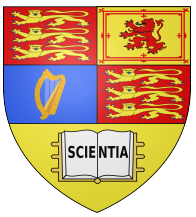


Imperial College London

Department of Computing

Hardware Acceleration of Power System Simulation

by

Yumeng Yang(yy1712)

Supervised by Prof. Wayne Luk

Submitted in partial fulfilment of the requirements for the MSc Degree in Advanced Computing of Imperial College London

September 6, 2013

Abstract

Real-time dynamics simulation of large-scale power systems is a computational challenge because of the need to solve a large set of stiff, nonlinear differential-algebraic equations. The main bottleneck in these simulations is the solution of the linear system during each nonlinear iteration of Newtons method.The need for faster, and accurate, power system dynamics simulation (or transient stability analysis) has been a primary focus of the power system community in recent years.

FPGAs have become an attractive choice for scientifis computing. This project is about exploring how the huge computational power and memory optimizations of FPGA based hardware accelerators can be used in the dynamic simulation of power systems.

Issues are threefold.

The study begins with the available power system simulation model, which deals with relatively simple structure and data size (a 4 machines, and 11 buses study system). Firstly,we present an optimised version of the simulation in a compiled language (C language) and demonstrate performance gains of approximately 500 times faster than the original run time using SIMULINK as the simulation environment.

In this paper, we also propose two high performance design for LU decomposition, a key kernel in the power system simulation applications. And a parallelism design for solving the nonlinear differential-algebraic equations among a number of machines in the power system.

Although the experiments targeting Maxeler systems show that our FPGA-based design can not improve the time efficiency from the C application of the study system. We builds the relationship between the speedup of simulation and the data size of the power system which indicates that our acceleration design can give a significant acceleration to larger-scale larger power systems.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor.Wayne Luk, for his continuous guidance, support and enthusiasm throughout the development of this project.

I would also like to thank Dr. Thomas Chua for his valuable ideas and for helping me with all my difficulties throughout the project.

As well, I would like to thank my beloved family for always believing in me and being by my side, even from far away.

Finally, I would like to thank my professors and my friends at Imperial College, without whom this past year would not have been half as much enlightening, or fun.

Contents

Chapter 1

Introduction

1.1 Problem Motivation

The need for power system dynamic analysis has grown significantly in recent years. This is due largely to the desure to utilize transmission networks for more flexible interchange transactions.[29]

Power system analysis is intensive in computational terms [32]. In fact, the power industry and the associated academic research are requiring complex developments in high performance computing tools, such as parallel computers, efficient compilers, graphic interfaces and algorithms including artificial intelligence [10]. Power system dynamic simulation is one of these problems needing a special treatment to reduce time and memory requirements. The dynamic simulation is present in design, planning, operation and control stages of power systems and have been largely used for testing methods, eigenvalue analysis and optimal control.[23]

In the engineering applications, it is frequently desirable to make many response simulations to calculate, for example, the effects of different fault locations and types, initial power system operating states and in design studies, different network, machine and control-system characteristics. However, the volume of computation imposes very severe constrain to such studies. For a large system, thousands of equations must be solved and each case can take an hour of CPU time on a large modem computer. Hence, there is always considerable incentive to find superior calculation methods.[32]

Recent years have seen significant improvements in the application of numerical and computational methods to the problem. Also, hardware developments are continuing to reduce the cost of computation spectacularly. Unfortunately, while stability is increasingly a limiting factor in secure system operation, the simulation of system dynamic response is grossly overburdening on present-day digital computing resources. It becomes necessary to solve larger systems, with increased detail of modeling, over longer response times, more frequently.[32]

The need for faster, and accurate, power system dynamics simulation (or transient stability analysis) has been a primary focus of the power system community in recent years. [1] Therefore, the next challenge is to accelerate the simulation of such large-scale power systems. To achieve this, the use of Field-programmable gate array (FPGA) based hardware accelerators is highly commended due to the nature of the algorithms involved in the simulation that allow parallel computations. Also, in a practice power grid system, the huge amount of data that are processed can be transferred closer to the processing units, in the FPGAs huge and memories, reducing the latency of the simulation even more. Previous work has shown that FPGA-based reconfigurable computing machines can achieve order of magnitude speedups compared to microprocessors for many important computing applications [2], [6], [22]. Therefore, this project takes the attempt to discuss how and how well FPGA based hardware accelerators can be used to speed up the simulation process of a power system.

1.2 Problem Specfication

At the first stage of the project, we should first focus on a study system for analyzing the dynamic behavior of different components in the power system provided by Dr Chaudhuri form his book Robust Control in Power Systems [25]. This is a relatively simple power system simulation without the use of electric springs but using the Multiple-model adaptive control (MMAC) approach for robust control. This 4-machine, 2 -area study system model is considered as one of the benchmark models for performing studies on inter-area oscillation because of its realistic structure and availability of system parameters [15], [17].

As a block diagram environment for multi-domain simulation and Model-based design, SIMULINK has been used to analyze and design of power systems. The simulation of 4-machine, 2 -area study system is initially in the form of a SIMULINK model based on some standard approaches to modeling of several power system components in Dr Chaudhuri's book, and an overview of these models is given as the background knowlegde in Chapter 2. However, although the SIMULINK tool provides a graphical user interface for building visualized model as block diagrams, It can't avoid the restrictions on its simulation performance. For example, large images and complex graphics take a long time to load and render. As a result, masked blocks that contain images might make your model less responsive. [20] In this project we will explore the benefits of accelerating from using Field Programming Gate Arrays (FPGAs) to power system simulation. There are many variations of FPGA designs. We will use the hardware and the complier by Maxeler Technologies. The implementation can be built using the programming paradigm known as data flow programming. Maxeler's dataflow system is a hybrid CUP-FPGA system. Therefore, the first step is optimise the simulation process in a complied language (we use C language) application so that it can run an CPU, and then further accelerate the slow operations by mapping the dataflow to the data flow engines(DFEs) provided by Maxeler. Specially, we should follow the design flow shown in Fig.1.1.

So the task can be divided into steps of progresses:

- Step 1: Study the Maxcompiler system, and dataflow programming technology.
- Step 2: Understanding the SIMULINK model and analyzing the program.
- Step 3: Rewrite the SIMULINK code to C code. Measure how long it takes to run the application on CPUs given a set of large datasets.
- Step 4: Identify areas of code for acceleration: a more detailed analysis provides the distribution of runtime of various parts of the application using time counter and profiling tools such as gprof, oprofile etc.
- Step 5: For the code to be accelerated, create dataflow graph, data layout and representation.
- Step 6: Optimize dataflow, data access and data representation options by the principle: maximizing regularity of computation and minimizing communication between CPU and dataflow engines.
- Step 7: use Maxcompiler to configure application for FPGA and analyze the performance.

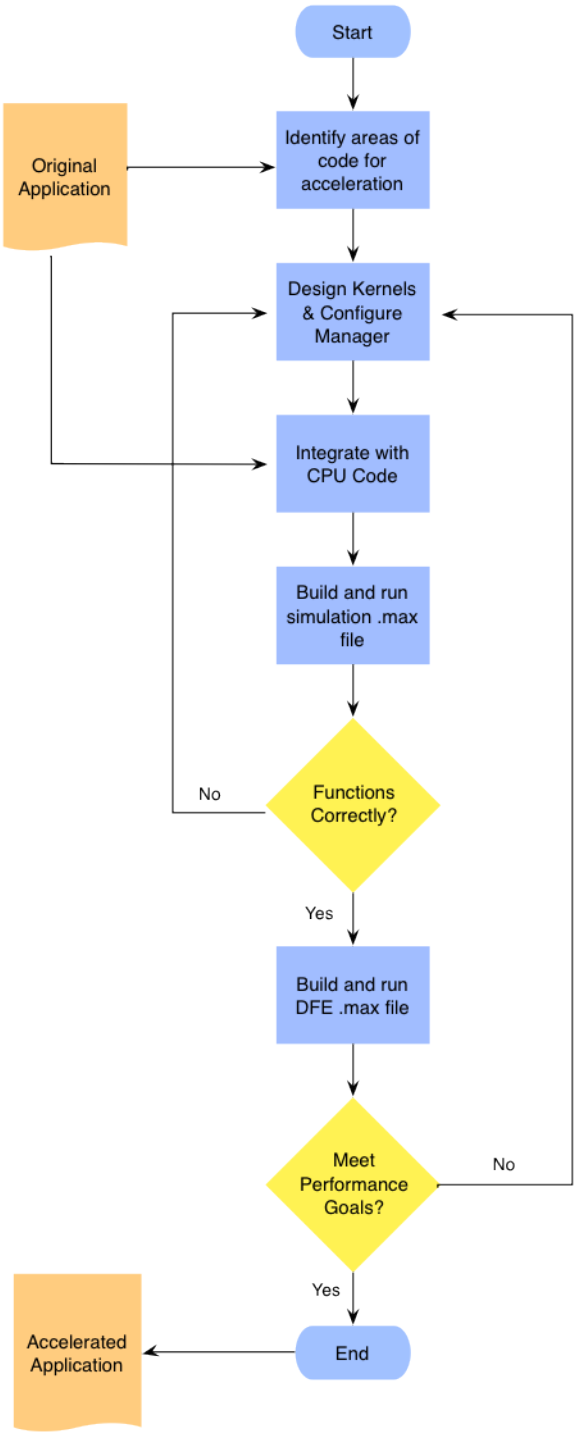


Figure 1.1: Diagram of Design Flow

1.3 Objectives & Achievements

The object of this project is to explore how the huge computational power and memory optimizations of FPGA based hardware accelerators can be used in the dynamic simulation of power systems. And the study begins with the available power system simulation model, which deals with relatively simple structure and data size.

In details, we show that to meet the milestones and targets set above we were to follow various objectives. In the case of migrating the SIMULINK model to a compiled language, we needed to first identify the performance gains we could make. Since the SIMULINK model was now serving as our

only description of the algorithms, it was in our best interests to rewrite the graphical descriptions as functions and refactor code which appeared slow or unnecessary. Once the power system simulation in SIMULINK was optimised to a complied language, we should check for correctness. Once we verified the program's outputs, we could look at performance gains and consider acceleration. For this, the Field Programmable Gate Array (FPGA) applied by the Maxeler Technologies was used. We present an evaluation of various design of the Kernels to accelerate the power system, and demonstrate our achievement of acceleration.

The following objectives were key in directing this project to its end and producing our contributions:

- Analyised the simulation model in SIMULINK (chapter 3). Proposed a method to trace the sorted order of different subsystems and blocks in SIMULINK. This was paramount in providing a base to begin migrating to a compiled language.
- An optimised version of the simulation in C language (chapter 4). We maximise the performance gains available without hardware acceleration by 500 times' faster than SIMULINK. We re-implemented everything from scratch so that have concluded an efficient scheme of converting SIMULINK diagrams to C(C++) language.
- An accelerated version of the LU decomposition solver used in the power system simulation. (chapter 5) And implementation of different FPGA kernel designs for pipeline. During this process, we successfully solved the problem of handling complex numbers in maxeler.
- Analysed the performance of the designed kernels(chapter 6). A relationship between the speedup of simulation and the data size of the power system is built which indicated that our acceleration design can give a significant acceleration to larger-scale larger power systems.

1.4 Report Structure

In this chapter we looked at an overview of the main motivations, objectives and steps for our project. The rest of this report is divided into 7 chapters and organized as follows.

Chapter 2 introduces the background knowledge needed for understanding the project. The first part of chapter 2 illustrates the idea of power system simulation and an dynamic simulation model of the power system is introduced which support the primary mathematical principle underlaying our SIMULINK power system model. The second part of chapter 2 gives an introduction to the SIMULINK tool while the third part presents the principles of FPGAs and Hardware acceleration technology and also the FPGA programming platform we used in the project: a programming model based on data-flow programming provided by Maxeler Technologies call MaxCompiler.

Chapter 3 is about how I analysed the given Simulink Model in order to extract the algorithms for a complied language, I will show this from four aspects. 1. An overview of the simulink model, the structure. 2. What are the inputs and desired outputs of the model and where are they from. 3. How does the operarions linked together: the execution order of the blocks,their dependency. 4. What are the specific operations involved, and how to understand them, the mathematical meaning.

Chapter 4 gives a detailed illustration of an optimised application in C language that does the same thing with the simulation system. Depending on the analysis in chapter 2, we describe the ideas we used to re-implement each aspect of the simulation scheme that corresponding to each section of chapter 2.

Chapter 5 starts from showing the profiling result of the C application introduced in chapter 4

which determined the area of code to be accelerated by FPGA. And then, the design and implementation details using the Maxeler Compiler are represented.

Chapter 6 gives the noteworthy experimental results for both the C application, and the accelerated design of FPGA computing blocks. And by comparing them, we analysis the practicability of our hardware design.

Chapter 7 is the conclusion of the report and lists some further work the author would like to try later.

Chapter 2

Background

2.1 Power System Simulation

An electric power system is a network of electrical components used to supply, transmit and use electric power. An example of an electric power system is the network that supplies a region's homes and industry with power , this power system is known as the power grid and can be broadly divided into the **generators** that supply the power, the **transmission system** that carries the power from the generating centres to the load centres and the **distribution system** that feeds the power to nearby homes and industries. Smaller power systems are also found in industry, hospitals, commercial buildings and homes.

Simulation is historically one the principal tools used in the design of power system controls.The conventional power-system stability study computes the system response to a sequence of large disturbances, usually a network short circuit, followed by protective branch-switching operations. The process is a direct simulation in the time domain of duration varying between say 1 s and 20 min or m.ore. Different components of the power system have their greatest influences on stability at different of the response, and the system modeling (simulation) reflects this fact.

In power systems, the primary sources of electrical energy are the synchronous generators. The problem of power system stability is primarily to keep the interconnected synchronous machines in synchronism [17]. The stability is also dependent on several other components such as the speed governors, excitation systems of the generators, the loads, the FACTS devices etc. Therefore, an understanding of their characteristics and modeling of their performance are of fundamental importance for stability studies and control design. The general approach to modelling of several power system components is quite standard, and a quick overview of these models is given in next section.

2.2 Models of Different Components in Power System

Accurate modelling of the generators and their excitation systems is of fundamental importance for studying the dynamic behavior of power systems. Besides generators and excitation systems, other components such as the dynamic loads (e.g. induction motor type), controllable devices (e.g. thyristor controlled series capacitor (TCSC), power system stabilizer (PSS)), prime-movers etc. need to be modelled as well. The dynamic behavior of these devices is generally described through a set of differential equations. The power flow in the network is represented by a set of algebraic equations. This gives rise to a set of differential-algebraic equations (DAE) describing the power system behavior. Different types of model have been reported in the literature for each of the power system components depending upon their specific application [17], [29]. In this section, the relevant equations governing the dynamic behavior of only the specific types of models used in this project is described. The IEEE recommended practice regarding d-q axis orientation [5] of a synchronous generator is used. This results in a negative d axis component of stator current for an overexcited

generator delivering power to the system.

