{D}}{\partial t}. \tag{A.2}$$

which are used to describe the relationship between electric field $\mathbf{E}$(x,t) and magnetic field $\mathbf{H}$(x,t), and the underlying electromagnetic materials can be described by two material parameters: the permittivity ϵ and the permeability μ. In equation A.1 and equation A.2, we use the electric flux density $\mathbf{D}$(x,t) and magnetic flux density $\mathbf{B}$(x,t), which are related to the fields $\mathbf{E}$ and $\mathbf{H}$ through the constitutive relations given by

$$\mathbf{D} = \epsilon_0 \mathbf{E} + \mathbf{P} \equiv \epsilon \mathbf{E},\ \mathbf{B} = \mu_0 \mathbf{H} + \mathbf{M} \equiv \mu \mathbf{H}. \tag{A.3}$$

where ϵ_0 is the vacuum permittivity, μ_0 is the vacuum permeability, and $\mathbf{P}$ and $\mathbf{M}$ are the induced polarization and magnetization, respectively. Note that $\mathbf{P}$ and $\mathbf{M}$ are caused by the impinging fields, which can influence the organization of electrical charges and magnetic dipoles in a medium. How big the induced polarization $\mathbf{P}$ and magnetization $\mathbf{M}$ are depends on the particular material involved. For example, in vacuum, $\epsilon = \epsilon_0$, $\mu = \mu_0$, hence $\mathbf{P} = \mathbf{M} = 0$; while in pure water, $\epsilon = 80\epsilon_0$ and $\mu = \mu_0$, which lead to $\mathbf{P} = 79\epsilon_0\mathbf{E}$ and $\mathbf{M} = 0$.

For metamaterials, the permittivity ϵ and the permeability μ are not just simple constants due to the complicated interaction between electromagnetic fields and meta-atoms

(i.e., the unit cell structure). Since the scale of inhomogeneities in a metamaterial is much smaller than the wavelength of interest, the responses of the metamaterial to external fields can be homogenized and are described using effective permittivity and effective permeability. A popular model for metamaterial is the lossy Drude model [37, 38], which in frequency domain is described by:

$$\epsilon(\omega) = \epsilon_0(1 - \frac{\omega_{pe}^2}{\omega(\omega - j\Gamma_e)}) = \epsilon_0\epsilon_r. \tag{A.4}$$

$$\mu(\omega) = \mu_0(1 - \frac{\omega_{pm}^2}{\omega(\omega - j\Gamma_m)}) = \mu_0\mu_r. \tag{A.5}$$

where ω_{pe} and ω_{pm} are the electric and magnetic plasma frequencies, Γ_e and Γ_m are the electric and magnetic damping frequencies, and ω is a general frequency. A simple case for achieving negative refraction index $n = -\sqrt{\epsilon_r\mu_r} = -1$ is to choose $\Gamma_e = \Gamma_m = 0$ and $\omega_{pe} = \omega_{pm} = \sqrt{2}\omega$.

A derivation of A.4 is given in[39] for very thin metallic wires assembled into a periodic lattice. Assuming that the wires have radius r, and are arranged in a simple cubic lattice with distance a between wires, and σ is the conductivity of the metal, Pendry et al. [39] showed that

$$\omega_{pe}^2 = \frac{2\pi c^2}{a^2 \ln(a/r)}, \; \Gamma_e = \frac{\epsilon_0 a^2 \omega_{pe}^2}{\pi r^2 \sigma}. \tag{A.6}$$

where c denotes the speed of light in vacuum.

Using a time-harmonic variation of $exp(jwt)$. from A.3 to A.5 we can obtain the corresponding time domain equations for the polarization $\mathbf{P}$ and the magnetization $\mathbf{M}$ as follows:

$$\frac{\partial^2 \mathbf{P}}{\partial t^2} + \Gamma_e \frac{\partial P}{\partial t} = \epsilon_0 \omega_{pe}^2 \mathbf{E}. \tag{A.7}$$

$$\frac{\partial^2 \mathbf{M}}{\partial t^2} + \Gamma_m \frac{\partial M}{\partial t} = \mu_0 \omega_{pm}^2 \mathbf{H}. \tag{A.8}$$

Furthermore, if we denote the induced electric and magnetic currents

$$\mathbf{J} = \frac{\partial \mathbf{P}}{\partial t},\ \mathbf{K} = \frac{\partial \mathbf{M}}{\partial t}, \tag{A.9}$$

then we can obtain the governing equations for modeling the wave propagation in a DNG medium described by the Drude model [40]:

$$\epsilon_0 \frac{\partial \mathbf{E}}{\partial t} = \nabla \times \mathbf{H} - \mathbf{J}. \tag{A.10}$$

$$\mu_0 \frac{\partial \mathbf{H}}{\partial t} = -\nabla \times \mathbf{E} - \mathbf{K}. \tag{A.11}$$

$$\frac{1}{\epsilon_0 \omega_{pe}^2} \frac{\partial \mathbf{J}}{\partial t} + \frac{\Gamma_e}{\epsilon_0 \omega_{pe}^2} \mathbf{J} = \mathbf{E}. \tag{A.12}$$

$$\frac{1}{\mu_0 \omega_{pm}^2} \frac{\partial \mathbf{K}}{\partial t} + \frac{\Gamma_m}{\mu_0 \omega_{pm}^2} \mathbf{K} = \mathbf{H}. \tag{A.13}$$

Note that the two-dimensional transverse magnetic model of [37], Eq. (10) can be obtained directly from A.10 to A.13 by assuming that components $E_y, H_x, H_z \neq 0$, while the rest components are 0.

Another popular model used for modeling wave propagation in metamaterials is described by the so-called Lorentz model [38, 41, 42], which in frequency domain is given by

$$\epsilon(\omega) = \epsilon_0 (1 - \frac{\omega_{pe}^2}{\omega^2 - \omega_{e0}^2 - j\Gamma_e \omega}),\ \mu(\omega) = \mu_0 (1 - \frac{\omega_{pm}^2}{\omega^2 - \omega_{m0}^2 - j\Gamma_m \omega}). \tag{A.14}$$

where ω_{pe}, ω_{pm}, Γ_e and Γ_m have the same meaning as the Drude model. Furthermore, ω_{e0} and ω_{m0} are the electric and magnetic resonance frequencies, respectively.

A derivation of $\mu_r(\omega) = 1 - \frac{F\omega^2}{\omega^2 - \omega_{m0}^2 - j\Gamma_m \omega}$ is shown by Pendry et al. for a composite medium consisting of a square array of cylinders with split ring structure(cf. [43], Fig.3) formed by two sheets separated by a distance d. More specifically, they derived

$\omega_{m0}^2 = \frac{3dc^2}{\pi^2 r^3}$, $\Gamma_m = \frac{2\sigma}{\mu_0 r}$, $F = \frac{\pi r^2}{a^2}$.

where the parameters a, c, r and σ have the same meaning as in A.6. Later, Smith and Kroll [269] changed $F\omega^2$ to $F\omega_0^2$ to ensure that $\mu_r(\omega) \to 1$ as $\omega \to \infty$. This new choice results the Lorentz model A.14 with $\omega_{pm}^2 = F\omega_0^2$.

Transforming A.14 into time domain, we obtain the Lorentz model equations for metamaterials:

$$\epsilon_0 \frac{\partial \mathbf{E}}{\partial t} + \frac{\partial \mathbf{P}}{\partial t} - \nabla \times \mathbf{H} = 0. \tag{A.15}$$

$$\mu_0 \frac{\partial \mathbf{H}}{\partial t} + \frac{\partial \mathbf{M}}{\partial t} + \nabla \times \mathbf{E} = 0. \tag{A.16}$$

$$\frac{1}{\epsilon_0 \omega_{pe}^2} \frac{\partial^2 \mathbf{P}}{\partial t^2} + \frac{\Gamma_e}{\epsilon_0 \omega_{pe}^2} \frac{\partial \mathbf{P}}{\partial t} + \frac{\omega_{e0}^2}{\epsilon_0 \omega_{pe}^2} \mathbf{P} - \mathbf{E} = 0. \tag{A.17}$$

$$\frac{1}{\mu_0 \omega_{pm}^2} \frac{\partial^2 \mathbf{M}}{\partial t^2} + \frac{\Gamma_m}{\mu_0 \omega_{pm}^2} \frac{\partial \mathbf{M}}{\partial t} + \frac{\omega_{m0}^2}{\mu_0 \omega_{pm}^2} \mathbf{M} - \mathbf{H} = 0. \tag{A.18}$$

he last popular model we want to mention is a mixed model used by engineers and physicists [29, 41–44], in which the permittivity is described by the Drude model, while the permeability is described by the Lorentz model. More precisely, the permittivity is described by the Drude model [29, 43]:

$$\epsilon(\omega) = \epsilon_0 (1 - \frac{\omega_p^2}{\omega(\omega + jv)}) \tag{A.19}$$

where ω is the excitation angular frequency, $\omega_p > 0$ is the effective plasma frequency, and $v > 0$ is the loss parameter. On the other hand, the permeability can be described by the Lorentz model [41, 42]:

$$\mu(\omega) = \mu_0 (1 - \frac{F\omega_0^2}{\omega^2 + j\gamma\omega - \omega_0^2}) \tag{A.20}$$

where $\omega_0 > 0$ is the resonant frequency, $\gamma \geq 0$ is the loss parameter, and $F \in (0, 1)$ is a parameter depending on the geometry of the unit cell of the metamaterial.