2.2.1 Generators

All the generators are represented by a sub-transient model [29], [17] with four equivalent coils on the rotor. Besides the field coil, there is one equivalent damper coil in the direct axis and two in the quadrature axis. The mechanical input power to the generator is assumed to be constant during the disturbances such as a 3-phase fault, obviating the need for modelling the primemover. The differential equations governing the sub-transient dynamic behavior of the i^{th} generator is given by:

$$\frac{d\delta_i}{dt} = \omega_i - \omega_s \tag{2.1}$$

$$\begin{aligned}\frac{d\omega_i}{dt} =& \frac{\omega_s}{2H}[T_{mi} - D(\omega_i - \omega_s) - \frac{(X_{di}^{''} - X_{lsi})}{(X_{di}^{'} - X_{lsi})}E_{qi}^{'}I_{qi} - \frac{(X_{di}^{'} - X_{di}^{''})}{(X_{di}^{'} - X_{lsi})}\psi_{1di}I_{qi} - \\ & \frac{(X_{qi}^{''} - X_{lsi})}{(X_{qi}^{'} - X_{lsi})} + \frac{(X_{qi}^{'} - X_{qi}^{''})}{(X_{qi}^{'} - X_{lsi})}\psi_{2qi}I_{di} + (X_{qi}^{''} - X_{di}^{''})I_{qi}I_{di}]\end{aligned} \tag{2.2}$$

$$\frac{dE_{qi}^{'}}{dt} = \frac{1}{T_{doi}^{'}}[-E_{qi}^{'} - (X_{di} - X_{di}^{'})\{-I_{di} - \frac{(X_{di}^{'} - X_{di}^{''})}{(X_{di}^{'} - X_{lsi})^2}(\psi_{1di} - (X_{di}^{'} - X_{lsi})I_{di} - E_{qi}^{'})\} + E_{fdi}] \tag{2.3}$$

$$\frac{dE_{di}^{'}}{dt} = \frac{1}{T_{qoi}^{'}}[-E_{di}^{'} - (X_{qi} - X_{qi}^{'})\{I_{qi} - \frac{(X_{qi}^{'} - X_{qi}^{''})}{(X_{qi}^{'} - X_{lsi})^2}(\psi_{2qi} + (X_{qi}^{'} - X_{lsi})I_{qi} - E_{di}^{'})\}] \tag{2.4}$$

$$\frac{d\psi_{1di}}{dt} = \frac{1}{T_{doi}^{''}}[-\psi_{1di} + E_{qi}^{'} + (X_{di}^{'} - X_{lsi})I_{di}] \tag{2.5}$$

$$\frac{d\psi_{2qi}}{dt} = \frac{1}{T_{qoi}^{''}}[\psi_{2qi} + E_{di}^{'} + (X_{qi}^{'} - X_{lsi})I_{qi}] \tag{2.6}$$

for $i = 1, 2, ..., m$,where,
m : total number of generators,
δ_i : generator rotor angle,
ω_i : rotor angular speed,
$E_{qi}^{'}$: transient emf due to field flux-linkage,
$E_{di}^{'}$: transient emf due to flux-linkage in q-axis damper coil,
ψ_{1di} : sub-transient emf due to flux-linkage in d-axis damper,
ψ_{1qi} : sub-transient emf due to flux-linkage in q-axis damper,
I_{di} : d-axis component of stator current,
I_{qi} : q-axis component of stator current,
$X_{di}, X_{di}^{'}, X_{di}^{''}$: synchronous, transient and sub-transient reactances, respectively along d-axis,
$X_{qi}, X_{qi}^{'}, X_{qi}^{''}$: synchronous, transient and sub-transient reactances, respectively along q-axis,
$T_{do}^{''}, T_{do}^{''}$: d-axis open-circuit transient and sub-transient time constants, respectively
$T_{qo}^{''}, T_{qo}^{''}$: q-axis open-circuit transient and sub-transient time constants, respectively

The stator transients are generally much faster compared to the swing dynamics. Hence, for stability studies, the stator quantities are assumed to be related to the terminal bus quantities through algebraic equations rather than state equations. The stator algebraic equations are given by:

$$V_i cos(\delta_i - \theta_i) - \frac{(X_{di}^{''} - X_{lsi})}{(X_{di}^{'} - X_{lsi})}E_{qi}^{'} - \frac{(X_{di}^{'} - X_{di}^{''})}{(X_{di}^{'} - X_{lsi})}\psi_{1di} + R_{si}I_{qi} - X_{di}^{''}I_{di} = 0 \tag{2.7}$$

$$V_i sin(\delta_i - \theta_i) + \frac{(X_{qi}^{''} - X_{lsi})}{(X_{qi}^{'} - X_{lsi})}E_{di}^{'} + \frac{(X_{qi}^{'} - X_{qi}^{''})}{(X_{qi}^{'} - X_{lsi})}\psi_{2qi} - R_{si}I_{di} - X_{qi}^{''}I_{di} = 0 \tag{2.8}$$

for $i = 1, 2, ..., m$, where,
V_i : generator terminal voltage magnitude,

θ_i : generator terminal voltage angle,
R_{si} : resistance of the armature,
X_{lsi} : armature leakage reactance.
The notation is standard as in [29]. The parameters used for the study system are given in Appendix A.

2.2.2 Excitation systems

The generators are equipped with slow excitation systems (IEEE-DC1A) to ensure adequate damping for its local modes. The rest of the generators are under manual excitation control. The differential equations governing the behavior of an IEEE-DC1A type excitation system are given by:

$$\frac{dV_{tri}}{dt} = \frac{1}{T_{ri}}[-V_{tri} + V_{ti}] \tag{2.9}$$

$$\frac{dE_{fdi}}{dt} = -\frac{1}{T_{Ei}}[K_{Ei}E_{fdi} + E_{fdi}A_{ex}e^{B_{Ex}E_{fdi}} - V_{ri}] \tag{2.10}$$

$$\frac{dV_{ri}}{dt} = \frac{1}{T_{Ai}}[\frac{K_{Ai}K_{Fi}}{T_{Fi}}R_{Fi} + K_{Ai}(V_{refi} - V_{tri}) - \frac{K_{Ai}K_{Fi}}{T_{Fi}}E_{fdi} - V_{ri}] \tag{2.11}$$

$$\frac{dR_{Fi}}{dt} = \frac{1}{T_{Fi}}[-R_{Fi} + E_{fdi}] \tag{2.12}$$

where,
E_{fdi} : field voltage,
V_{tri} : measured voltage state variable after sensor lag block,
and the rest of the notation carries their standard meaning [29].

2.2.3 Network power flow model

The network power balance equation for the i^{th} generator bus is given by:

$$V_i cos(\delta_i - \theta_i)I_{qi} - V_i sin(\delta_i - \theta_i)I_{di} - S_{pi} = 0 \tag{2.13}$$

$$-V_i sin(\delta_i - \theta_i)I_{qi} - V_i cos(\delta_i - \theta_i)I_{di} - S_{qi} = 0 \tag{2.14}$$

where,

$$S_{pi} = \sum_{k=1}^{k=n} V_i V_k[G_{ik}cos(\theta_i - \theta_k) + B_{ik}sin(\theta_i - \theta_k)] \tag{2.15}$$

$$S_{qi} = \sum_{k=1}^{k=n} V_i V_k[G_{ik}sin(\theta_i - \theta_k) - B_{ik}cos(\theta_i - \theta_k)] \tag{2.16}$$

for $i = 1, 2, ..., m$

Power balance equations for the i^{th} non-generator bus is given by:

$$P_{Li}(V_i) + \sum_{k=1}^{k=n} V_i V_k[G_{ik}cos(\theta_i - \theta_k) + B_{ik}sin(\theta_i - \theta_k)] = 0 \tag{2.17}$$

$$Q_{Li}(V_i) + \sum_{k=1}^{k=n} V_i V_k[G_{ik}sin(\theta_i - \theta_k) - B_{ik}cos(\theta_i - \theta_k)] = 0 \tag{2.18}$$

for $i = m + 1, ..., n$
where, n is the total number of buses in the system and $Y_{ik} = G_{ik} + jB_{ik}$ is the element of the i^th row and k^{th} column of the bus admittance matrix Y .

2.2.4 Thyristor conrolled series capacitor (TCSC)

A TCSC is a capacitive reactance compensator which consists of a series capacitor bank shunted by a thyristor controlled reactor (TCR) in order to provide a smooth variation in series capacitive reactance [11], [31].

The dynamic characteristics of the TCSC is assumed to be modelled by a single time constant (T_{tcsc} = 0.02 s) representing the response time of the TCSC control circuit as follows:

$$\frac{d}{dt}\Delta k_c = \frac{1}{T_{tcsc}}(-\Delta k_c + \Delta k_{c-ref} + \Delta k_{c-ss}) \tag{2.19}$$

The small-signal dynamic model is given in Fig. 2.1 where, Δk_c is the incremental change in value of k_c about the nominal value of 0.5 (50% compensation). The reference setting $\Delta k_(c-ref)$ is augmented by Δk_{c-ss} within a limit of Δk_{c-max} =0.3 and Δk_{c_min} = -0.4 in the presence of supplementary damping control.

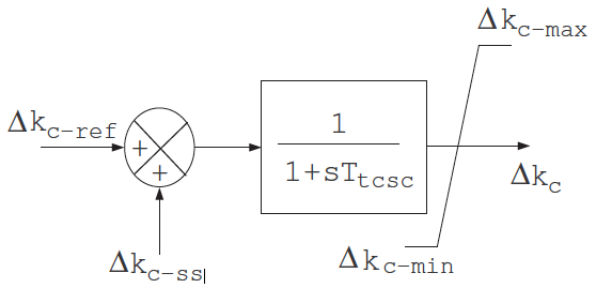


Figure 2.1: Small-signal dynamic model of TCSC

2.3 The Simulation Challenge

Engineers in the power industry face the problem that, while stability is increasingly a limiting factor in secure system operation, the simulation of system dynamic response is grossly overburdening on present-day digital computing resources. Each individual response case involves the step-by-step numerical solution in the time domain of perhaps thousands of nonlinear differential-algebraic equations, at a cost of up to several thousand dollars A high premium is thus to be placed on the use of the most efficient and reliable modern calculation techniques.

The need for faster, and accurate, power system dynamics simulation (or transient stability analysis) has been a primary focus of the power system community in recent years. This view was reiterated in the recent DOE and EPRI work- shops [9]. Indeed, more than two decades ago, real-time dynamics simulation was identified as a grand computing challenge [16]. As processor speeds were increasing, real- time dynamics simulation appeared possible in the not-too-distant future. Unfortunately, processor clock speeds saturated about a decade ago, and real-time dynamics simulation re- mains a grand computing challenge.

Dynamics simulation of a large-scale power system is computationally challenging because of the presence of a large set of stiff, nonlinear differential-algebraic equations (DAEs).The electrical power system is expressed as a set of nonlinear DAEs.The solution of the dynamic model given only one DAE needs the following:

- A numerical integration scheme to convert the differential equations in algebraic form.
- A nonlinear solution scheme to solve the resultant nonlinear algebraic equations.

- A linear solver to solve the update step at each iteration of the nonlinear solution.

Fig. 2.2 shows the wall-clock execution time of a series of dynamics simulations on a single processor for a temporary three-phase fault applied for 0.1 seconds illustrated in [1].

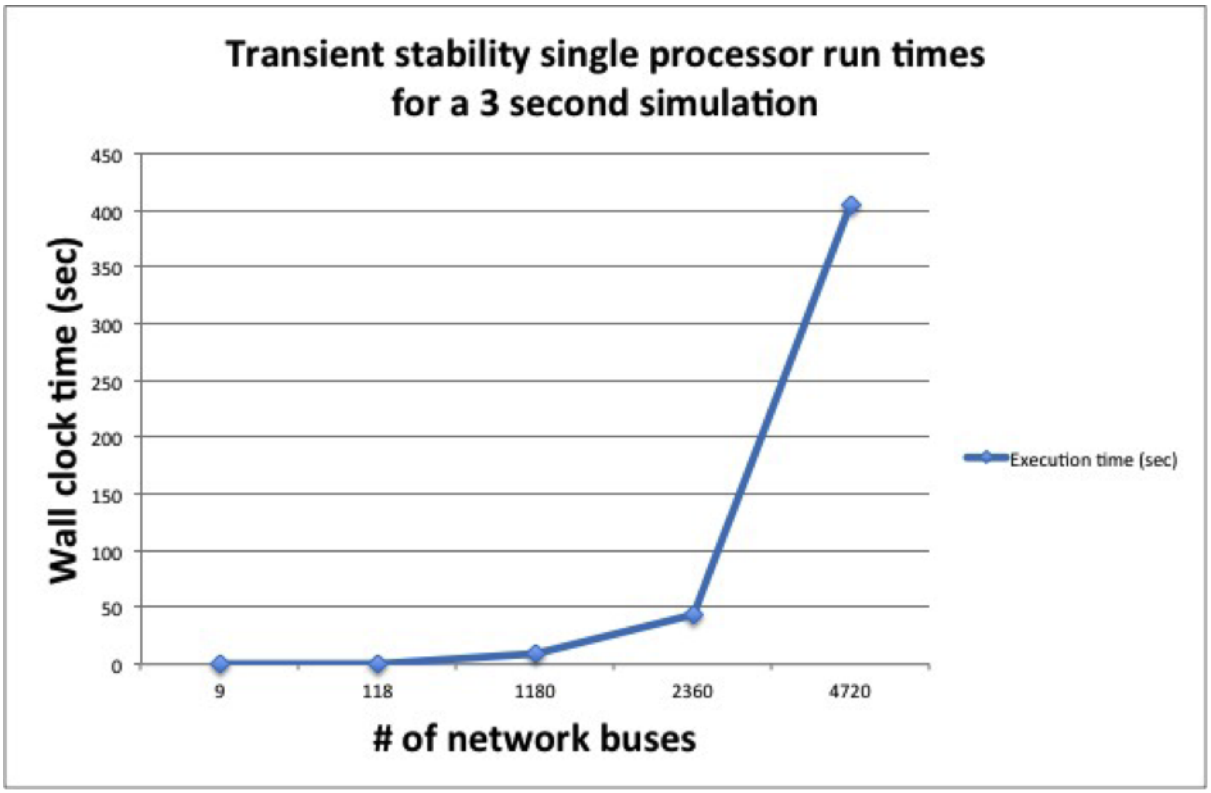


Figure 2.2: Single processor dynamic simulation execution times for a 3 second simulation period on different systems for 0.1 second three-phase balanced temporary fault.

The test cases with bus sizes greater than 1000 were obtained by duplicating the 118 bus test system 10, 20, and 40 times respectively, and connecting each 118 bus area by five randomly chosen tie lines. As system size increases, execution time grows dramatically. Thus real- time dynamics analysis of a utility or a regional operator network is an enormous computing challenge.

For example, PJM, a regional transmission organization (RTO) covering 168,500 square miles of 12 different states, monitors approximately 13,500 buses [14]. Similarly, the Electric Reliability Council of Texas (ERCOT) monitors approximately 18,000 buses [8]. High-level Eastern Interconnection models contain more than 50,000 buses. To perform dynamics simulation in real time, the simulator must compute the solution to a set of equations containing more than 150,000 variables in a few milliseconds. Because of this high computational cost, dynamics analysis is usually performed on relatively small interconnected power system models, and computation is mainly performed offline. Researchers at Pacific Northwest National Laboratory have reported that a simulation of 30 seconds of dynamic behavior of the Western Interconnection requires about 10 minutes of computation time today on an optimized single processor [13].

2.4 The SIMULINK Tool

2.4.1 What is SIMULINK

Simulink [30],[19] developed by The MathWorks, is a software package for modeling, simulating, and analyzing dynamical systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, such as signal processing, control and communication applications. It can now also be used to analyze and design of power systems.[24] During last four decades simulation of power systems have gained more importance. Recently published IEEE paper discussing different approaches to modeling protective relays and related power system events indicates a variety of possible software tools that may be used for this purpose [36]. But rather than MATLAB/SIMILINK software it is difficult to add the modeling and simulation features to teach specific protective relaying concepts that go beyond the level of detail originally provided by the software.[33]

For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. With this interface, you can draw the models just as you would with pencil and paper (or as most textbooks depict them). Simulink includes a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors. You can also customize and create your own blocks. For information on creating your own blocks, see the separate Writing S-Functions guide.

Models are hierarchical, so you can build models using both top-down and bottom-up approaches. You can view the system at a high level, then double-click on blocks to go down through the levels to see increasing levels of model detail. This approach provides insight into how a model is organized and how its parts interact.

After you define a model, you can simulate it, using a choice of integration methods, either from the Simulink menus or by entering commands in MATLABs command window. Using scopes and other display blocks, you can see the simulation results while the simulation is running. In addition, the simulation results can be put in the MATLAB workspace for post-processing and visualization.

2.4.2 Basic elements of Simulink

Models in Simulink can be thought of as executable specifications, Simulink's graphical editor is used for modeling dynamic systems with a block diagram, consisting two major classes of elements in Simulink: blocks and connections. Blocks are used to generate, modify, combine, output, and display signals. Narrows are used to transfer signals from one block to another.

Blocks

Each block represents a set of equations called block methods, which define a relationship between the blocks input signals, output signals and the state variables. In the Simulink User Guide a block is characterised by a combination of three functions.

1. A function which computes the output from the state and the input values.

2. A function which computes the next value of the discrete components of the state.

3. A function which yields the rate of change of the continuous components of the state. Blocks are frequently parameterized with constants or arithmetical expressions over constants.

Examples of simulink blocks

The block is an entity which defines a relation between its inputs and outputs. The block's functionality can vary, depending on the blocks parameters and inputs. It can also be influenced by other blocks. The number of blocks inputs and outputs can vary also. Each input and output has a dimensionality and it can be a scalar, vector or a matrix signal. We will give a representative example for most of these cases.

The Gain block (Fig. 2.3) multiplies the input by a constant value which is specified by the Gain parameter. The Gain can be a scalar or a vector.

Gain	Input	Output
5	7	35
5	[3,7]	[15,35]
[5,4]	7	[35,28]
[5,4]	[3,7]	[15,28]

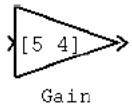


Figure 2.3: A Gain block and sample values

The Sum block (Fig. 2.4) performs addition or subtraction on its inputs. This block can add or subtract scalar or vector inputs. If the block has a single input signal, then it collapses its elements into a scalar by summing or subtracting them. The operation of the block is specified with the "List of signs" parameter, which is a list of Plus (+) and minus (-) signs, and indicates the operations to be performed on the inputs. If there are two or more inputs, then the number of + and characters must equal the number of inputs. For example, "+-+" requires three inputs and configures the block to subtract the second (middle) input from the first input, and then add the third input. All non-scalar inputs must have the same dimensions. Scalar inputs are expanded to have the same dimensions as the other inputs.

List	Input1	Input2	Output
+	[5,7]	—	12
++	[5,7]	[3,4]	[8,11]
++	[5,7]	3	[8,10]
+-	[5,7]	3	[8,4]

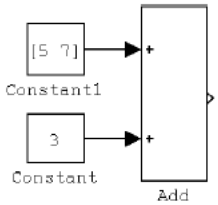


Figure 2.4: An Add block and sample computations.

The Mux block (Fig. 2.5) combines its inputs into a single vector output. An input can be a scalar or a vector signal. The Mux block's "Number of Inputs" parameter allows to specify input signal names and sizes as well as the number of inputs.