Using a time-harmonic variation of $exp(jwt)$, and substituting A.19 and A.20 into A.3, respectively, we obtain the time-domain equation for the polarization:

$$\frac{\partial^2 \mathbf{P}}{\partial t^2} + v\frac{\partial \mathbf{P}}{\partial t} = \epsilon_0 \omega_p^2 \mathbf{E}. \tag{A.21}$$

and the equation for the magnetization:

$$\frac{\partial^2 \mathbf{M}}{\partial t^2} + \gamma\frac{\partial \mathbf{M}}{\partial t} + \omega_0^2 \mathbf{M} = \mu_0 F \omega_0^2 \mathbf{H}. \tag{A.22}$$

To facility the mathematical study of the model, by introducing the induced electric current $\mathbf{J} = \frac{\partial \mathbf{B}}{\partial t}$ and magnetic current $\mathbf{K} = \frac{\partial \mathbf{M}}{\partial t}$, we can write the time domain governing equations for the Drude-Lorentz model as following:

$$\epsilon_0 \frac{\partial \mathbf{E}}{\partial t} = \nabla \times \mathbf{H} - \mathbf{J}. \tag{A.23}$$

$$\mu_0 \frac{\partial \mathbf{H}}{\partial t} = -\nabla \times \mathbf{E} - \mathbf{K}. \tag{A.24}$$

$$\frac{1}{\mu_0 \omega_0^2 F} + \frac{\gamma}{\mu_0 \omega_0^2 F}\mathbf{K} + \frac{1}{\mu_0 F}\mathbf{M} = \mathbf{H}. \tag{A.25}$$

$$\frac{1}{\mu_0 F}\frac{\partial \mathbf{M}}{\partial t} = \frac{1}{\mu_0 F}\mathbf{K}. \tag{A.26}$$

$$\frac{1}{\epsilon_0 \omega_p^2}\frac{\partial \mathbf{J}}{\partial t} + \frac{v}{\epsilon_0 \omega_p^2}\mathbf{J} = \mathbf{E}. \tag{A.27}$$

Appendix B

Finite Element Method

Computational electromagnetics [45] can be classified into either frequency-domain simulation or time-domain simulation. Each category can be further classified into surface-based or volume-based methods. The method of moments (MoM) or boundary element method (BEM) is formulated as integral equations given on the surface of the physical domain. Note that MoM [46] or BEM [47, 48] is applicable to problems for which Green's functions of the underlying partial differential equations are available, which limits its applicability.

Hence the volume-based methods such as the finite element method, the finite difference method, the finite volume method (e.g. [49, 50]), and the spectral method (direct applications in computational electromagnetics [51, 52]; applications in broader areas [28, 53]) are quite popular.

One of the most favorite methods is the so-called finite-difference time-domain (FDTD) method proposed by Yee in 1966 [54]. Due to its simplicity, the FDTD method is very popular in the electrical engineering community, and it is especially useful for broadband simulations, since one single simulation can cover a wide range of frequencies.

But the FDTD method has a major disadvantage when it is used for complex geometry simulation. In this case, the finite element method (FEM) is a better choice as evidenced by several published books in this area. For example, books [55] and [56] focus on how to develop and implement FEMs for solving Maxwell's equations. Many other books mainly focus on Maxwell's equations in free space, except that [57] is devoted

to Maxwell's equations in metamaterials. In this Appendix, we will focus on the finite element method.

B.1 Overview

The finite element method is considered as the most general and well understood PDEs solution available. "*The finite element method replaces the original function with a function that has some degree of smoothness over the global domain but is piecewise polynomial on simple cells, such as small triangles or rectangles.*" [7]

The essential idea of the finite element method is to approach the continuous functions of the exact solution of the PDE using piecewise approximations, generally polynomials [58]. A complex system is constructed with points called nodes which constitute a grid called a mesh. This mesh is programmed to contain the material and structural properties which define how the structure will react to predefined loading conditions in the case of structural analysis. Nodes are assigned at a predefined density throughout the material depending on the anticipated stress levels of a particular area.

B.2 Finite Element Algorithm

A basic flow chart of FEM is shown in Figure B.1. First, the spatial domain for the analysis is sub-divided by a geometric discretization based on a variety of geometrical data and material properties using a number of different strategies. Generally, the solution domain is descretized into triangular elements or quadrilateral elements, which are the two most common forms of two-dimensional elements. Then, the element matrices and forces are formed; then the system equations are assembled and solved. Finally, the results are post-processed so that the results are presented in a suitable form for human interaction.

The finite element method is used to model and simulate complex physical systems. The continuous functions are discretized into piecewise approximations, so the whole system is broken to many, but finite parts. However, the finite element method can be extremely computationally expensive and the available memory can be exhausted, especially when the number of grid points becomes large. The resulting system of

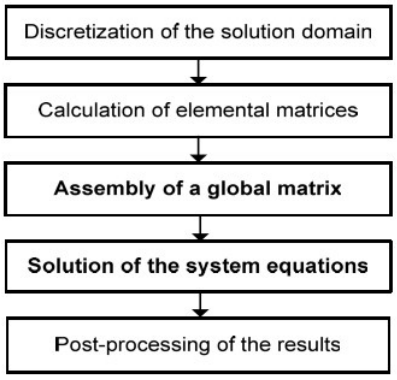

FIGURE B.1: Flow chart for the finite element algorithm.

equations may be solved either by direct methods or iterative methods such as Jacobi, Gauss Seidel, Conjugate Gradients or other advanced iterative methods such as the Preconditioned Conjugate Gradient (PCG) method, Incomplete Cholesky Conjugate Gradient method and GMRES.

The direct method can provide the accurate solution with minimal round-off errors, but it is computationally expensive in terms of both processing and memory requirements, especially for large matrices and three dimensional problems because the original zero entries will be filled in during the elimination process. The global matrix for linear structural problems is a symmetric positive definite matrix. In general, it is also large and sparse. Consequently, the iterative methods are more efficient and more suitable for parallel computation but with lower accuracy (though higher accuracy can be obtained at the expense of computational time) and the risk of a slow convergence rate.

Instead of assembling the global matrix, element stiffness matrices can be used directly for iterative solution techniques. An element-by-element approximation for finite element equation systems was presented in [59], and applied in the context of conventional parallel computing in [60]. This approach is very memory-efficient (despite the fact that more memory is required than storing just the non-zero elements as in the CSR structure), computationally convenient and retains the accuracy of the global coefficient matrix.

Appendix C

Solving system of linear equations

In this appendix, we provide the background material related to systems of linear equations in general and the mathematical methods commonly used in solving them.

C.1 Systems of Linear Equations

A system of M linear equations with N unknown variables, as shown in Equation C.1, is often represented in a matrix and vector form, as shown in Equation C.2. The coeffcients of the variables in each linear equation are represented in each row of an M x N matrix (A) and they are multiplied by the N-element vector of unknown variables (x). A solver must determine the values of x for which the product generates the Mdimensional constant (b). Given a system of linear equations, there can be one solution, an infinite number of solutions or no solution. Here, we only focus on sparse linear systems generated from the FEM.

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,N}x_N &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,N}x_N &= b_2 \\ &\vdots \\ a_{M,1}x_1 + a_{M,2}x_2 + \ldots + a_{M,N}x_N &= b_M \end{aligned} \tag{C.1}$$

$$Ax = b$$

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M,1} & a_{M,2} & \cdots & a_{M,N} \end{pmatrix} x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_M \end{pmatrix} \tag{C.2}$$

C.2 Different methods of solving sparse linear systems

In general, Methods to solve sparse linear systems (SLS) of equations fall under two broad categories: *direct methods* and *iterative methods*. Iterative methods start with an initial guess and compute a sequence of intermediate results that converge to a final solution while direct methods determine the solution in a single process. Moreover, iterative methods are useful for finding solutions to large systems of linear equations where direct methods are prohibitively expensive. However, convergence is not always guaranteed for iterative methods since they are more sensitive to the matrix type being solved which limits the scope of the range of the matrices that can be solved.