The Subsystem block. (Fig. 2.6) A subsystem is a set of blocks that have been replaced by a single block called a Subsystem block. As a model increases in size and complexity, it can be simplified by grouping blocks into subsystems. Using subsystems helps to reduce the number of blocks displayed in the model window, allows to keep functionally-related blocks together, and enables to establish a hierarchical block diagram. The whole Simulink model, composed of blocks and subsystems is placed in a single system block referred to as a root system.

The Outport block. The outport block in a subsystem represents its outputs. A signal arriving

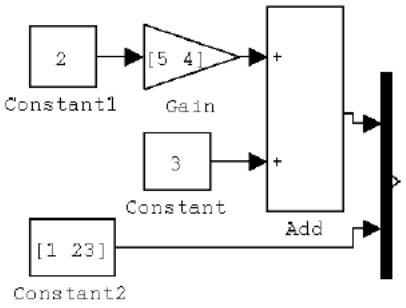


Figure 2.5: A Mux block. The number of inputs parameter is 2

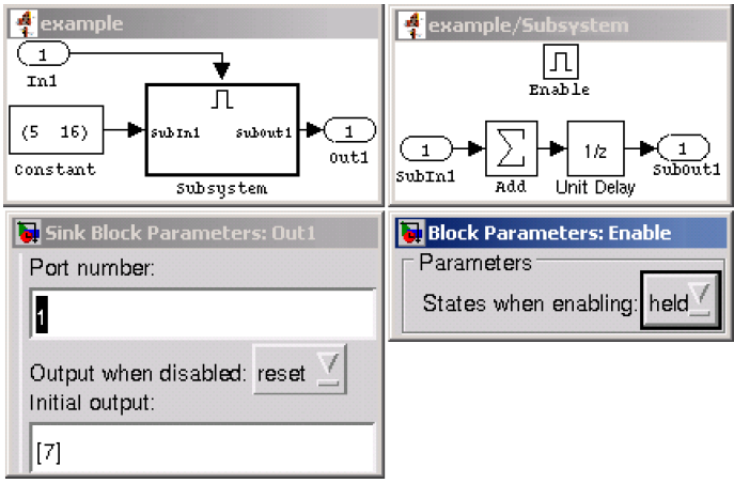


Figure 2.6: An Enabled subsystem. The subsystems control input is associated with the Enable block. This subsystem will output the value 7 when disabled and its states will be held at their previous values.

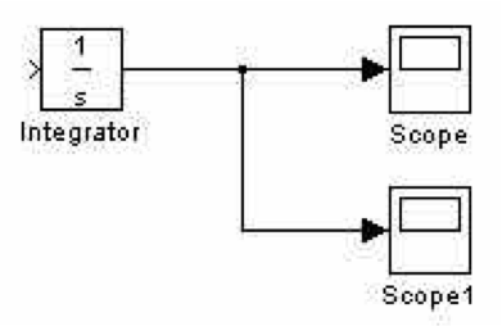
to an Outport block in a subsystem flows out of the associated output port on that Subsystem block. For example in Fig. 2.5), the Outport block SubOut1 on the right represents the output port SubOut1 of the subsystem on the left.

Block Types

Simulink blocks can be categorized. This categorization was described in [7] and [8]. The generation of the verification condition in tvs is organized according to these categories. Simulink blocks can be categorized into virtual and non-virtual. **A virtual block** is one that defines only the interconnections of signals and has no memory element (For example Mux, Outport and Subsystem blocks). Such a block has no explicit representation in the generated code. Non-virtual blocks normally represent some mathematical operation on their input values (for example the Gain and Sum blocks). **A non-virtual block** can be represented in the generated code by a variable or its operation can be propagated.

Connections(Lines)

Lines transmit signals in the direction indicated by the arrow. Lines must always transmit signals from the output terminal of one block to the input terminal of another block. One exception to this is that a line can tap off of another line. This sends the original signal to each of two (or more) destination blocks, as shown below:



Lines can never inject a signal into another line; lines must be combined through the use of a block such as a summing junction.

Signal

Signals are what kind of information is carried by the connections in a diagram. According to the Using Simulink manual, a wide range of signal attributes can be specified, including signal name, data type (e.g., 16-bit or 32-bit integer, double, single, uint8, uint16, uint32), numeric type (real or complex), and dimensionality (e.g., one-dimensional or multidimensional array), and introduces the type Boolean.

2.4.3 Textual Representation of the Model

There are two textual representations of the model:

- The model.mdl file, which is written in a Mathworks propriety markup language. The file contains the graphical model description and assignments to parameters of template blocks. Simulink allows not specifying blocks parameters that can be derived, i.e., propagated from other blocks automatically (for example input signals types. Therefore, in the model file not all parameters are contained explicitly for all blocks. Blocks parameters can be defined in terms of Matlab workspace variables, but those values are also not included in the model.mdl file. Thus, the model.mdl file is tightly coupled with the MATLAB environment.
- The model.rtw file, which is derived from model.mdl during code generation. It is an intermediate representation created by removing graphical information from model.mdl, and evaluating parameters of blocks. Although it contains more information, its format is not described in Mathworks' documentation and is difficult to understand by reverse engineering.

2.5 Hardware Acceleration & FPGAs

2.5.1 Hardware Accelerators

In computing, hardware acceleration is the use of computer hardware to perform some function faster than is possible in software running on the general-purpose CPU. Examples of hardware acceleration include blitting acceleration functionality in graphics processing units (GPUs) and instructions for complex operations in CPUs. Many hardware accelerators are built on top of field-programmable gate array chips. Also, machines such as the gaming machine Sony-PlayStation can be used as hardware accelerators.

Normally, processors are sequential, and instructions are executed one by one. Conventional processors are hitting the limits of attainable clock frequencies and thus, future significant increases in performance must come from exploiting parallelism. [26] Various techniques are used to improve

performance; hardware acceleration is one of them. The main difference between hardware and software is concurrency, allowing hardware to be much faster than software. The hardware that performs the acceleration, when in a separate unit from the CPU, is referred to as a hardware accelerator. [35]

As shown in [18] The overarching goal of hardware acceleration is to increase the speed at which data can be processed by using custom hardware specifically designed to implement a specific routine. By doing so, the software can be sped up in two ways. The first advantage is that the CPU is able to process other data, while the computation necessary for the accelerated routine is offloaded to the coprocessor. This makes the computation appear to be essentially free to the processor. The only time the processor must spend on the computation is the time that it takes to set up the coprocessor to begin its calculation and the time it takes to receive the results. As long as the overhead necessary to communicate with coprocessor is less costly than performing the actual computation, a speedup is realized.

The second potential gain is realized when the hardware accelerator is structured in such a way that it is able to calculate the result a faster than the software. In this case, the communications overhead, and the runtime of the hardware accelerator must be less than the time the software implementation of the same algorithm would take. If this condition is met the algorithm will be accelerated whether or not the processor is processing data in parallel with the coprocessor.

2.5.2 Field-programmable gate array (FPGA)

FPGAs are reconfigurable hardware chips that can be reprogrammed to implement varied combinational and sequential logic.For the purpose of this project we will focus on FPGA based acceleration as the significant improvements in FPGA size, speed, and storage capacity have made them excellently potential as hardware accelerators for a wide class of applications. Significant speedups achieved using FPGAs to accelerate cryptography, sparse matrix-vector multiplication, Viterbi decoding, and financial computing systems have been reported in recent literature [21],[34],[3]. As an example, the study described in [34] demonstrated a 2 times speedup for floating-point sparse matrix-vector multiplication (a computational kernel at the heart of many scientific computing applications) implemented on a Virtex II FPGA compared to the fastest single processor system at the time and even greater speedups for multi- FPGA systems (compared to multi-processor systems)[12]. And an FPGA accelerated system can be 31-37 times faster than an equivalently sized conventional machine, and consume 1/39 of the power. [7]

An FPGA is made up of an array of programmable logic blocks (programming cells). These logic blocks are connected by reconfigurable sets of wires as shown in Fig. 2.7, which allow for signals to be routed according to the definition of the circuit. [18] Modern FPGAs allow the user to reconfigure these circuits many times each second, making FPGAs fully programmable and general purpose. The price of reconfigurability is a 10x slower dynamic clock frequency compared to todays state-of-the-art Pentium and Opteron processors. This slower clock frequency is compensated for by support for massive fine-grained parallelism. [26]

2.5.3 Maxeler Platform

In our project we will use a programming model based on data-flow provided by Maxeler Technologies which is entirely driven by Java. The user specifies kernels, which are statically scheduled, pipelined data-paths, and a manager that controls the routing of streaming data between multiple kernels and off-chip connections. Kernels use arbitrary precision floating/fixed point types and our compiler takes care of type conversions. [7]]

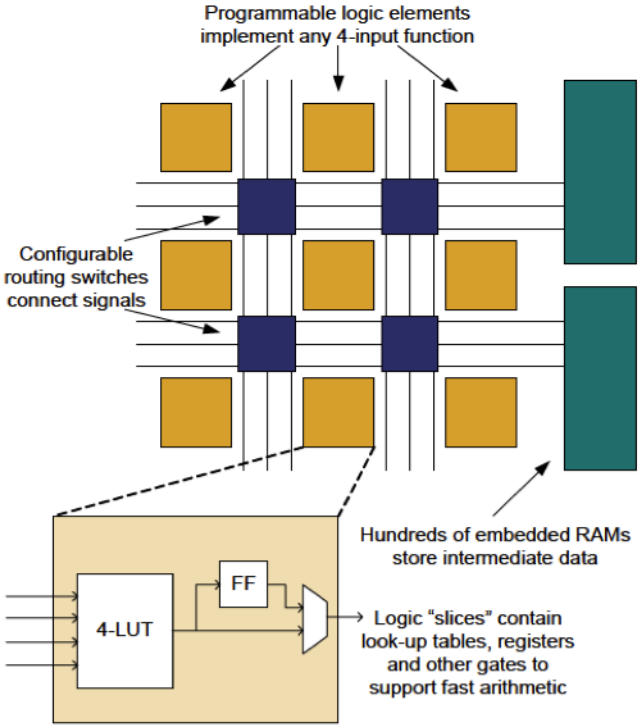


Figure 2.7: Simplified view inside an FPGA

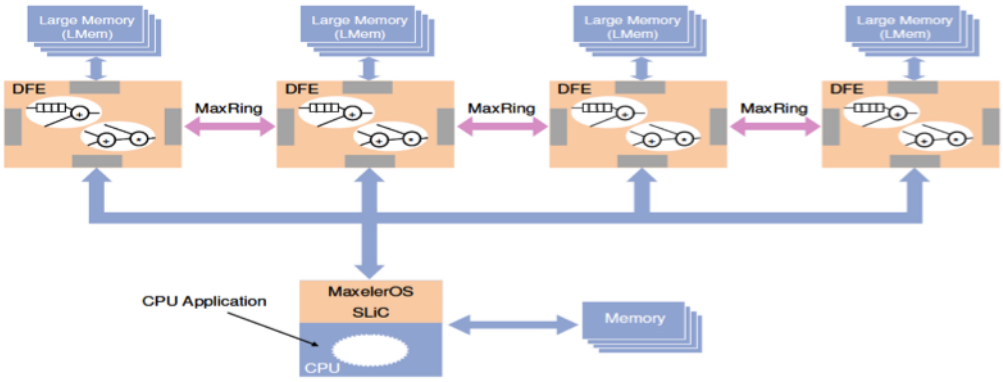


Figure 2.8: Maxelers dataflow system architecture

Maxelers dataflow system architecture as shown in Fig. 2.8 is a hybrid CPU-FPGA system. The system comprises CPU, dataflow engines (DFE) as well as from communication and memory systems. DFE can implement multiple kernels, which perform computation as data flows between the CPU, DFE and its associated memories. The DFE has two types of memory: FMem (Fast Memory) which can store several megabytes of data on-chip with terabytes/second of access bandwidth and LMem (Large Memory) which can store many gigabytes of data off-chip. In a Maxeler dataflow supercomputing system, multiple dataflow engines are connected together via high-bandwidth MaxRing interconnect. The MaxRing interconnect allows applications to scale linearly with multiple DFEs in the system while supporting full overlap of communication and computation.

Form the tutorials provide by Maxeler Techologies and some papers [27] I have got an overview of Maxelers dataflow programming technology: In the Maxeler platform, we develop the kernel

code by the hardware compiler called MaxCompiler. The Maxeler platform utilizes Java as a meta-programing language for implementation. The MaxCompiler IDE isa modified version of the Eclipse IDE, with a modified version of Java serving as the descriptive code for the dataflow. Using a meta-program that describes the structure of dataflow as an input, MaxCompiler generates the .max file which contains the FPGA bitstream. Also, data exchange between a host CPU and a dataflow engine is performed using a run-time library API called MaxCompilerRT.

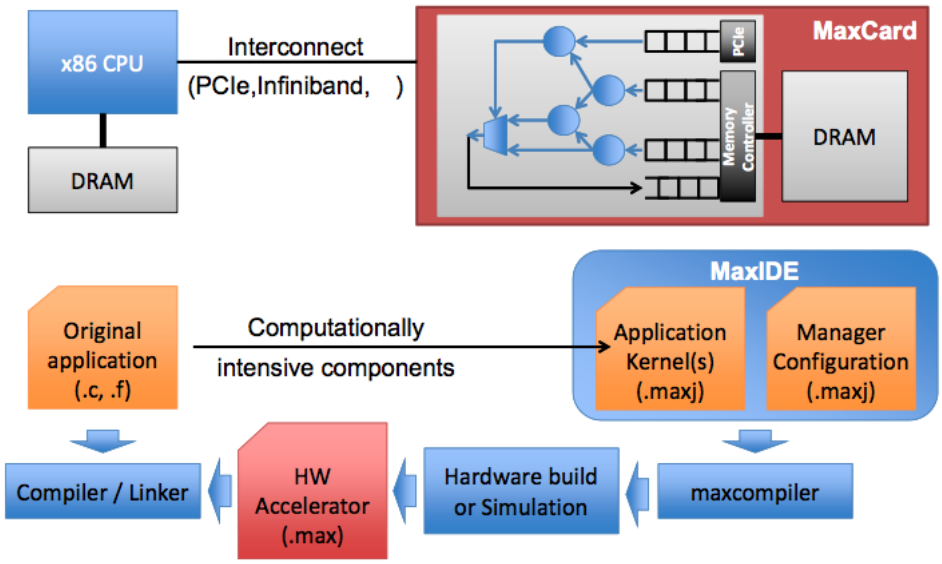


Figure 2.9: Maxeler MaxCompiler Development Flow

In the Maxeler system, data streaming and execution are performed as follows. The dataflow engine is initialized and the .max file generated by MaxCompiler is loaded from the CPU to the engine, configuring the FPGA. After input data are streamed from CPU memory onto the FPGA chip, these are processed by forwarding intermediate results from one functional unit to another where the results are needed, without ever being written to the off-chip memory until the chain of processing is complete. After all processing on the dataflow engine is completed, the final output data are transferred to CPU memory.

Chapter 3

Analysing the Power System Simulation

The original power system simulation system (a 4-machine, 2 -area study system) is initially built as a SIMULINK model. In order to apply the FPGA accelerators to this simulation problem, we therefore take first steps in converting the Simulink diagram to a software application. Before we can present our conversions, we must first understand how the power simulation is modeled in the Simulinks grammar and extract the underlying updating algorithms from the diagram representation.

In this chapter we present how we analysed the Power System simulation model to extract the dynamic behavior of the power system from a software optimisations point of view that allow performance gains in SIMULINK and a departure from the SIMULINKs graphical user interface to a compiled language.

In detail, we will firstly introduce the power system that the simulation built on, then from the Simulation program we analysed the properties of the program: the requirements of the program, the data flow and the control flow of the program.Section 3.1 describes the power system being simulated. Section 3.2 covers an detailed analysis of the power system simulation in the form of a SIMULINK model. And section 3.3 gives a brief illustration of the performance of the SIMULINK model from the efficiency prospect.

3.1 The Study System

A simple 4-machine, 2-area study system, shown in Fig. 3.1, is considered first. This system is one of the benchmark models for performing studies on inter-area oscillation because of its realistic structure and the availability of system parameters [2, 8] in the public domain. All four generators are represented using the sub-transient model with DC (IEEE-DC1A type) excitation system, as described in Chapter 2. Power flow and dynamic data for the system can be found in [8].

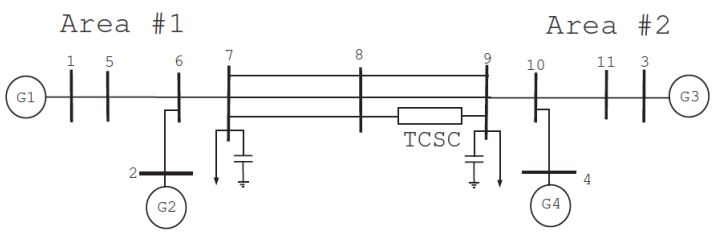


Figure 3.1: 4-machine, 2-area study system with a TCSC

The system consists of two areas connected by a weak transmission corridor. To enhance the transfer capability of the corridor, a TCSC is installed in one of the lines connecting buses #8 and #9. From the transfer capacity enhancement point of view, the percentage compensation K_c of the TCSC is set to 10%. A maximum and minimum limit of 50% and 1%, respectively, is imposed on the dynamic variation of kc. Under normal operating conditions, the power flow from Area #1 to Area # 2 is 400 MW.