The most common direct methods are Gaussian elimination, Gauss-Jordan and LU decomposition. All of them have a computational complexity of order n^3 and are sensitive to numerical errors. For iterative technique, Jacobi and Gauss Seidel methods are used typically and have computational complexity of order n^2 for each iteration step, and can produce results with higher precision than direct methods. Hence, both iterative and direct solvers are widely used. More details on methods of solving linear equations can be found in [61] and [62].

C.2.1 Iterative methods for sparse linear systems

There are two main iterative methods used to solve the SLS expressed as $Ax = b$. They are described in the following sections.

C.2.1.1 Jacobi method

Consider a SLS represented as Ax = b. Then the i^{th} equation can be represented as

$$\sum_{j=1}^{n} a_{i,j}x_j = b_i \tag{C.3}$$

To solve for x_i iteratively, equation C.3 can be rearranged to have a relation between ${x_i}^{(t+1)}$ and ${x_i}^{(t)}$

$${x_i}^{(t+1)} = \frac{b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{i,j}{x_j}^{(t)}}{a_{i,i}} \tag{C.4}$$

The Jacobi method starts with an initial guess of vector x and solves each unknown x using equation C.4. The obtained ${x_i}^{(t+1)}$ is then used as the current solution (acts as ${x_i}^{(t)}$) and the process is iterated again. This process continues until it converges to a solution.

From equation C.4, each x_i can be solved independently and thus the Jacobi method has inherent parallelism. Furthermore, the most computationally intensive operation is to find the summation. Besides, the division operation takes more computational time than any other numerical operation used in equation C.4. Therefore, an architecture that finds the summation quickly and maximally overlaps the division operation with the computation of numerator of equation C.4, should result in faster system operation.

C.2.1.2 Gauss seidel method

In contrast to the Jacobi method, the Gauss Seidel method not only uses ${x_i}^{(t)}$ but also uses ${x_i}^{(t+1)}$ as soon as they become available. The independent nature of x_i variables in any particular iteration, as in the Jacobi method, does not hold here.

Mathematically, in order to find the solutions of any SLS represented in a matrix format $Ax = b$ is given below:

$${x_i}^{(t+1)} = \frac{b_i - \sum_{j<1}^{n} a_{i,j}{x_j}^{(t+1)} - \sum_{j>1}^{n} a_{i,j}{x_j}^{(t)}}{a_{i,i}} \tag{C.5}$$

From equation C.5, the computations of x_i appear to be serial. Thus the updates cannot be done simultaneously as in the Jacobi method.

C.2.1.3 Convergence criterion

It is important to consider that not all matrices can be solved using iterative methods. A matrix can be solved only when it is diagonally dominant. A matrix is said to be diagonally dominant if the magnitude of every diagonal entry is more than the sum of the magnitude of all the non-zero elements of the corresponding row. Mathematically, a matrix is diagonally dominant if it satisfies equation C.6 for all i = 1..n

$$|a_{i,i}| > \sum_{j=1, j\neq i}^{n} |a_{i,j}| \tag{C.6}$$

Both methods sometimes converge even if this condition is not satisfied. However, it is necessary that the magnitude of diagonal terms in a matrix is greater than the magnitude of other terms.

C.2.2 Direct methods for sparse linear systems

Here, we will focus on Gaussian elimination, the one used in the second hardware implementation.

C.2.2.1 Gaussian elimination method

Direct methods for solving the system equations theoretically deliver an exact solution in arbitrary-precision arithmetic by a (predictable) finite sequence of operations based on algebraic elimination. Gauss elimination, Gauss Jordan elimination and LU factorization are some of the examples of direct methods.

Consider the system of linear algebraic equations,

$$Ax = b \tag{C.7}$$

where matrix A is the coefficient nxn matrix obtained from the system of equations.

The Gauss elimination procedure is summarized as follows [63]:

1. Define the nxn coefficient matrix A, and the nx1 column vectors x and b,

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \tag{C.8}$$

2. Perform elementary row operations to reduce the matrix into the upper triangular form

$$\begin{pmatrix} a'_{1,1} & a'_{1,2} & \cdots & a'_{1,n} \\ 0 & a'_{2,2} & \cdots & a'_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a'_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_m \end{pmatrix} \tag{C.9}$$

3. Solve the equation of the n^{th} row for x_n, then substitute back into the equation of the $(n-1)^th$ row to obtain a solution for x_{n-1}, etc., according to the formula

$$x_i = \frac{1}{a'_{i,j}}(b'_i - \sum_{j=i+1}^{n} a'_{i,j}x_j) \tag{C.10}$$

The number of operations required by Gauss elimination method is $N = (n^3/3 - n/3) + n^2$ The Gauss-Jordan method, the matrix inverse method, the LU factorization method and the Thomas algorithm are variations or modifications of the Gauss elimination method. The Gauss-Jordan method requires more operations than the Gauss elimination method, which is $N = (n^3/2 - n/2) + n^2$. The matrix inverse method is simple but not all matrices have an inverse (there is no inverse matrix if the matrix's determinant is zero, i.e. singular, and no unique solution for the corresponding system of equations). The LU method requires $N = (4n^3/3 - n/3)$ multiplicative operations, which is much less than Gauss elimination, especially for large systems.

When either the number of equations is small (100 or less), or most of the coefficients in the equations are non-zero, or the system domain is not diagonal, or the system of equations is ill conditioned *(The condition number of a matrix is the ratio of the magnitudes of its maximum and minimum eigenvalues. A matrix is ill-conditioned if its*

condition number is very large.) direct elimination methods would normally be used. Otherwise, an alternative solution method for the system of equations is an iterative method. This is desirable when the number of equations is large, especially when the system matrix is sparse [64].

Appendix D

Floating Point

In this this Appendix we are going to introduce the floating point standards and its rule in this project

D.1 Overview

In this section we introduce some general information about Floating point IEEE standard and how it works.

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating point implementations that made them difficult to reliably and portably use. The current version, IEEE 754-2008 published in August 2008, includes nearly all of the original IEEE 754-1985 standard and the IEEE Standard for Radix-Independent Floating-Point Arithmetic (IEEE 854-1987).

IEEE 754-1985 was implemented in software, in the form of floating-point libraries, and in hardware, in the instructions of many CPUs and FPUs. The first integrated circuit to implement the draft of what was to become IEEE 754-1985 was the Intel 8087.

IEEE 754-1985 represents numbers in binary, providing definitions for four levels of precision, but the most used levels of precision are **single** and **double** precision as shown in Figure D.1.

level	width	range	precision*
single precision	32 bits	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	approx. 7 decimal digits
double precision	64 bits	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$	approx. 15 decimal digits

FIGURE D.1: Levels of precision implemented by IEEE standard.

Precision is the number of decimal digits precision is calculated via number_of_mantissa_bits * Log10(2). Thus 7.2 and 15.9 for single and double precision respectively.

D.2 IEEE standard representations

The standard also defines the normailzed and denormalized numbers representation, a negative zero representation, positive and negative infinity, NaN specaial value representation for invalid floating-point arithmatic operations, and four rounding modes. All these definitions are discused in the next sections.

D.2.1 Normalized numbers representation

Floating-point numbers in IEEE 754 format consist of three fields: a sign bit, a biased exponent, and a fraction as shown in Figure D.2.

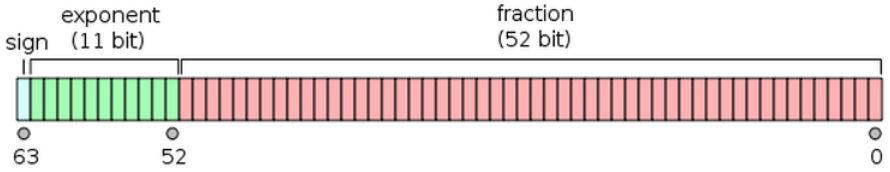

FIGURE D.2: The three fields in a 64 bit IEEE 754 float.

IEEE 754 adds a bias to the exponent so that numbers can in many cases be compared conveniently by the same hardware that compares signed 2's-complement integers. Using a biased exponent, the lesser of two positive floating-point numbers will come out "less than" the greater following the same ordering as for sign and magnitude integers. If two floating-point numbers have different signs, the sign-and-magnitude comparison also works with biased exponents. However, if both biased-exponent floating-point numbers are negative, then the ordering must be reversed. If the exponent were represented as, say, a 2's-complement number, comparison to see which of two numbers is greater would

not be as convenient. The leading 1 bit is omitted since all numbers except zero start with a leading 1; the leading 1 is implicit and doesn't actually need to be stored which gives an extra bit of precision for "free".