3.2 Analysing the SIMULINK Model

As mentioned before, Simulink is a block diagram environment for multi-domain simulation and Model-based design. It supports system-level design, simulation, and continuous test and verification of embedded systems. It can now also be used to analyze and design of power systems. The original simulation system is just initially written as a SIMULINK model. The sium4mac.mdl file, containing the graphical model description and assignments to parameters of template blocks, is the main entry point of our analysis. Form the Model, we can know:

- What is simulation process, the time.
- What are the inputs and desired outputs of the model and where are they from (How are they initialized and updated).
- How do the operations linked together: the execution order of the blocks, their dependency.
- What are the specific operations involved, and how to understand them: their mathematical meaning (control flow)

3.2.1 An overview of the simulation process

From the sium4mac.mdl, we get a power system shown in Figure:

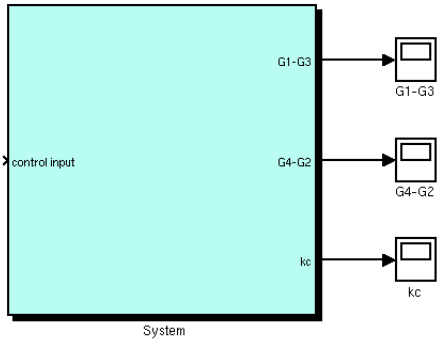


Figure 3.2: An Overview of the Model in SIMULINK

As we can see from the Fig.3.2, the root system gets one input data called control input which is now defined as a constant 0 and produce outputs of the simulation as the phase differences between generator 1 and generator 3 (G1-G3) and G2-G4 respectively. These phase difference can be used to measure whether the difference between the voltage and the current in an AC circuit are in phase. In addition, only when the voltage between the hots is zero, namely, in phase, can two generators connected together. As we can see, scope blocks are used to display signals generated during simulation. An example result of 20 seconds' simulation of the phase difference between generator 2 and generator 4 is shown in Fig. 3.3, the signal fluctuated regularly and gradually leveled off.

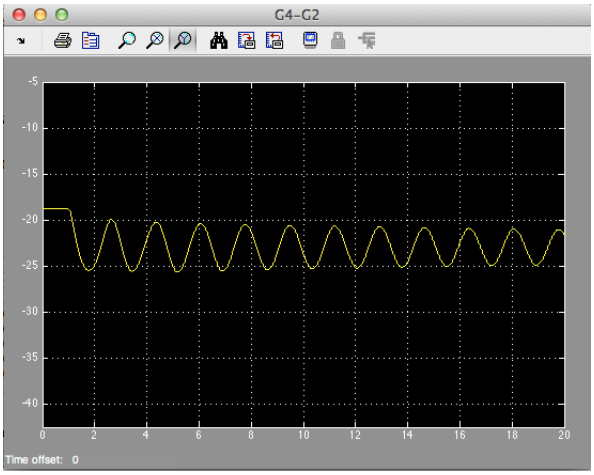


Figure 3.3: The Result of 20s' simulation of G2-G4

As a simulation system in SIMULINK, the input data streaming through the model, doing operations behind it (more detailed in the subsystems) once each *step size* and update their states, which will be used in next step. So the total number of iterations is:

$$Updatetimes = \frac{Simulationtime}{Stepsize} \tag{3.1}$$

Where *step size* is set to 1e-3 by default, and Simulation Time is the total length of time the simulation runs (the period between start time and end time), these values can be set from the **solver pane** of the graphical user interface: **Simulation →Configure Parameters→Solver**. The solver plan specifies not only the simulation start and stop time but also the solver configuration for the simulation. As we mentioned before, a solver computes a dynamic system's states at successive time steps over a specified time span, using information provided by the model, which is the fundamental mechanism of the simulation process.

The Simulink product provides an extensive library of solvers (e.g., the *Dormand-Prince* method, the *Runge-Kutta* method, the *Bogacki-shampine* method, the *Heuns* method and the Euler method) each of which determines the time of the next simulation step and applies a numerical method to solve the set of ordinary differential equations (ODEs) that represent the model. In the process of solving this initial value problem, the solver also satisfies the accuracy requirements that we specify. The default setting is the ode3 solver (*Bogacki-shampine*).

3.2.2 Subsystems

To see how is the power system structured and how is the data updated, we can just trace down hierarchical block diagram level by level.

As shown in Fig. 3.4, the power system is configured by 5 main subsystem blocks, and each of these subsystems also contains its own subsystems, totally 19 subsystems:

- TSCS system
- The '*machine network interface*' system
- The excitation system
  1. DC Exciter_R_f
  2. DC Exciter_V_r

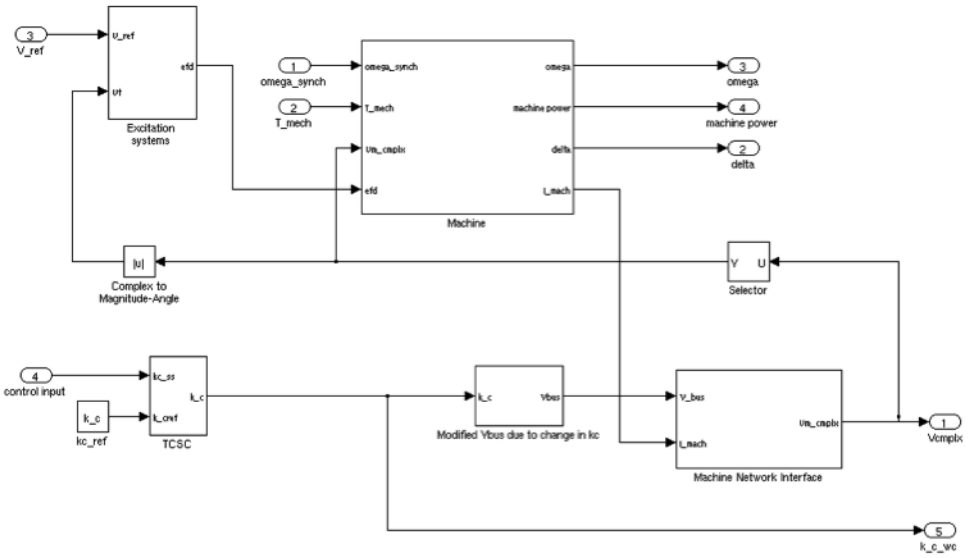


Figure 3.4: block diagram of the power system model

3. DC Exciter_E_df
4. DC Exciter_V_tr

- The machines (generators)

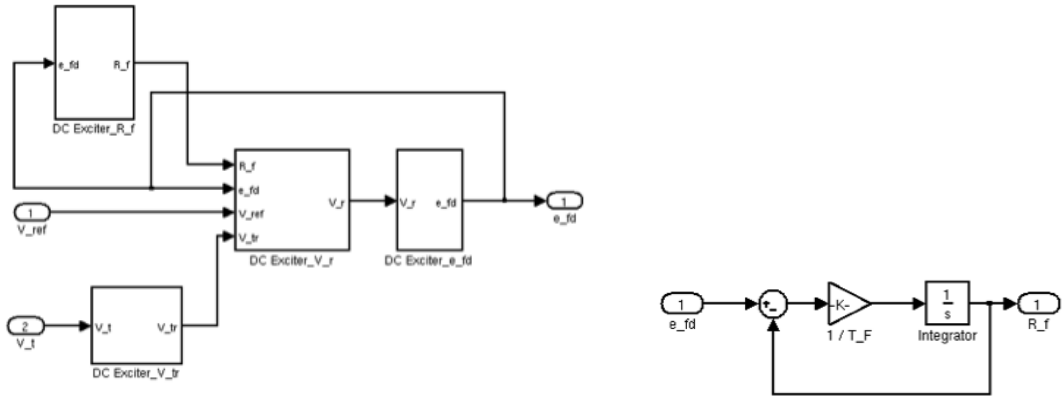
1. Machine state equations: Machine omega, Machine delta, Machine psid, Machine psiq, Machine eq_dash, Machine ed_dash.
2. Machine T_elec
3. Machine currents

- The Modified Ybus due to change in k_c block: Some intermediate processing blocks used for data (Matrix) transformation.

1. Switching logic
2. Multiport switch
3. Modified TCSC line admittance
4. Modified Ybus entries due to change in kc
5. Ybus modification

Amount these subsystems, some of them are for flow control, namely, the '*Switching logic*' and '*Multiport switch*' subsystems. They together determine what $Ybus$ data to be used in each iteration. And the '*Ybus modification*' system performs an intermediate operation on the $Ybus$ data which gets the new bus data used for each iterations calculations by changing some entries in the $Ybus$ data matrix. The remaining subsystems are all for updating the desire parameters: *Delta, Eq_ dash, Ed_ dash, Psid, Psiq, Omega, I_mach, T_q, T_d, Vm_cmplx, Efd, Rf, Vr, Vtr, T_ elec, Kc.* In other words, each of these subsystems encapsulates the dynamic updating rule for one of these 16 parameters (the '*Machine currents*' updates 3 of these parameters together: *I_mach, I_q* and *I_d*).

The dynamic behavior of these devices (subsystems) is generally described through a set of differential equations. The power flow in the network is represented by a set of algebraic equations. This gives rise to a set of differential-algebraic equations (DAEs) describing the power system behavior.



(a) Inside the excitation system (b) Inside the Exciter_R_f block

Figure 3.5: The Excitation Subsystem

The relevant equations governing the dynamic behavior of only the specific types of models used in this study system are given in Chapter 2.

By tracing down the subsystems, we can extract the operation blocks and their mathematical representations to form the algorithms used to simulate the power flow and finally they can be matched with the differential-algebraic equations given. For example, the Excitation system and can be further extended and finally matched the governing equations for the IEEE-DC1A type excitation system given by:

$$\frac{dV_{tri}}{dt} = \frac{1}{T_{ri}}[-V_{tri} + V_{ti}] \tag{3.2}$$

$$\frac{dE_{fdi}}{dt} = -\frac{1}{T_{Ei}}[K_{Ei}E_{fdi} + E_{fdi}A_{ex}e^{B_{Ex}E_{fdi}} - V_{ri}] \tag{3.3}$$

$$\frac{dV_{ri}}{dt} = \frac{1}{T_{Ai}}[\frac{K_{Ai}K_{Fi}}{T_{Fi}}R_{Fi} + K_{Ai}(V_{refi} - V_{tri}) - \frac{K_{Ai}K_{Fi}}{T_{Fi}}E_{fdi} - V_{ri}] \tag{3.4}$$

$$\frac{dR_{Fi}}{dt} = \frac{1}{T_{Fi}}[-R_{Fi} + E_{fdi}] \tag{3.5}$$

As we can see form Fig. 3.5 below, block DC Exciter_R_f, DC Exciter_V_r, DC Exciter_E_df and DC Exciter_V_tr corresponding to the R_Fi,V_ri,E_fdi,V_tri respectively in the formulas. And Figure is extended form the Exciter_R_f block, it correspongs to the equation (3.2.12) which calculates the R_{Fi} by an integration operation.

3.2.3 Data Initialisation

Obviously, the input data of this power system is never only the '*control input*'. However it is only one that can be specified by users (it is now set to 0). For each block, the input data stream in and give rise to outputs after calculations. The input of some blocks depends on the output of some blocks. For each time step, the current states of the input data are used to update the dynamic system's states, which will be used in next step. In this way, the variables in the system keeping changing their states like a state machine. For example, the DC Exciter_V_r block of the excitation system takes R_F,V_{ref},E_{fd} and V_{tr} as inputs and output the value of V_r which is the input of the DC Exciter_E_df block. At the same time R_F is also the output of Exciter_R_f block, etc.

Therefore, before the simulation begins, all these parameters in the system should already have their initial states. The question is where are from? The answer can be found in the callback setting of the model properties. (Fig. 3.6)

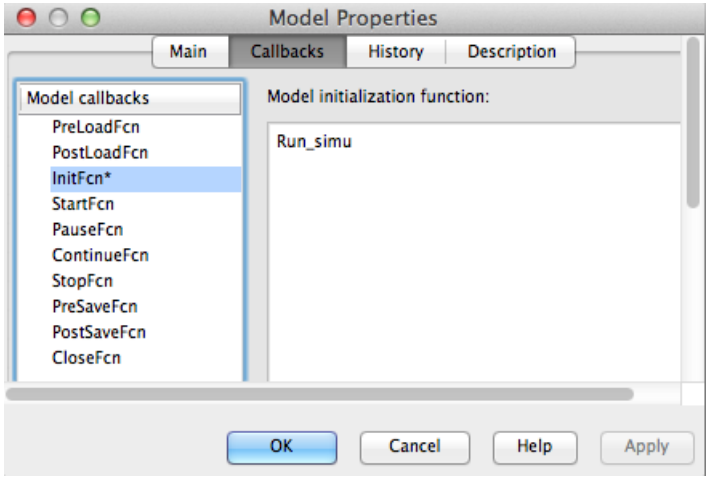


Figure 3.6: The Model Properity Pane

Callbacks are a series of user-defined commands that execute in response to a specific modeling action, such as opening a model or stopping a simulation. one can also use callbacks to execute MATLAB code. You can use model, block, or port callbacks to perform common tasks. In our system, only the initial function is specified: *Run_simu*. This function is called each time the model is initialized and is therefore, before the simulation starts. The *Run_simu.m* is a MATLAB script that calls other MATLAB scripts and functions to calculate the initial states of the input value of the Model. We can find all the variables appear in the model in these .m files and their descriptions as comments.

To be more detailed, the initialization process involves 11 MATLAB files. The *Run_simu.m* file does totally 3 things:

- Call the *init_cond.m* script: Creating the parameters in the workspace form the data files.

  1. Call the *data_kundur_mod.m* scrip which is the 4-machine 11-bus system from Kundur's book[].

     Provides the 11*10 Bus Data, 10*7 Line data, 4*17 Exciter Data, 1*7 CSC data, and 4*21 machine data. And call the *mac_sat_kundur.m* script where some operations are performed on the above data to get more new data.

  2. Converting the machine parameters to the system base.

  3. Define the parameters used for the machine, TCSC system and the exciter system. *E.g., T_do_p, T_qo_p, X_d, X_q, k_c, T_tcsc, tcsc_from_bus, tcsc_to_bus.*

  4. Give condition initialization and zero initialization of all the states.

  5. Use the bus data, line data and etc., to get the load flow solution by calling the MATLAB function *loadflow_mod*.

     This function solve the load-flow equations of power systems modified to eliminate do loops and improve the use sparse matrices and may produce a Load-Flow Study Report at the end. The algorithm is the Newton-Raphson method using the polar form of the

equations for P(real power) and Q (reactive power). So it calls other MATLAB functions: *calc, form_jac, chq_lim, ybus_mod.*

6. Use the load flow solution to get the initial values of the inputs in the model. E.g., *Theta, Ign, Temp, Delta, Omega, Igm, I_q, eq_dash, ed_dash, psid, psiq, E_dc, Efd, K_E, Rf, Vtr, Vr, T_elec, T_mech, I_curr.*

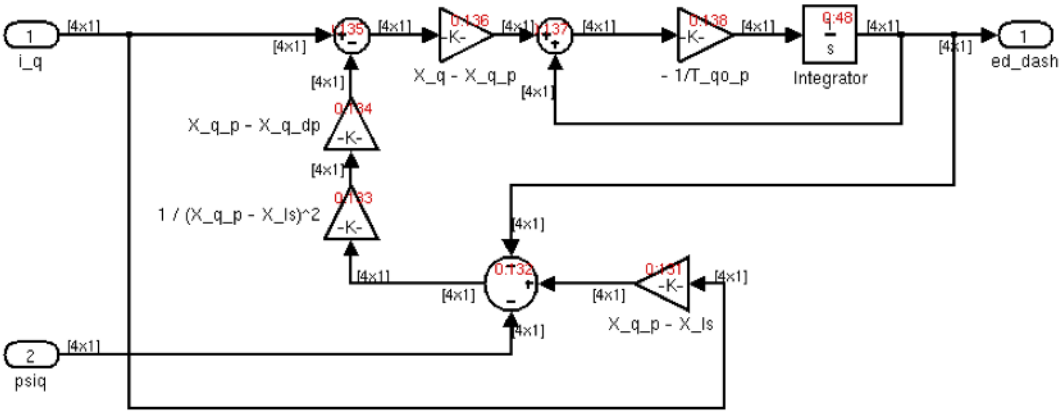
- Call the *init_sim.m* script: Declarifing the simulation parameters used in the model, such as the sampling time, pre-fault time, fault duration and the post-fault time.
- Load the bus admittance matrix Y: *Ybus_data89.mat/ Ybus_data79.mat.*

As we know that, the data initialization is before the simulation process starts, after we get the initial states of the data, how to update the states is what we should focus on in order to analysis the simulation process. We analysis the simulation process from two software aspects: the control flow and dataflow. The control flow is about how executions of the blocks are sorted according to the data dependency. From the data flow perspective we illustrate the specific operations that done on the data.

3.3 Control Flow (Execution Order of the Blocks, Data Dependency)

From a macroscopical view, we can imagine that the model works like a state machine, data stream in and out with all the blocks are executed in parallel. However, as a software implementation, there must be some kind of sort in the execution of SIMULINK.