D.2.2 Denormalized numbers representation

As discussed in **Normalized numbers representation** section, the implicit leading binary digit is a 1. To reduce the loss of precision when an underflow occurs, IEEE 754 includes the ability to represent fractions smaller than are possible in the normalized representation, by making the implicit leading digit a 0. Such numbers are called "denormal". They don't include as many significant digits as a normalized number, but they enable a gradual loss of precision when the result of an arithmetic operation is not exactly zero but is too close to zero to be represented by a normalized number.

A denormal number is represented with a biased exponent of all 0 bits, which represents an exponent of -126 in single precision (not -127), or -1022 in double precision (not -1023). In contrast, the smallest biased exponent representing a normal number is 1.

D.2.3 Negative zero representation

The number zero is represented specifing **sign = 0** for positive zero, and **sign = 1** for negative zero. The biased exponent value equals zero, and the fraction value also equals zero.

D.2.4 Positive and negative infinty representation

Positive and negative infinity are represented by specifing **sign = 0** for positive infinity, and **sign = 1** for negative infinity. The biased exponent value is set to one for all its bits, and the fraction value is set to zero for all its bits.

D.2.5 NaN special value

Some operations of floating-point arithmetic are invalid, such as dividing by zero or taking the square root of a negative number. The act of reaching an invalid result is

called a floating-point exception. An exceptional result is represented by a special code called a NaN, for "Not a Number". All NaNs in IEEE 754-1985 have this format:

- sign = either 0 or 1.
- biased exponent = all 1 bits.
- fraction = anything except all 0 bits (since all 0 bits represents infinity).

D.2.6 Rounding modes

The IEEE standard has four different rounding modes; the first is the default; the others are called directed roundings.

- **Round to Nearest** : rounds to the nearest value; if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit, which occurs 50% of the time (in IEEE 754-2008 this mode is called roundTiesToEven to distinguish it from another round-to-nearest mode).
- **Round toward 0** : directed rounding towards zero.
- **Round toward positive infinity** : directed rounding towards positive infinity.
- **Round toward negative infinity** : directed rounding towards negative infinity.

D.3 Floating point modules

In this section we are going to discuss the main floating-point modules used and how they work.

D.3.1 Arithmetic Operations

First analyze the main arithmetic operations and generate the corresponding computation algorithms. In what follows it will be assumed that the significand s is represented in base B (in binary if $B = 2$, in decimal if $B = 10$) and that it belongs to the interval $(1 \leq Bs \leq B - ulp)$, where ulp stands for the unit in the last position or unit of least

precision. Thus s is expressed in the form (s_0, s_1, s_2, s_p). B^e where $e_{min} \leq e <= e_{max}$ and $1 \leq s_0 \leq B-$.

The binary subnormals and the decimal floating point are not normalized numbers and are not included in the following analysis. This situation deserves some special treatment and is out of the scope of this section.

D.3.1.1 Addition of positive numbers

Given two positive floating-point numbers $s_1.B^{e_1}$ and $s_2.B^{e_2}$ their sum $s.B^e$ is computed as follows: assume that e_1 is greater than or equal to e_2; then (alignment) the sum of $s1 * (B^e1)$ and $s2 * (B^e2)$ can be expressed in the form $s.B^e$ where

$$s = s_1 + s_2/B^{e_1-e_2} \; and \; e = e_1 \tag{D.1}$$

The value of s belongs to the interval

$$1 \leq s \leq 2.B - 2.ulp \tag{D.2}$$

so that s could be greater than or equal to B. If it is the case, that is if

$$B \leq s \leq 2.B - 2.ulp \tag{D.3}$$

then (normalization) substitute s by s/B, and e by e+1, so that the value of $s.B^e$ is the same as before, and the new value of s satisfies

$$1 \leq s \leq 2 - (2/B).ulp \leq B - ulp \tag{D.4}$$

The significands s_1 and s_2 of the operands are multiples of ulp. If e_1 is greater than e_2, the value of s could no longer be a multiple of ulp and some rounding function should be applied to s. Assume that $s^{'} < s < s^{"} = s^{'} + ulp$, $s^{'}$ and $s^{"}$ being two successive multiples of ulp. Then the rounding function associates to s either $s^{'}$ or $s^{"}$, according to some rounding strategy. According to D.4 and to the fact that 1 and $B - ulp$ are

multiples of ulp, it is obvious that $1 \leq s^{'} < s^{"} \leq B - ulp$. Nevertheless, if the condition in equation D.3 does not hold, that is if $1 \leq s < B$, s could belong to the interval so that rounding(s) could be equal to B, then a new normalization step would be necessary. i.e. substitution of $s = B$ by $s = 1$ and e by $e + 1$.

$$B - ulp < s < B \tag{D.5}$$

D.3.1.2 Difference of positive numbers

Given two positive floating-point numbers $s_1.B^{e_1}$ and $s_2.B^{e_2}$their difference $s.B^e$ is computed as follows: assume that e_1 is greater than or equal to e_2; then (for alignment) the difference between $s_1.B^{e_1}$ and $s_2.B^{e_2}$ can be expressed in the form $s.B^e$ where

$$s = s_1 - s2/(B^{e_1 - e_2}) and e = e_1 \tag{D.6}$$

The value of s belongs to the interval $-s. - (B - ulp) \leq s \leq B - ulp$. If s is negative, then it is substituted by and the sign of the final result will be modified accordingly. If s is equal to 0, then an exception $equal_zero$ could be raised. It remains to consider the case where $0 < s \leq Bulp$. The value of s could be smaller than 1. In order to normalize the significand, s is substituted by $s.B^k$ and e by $e - k$, where k is the minimum exponent k such that $s.B^k \geq 1$: Thus, the relation $1 \leq sleqB$ holds. It remains to round (up or down) the significand and to normalize it if necessary.

D.3.1.3 Addition and subtraction

Given two floating-point numbers $(-1)^{sign_1}.s_1.B^{e_1}$ and $(-1)^{sign_2}.s_2.B^{e_2}$; and a control variable operation, an algorithm is defined for computing

$z = (-1)^{sign}.s.B^e = (-1)^{sign_1}.s_1.B^{e_1} + (-1)^{sign_2}.s_2.B^{e_2}$, $if operation = 0$;

$z = (-1)^{sign}.s.B^e = (-1)^{sign_1}.s_1.B^{e_1} - (-1)^{sign_2}.s_2.B^{e_2}$, $if operation = 1$;

Once the significands have been aligned, the actual operation (addition or subtraction of the significands) depends on the values of operation, $sign_1 and sign_2$, shown in figure D.3.

Operation	$Sign_1$	$Sign_2$	Actual operation
0	0	0	$s_1 + s_2$
0	0	1	$s_1 - s_2$
0	1	0	$-(s_1 - s_2)$
0	1	1	$-(s_1 - s_2)$
1	0	0	$s_1 - s_2$
1	0	1	$s_1 + s_2$
1	1	0	$-(s_1 + s_2)$
1	1	1	$-(s_1 - s_2)$

FIGURE D.3: Effective operation in floating point adder-subtractor.

D.3.1.4 Multiplication

Given two flooating-point numbers $(-1)^{sign_1}.s_1.B^{e_1}$ and $(-1)^{sign_2}.s_2.B^{e_2}$ their product $(-1)^{s}ign.s.B^{e}$ is computed as follows:

$$sign = sign_1 \; xor \; sign_2, \; s = s_1.s_2, \; e = e_1 + e_2 \tag{D.7}$$

The value of s belongs to the interval $1 \leq s \leq (B - ulp)^2$, and could be greater than or equal to B. If it is the case, that is if $B \leq s \leq (B - ulp)^2$, then (normalization) substitute s by s/B, and e by $e + 1$. The new value of s satisfies

$$1 \leq s \leq (B - ulp)^2/B = B - 2.ulp + (ulp)^2/B < B - ulp \tag{D.8}$$

($ulp < B$ so that $2 - ulp/B > 1$). It remains to round the significand and to normalize if necessary.

D.3.1.5 Division

Given two floating-point numbers $(-1)^{s}ign_1.s_1.B^{e_1}$ and $(-1)^{s}ign_2.s_2.B^{e_2}$ their quotient $(-1)^{s}ign.s.B^{e}$ is computed as follows:

$$sign = sign_1 \; xor \; sign_2, \; s = s_1/s_2, \; e = e_1 - e_2 \tag{D.9}$$

The value of s belongs to the interval $1/B < s \leq B - ulp$, and could be smaller than 1. If that is the case, that is if $s = s_1/s_2 < 1$; then $s_1 < s_2$, $s_1 \leq s_2 - ulp$, $s_1/s_2 < 1 - ulp/s_2 < 1 - ulp/B$, and $1/B < s < 1 - ulp/B$ Then (normalization) substitute s by $s.B$, and e by $e - 1$. The new value of s satisfies $1 < s < B - ulp$. It remains to round the significand.

D.3.2 Rounding schemes

Given a real number x and a floating-point representation system, the following situations could happen: $|x| < s_{min}.B^{e_{min}}$, that is, an underflow situation, $|x| > s_{max}.B^{e_{min}}$, that is, an overflow situation, $|x| = s.B^e$, where $e_{min} \leq e \leq e_{max}$ and $s_{min} \leq s \leq s_{max}$,

In the third case, either s is a multiple of ulp, in which case a rounding operation is not necessary, or it is included between two multiples $s^{'}$ and $s^{"}$ of ulp:

$s^{'} < s < s^{"}$.

The rounding operation associates to s either $s^{'}$ or $s^{"}$, according to some rounding strategy. The most common are the following ones:

- The truncation method (also called round toward 0 or chopping) is accomplished by dropping the extra digits, i.e. round$(s) = s^{'}$ if s is positive, round$(-s) = s^{"}$ if s is negative.
- The round toward plus infinity is defined by round$(s) = s^{"}$.
- The round toward minus infinity is defined by round$(s) = s^{'}$.
- The round to nearest method associates s with the closest value, that is, if $s < s^{'} + ulp/2$, round$(s) = s^{'}$, and if $s > s^{'} + ulp/2$, round$(s) = s^{"}$.

  If the distances to $s^{'}$ and $s^{"}$ are the same, that is, if $s = s^{'} + ulp/2$, there are several options.
  beginitemize

- round$(s) = s^{'}$;
- round$(s) = s^{"}$;

- round(s) = s_0 if s is positive, round(s) = $s^{"}$ if s is negative. It is the round to nearest, ties to zero scheme.
- round(s) = $s^{"}$ if s is positive, round(s) = $s^{'}$if s is negative. It is the round to nearest, ties away from zero scheme.
- round(s) = $s^{'}$ if $s^{'}$ is an even multiple of ulp, round(s) = $s^{"}$ if $s^{"}$ is an even multiple of ulp. It is the default scheme in the IEEE 754 standard.
- round(s) = $s^{'}$ if $s^{'}$ is an odd multiple of ulp, round(s) = $s^{"}$ if $s^{"}$ is an odd multiple of ulp.

The preceding schemes (round to the nearest) produce the smallest absolute error, and the two last (tie to even, tie to odd) also produce the smallest average absolute error (unbiased or 0-bias representation systems). Assume now that the exact result of an operation, after normalization, is

$s = 1.s_{-1}.s_{-2}.s_{-3}....s_{-p}|s_{-(p+1)}.s_{-(p+2)}.s_{-(p+3)}\cdots$

where ulp is equal to B^{-p} (the — symbol indicates the separation between the digit which corresponds to the ulp and the following). Whatever the chosen rounding scheme, it is not necessary to have previously computed all the digits $s_{-(p+1)}s_{-(p+2)}...$, it is sufficient to know whether all the digits $s_{-(p+1)}s_{-(p+2)}...$ are equal to 0, or not. For example the following algorithm computes round(s) if the round to the nearest, tie to even scheme is used.

D.4 FloPoCo

FloPoCo is an open-sourcee generator of arithmetic cores (Floating-Point Cores, but not only) for FPGAs (but not only).

The purpose of the FloPoCo project is to explore the many ways in which the flexibility of the FPGA target can be exploited in the arithmetic realm, with a focus on floating-point.

FloPoCo focuses on exotic operators and exotic precisions. However it also provides basic operators(+,-,*,/ and square root) whose performance matches vendor-supplied operators while offering more flexibility.

FloPoCo is not a library of operators, but a generator of operators written in C++. It inputs operator specifications, and outputs synthesizable VHDL. This approach allows much better optimization and customization than what VHDL alone permits. In addition, FloPoCo is to our knowledge the easiest way to design complex operators with flexible pipeline.

FloPoCo supersedes FPLibrary, and is compatible with it.

There are two main differences between the format (wE=8, wF=23) and the IEEE-754 single precision format (the same holds for double).

- Exceptional cases (zeroes, infinities and Not a Number or NaN) are encoded as separate bits in FloPoCo, instead of being encoded as special exponent values in IEEE-754. This saves quite a lot of decoding/encoding logic. The main drawback of this format is when results have to be stored in memory, where they consume two more bits. However, FPGA embedded memory can accomodate 36-bit data, so adding two bits to a 32-bit IEEE-754 format is harmless as long as data resides within the FPGA.

  As a side effect, the exponent can take two more values in FloPoCo than in IEEE-754 (one for very large numbers, one for very small ones).

- FloPoCo does not support subnormal numbers. If you think you need subnormal numbers, consider adding one bit to your exponent field instead: you will thus get all the subnormals, and many more. If you are still not convinced, maybe you are right: please get in touch with us.

  Note that anyway, FloPoCo provides conversion operators from and to IEEE-754 formats (single and double precision).

Appendix E

Direct Programming Interface

E.1 Introduction

Currently system-level design and functional verification methodology based on high-level abstraction becomes more important to increase the productivity of SoC design. In system-level design and verification, hardware/software partitioning of a system through design space exploration affects the structure and performance of the final system, and the importance of the verifying functional interaction between hardware part and software part is increasing ever [65–68].

SystemC [69–71], is a design language at multiple abstraction levels, and enables the design on a higher abstraction level to proceed to a synthesizable RT-level design through a progressive refinement. SystemVerilog [72–76] is a set of extensions to the Verilog HDL that allows higher level modeling and efficient verification of large digital systems. SystemVerilog adds hardware functional verification constructs such as object-oriented programming (OOP), randomization, thread, etc.

SystemVerilog DPI (Direct Programming Interface) is an interface which can be used to interface SystemVerilog with foreign languages. These Foreign languages can be C, C++, SystemC as well as others. DPIs consist of two layers: A SystemVerilog Layer and a foreign language layer. Both layers are isolated from each other. Which programming language is actually used as the foreign language is transparent and irrelevant for the System-Verilog side of this interface. Neither the SystemVerilog compiler nor the foreign language compiler is required to analyze the source code in the other's language.

Different programming languages can be used and supported with the same intact SystemVerilog layer. For now, however, SystemVerilog defines a foreign language layer only for the C programming language.

The motivation for this interface is two-fold. The methodological requirement is that the interface should allow a heterogeneous system to be built (a design or a testbench) in which some components can be written in a language (or more languages) other than SystemVerilog, hereinafter called the foreign language. On the other hand, there is also a practical need for an easy and efficient way to connect existing code, usually written in C or C++, without the knowledge and the overhead of Programming Language Interface (PLI) or Verilog Procedural Interface (VPI).

DPI follows the principle of a black box: the specification and the implementation of a component are clearly separated, and the actual implementation is transparent to the rest of the system. Therefore, the actual programming language of the implementation is also transparent, although this standard defines only C linkage semantics. The separation between SystemVerilog code and the foreign language is based on using functions as the natural encapsulation unit in SystemVerilog. By and large, any function can be treated as a black box and implemented either in SystemVerilog or in the foreign language in a transparent way, without changing its calls.

E.2 Tasks and functions

DPI allows direct inter-language function calls between the languages on either side of the interface. Specifically, functions implemented in a foreign language can be called from SystemVerilog; such functions are referred to as imported functions. SystemVerilog functions that are to be called from a foreign code shall be specified in export declarations. DPI allows for passing SystemVerilog data between the two domains through function arguments and results. There is no intrinsic overhead in this interface.

It is also possible to perform task enables across the language boundary. Foreign code can call SystemVerilog tasks, and native Verilog code can call imported tasks. An imported task has the same semantics as a native Verilog task: it never returns a value, and it can consume simulation time.

All functions used in DPI are assumed to complete their execution instantly and consume zero simulation time, just as normal SystemVerilog functions. DPI provides no means of synchronization other than by data exchange and explicit transfer of control.

Every imported task and function needs to be declared. A declaration of an imported task or function is referred to as an import declaration. Import declarations are very similar to SystemVerilog task and function declarations. Import declarations can occur anywhere where SystemVerilog task and function definitions are permitted. An import declaration is considered to be a definition of a SystemVerilog task or function with a foreign language implementation. The same foreign task or function can be used to implement multiple SystemVerilog tasks and functions (this can be a useful way of providing differing default argument values for the same basic task or function), but a given SystemVerilog name can only be defined once per scope.

Imported task and functions can have zero or more formal input, output, and inout arguments. Imported tasks always return a void value and thus can only be used in statement context. Imported functions can return a result or be defined as void functions.