During the updating phase of simulation, Simulink determines the order in which to invoke the block methods during simulation. This block invocation ordering is the sorted order. In order to translate the simulation process to a C code, we must find out this order. Fortunately, we can view the sorted order in the model window by selecting **Display →Blocks →Sorted Execution Order**. Simulink displays a notation in the top-right corner of each nonvirtual block and each nonvirtual subsystem. These numbers indicate the order in which the blocks execute. The first block to execute has a sorted order of 0. Here is an example in our system:



However, our system is actually complicated and the data dependency is intricate. If we want to understand the execution order only from those numbers we must understand how Simulink determines the sorted order. To ensure that the sorted order reflects data dependencies among blocks, Simulink categorizes block input ports according to the dependency of the block outputs

on the block input ports. An input port whose current value determines the current value of one of the block outputs is a direct-feedthrough port. Examples of blocks that have direct-feedthrough ports include: Gain, Product, Sum. Examples of blocks that have non-direct-feedthrough inputs: Integrator, constant, memory[20].

By studying the tutorial[20], the rules for sorting blocks can be concluded as:

If a block drives the direct-feedthrough port of another block, the block must appear in the sorted order ahead of the block that it drives. This rule ensures that the direct-feedthrough inputs to blocks are valid when Simulink invokes block methods that require current inputs. Blocks that do not have direct-feedthrough inputs can appear anywhere in the sorted order as long as they precede any direct-feedthrough blocks that they drive. Placing all blocks that do not have direct-feedthrough ports at the beginning of the sorted order satisfies this rule. This arrangement allows Simulink to ignore these blocks during the sorting process. Applying these rules results in the sorted order. Blocks without direct-feedthrough ports appear at the beginning of the list in no particular order. These blocks are followed by blocks with direct-feedthrough ports arranged such that they can supply valid inputs to the blocks which they drive.

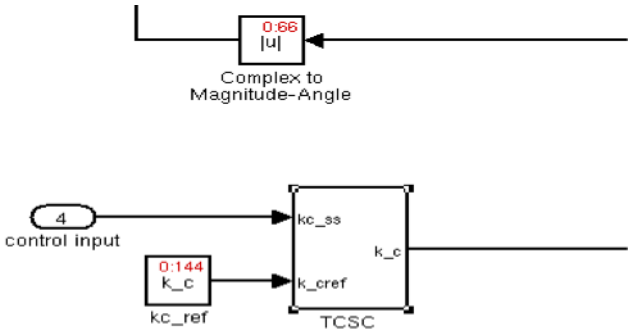


Figure 3.7

As an example in our system, The TCSC subsystem is virtual it does not have a sorted order (Fig. 3.7) and does not execute as an atomic unit. However, the blocks within the subsystem execute at the root level, so the Integrator block in the TCSC subsystem executes first. As we can see from Fig. 3.8 The Integrator block sends its output K_c to the connecting block in the upper-level model, which executes next.

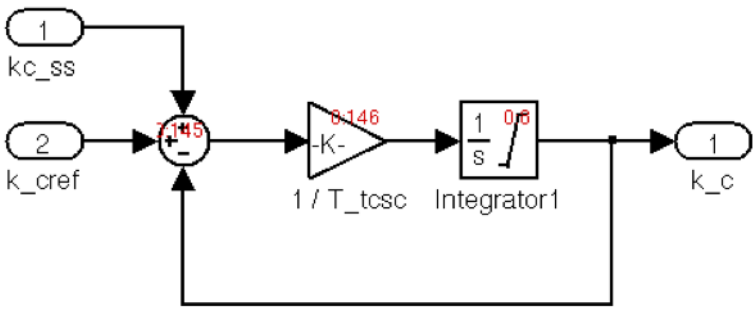


Figure 3.8: The TCSC subsystem with Block Sorted Order

By tracing the sorted order notation, we found that the Integrator block inside the *Machine Delta*

subsystem has a sorted order of 0:0, indicating that this Integrator block is the first block executed in the context of the entire model. And then is the TCSC subsystem that update the value of K_c with the Integrator block inside the TCSC subsystem has a sorted order of 0:8. In next step, the *Machine Eq_dash* subsystem is invoked etc., the author noticed that it is a good approach to find the execution order by compare the sorted order notations of the Integrator blocks in each subsystems. This is because Integrator block always executes the first in one subsystem.

Finally, the execution order of the subsystems can conclude as:

Machine delta →TSCS system →Modified Ybus due to change in K_c →Machine psid →Machine ed_dash →Machine psiq →Machine eq_dash →Machine currents (I_mach) →machine network interface →DC Exciter_E_df →DC Exciter_R_f →DC Exciter_V_r →DC Exciter_V_tr →Machine omega →Machine currents (I_q, I_d)→Machine T_elec

3.4 Dataflow (Block Operations)

Other then other software applications, Simulink provides the graphical user interface that represent all the knowledge in a form of a diagram. To convert the model to C code, we should also understand the operation blocks and extract their mathematical representations form the diagram presentation to see what calculations are performed. As we know, non-virtual blocks normally represent some mathematical operation on their input values. A set of non-virtual blocks connected together may give rise to an algorithm. Under each of the 19 subsystems, we can find the detailed operations.

In this section, some very important and not common operation blocks used in the simulation process are introduced with their underlying mathematical principles discussed.

3.4.1 Arithmetic operations

In Simulink, different operators are represented as different blocks. The most simple and common ones should be the arithmetic operators (addition, subtraction, multiplication and division). As a example, see Fig. 3.9 the *Machine T_elec* subsystem which updates the value of the time constant *T_elec* contains only the arithmetic operators.

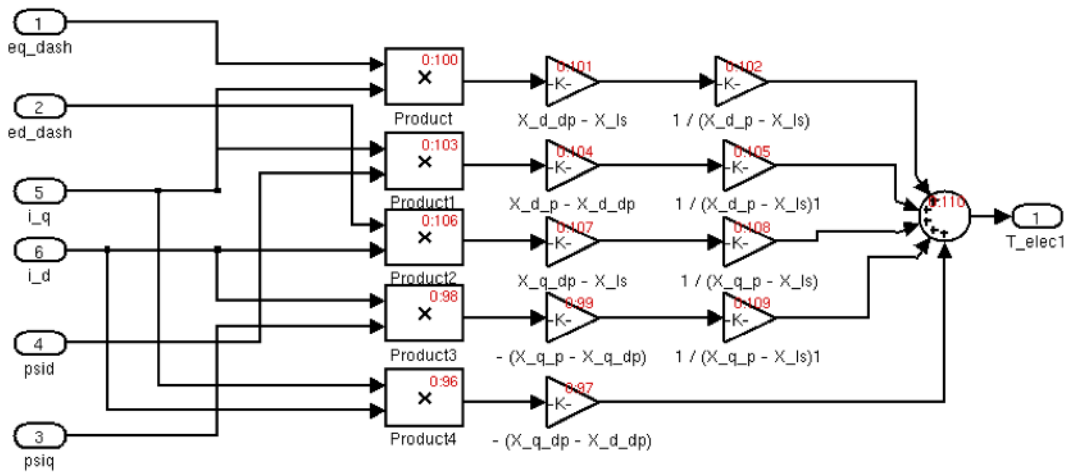


Figure 3.9: the *Machine T_elec* subsystem

Form the diagram we can see two forms of operators that can represents the production operation, Product and Gain. We can extract the update formula form the diagram as:

$T_{elec} = (eq\_dash * I\_q * \frac{X\_d\_dp - X\_ls}{X\_d\_p - X\_ls}) + (ed\_dash * I\_d * \frac{X\_q\_dp - X\_ls}{X\_q\_p - X\_ls}) + (psid * I\_q * \frac{X\_d\_p - X\_d\_dp}{X\_d\_p - X\_ls}) - (psiq * I\_d * \frac{X\_q\_p - X\_q\_dp}{X\_q\_p - X\_ls}) - (I\_q * I\_d * (X\_q\_dp - X\_d\_dp)$

3.4.2 Integration

As we mentioned before, the dynamic behavior of the power system is described by a set of nonlinear differential-algebraic equations (DAE). After analyzing the SIMULINK model, it is not hard to see that most of out update rules are dependent on the Integrator block, which outputs the value of the integral of its input signal with respect to time. For example, the dynamic characteristics of the TCSC system (Fig. 3.10)is assumed to be modeled by a single time constant T_tcsc representing the response time of the TCSC control circuit as follows:

$$\frac{d}{dt}\Delta k_c = \frac{1}{T_{tcsc}}(-\Delta k_c + \Delta k_{c-ref} + \Delta k_{c-ss})$$

In Simulink, it is represented as:

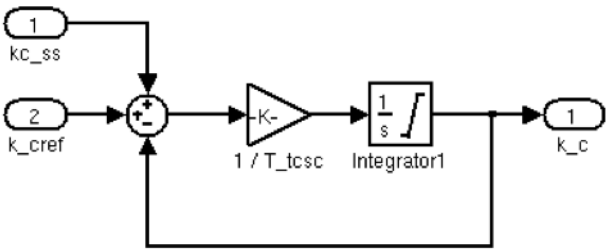


Figure 3.10: The TCSC System in SIMULINK

While this equation defines an exact relationship in continuous time, Simulink uses numerical approximation methods to evaluate them with finite precision. Simulink can use a number of different numerical integration methods, called ODE solver, to compute the Integrator block's output, each with advantages in particular applications. With the ode3 (*Bogacki-shampine*) solver is the default one, we can also use the **Solver pane** of the Configuration Parameters dialog box to select the technique we want to use.

Mathematical Principle of ODEs:

The following is the general definition of ODE [35]: Let F be a given function of x,y, and derivatives of y. Then an equation of the form:

$$F(x, y, y^{'}, ..., y^{(n-1)}) = y^{n}$$

is called an explicit ordinary differential equation of order n. More generally, an implicit ordinary differential equation of order n takes the form:

$$F(x, y, y^{'}, y^{''}, ..., y^{(n)}) = 0$$

There are plenty of numerical methods has been proposed for solving ODE. In our program, for simplicity, we chose the ODE Solver called the **Euler Method**.

Mathematical Principle of Euler Method:

In mathematics and computational science, the Euler method is a first-order numerical procedure for solving ordinary differential equations (ODEs) with a given initial value. And the following is

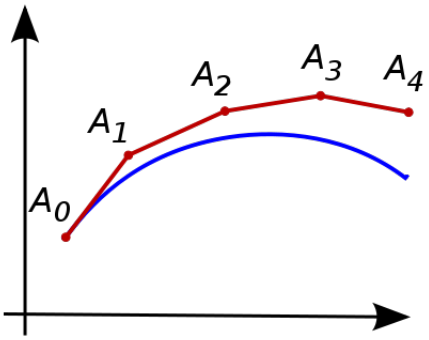


Figure 3.11: Graphical Illustration of Euler Method

the idea of the Euler Method:

Consider the problem of calculating the shape of an unknown curve which starts at a given point and satisfies a given differential equation. Here, a differential equation can be thought of as a formula by which the slope of the tangent line to the curve can be computed at any point on the curve, once the position of that point has been calculated. The idea is that while the curve is initially unknown, its starting point, which we denote by A_0 is known (see Fig. 3.11). Then, from the differential equation, the slope to the curve at A0 can be computed, and so, the tangent line. Take a small step along that tangent line up to a point A_1 Along this small step, the slope does not change too much, so A_0 will be close to the curve. If we pretend that A1 is still on the curve, the same reasoning as for the point A_0 above can be used. After several steps, a polygonal curve $A0, A1, A2, A3$.is computed. In general, this curve does not diverge too far from the original unknown curve, and the error between the two curves can be made small if the step size is small enough and the interval of computation is finite. And the function looks like:

Suppose that we want to approximate the solution of the initial value problem

$$y^{'}(t) = f(t, y(t)),\ y(t_0) = y_0$$

Choose a value h for the size of every step and set $t_n = t_0 + nh$. Now, one step of the Euler method from t_n to$t_{n+1} = t_n + h$ is

$$y_{n+1} = y_n + h * y^{'}(t)$$

The value of y_n is an approximation of the solution to the ODE at time : $t_n : y_n \approx y(t_n)$.

In our system, h is the *step-size*, $y^{'}(t)$ is the derivative of y at time t. From the formula we can see that if we have y_0, *step-size* and the formula of how to calculate the derivative then we can calculate the value of the variable at any time.In our case, the *step-size* is already known, the default value y_0 is calculated using several .m files and can be known directly as well. What's more the formula of how to calculate the derivative of each variable can be derived from the SIMULINK model as shown in Chapter 4.

3.4.3 Multiport Switch

The Multiport Switch block chooses between a number of inputs passing through the input signals corresponding to the truncated value of the first input. The inputs are numbered top to bottom (or left to right) with the first input port is the control port and the other input ports are data ports.
In our system, the Multiport Switch block is used in the '*Modified Ybus due to change in kc*'

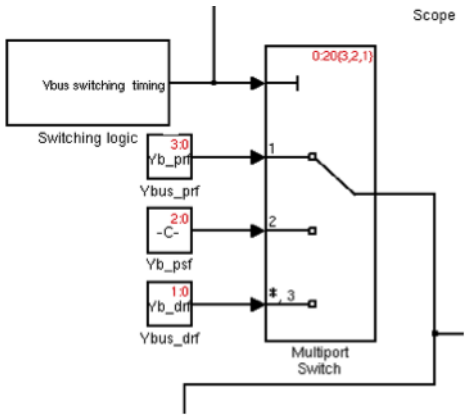


Figure 3.12: The Multiport Switch Block

subsystem as shown in Fig. 3.12. Like the usage of switch statement in a programing language, based on the value of Ybus switching timing (1,2 or 3), the system choose different $Ybus$ data from *Ybus_prf, Ybus_psf or Ybus_drf* to pass to the next operation and if no value matching, the system will pass the *Ybus_drf*.

3.4.4 Pulse Generator

From the above section, we know that which $Ybus$ data to use is depending on the value passed to the '*Switching logic*' block (Fig. 3.13). This block uses two **Discrete Pulse Generators** each of which generates the value of 1 intermittently. So that, adding up with a constant of 1, it can finally output a value of 1,2 or 3 changing with the simulation time. (This scheme is to simulation an occur of short-circuit as I observed that after 1.0s the value of *Ybus switching timing*' will always be 2)

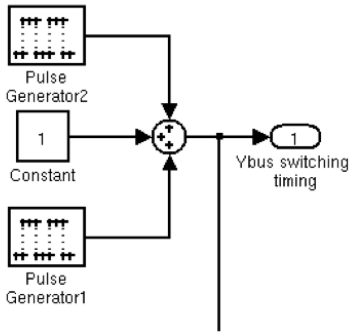


Figure 3.13: The Switching Logic Block

The working principle of a Discrete Pulse Generator block is that it generates a series of pulses at regular intervals. We specify the **Amplitude** as the amplitude of the pulse; the **Pulse width** as the number of sample periods the pulse is high; the **Period** is the number of sample periods the pulse is high and low and the **Phase delay** is the number of sample periods before the pulse starts. The **Sample time** is the length of the one sample period, here is the *step size*: 0.001 seconds. For example, our Pulse Generator 2 as the parameters shown in Fig. 3.14:
With $prd = \frac{prft+psft}{Tsmp} = 26000$, $phdelay = \frac{prft}{Tsmp} = 1000$; $pulswdth2 = \frac{fdtn}{Tsmp} = 80$ predefined in the *init_sim.m* file. Where $Tsmp$ is sample time = 0.001s. This means that the generator will begin to output value 1 every 26 seconds after 1 seconds simulation and each time lasts for 0.08 seconds. For the remaining time, outputs 0.

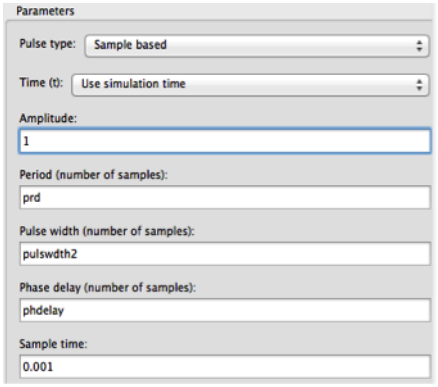


Figure 3.14: Parameters for an Pulse Generator

3.4.5 LU Solver

As a nonlinear solution scheme to solve the resultant nonlinear algebraic equations, the LU Solver block solves the linear system $AX = B$ by applying LU decomposition to the M-by-M matrix (must be square) at the A port. The input to the B port is the right side M-by-N matrix, B. The M-by-N matrix output X is the unique solution of the equations. The block treats length-M unoriented vector input to the input port B as an M-by-1 matrix. In our system, a LU Solver is used in the '*Machine Network interface*' subsystem which takes the 11*11 *Ybus* data and 11*1 *i_bus_mod* to get the *Vm_cmplx* values of the 11 bus.

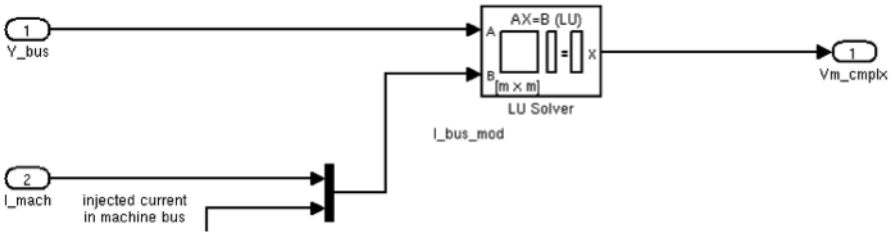


Figure 3.15: LU Solver Block in SIMULINK

Algorithm

The LU algorithm factors a row-permuted variant (A_p) of the square input matrix A as $A_P = LU$ where L is a lower triangular square matrix with unity diagonal elements, and U is an upper triangular square matrix. The matrix factors are substituted for A_p in $A_pX = B_p$ where B_p is the row-permuted variant of B, and the resulting equation $LUX = B_p$ is solved for X by making the substitution $Y = UX$, and solving two triangular systems: $LY = B_p$, $UX = Y$.