DPI is based entirely upon SystemVerilog constructs. The usage of imported functions is identical to the usage of native SystemVerilog functions. With few exceptions, imported functions and native functions are mutually exchangeable. Calls of imported functions are indistinguishable from calls of SystemVerilog functions. This facilitates ease of use and minimizes the learning curve. Similar interchangeable semantics exist between native SystemVerilog tasks and imported tasks.

E.2.1 Function Import and Export

Function Import: A function implemented in Foreign language can be used in SystemVerilog by importing it. A Foreign language function used in SystemVerilog is called imported function.

Properties of Imported Function and Task:

1. An imported function shall complete their execution instantly and consume zero simulation time. Imported task can consume time.

2. Imported function can have input, output, and inout arguments.

(a) The formal input arguments shall not be modified. If such arguments are changed within a function, the changes shall not be visible outside the function.

(b) Imported function shall not assume any initial values of formal output arguments. The initial value of output arguments is undetermined and implementation dependent.

(c) Imported function can access the initial value of a formal inout argument. Changes that the imported function makes to a formal inout argument shall be visible outside the function.

3. An imported function shall not free the memory allocated by SystemVerilog code nor expect SystemVerilog code to free memory allocated by Foreign code or (Foreign Compiler).

4. A call to an imported task can result in suspension of the currently executing thread. This occurs when an imported task calls an Exported task, and the exported task executes a delay control, event control or wait statement. Thus it is possible for an imported task to be simultaneously active in multiple execution threads.

5. An imported function or task can be equip with special properties called pure or context.

E.2.2 Pure Functions

A pure function call can be safely eliminated if its result is not needed or if the previous result for the same values of input arguments is available somehow and can be reused without needing to recalculate. Only non void functions with no output or inout arguments can be specified as pure. Functions specified as pure shall have no side effects whatsoever; their results need to depend solely on the values of their input arguments. Calls to such functions can be removed by SystemVerilog compiler optimizations or replaced with the values previously computed for the same values of the input arguments.

Specifically, a pure function is assumed not to directly or indirectly (i.e., by calling other functions) perform the following:

- Perform any file operations.
- Read or write anything in the broadest possible meaning, including input/output, environment variables, objects from the operating system or from the program or other processes, shared memory, sockets, etc.
- Access any persistent data, like global or static variables.

If a pure function does not obey the above restrictions, SystemVerilog compiler optimizations can lead to unexpected behavior, due to eliminated calls or incorrect results being used.

E.3 Data types

SystemVerilog data types are the sole data types that can cross the boundary between SystemVerilog and a foreign language in either direction (i.e., when an imported function is called from SystemVerilog code or an exported SystemVerilog function is called from a foreign code). It is not possible to import the data types or directly use the type syntax from another language. A rich subset of SystemVerilog data types is allowed for formal arguments of import and export functions, although with some restrictions and with some notational l extensions. Function result types are restricted to small values, The following SystemVerilog data types are allowed for imported function results:

- void, byte, shortint, int, longint, real, shortreal, chandle, and string.
- Scalar values of type bit and logic.

However, formal arguments of an imported function can be specified as open arrays. A formal argument is an open array when a range of one or more of its dimensions, packed or unpacked, is unspecified. An open array is like a multidimensional dynamic array formal in both packed and unpacked dimensions and is thus denoted using the same syntax as dynamic arrays, using [] to denote an open dimension. This is solely a relaxation of the argument-matching rules. An actual argument shall match the formal one regardless of the range(s) for its corresponding dimension(s), which facilitates writing generalized code that can handle SystemVerilog arrays of different sizes.

Table E.1 defines the mapping between the basic SystemVerilog data types and the corresponding C types.

TABLE E.1: Mapping data types.

SystemVerilog type	C type
byte	char
shortint	short int
int	int
longint	long long
real	double
shortreal	float
chandle	void*
string	const char*
bit	unsigned char
logic/reg	unsigned char

DPI does not add any constraints on how SystemVerilog-specific data types are actually implemented. Opti mal representation can be platform dependent. The layout of 2- or 4-state packed structures and arrays is implementation and platform dependent.

The implementation (representation and layout) of 4-state values, structures, and arrays is irrelevant for SystemVerilog semantics and can only impact the foreign side of the interface.

E.4 Importance of DPI In our work

In order to make any module ready to work, we need to compile it first and then run the emulator. After the emulator finishes its work, it dumps the contents of the memory to files, so we can use it. Each time we change the input memory contents and test another design, we have to repeat that operation. However, if you need to update the memory and catch the output in the running time that what DPI will guarantee.

We attempt to use DPI with our hardware iterative solution (Jacobi) to catch the output that the system give and save it in the software side. We succeeded in doing this on ModelSim PE student Edition 10.3 under windows and Questa simulator under linux .

For applying this simple example we chosen to do on the emulation we found that we have to use TestBench-Xpress (TBX) beside the DPI. We searched for examples which demonstrate how to use the dpi in our work, we found that to use the DPI we have to make our program consists of two layers SystemVerilog layer and the SystemC layer and to connect between them we use Makefile, And after applying this in our work it worked correctly on the windows DPI under windows. After this we began to work on DPI for Questa "under linux" the simulation worked correctly. So, we started working on the emulator using TBX. A brief overview about TBX would be given in the following section.

E.5 TBX

E.5.1 Introduction

TestBench-Xpress (TBX) is a verification accelerator. TBX uses a transaction-based test bench compilation technology to deliver orders of magnitude higher performance. TBX can be used to accelerate existing environments (including those based on SystemC and SystemVerilog), and to create transaction-based verification environments.

TBX enables efficient verification techniques, such as random stimulus, directed random stimulus, and coverage driven verification coupled to the Veloce emulation platform.

TBX is ideally suited for system tests where long periods of stimulus are required to fully test the design functionality. Examples include the following:

- Booting a real time OS on an embedded processor.
- Running a particular duration of a device protocol.
- Acquiring the pilot or initiating a call on a cell phone.
- Setting up and dropping a telecom connection.
- Simulating Transmission Control Protocol/Internet Protocol (TCP/IP) traffic over the Ethernet.

These examples share the need to reactively simulate long running complex protocols or instruction sequences to gain confidence in the correctness of a design at the RTL and system levels.

TBX enables design and verification engineers to construct environments that include streaming data, multiple asynchronous data and control channels, multiple transaction interfaces, constrained random data generation, transaction-based verification, and bus functional models.

TBX improves performance and productivity gains through technology that eliminates co-simulation performance limitations. TBX compiles transaction-based test benches written at the eXtended RTL (XRTL) SystemVerilog level into the Veloce emulator and executes it along with a High-Level Verification Language (HVL) based test bench (see page 477) running on the workstation. TBX also creates a co-modeling interface between the two. This co-modeling interface interacts at a transaction-level. This interface enables C, SystemC, or HVL models to connect efficiently to Veloce.

XRTL, combined with the SystemVerilog DPI interface to create trans-actors, provides you with a level of performance that matches the performance obtained by models written in the traditional RTL level of abstraction.

The XRTL modeling abstraction encompasses all the RTL modeling abstraction as supported by standard synthesis tools including the Mentor Graphics RTL Compiler. In addition, XRTL supports the following major special extensions:

- Clock and reset specifications
- SystemVerilog DPI function/task calls as building blocks for transactions
- Relevant non-RTL constructs:
  - Initial blocks for signal initialization
  - Named event triggers/synchronous for signaling mechanisms
  - Procedural assigns on a register from multiple processes
- Implicit state machines with initial blocks

These extensions can be simulated on any SystemVerilog simulator as is, including the clock and reset specifications. TBX is compliant with SCE-MI 1.1 and 2.0 modeling interface standards.

TBX supports designs written in SystemVerilog, RTL/VHDL, and mixed-language RTL. Testbenches and stimulus generation for TBX can be in C, SystemC, Verilog RTL, or a combination of these languages. Stimulus to the DUT can also be provided through in-circuit devices.

E.5.2 Compiler

The TBX compiler performs three major functions:

- RTL compilation
- XRTL transactor compilation
- Construction of an infrastructure to create a connection between RTL SystemVerilog and a C or SystemC environment.

TBX uses SystemVerilog DPI to provide the building blocks that create the transaction-based communication link between HVL and HDL. Based on these functions and tasks, TBX automatically generates a connection between a C or SystemC environment. This connection transparently handles inter-language calls, coordination of threads between SystemC and Verilog, argument translation, and SCE-MI callbacks. This level of automation makes it simple to create a transaction-based verification environment that combines the abstraction and performance of untimed SystemC models with the performance of emulation.

E.5.3 Modeling Constructs

The basic transaction modeling constructs in TBX are the SystemVerilog DPI exported functions/tasks and imported functions/tasks. They all illustrated before.