LU Decomposition

From the algorithm we can see that the first step for a LU Solver is to factor a matrix as the product of a lower (L) triangular matrix and an upper (U) triangular matrix. This process is the so-called LU decomposition (or LU Factorization).
The LU decomposition is basically a modified form of Gaussian elimination. We transform the matrix A into an upper triangular matrix U by eliminating the entries below the main diagonal. [35] The idea is shown in Fig. 3.16. There are many methods proposed to solve LU decomposition, for example, the most common one, the Doolittle algorithm, does the elimination column-by-column starting from the left, by multiplying A to the left with atomic lower triangular matrices. It results

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} . \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Figure 3.16: An LU decomposition of a 3x3 matrix

in a unit lower triangular matrix and an upper triangular matrix. And the Crout algorithm is slightly different and constructs a lower triangular matrix and a unit upper triangular matrix. In our C code implementation, we adapt the **Doolittle algorithm** for LU decomposition, please see more detail in next Chapter.

3.4.6 Real-imag to Complex and Complex to Real-imag

Complex numbers are very common in electrical engineering. In our system, we have to deal with complex numbers, for examples, the *Ybus* data are complex numbers, the machine currents, injected current in machine bus (*I_mach*) is in complex number and the same as the *Vm_cmplx* values. So, in some circumstance, we have to do the job of convert real numbers to complex numbers and sometimes inversely. In the Machine currents subsystem, we can experience the use of this kind of blocks: the Real-imag to Complex block and Complex to Real-imag block.

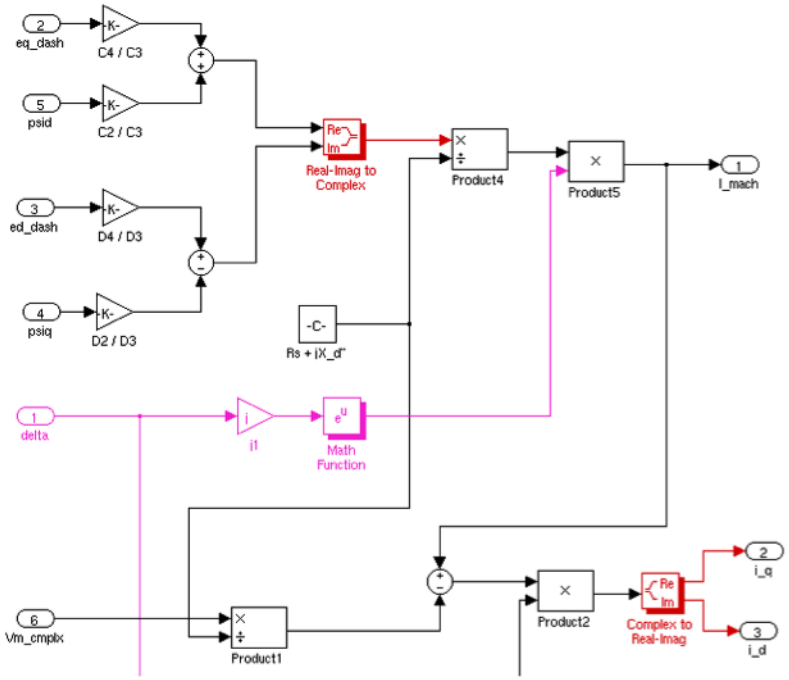


Figure 3.17

In Simulink, the Real-imag to Complex block (red block in Fig. 3.17) converts real and/or imaginary inputs to a complex-valued output signal. The inputs must be real-valued signals of type double. The data type of the complex output signal is double. The Complex to Real-imag block is just the opposite.

From Wikipedia, **a complex number** is a number that can be expressed in the form $a + bi$, where a and b are real numbers and i is the imaginary unit, where $i^2 = 1$.[1] In this expression, a is the real part and b is the imaginary part of the complex number.

In Simulink, the inputs may be both vectors of equal size, or one input may be a vector and the

other a scalar. If the block has a vector input, the output is a vector of complex signals. The elements of a real input vector are mapped to real parts of the corresponding complex output elements. An imaginary input vector is similarly mapped to the imaginary parts of the complex output signals. If one input is a scalar, it is mapped to the corresponding component (real or imaginary) of all the complex output signals. For example the Real-imag to Complex block in the diagram does the operation as:

$Output = (eq\_dash * C4/C3 + psid * C2/C3 + (ed\_dash * D4/D3 - psiq * D2/D3 * I)$

3.4.7 Exponential form of complex number

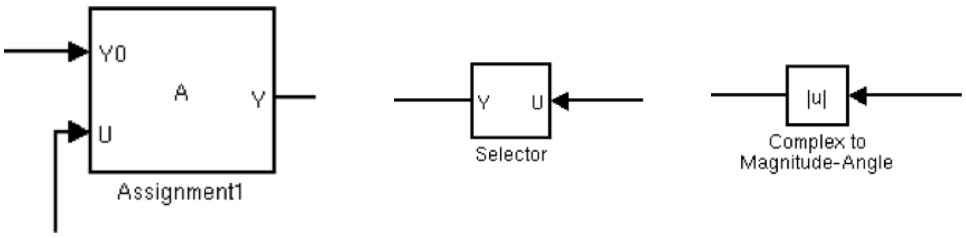
As we can see from the Fig. 3.17 (pink part), It gives an exponential form of complex number as the output is: $e^{i*delta}$, where $i * delta$ is a complex number that equals $0 + i * delta$.

Mathematical Principle

We have, so far, considered two ways of representing a complex number: Cartesian form $z = a + ib$ or polar form $z = r(cos + isin)$. And since $e^i = cos + isin$, we therefore obtain another way in which to denote a complex number: $z = a + ib = r(cos + isin) = re^i$

So we get $e^{i*delta} = cos(delta) + I * sin(delta)$;

3.4.8 Others



(a) Assignment (b) Selector (c) Complex to Mag

Figure 3.18: Other Noteworthing Blocks

Assignment Block (Fig. 3.18(a)): Assign values to specified elements of a multidimensional output signal. The index to each element is identified from an input port or this dialog.

Assignment blocks are used after the modified Ybus entries $Y(9,9)Y(8,8), Y(8,9)$ due to the change in the percentage compensation of the TCSC (K_c) are calculated and we have to exchange the old values to the new values.

Selector (Fig. 3.18(b)): Select or reorder specified elements of a multidimensional input signal. The index to each element is identified from an input port or this dialog. In our system, the selector block is used when passing the *Vm_cmplx* values, as the *Vm_cmplx* is a 11*1 array, only the first 4 units are meaningful values for each of the four machines. So the selector block selectors the 4 elements.

Complex to Magnitude-Angle (Fig. 3.18(c)): Compute magnitude and/or radian phase angle of the input.

Consider the complex number $z = abi$. The **complex conjugate** of the number z, denoted z^*, is obtained by simply taking every i that you see in the expression for z and replacing it by i. The

magnitude of a complex number z, denoted $|z|$, is defined to be the positive square-root of the complex number times its complex conjugate. That is, $|z| = \sqrt{zz^*}$

In general, for the generic complex number $z = a + bi$, the magnitude of z is given by:

$$|z| = \sqrt{zz^*} = \sqrt{(a+bi)(a-bi)} = \sqrt{a^2 - abi + abi - b^2 i^2}$$

And, since $i^2 = 1$,

$$|z| = \sqrt{a^2 + b^2}$$

In our system, we compute the magnitudes of the 4 complex numbers from the *Vm_cmplx* array to get the measured terminal voltages (V_t) for the 4 machines to be used in the Excitation system.

3.5 Performance Gain

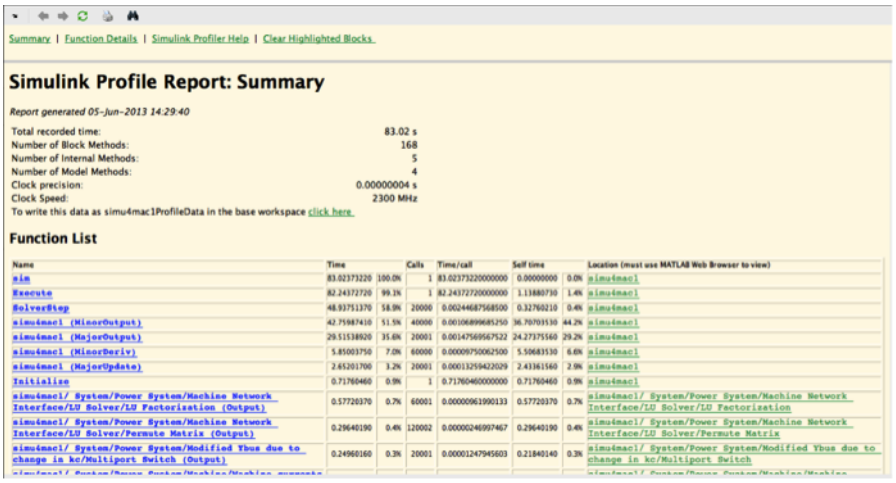


Figure 3.19: SIMULINK Profiling Report

The second analysis of the existing model is about the running time, by looking at a Simulink Profile Report as shown in Fig. 3.19, we can get the total recorded time of each simulation as well as the detailed records such as, the times per call and self time. By running multiple simulations on different time periods (here we show 5s, 10s, 20s, 40s), we get results like this:

Simulation Time	5s	10s	20s	40s
Recorded Time	22.64s	42.87s	83.02s	163.35s

Table 3.1: f

As shown in the figures, although the recorded times are increasing linearly, it is much (4 times) bigger than the real simulation times. This means that the simulation is far from optimal and has a large space for acceleration.

3.6 Summary

In this chapter, we describes the process of how we get to understand the SIMULINK model as a new player. In detail, we firstly introduced the power system that the simulation built on, then from the Simulation program we analysed the properties of the program: the requirements of the

program, the data flow and the control flow of the program.Section 3.1 described the power system being simulated. Section 3.2 covered an detailed analysis of the power system simulation in the form of a SIMULINK model. And section 3.3 gave a brief illustration of the performance of the SIMULINK model from the efficiency prospect.

Chapter 4

Optimised the Power System Simulation

4.1 Simulink Coder & Initial Design

The Simulink Coder (formally Real-Time Workshop code generator)[20], is an extension of capabilities of Simulink and MATLAB that automatically generates, packages and compiles C or C++ source code from Simulink diagrams to create real-time software applications on a variety of systems.

Firstly, we tried to use the Simulink Coder to automatically generate the C code form our Simulink system. The Coder generated a folder called simu4mac1_grt_rtw which contains 28 files including header files, data files, etc. And only the main C file has almost 2500 lines of code with very elusive structure. It seems that because of the automaticity, the generated code is not very efficient. Therefore, if we want to use the generated code we have to experience large amount of refactoring and reimplementation work. And it may be time consuming to understand the generated code first. According to the above reasons, we decided to write the C code from a scratch.

Drawing lessons from the Simulink Coder [28], we know that in the first phase the Simulink coder converts the model to an executable form following these steps:

- It evaluates the parameters (given as expressions over constants and Matlab's workspace constants) of the model's blocks;
- It determines signal attributes, e.g., name, data type, numeric type, dimensionality and sample rate (these are not explicitly specified in the model file) and performs type-checking, i.e., it checks that each block can accept the signals connected to its inputs;
- It flattens the model hierarchy by replacing virtual subsystems with the blocks that they contain;
- It sorts the blocks according to the execution order.

After this phase, the blocks are executed in a single loop corresponding to a single step of the model.

As long as we know the sort of execution for each block we are now able to realize exactly what simulink does in a time step as follows:

*In a specific time i*step time:*

For each machine{

double delta_value[machine]=delta[i][machine]; //The Integrator block in sub-system Machine delta which has a sorted order of 0.

double delta_in_radian= delta_value[machine] 180/PI; //The gain block in sub-system System which has a sorted order of 1(However, this value is not important here due to we focus on how the value of all the states change in one time step here. This gain block only matters the output and won't have effect on any of the states.)*

double k_c_value=k_c[i]; //The Integrator block in sub-system TCSC which has a sorted order of 8.(Here we move from 1 directly to 8 for the same reason

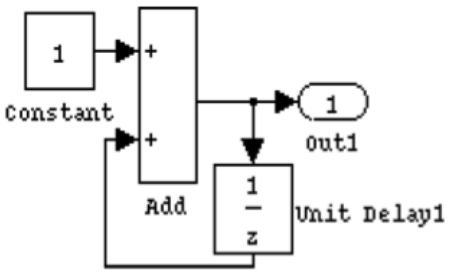
details please refer to the full sorted list in the appendix.)
.
.
double omega_value=omega[i][machine]; //The Integrator block in sub-system Machine omega who has a sorted order of 69.
.
.
double delta[i][machine]=omega_value[machine]-omega_synch
}

However, if we are doing as above, then we can find that we are assigning the updated value to the variable in the same round, and when the time becomes $(i+1) * stepsize$, $delta[i+1][machine]$ and all other variables will become undefined. And the problem is according to the principle of integration. As we use Euler's method, the update rule follows $f(x+1) = f(x) + h * f'(x)$, here the updated value should be assigned to the variable in the next round. Therefore, we need to identify which blocks are assigning value to current variables and which blocks are assigning value to next time step's variables. Which can be very time consuming and fallible and it will be very difficult to locate such bugs when debugging.

Therefore, instead of exactly re-implement all the operation steps in sequence, we may take the computation in a subsystem as a whole, which means we re-implement in the order of subsystems.

We can do so because for the sub-systems who contain Integrator blocks, they are essentially implementing Euler's method: $f(x+1) = f(x) + h * f'(x)$, which means in the current round they are using the value in previous round so that in the current round there are no dependencies between any pair of variables and the order is not important as long as they updated the value and assigned the value to variables in the next round. For the sub-systems who don't contain Integrator blocks, we can also treat a sub-system as a whole is because the sorted order is data-driven. For instance, in sub-system *'Ybus modification'*, block Assignment needs the value of *Y_8_9*, therefore the sum block in modified Ybus entries due to change in kc will be executed first so that it has a sorted order of 31 and the Assignment block has a sorted order of 32, which means if the whole sub-system modified $Ybus$ due to change in kc is executed before the sub-system $Ybus$ modification, then we will still get the same expected result. In fact, virtual sub-systems have no sorted order, however we may determine the order of the sub-systems according to their Integrator blocks' sorted order and the order of data flow. And as a result, the derived order is just as mentioned in section

RTW also affirms this idea that Simulink diagrams represent synchronous systems, and can therefore be translated naturally into an initialization function and a step function. [28]The block diagram in the left of Fig. 4.1, for example, represents a counter; the corresponding initialization and step functions that were generated RTW appear to the right of the same figure. The generated code has the form of two functions corresponding to the initial state and the step.




```
void example_state_initialize(void)
{
Out1 = 0.0;
UnitDelay1_DSTATE = UnitDelay1_X0;
}
void example_state_step(void)
{
UnitDelay1 = UnitDelay1_DSTATE;
Add = Constant_Value + UnitDelay1;
Out1 = Add;
UnitDelay1_DSTATE = Add;
}
```

Figure 4.1

Based on this guidance, we got to our initial train of thought: we firstly converted each subsystem to a single C function. That is we encapsulated the update operations of each subsystem in a step function as a black box hiding the dataflow. And then have to focus on two aspects: the initialization of the parameters and updating the parameters by calling the step functions abiding to the right order.

4.2 Reimplementation

4.2.1 Data Initialisation

As we have discussed in previous chapter, before the simulation begins, all these parameters in the system should already have their initial states. In our system, the initial function: *Run_simu* calls other MATLAB scripts and functions to calculate the initial states of the input value of the Model. Some of the .m files are only for data supplying and some of them perform some operations on the loaded data. We can find all the variables appear in the model in these .m files and their descriptions as comments.

Therefore, the re-implementation of the data initialization functions is actually the work of converting the MATLAB scripts and functions to C code. However, after analyzing the Matlab code we found that converting all the .m files which initialize the model is a time-consuming work, especially the function *loadflow_mod.m* who solves the load-flow equations of power systems using the Newton-Raphson algorithm. And this function also needs to call 4 other functions that make the code more complex.

Fortunately, the initialistion process is before the simulation starts and it has little influence to the performance of the simulation. So we realized the initialization of all the input by directly utilizing the data generated from the .m files instead of actually converting them into C code. Specially, we hard coded the data that is needed for the model, i.e., the data in *data_kundur_mod.m*. And we also re-wrote the function *mac_sat_kundur.m* code to initialize the variables and states that are used in the model. But for the later processing: load-flow solution. We only hard coded the results from the MATLAB workspace.

4.2.2 Timing and Update Sequence

As we know, in Simulink, the simulation process is time based. A solver computes a dynamic system's states at successive time steps over a specified time span, using information provided by the model. So we should specify the Simulation Time which is the total length of time the simulation runs (the period between start time and end time) and the Step size which determines the update frequency. As for the Euler solver: $F(t) = F(t-1) + h * F'(t-1)$, $stepsize = h$. We can imagine that, the system updates its states every step size. So the total number of iterations is:

$$Iterations = \frac{Simulationtime}{stepsize}$$

For example, if the Step size is set to the default value 1e-3, and Simulation Time is the 20s, the total number of iterations is: 20000. We can simulate this timing scheme in C code as a single for loop in the main function: $for(i = 0; i * stepSize <= simulation_time; i++)$

In each iteration, we update all the states by calling the step functions of the subsystems in the order:

Machine delta →TSCS system →Modified Ybus due to change in K_c →Machine psid →Machine ed_dash →Machine psiq →Machine eq_dash →Machine currents (I_mach) →machine network interface →DC Exciter_E_df →DC Exciter_R_f →DC Exciter_V_r →DC Exciter_V_tr →Machine omega →Machine currents (I_q, I_d)→Machine T_elec

However, during the implementation phase, we noticed that, keep calling functions outside the main function needs declaration of a large amount of variables which in some extent embarrassed the algorithm.

As we know, Amount these subsystems, the '*Switching logic*' and '*Multiport switch*' subsystems are for flow control, and the '*Ybus modification*' system performs an intermediate matrix operations on the *Ybus* data which gets the new bus data used for each iterations calculations by changing some entries in the *Ybus* data matrix. And each of the remaining subsystems encapsulates the dynamic updating rule for one of these 16 parameters: *Delta, Eq_dash, Ed_dash, Psid, Psiq, Omega, I_mach, T_q, T_d, Vm_cmplx, Efd, Rf, Vr, Vtr,T_elec, Kc*

The dynamic behavior of the some of these subsystems is generally described through a set of differential equations while some them (the power flow in the network) are represented by a set of algebraic equations. Therefore, for each of the parameters, we can simplify the step function as a single equation (formula). For example, by the Euler method, the integrator just updates the states following the formula: $F(t) = F(t-1) + h * F'(t-1)$ and $F(0) = x_0$, where $F(t)$ is the value of a variable at time t, h is the step size, $F'(t)$ is the derivative at time t. The default value x_0 is calculated using several .m files and can be known directly as well. What's more the formula of how to calculate the derivative of each variable can be derived from the SIMULINK model. Therefore, we can calculate the value of the variable at any time. And for the algebraic subsystems, we can extract algebraic equations straightforward from the model diagram.

In details, instead of using only one variable to store the current value of state, I created two-dimensional arrays for every variable respectively to store the value of each variable in every round, specifically, the first dimension is the time (iteration) and the second dimension is the machine number (totally 4 machines). And we do not need to call any step functions, we just use the formulas to update the parameters in each iteration by the sorted order:

Delta →k_c →Psid →Ed_dash →Psiq →I_mach →Vm_cmplx →Efd →Rf →Vr →Vtr →Omega →I_q →I_d →T_elec.

However, a noteworthy thing to mention is that, as the subsystems contain Integrator blocks, they are essentially implementing Euler's method: F(t)=F(t-1)+H*F'(t-1) and F(0)=x0, which means in the current round they are using the value in previous round except the first round. So for update of these kind of parameters, we should add a control scheme: if (i >0) before using the update formula F(t)=F(t-1)+H*F'(t-1). For the sub-systems who don't contain Integrator blocks, they use the current values to update in current round. Therefore, based on all the principles we discussed above, we get the most compact and efficient version of our main function:

```
int main(int argc, char** argv) {
   for (i = 0; i * stepSize <= 20; i++) {
       initialize();
       modified_Ybus_entries_due_to_change_in_kc();
         Ybus_modification();
          int it;
          for (it = 0; it <= 3; it++) {
                if (i > 0) {
                  //Delta
                  delta[1][it] = delta[0][it] + (omega[0][it] -
                                 Omega_synch)* stepSize;
           . . .
         }
       //I_mach
       I_mach[1][it] = (eq_dash[1][it] * C4[it] / C3[it] + psid[1][it]
                        * C2[it] / C3[it] + (ed_dash[1][it] * D4[it]
                       / D3[it] - psiq[1][it] * D2[it] / D3[it]) * I)
                      / (R_s[it]+ X_d_dp[it] * I) * (cos(delta[1][it])
                         + I * sin(delta[1][it]));
       . . .
       }

       LUSolver1(&Ybus[0][0], &b[0], 11, i);

       for (it = 0; it <= 3; it++) {
         if (i > 0) {
                  //Efd
            efd[1][it] =  (. . .)
          . . .
         }
      //I_q
      I_q[1][it] =  (. . .)
       . . .
      }
}
```

4.2.3 Reimplementation of the Multiprt Switch Scheme

We implement the updating of the power system parameters in the way we discussed above. Beside these subsystems that describe the dynamic behavior of devices in power system, in the simulation model, we also have some subsystem for intermediate data handling. Namely, the whole*Modified Ybus due to change in Kc*' system finally outputs the modified bus admittance matrix Y used in each iteration due to the power injection of the TCSC system represented by the value of percentage compensation (Kc). However, before the modification, we actually use different original $Ybus$

data.

The '*Switching logic*' and '*Multiport switch*' subsystems together provide the control scheme for choosing from 3 different kinds of $Ybusdata$ (*Ybus_prf, Ybus_psf, Ybus_drf*). As we have detailedly introduced these two subsystems in section **3.4.3** and **3.4.4**, '*Multiport switch*' chose from the 3 options based on the value of '*Ybus switching timing*' (1,2 or 3) generated by '*Switing logic*. This scheme can be implemented as a switch statement in C code.

The '*Switing logic*' system determines the value of *switching_time* with the help of two Pulse Generators. The working principle of a Pulse Generator block is that it generates a series of pulses at regular intervals. We specify the Amplitude as the amplitude of the pulse; the Pulse width as the number of sample periods the pulse is high; the Period is the number of sample periods the pulse is high and low and the Phase delay is the number of sample periods before the pulse starts. The Sample time is the length of the one sample period, here is the setp size: 0.001 seconds. For example, our Pulse Generator 2 as the parameters: With $prd = (prft + psft)/Tsmp = 26000$, $phdelay = prft/Tsmp = 1000$; $pulswdth2 = fdtn/Tsmp = 80$ predefined in the *init_sim.m* file. Where $Tsmp$ is sample time = 0.001s. This means that the generator will begin to output value 1 every 26 seconds after 1 seconds simulation and each time lasts for 0.08 seconds. For the remaining time, outputs 0.

To implement this scheme in our main, we defined 2 variables *Pulse1* and *Pulse 2*. In each iteration i, we should determine whether each of Pulse values is 1 or 0. Lets consider a simple case first, we calculate the value i modulo prd, and if the value is less than $pulswdth1$ then the amplitude of the pulse at time i is 1, otherwise 0. Here in our case, the delay needs to be taken into account, therefore if $(i - phdelay)$ modulo prd is less than $pulswdth1$, then at time i the amplitude of the pulse is 1. In addition, due to in our case, time should always larger than 0, the first amplitude of 1 should appear when $(i - phdelay) >= 0$. In code, it looks like:

```
if ((i-(int)phdelay)%(int)prd < pulswdth1 && (i-(int)phdelay>= 0)) {
                        pulse1 = 1;
} else {
                        pulse1 = 0;
}
```

And then, the get **Switching_time(pulse1, pulse2)** function just returns the addition of pulse1, pulse2 and a constant 1 which is 1, 2 or 3. And then, by use a switch statement to implement the Multiple Switch Scheme:
switch (getSwitching_time(pulse1, pulse2)
This statement has 3 cases corresponding to the three kinds of $Ybus$ data and the default case is to use the $Ybus\_drf$.

4.2.4 Reimplementation of the Modification of $Ybus$ Data

In the '*Modified Ybus due to change in Kc*' system, we have two subsystems: '*modified Ybus entries due to change in kc*' and '*Ybus modification*' who modify the bus admittance matrix Y used in each iteration due to the change in *kc*. In our C code implementation, we wrote two functions one for each of these two subsystems.

```
//BLOCK: modified Ybus entries due to change in kc
//return the new values for changing the original ones

Bus_changes modified_Ybus_entries_due_to_change_in_kc
(complex* Ybus, complex modified_tcsc_line_y) {
        Bus_changes changes = { 0, 0, 0 };
        changes.Y_9_9 = *(Ybus + 8 * 11 + 9 - 1) - y_tcsc_line_with_kc
                        + modified_tcsc_line_y;
        changes.Y_8_8 = *(Ybus + 7 * 11 + 8 - 1) - y_tcsc_line_with_kc
                        + modified_tcsc_line_y;
        changes.Y_8_9 = *(Ybus + 7 * 11 + 9 - 1) + y_tcsc_line_with_kc
                        - modified_tcsc_line_y;
        return changes;
}
```

```
//BLOCK: Ybus modification
//Replace the corresponding entries in Ybus matrix with the new values
//and return the new Ybus matrix

double complex * Ybus_modification(complex* Ybus, complex Y_8_9,
                complex Y_8_8, complex Y_9_9) {
        *(Ybus + 7 * 11 + 9 - 1) = Y_8_9;
        *(Ybus + 8 * 11 + 8 - 1) = Y_8_9;
        *(Ybus + 7 * 11 + 8 - 1) = Y_8_8;
        *(Ybus + 8 * 11 + 9 - 1) = Y_9_9;
        return Ybus;
}
```

4.2.5 Reimplementation of LU Solver

In our system, a **LU Solver** is used in the '*Machine Network interface*' subsystem which takes the 11*11 *Ybus* data and 11*1 *i_bus_mod* to get the *Vm_cmplx* values of the 11 bus. The LU Solver block solves the linear system $AX = B$ by

1. Applying LU decomposition to the square matrix A to get $LUX = B$ and then substitute $Y = UX$.

2. And solving two triangular systems. $LY = B$, $UX = Y$

We should re-implement this process in C code as a function void **LUSolver1(double complex*a, double complex*b, int n)**. And in each iteration update *Vm_cmplx* by calling this function.

LU Decomposition Implementation

For the LU decomposition process, we adapt the Doolittle algorithm:

An LU decomposition of A may be obtained by applying the definition of matrix multiplication to the equation $A = LU$. If L is unit lower triangular and U is upper triangular, then

$$a_{ij} = \sum_{p=1}^{min(i,j)} l_{ip} u_{pj} \tag{4.1}$$

where $1 <= i$ and $j <= n$. Rearranging the terms of Equation 4.1 yields

$$l_{ij} = \frac{a_{ij} - \sum_{p=1}^{j-1} l_{ip} u_{pj}}{u_{jj}} \tag{4.2}$$

where $i > j$, and

$$u_{ij} = a_{ij} - \sum_{p=1}^{i-1} l_{ip} u_{pj} \tag{4.3}$$

where $i < j$ Jointly Equation 4.2 and Equation 4.3 are referred to as Doolittles method of computing the LU decomposition of A. We implemented the same algorithm as:

```
\\ LU Decomposition

for (k = 0; k < n; k++) {
        u[k][k] = 1;
            \\ Find L part using Equation 4.2
             for (i = k; i < n; i++) {
                  sum = 0;
                  for (p = 0; p <= k - 1; p++) {
                          sum += l[i][p] * u[p][k];
                  }
                  l[i][k] = *(a + i * n + k) - sum;
             }
            \\ Find U part using Equation 4.3
             for (j = k + 1; j < n; j++) {
                   sum = 0;
                   for (p = 0; p <= k - 1; p++) {
                          sum += l[k][p] * u[p][j];
                  }
                  u[k][j] = (*(a + k * n + j) - sum) / l[k][k];
         }
}
```

4.3 Performance Gains

4.3.1 Correctness

The program can generate exactly same results as SIMULINK does up to 10 digits after decimal point after 20000 times calculations (20s' simulation). And can give the exact same fluctuate patterns of the signals as in the SIMULINK model. Take the desired phase differences between generator 1 and generator 3 (G1-G3) of 20's simulation as an example, the upper diagram (Fig. 4.2)is generated by SIMULINK, and the lower one (Fig. 4.3) is generated from the results of our C code:

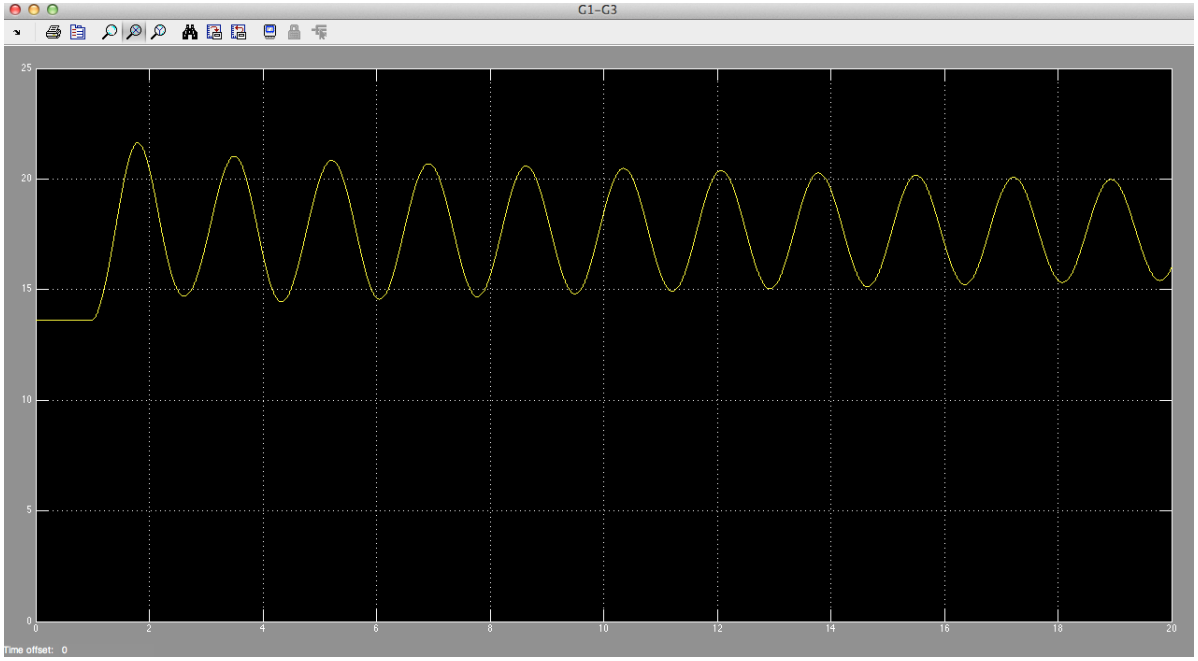


Figure 4.2: G1-G3 in SIMULINK of 20s' simulation

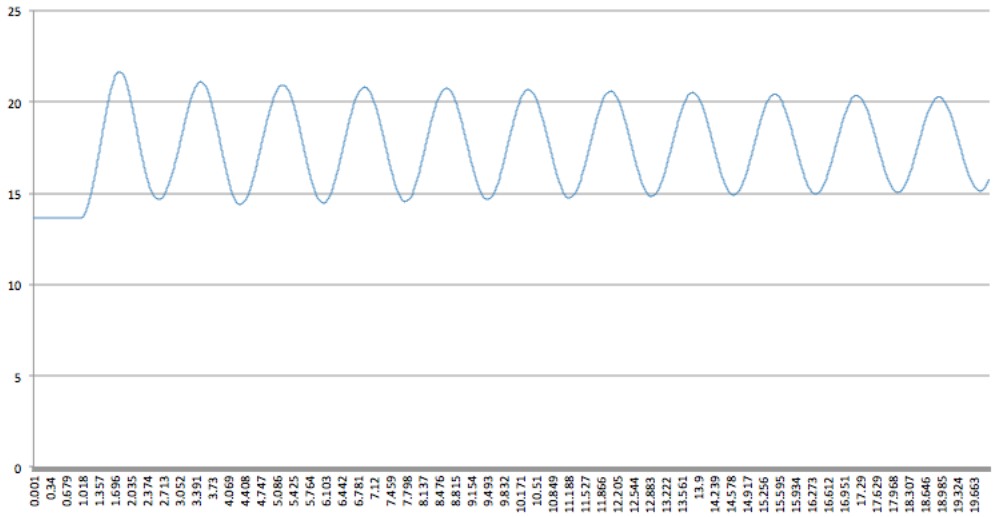


Figure 4.3: G1-G3 from C code of 20000 times' Calculation

4.3.2 Speed

We can get the total recorded time of each simulation in SIMULINK from the Simulink Profile Reports while recording the running time of the C program using the function **gettimeofday()**.By running multiple simulations on different time periods (here we show 5s, 10s, 20s, 40s), we get results like this:

Simulation Time	5s	10s	20s	40s
Recorded Time	22.64s	42.87s	83.02s	163.35s

Table 4.1: Recorded Time using SIMULINK

Simulation Time	5s	10s	20s	40s
Run Time	0.037s	0.098s	0.158s	0.223s

Table 4.2: Running Time of the C Program

By comparing the results we see that although the recorded times are increasing linearly, it is much (4 times) bigger than the real simulation times and our C program optimises the simulaiton system by a approximately 500 times' speed up.