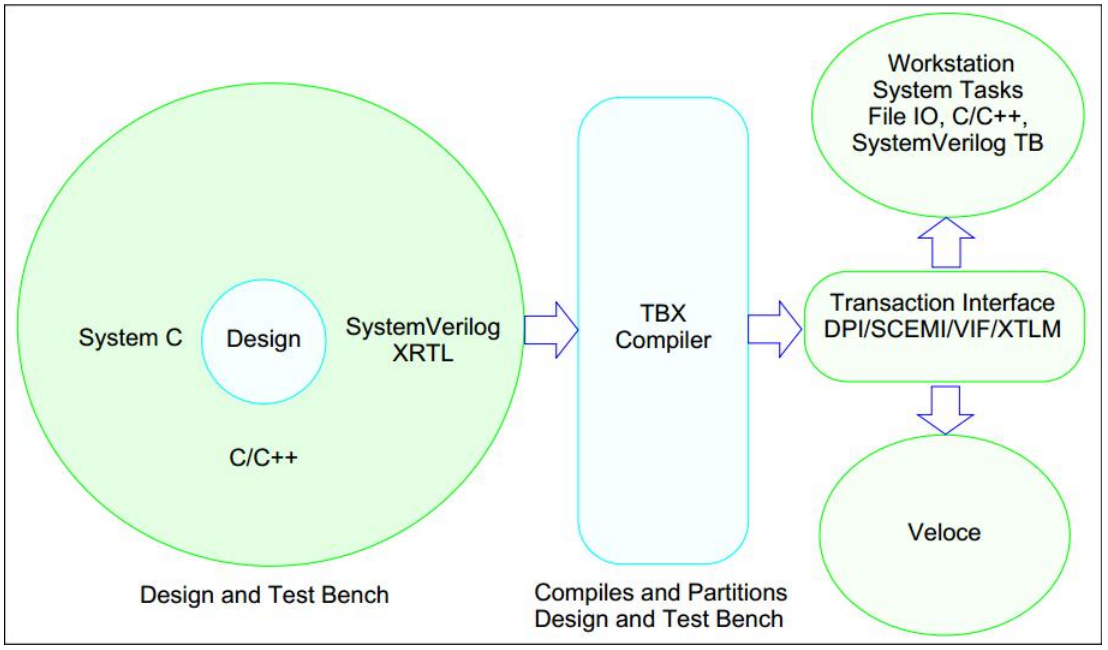

FIGURE E.1: TBX Compile and Runtime Views.

E.5.4 Transaction-Based Interface for C Testbenches

In transaction-based testbenches, the system-level tests that algorithmically generate high-level test data (transactions) can be written in SystemC, C, and C++. The transaction processing part of the test bench (transactors) that applies the transaction to the DUT can be written more naturally in XRTL SystemVerilog RTL. This is because interaction with the DUT requires timing, which is often awkward to model in C.

The direct C testbench interface is available by coupling the C testbench with the XRTL transactors through the HDL-to-C and C-to-HDL function call mechanism.

Transaction-based testbenches enable the following:

- Separation of untimed (algorithmic) test generation from the timed DUT model.
- Standardization of the coupling interface between untimed and timed parts.
- Use of a higher level of the verification methodology.
- Assertions for interface and protocol verification.
- Constrained test generation for intelligent tests.
- Use of reusable verification IP (for example, Ethernet, PCI).

E.5.5 SystemVerilog Testbenches

Besides C and SystemC models using DPI, TBX also supports SystemVerilog testbenches. These testbenches could be run on the workstation using any standard high level simulator. TBX is tightly integrated with the Questa simulator, which already supports C, C++, SystemC, and SystemVerilog. With Questa as your HVL simulator, any of these languages or their combination could be used to model the test bench (HVL). The only precondition is the communication between the HVL and HDL must be at a transaction level. Following some basic rules of modeling, modern verification methodologies could be natively accelerated using TBX.

Most modern SystemVerilog testbenches use virtual interfaces for connection between the dynamic test bench and the static DUT. In the context of co-emulation, the same concept is used. You can define functions or tasks in a synthesizable SystemVerilog interface. These tasks could be called from the HVL by using the virtual interface handle pointing to it. Along the same lines, the interface could contain a pointer to a SystemVerilog class. Using this pointer, the HDL could call functions defined in that class. Just like DPI, TBX will natively support the aforementioned modeling style for SystemVerilog test benches. In addition, TBX also ships an accelerated version of TLM libraries, known as XTLM. This library could also be used for direct communication between non-synthesizable HVL and synthesizable HDL.

Bibliography

[1] G. D. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods.* Oxford University Press, Oxford, 1978.

[2] G. Strang and G. Fix. *An analysis of the finite element method: Englewood Cliffs.* Prentice Hall, 1973.

[3] R. LeVeque. *Finite volume methods for hyperbolic problems.* Cambridge University Press, Cambridge, 2002.

[4] P. K. Banerjee. *Boundary element methods in engineering.* McGraw-Hill, London, 1994.

[5] G. R. Liu. *Mesh Free Methods.* CRC Press, Singapore, 2002.

[6] D. Gottlieb and Orzag S. *Numerical Analysis of Spectral Methods: Theory and Applications.* SIAM, Philadelphia, 1977.

[7] A. Saad. *Iterative methods for sparse linear systems.* SIAM, Philadelphia, 2003.

[8] *Veloce Emulator.* Mentor Graphics, http://www.mentor.com/products/fv/emulation-systems/.

[9] Ling Zhuo and K. Prasanna. Viktor. Sparse matrix-vector multiplication on fpgas. In *13th International Symposium on Field Programmable Gate Arrays*, pages 63–74. ACM/SIGDA, 2005.

[10] Micheal deLorimier and Andre DeHon. Floating-point sparse matrix-vector multiply for fpgas. In *13th International Symposium on Field Programmable Gate Arrays*, pages 75–85. ACM/SIGDA, 2005.

[11] R. Morris Gerald and K. Prasanna Viktor. *Sparse matrix computations on reconfigurable hardware.* Computer, 40(3), March, 2007.

[12] *SPARSKIT*. http://www-users.cs.umn.edu/ saad/software/SPARSKIT/sparskit.html.

[13] A. R. Lopes and G. A. Constantinides. *A high throughput FPGA-based floating point conjugate gradient implementation*. 2008.

[14] Ling Zhuo and K. Prasanna Viktor. High-performance and parameterized matrix factorization on fpgas. In *International Conference on Field Programmable Logic and Applications*, pages 1–6, 2007.

[15] Vikash Daga, Gokul Govindu, Sridhar Gangadharpalli, V. Sridhar, and K. Prasanna Viktor. Efficient floating-point based block lu decomposition on fpgas. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2004.

[16] *AMD Core Math Library*. AMD.

[17] V. G. Veselago. *The electrodynamics of substances with simultaneously negative values of epslon and mu*, volume 10. Soviet Phys Usp, 1968.

[18] R. W. Ziolkowski. *Wave propagation in media having negative permittivity and permeability*, volume 64. Physical Review E, 2001.

[19] N. Engheta and RW. Ziolkowski. A positive future for double negative metamaterials. In *IEEE Trans Microwave Theory Tech*, volume 53, pages 1535–1556, 2005.

[20] J. L. Young and R. O. Nelson. A summary and systematic analysis of fdtd algorithms for linearly dispersive media. In *IEEE Antennas and Propagation Magazine*, volume 43, pages 61–77, 2001.

[21] M. F. Su, I. El-Kady, D.A. Bader, and S.Y. Lin. A novel fdtd application featuring openmp-mpi hybrid parallelization. In *International Conference on Parallel Processing (ICPP)*, volume 33rd, pages 373–379, Canada, 2004.

[22] Jichun Li and Yunqing Huang. *Time-Domain Finite Element Methods for Maxwell's Equations in Metamaterials*. Springer Series in Computational Mathematics, 2013.

[23] MathWorks. http://www.mathworks.com/help/matlab/ref/mldivide.html.

[24] D.W. Zinng and T.T. Chisholm. Runge-kutta methods for linear ordinary differential equations. In *Applied Numerical Mathematics*, volume 31, pages 227–238, 1999.

[25] W.E. Boyse, D.R. Lynch, K.D. Paulsen, and G.N. Minerbo. Nodal-based finite-element modeling of maxwell's equations. In *Antennas and Propagation, IEEE Transactions on*, volume 40(6), pages 642–651, 1992.

[26] V.G. Veselago. Electrodynamics of substances with simultaneously negative values of sigma and mu. *Sov. Phys. Usp.*, 10:504–514, 1968.

[27] D.R. Smith, W.J. Padilla, D.C. Vier, S.C. Nemat-Nasser, and S. Schultz. Composite medium with simultaneously negative permeability and permittivity. *Phys. Rev. Lett.*, 84:4187–4187, 2000.

[28] J. Shen, T. Tang, and L.-L. Wang. *Spectral Methods: Algorithms, Analysis and Applications*. Springer, New York, 2011.

[29] J.B. Pendry. Negative refraction makes a perfect lens. *Phys. Rev. Lett.*, 85:3966–3969, 2000.

[30] L. Solymar and E. Shamonina. *Waves in Meta-materials*. Oxford University Press, Oxford, 2009.

[31] R.W. Ziolkowski and A. Erentok. Metamaterial-based efficient electrically small antennas. In *IEEE Trans. Antennas Propag.*, volume AP-54(7), pages 2113–2130, 2006.

[32] A. Alu, E. Bilotti, N. Engheta, and L. Vegni. Sub-wavelength, compact, resonant patch antennas loaded with meta-materials. In *IEEE Trans. Antennas Propag.*, volume AP-55(1), pages 13–25, 2007.

[33] E. Bilotti, A. Alu, and L. Vegni. Design of miniaturized patch antennas with mu negative loading. In *IEEE Trans. Antennas Propag.*, volume AP-56(6), pages 1640–1647, 2008.

[34] A. Alu, E. Bilotti, N. Engheta, and L. Vegni. A conformal omni-directional sub-wavelength meta-material leaky-wave antenna. In *IEEE Trans. Antennas Propag.*, volume AP-55(6), pages 1698–1708, 2007.

[35] S.A. Cummer, B. I. Popa, D. Schurig, D.R. Smith, and J. Pendry. Full-wave simulations of electromagnetic cloaking structures. *Phys. Rev.*, 74, 2006.

[36] D. Schurig, J.J. Mock, B.J. Justice, S.A. Cummer, J.B. Pendry, A.F.S. Starr, and D.R. Smith. Metamaterial electromagnetic cloak at microwave frequencies. *Sciencemag*, 314:977–980, 2006.

[37] R. W. Ziolkowski. Pulsed and cw gaussian beam interactions with double negative metamaterial slabs. *Opt. Express 11*, 11:662–681, 2003.

[38] R. W. Ziolkowski. Wave propagation in media having negative permittivity and permeability. *Phys. Rev. E*, 64:056625, 2001.

[39] J.B. Pendry, A.J. Holden, W.J. Stewart, and I. Youngs. Extremely low frequency plasmons in metallic meso structures. *Phys. Rev. Lett.*, 76:4773–4776, 1996.

[40] J. Li and A. Wood. Finite element analysis for wave propagation in double negative metamaterials. *J. Sci. Comput.*, 32:263–286, 2007.

[41] R.A. Shelby, D.R. Smith, S.C. Nemat-Nasser, and S. Schultz. Microwave transmission through a two-dimensional, isotropic, left-handed metamaterial. *Appl. Phys. Lett.*, 78:489–491, 2001.

[42] D.R. Smith and N. Kroll. Negative refractive index in left-handed materials. *Phys. Rev. Lett.*, 85:2933–2936, 2000.

[43] J.B. Pendry, A.J. Holden, D.J. Robbins, and W.J. Stewart. Magnetism from conductors and enhanced nonlinear phenomena. *IEEE Trans. Microw. Theory Tech*, 47:2075–2084, 1999.

[44] S.N. Galyamin and A.V. Tyukhtin. Electromagnetic field of a moving charge in the presence of a left-handed medium. *Phys. Rev. B*, 81(23):235134, 2010.

[45] A. Bossavit. *Computational Electromagnetism.* Academic, San Diego, 1998.

[46] R.F. Harrington. *Field Computation by Moment Methods.* Wiley-IEEE, Hoboken, 1993.

[47] A. Buffa, M. Costabel, and C. Schwabs. Boundary element methods for maxwell's equations on nonsmooth domains. In *Numer. Math.*, 92, pages 679–710, 2002.

[48] A. Buffa, R. Hiptmair, T. von Petersdorff, and C. Schwab. Boundary element methods for maxwell's equations on lipschitz domains. In *Numer. Math.*, 95, pages 459–485, 2003.

[49] E.T. Chung and B. Engquist. Convergene analysis of fully discrete finite volume methods maxwell's equations in nonhomogenous media. In *SIAM J. Numer. Anal.*, 43, pages 303–317, 2005.

[50] E.T. Chung, Q. Du, and J. Zou. Convergene analysis of fully discrete finite volume methods for maxwell's equations in non-homogenous media. In *SIAM J. Numer. Anal.*, 41, pages 37–63, 2003.

[51] D.A. Kopriva, S.L. Woodruff, and M.Y. Hussaini. Computation of electromagnetic scattering with a non-conforming discontinuous spectral element method. In *Int. J. Numer. Mech. Eng.*, 53, pages 105–122, 2002.

[52] J.-H. Lee, T. Xiao, and Q.H. Liu. A 3-d spectral-element method using mixed-order curl conforming vector basis functions for electromagnetic fields. In *IEEE Trans. Microw. Theory Tech.*, 54, pages 437–444, 2006.

[53] L.N. Trefethen. *Spectral Methods in MATLAB*. SIAM, Philadelphia, 2001.

[54] K.S. Yee. Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media. In *IEEE Trans. Antennas Propag.*, 14, pages 302–307, 1966.

[55] P.P. Silvester and R.L. Ferrari. *Finite Elements for Electrical Engineers*. Cambridge University Press, 3 edition, London, 1996.

[56] J. Jin. *The Finite Element Method in Electromagnetics*. Wiley-IEEE, 2 edition, Hoboken, 2002.

[57] Y. Hao and R. Mittra. *FDTD Modeling of Metamaterials: Theory and Applications*. Artech House Publishers, Boston, 2008.

[58] A. J. Davies. *The finite element method: a first approach*. Oxford University Press, Oxford, 1980.

[59] T. J. R. Hughes, I. Levit, and J. Winget. An element-by-element solution algorithm for problems of structural and solid mechanics. In *Computer Methods in Applied Mechanics and Engineering*, volume 36, page 241, 1983.

[60] G. F. Carey, E. Barragy, R. McLay, and M. Sharma. Element-by-element vector and parallel computations. In *Communications in Applied Numerical Methods*, volume 4, pages 299–307, 1988.

[61] William W. Hager. *Applied Numerical Linear Algebra.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[62] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers.* SIAM, Philadelphia, PA, 1998.

[63] E. W. Weisstein. *Gaussian Elimination.m.* MathWorld-A Wolfram Web Resource., 2010.

[64] J. D. Hoffman. *Numerical Methods for Engineers and Scientists.* McGraw-Hill, Inc., Singapore, 1992.

[65] Jason R. Andrews. *Co-Verification of Hardware and Software for ARM SoC Design.* Elsevier Inc., 2005.

[66] Ando Ki. *SoC Design and Verification:Methodologies and Environments.* Hongreung Science, 2008.

[67] Yongjoo Kim, Kyuseok Kim, Youngsoo Shin, Taekyoon Ahn, Wonyong Sung, Kiyoung Choi, and Soonhoi Ha. An integrated hardware-software cosimulation environment for heterogeneous systems prototyping. In *ASP-DAC*, pages 101–106, 1995.

[68] S. Chikada, S. Honda, H. Tomiyama, and H. Takada. Cosimulation of itron-based embedded software with systemc. In *HLDVT*, pages 71–76, 2005.

[69] David C. Black and Jack Donovan. *SystemC:From the Ground Up.* Eklectic Ally, Inc., 2004.

[70] Thorsten Grotker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC.* Kluwer Academic Publishers, 2002.

[71] *SystemC Language Reference Manual,TestBench-Xpress User Guide*, volume Software Version 2.4.3. http://www.systemc.org, October, 2012.

[72] Stuart Sutherland, Simon Davidmann, and Peter Flake. *SystemVerilog for Design (2nd Edition): A Guide to Using SystemVerilog for Hardware Design and Modeling*, volume 2nd edition. Springer, 2006.

[73] Chris Spear. *SystemVerilog for Verification : A Guide to Learning the Testbench Language Features*, volume 2nd edition. Springer, 2008.

[74] Accellera. *SystemVerilog 3.1 Language Reference Manual: Accellera's Extensions to Verilog.* Accellera Organization, Inc., Napa, California, 2004.

[75] Stuart Sutherland. *SystemVerilog, ModelSim, and You.* Mentor User2User, 2004.

[76] Stuart Sutherland. *Integrating SystemC Models with Verilog and SystemCVerilog Models Using the SystemVerilog Direct Programming Interface.* SNUG Boston, 2004.