4.4 Summary

In conclusion, this chapter gave a detailed illustration of an optimised application in C language that does the same thing with the simulation system. Depending on the analysis in chapter 2, we described the ideas we used to re-implement each aspect of the simulation scheme that corresponding to each section of chapter 2. Namely, Data initialisation, control flow, and data flow of the program.

Chapter 5

Accelerating the Power System Simulation

In this chapter we present our acceleration solutions to running the Power System simulation and inform the reader of the choices made along the process. Since we identified the most time consuming part of the simulation fundamentally as a matrix inversion problem disguised as an LU matrix decomposition, and since we have optimised the C simulation to what we believe was the maximum possible extent in the timeframe of this project, we look to acceleration by the FPGA techniques to further speed up our simulation process.

5.1 Profiling

It is important to note that developers using Maxeler technology typically aim to accelerate the slowest portion of the software. To Identify areas of code for acceleration, we did a more detailed analysis on the distribution of runtime of various parts of the application using both time counter and profiling tool: gprof.

```
index % time    self  children    called     name
                0.06    0.00   20001/20001       main [2]
[1]     75.0    0.06    0.00   20001         LUSolver1 [1]
-----------------------------------------------
                                                  <spontaneous>
[2]     75.0    0.00    0.06                 main [2]
                0.06    0.00   20001/20001       LUSolver1 [1]
                0.00    0.00   20001/20001       getSwitching_time [5]
                0.00    0.00   20001/20001       modified_Ybus_entries_due_to_change_in_kc [6]
                0.00    0.00   20001/20001       Ybus_modification [4]
                0.00    0.00       1/1           initialize [7]
-----------------------------------------------
                                                  <spontaneous>
[3]     25.0    0.02    0.00                 __intel_ssse3_rep_memcpy [3]
-----------------------------------------------
                0.00    0.00   20001/20001       main [2]
[4]      0.0    0.00    0.00   20001         Ybus_modification [4]
-----------------------------------------------
                0.00    0.00   20001/20001       main [2]
[5]      0.0    0.00    0.00   20001         getSwitching_time [5]
-----------------------------------------------
                0.00    0.00   20001/20001       main [2]
[6]      0.0    0.00    0.00   20001         modified_Ybus_entries_due_to_change_in_kc [6]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[7]      0.0    0.00    0.00       1         initialize [7]
-----------------------------------------------
```

Figure 5.1: Profing Result using Gprof

From the result provided by Gprof (Fig. 5.1) we can see, that is the *LUSolver1* function that

dominated the most of the running time (75%). It is reasonable as it has nested loops. And it is clear to see, the other part in the *main* function only occupy 25% of the total running time. However, Gprof tells us that the calling of the function *modified_Ybus_entries_due_to_change_in_kc, Ybus_modification, initialization, getSwitching_time* do not matter to much to the running speed. that is the something called *intel_ssse3_rep_memcpy* in the system that takes this 25% of the time. As we can guess that, that is those parameter update operations which are not defined in functions.

By focusing on the *LUSolver1* function, we then use time counters in the program to test the running time of different blocks of operations. As in *LUSolver1*, there are 3 *for* loops, the first one for LU Decomposition, and the other two for forwarding substitution and finding the $X(Vm\_cmplx)$, respectively. For a 20's simulation, the first loop takes 122000ms while the other two loops together take 46000ms. Therefore, we can get the conclusion that the LU Decomposition process is the most complicated and inefficient part of the code. We should consider if we can accelerate it in Maxeler.

For the second time consuming component, the updating operations, we can also accelerate in Maxeler as these operations are done on 4 machines. As they do the same operations, we can pipeline the 4 machine. Therefore, we get to the initial design of a CUP-FPGA simulation shown in Fig. 5.2.

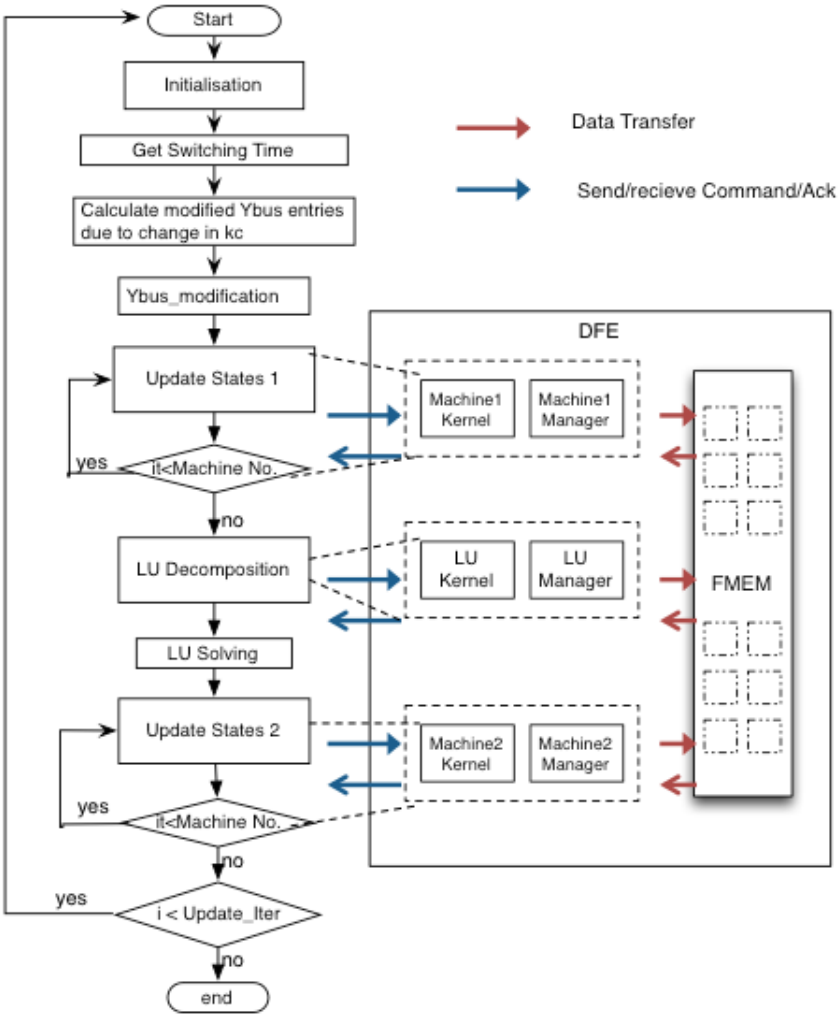


Figure 5.2: FPGA-based Algorithm Design

5.2 Pipeline of LU Decomposition

The difficulties I've met: 1. multiplexer does not support complex numbers which are the data types involved in our matrices. 2. For each element of matrix A, the iteration time is different, namely, the behavior is not regular. we show how we solved them in next two sections.

5.2.1 Analysis

As part of the LU Solver function, the LU decomposition process solves the problem of calculating a lower triangular matrix L and an upper triangular matrix U given matrix A, such that $LU = A$. The C code implementation can be found in .

Due to every calculation must use the up to date value of elements in L and U, we firslty initialize matrix L and U with all the entries on diagonals are one and the others zero. And the Inputs are L, U and A, outputs are new_L, new_U.

Memory Accessing

DFEs provide two basic kinds of memory: FMem and LMem. FMem (Fast Memory) is on-chip Static RAM (SRAM) which can hold several MBs of data. Off-chip LMem (Large Memory) is implemented using DRAM technology and can hold many GBs of data. As we know, the key to efficient dataflow implementations is to choreograph the data movements to maximize the reuse of data while it is in the chip and minimize movement of data in and out of the chip.

In our system, A, L, U are 11*11 matrixes and the matrix B is an array of length 11. They are all not of big datasize, so we decided to use the on-chip FMems in Kernels for storing values and computation.

For example: $L[x * 11 + i] * U[y + i * 11]$ can be written as: ***L.read(address)*U.read(address)*** $L[X][Y] = result$ can be written as: ***L.write(address,result,constant.var(true))***;(Here we may use a multiplexer to decide the value of the result to decide whether L or U to be changed)

Complex Numbers in Maxeler

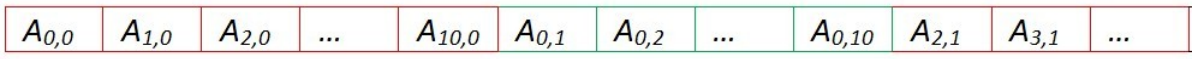
In our system, the LU Solver is used to calculate the Vm_cmplx value from the $Ybus$ data and the ,as we know, they are of the type of complex number. However, multiplexer does not support complex numbers. Therefore in order to use Multiplexer, we need to fisrtly divide the complexer number into real part and imaginary part which are both DFE variables that Multiplexer supports, and then combine the returned results by Multiplexer into a complexer number again to achieve this. Here is an illustration:

```
DFEVar divisionReal=x>=y?1.0:stream.offset(l.read(x * 11 + x), -128)
                                .getReal().cast(dfeFloat(11, 53));
DFEVar divisionImag=x>=y?0.0:stream.offset(l.read(x * 11 + x), -128)
                                .getImaginary().cast(dfeFloat(11, 53));
DFEComplex division = new DFEComplexType(dfeFloat(11, 53))
.newInstance(this);
division.setReal(divisionReal);
division.setImaginary(divisionImag);
```

Data Dependency

The above figure shows in what order the decomposition was executed, and the execution order was derived according to data dependency. For example, in the first row all the elements start from

the second column need the decomposed result of the first column's element. This emphasizes that the decomposition must be executed in some order to get the correct result. However, if we keep the algorithm unchanged and then try to rewritten it into Maxeler, then a lot of preparation work are needed. For example:



After that Maxeler can read input and perform the computation in the correct order. However, if we do what as what the above illustration shows, then the corresponding computation will change, see below:

```
for (i = k; i < n; i++) {
                    sum = 0;
                    for (p = 0; p <= k - 1; p++) {
                        sum += l[i][p] * u[p][k];
                    }
                    l[i][k] = *(a + i * n + k) - sum;
                }
```

Here `i,p,k` needs to be changed correspondingly, otherwise the computation will no longer be correct. And I find that to change them correspondingly is not an easy thing so that I proposed a new algorithm:

```
for each element in A (read row by row)
Iteration_Times= 11 (instead of min(row number of the element,
                                    column number of the element))
for(int i=0;i<Iteration_Times;i++){
        if(i < min(row number of the element,
                column number of the element)){
                sum+=L[x*11+i]*U[y+i*11];
        }
        else{
                sum+=0;
        }
}
if(row number >= column number){
        result=(A[row number][column number]-sum)/L[X][X];
}
else{
        result=(A[row number][column number]-sum)/1;
}
if(row number of the element>=column number of the element){
        L[X][Y]=result;
}
else{
        U[X][Y]=result;
}
```

This algorithm doesn't break the data dependency as shown in the original algorithm, and make the behavior regular.

5.2.2 Multi-tick Implementation

After refactoring the code, now the code looks like:

Listing 5.1: Application code

```
for(int a=0; a<121; a+=1){
        int x=a/11; //  x is the row number
        int y=a%11; //  y iis the column number
        int z; //       z equals min(x,y)
        if(x>=y){
                z=y;
        }
        else{
                z=x;
        }
        sum[a]=0;
        for(int i=0; i<11; i+=1){
                if(i<z){
                        sum[a]=sum[a]+l[x][i]*u[i][y];
                }
        }
        if(x>=y){
                l[x][y]=sum[a];
        }
        else{
                u[x][y]=sum[a]/l[x][x];
        }
}
```

For now we've re-implemented the dynamic control flow to make it regular so that it is now able to be moved into a dataflow Kernel. After refering to the `maxcompiler-loops-tutorial` which is a tutorial explaining how loop structure can be implemented in a Kernel, we've found an example which transforms the following code into dataflow Kernel:

Listing 5.2: Tutorial code

```
int count = 0;
for (int y=0; ; y += 1) {
        sum[y] = 0.0;
        for (int x=0; x<X; x += 1) {
                sum[y] = sum[y] + input[count]
                count += 1;
        }
        output[y] = sum[y];
}
```

We may find that both of them are in the same structure:

```
for(int i=0; ;i+=1){
        Initialization;
        for(int j=0;j<J;j+=1){
                Accumulated sum= Accumulated sum + Accumulation;
        }
        Output;
}
```

The only difference between `Application code` and `Tutorial code` is that the `Accumulation` in `Tutorial code` is a stream of input, however the `Accumulation` in `Application code` is a prodcut of two previously derived results. Although we may use `stream.offset` to get the two results, it

will be too difficult to hard code all the offset values due to the function `stream.offset` only receives `int` or `OffsetExpr`, which is set by the CPU Code, and neither of them is compatible with dataflow variable types, which means using dataflow variable types to make an offset expression is infeasible and the other alternative is to use general multiplexer in the following format:

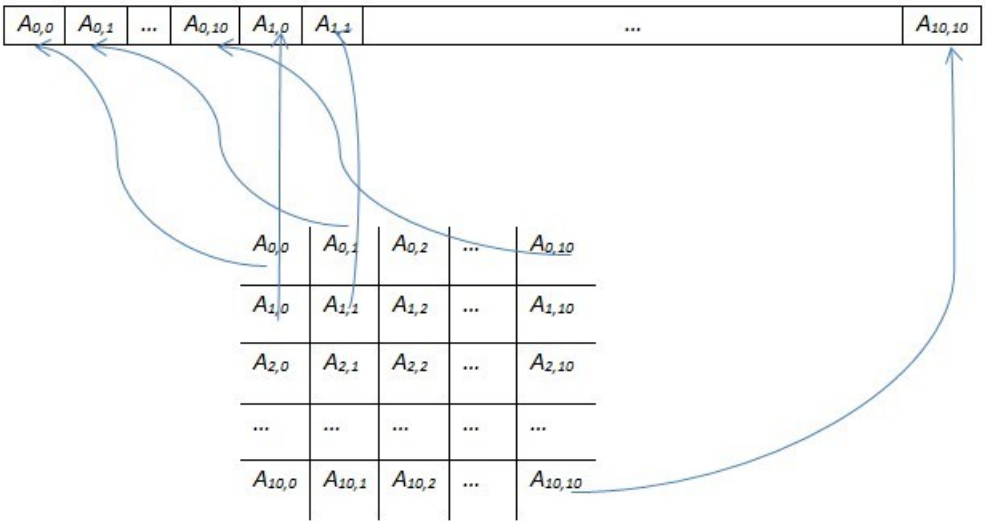
```
control.mux(offset,// Here the control stream
                    "offset" is a dataflow variable type
                stream.offset(x, 0),
                stream.offset(x, 1),
                stream.offset(x, 2),
                stream.offset(x, 3)) ;
```

Even though using general multiplexer may be possible, in our case there are 120 previously calcuated result, hard code all of them is too tedious. As a result, we turned to FMem which is on-chip static memory who can hold several megabytes of data as a solution, by using which we can store and retrieve generated results easily.
As a result, the following is how our Kernel design looks like what shown in `Figure 1`.
And the following are the implementation details:

Table 5.1: Data Dictionary

Variable Name	Data Type	Usage
loopLength	OffsetExpr	An offset expression by which Maxeler can infer the minimum positive value for scheduling if there exists one
loopLengthVal	dfeUInt(8)	The value of the offset expression **loopLength**
DATA_SIZE	int	The size of the matrix to be decomposed
addressCounter	dfeUInt(7)	This counter indicates the address where the calculated result will be write, which increments by 1 when the counter **loopCounter** wrapped
loopCounter	dfeUInt(4)	This counter indicates the current loop number, which increments by 1 when the counter **loopLengthCounter** wrapped
loopLengthCounter	dfeUInt(8)	This counter wraps when it counts to the value of **loopLengthVal-1**
x	dfeUInt(7)	The row number of the element in the input matrix A
y	dfeUInt(7)	The column number of the element in the input matrix A
z	dfeUInt7	The minimum between **x** and **y**, which determines the valid total loop number
l	Memory	The FMem for storing calculated lower triangular element
u	Memory	The FMem for storing calculated upper triangular element
AIn	DFEComplex	The current input element of A
carriedSum	DFEComplex	The accumulated sum calculated from previous loop
newSum	DFEComplex	The newly generated accumulated sum in current loop
division	DFEComplex	The dividend that will be divided by newSum
lcontent	DFEComplex	The value to be output and to be written to the FMem **l**
ucontent	DFEComplex	The value to be output and to be written to the FMem **u**

The implementation idea is that when the counters **loopCounter** and **loopLengthCounter** both count to 0,which means a new 11 loops begins, an element of A will be read into the kernel and stored in the variable **AIn**. As a supplement, the input stream of Matrix A will be read in as the following illustration:



At the same time, the variable **carriedSum** is initialized with the 0, and when the **loopCounter** is smaller then **z**, the carriedSum is added by **l**[**x**][**loopCounter**] $*$ **u**[**loopCounter**][**y**] , and this is achieved by reading value from FMem **l** and **u** with address **x*11+loopCounter** and **y+loopCounter*11** respectively, otherwise 0, here is the illustration:

When the **loopCounter** equals 10 and **loopLengthCounter** equals **loopLengthVal-1**, which

Figure 5.3: x=1,y=1,loopCounter=0

(a) Element to be read

$L_{0,0}$	$L_{0,1}$	$L_{0,2}$	...	$L_{0,10}$
$L_{1,0}$	$L_{1,1}$	$L_{1,2}$	...	$L_{1,10}$
$L_{2,0}$	$L_{2,1}$	$L_{2,2}$	...	$L_{2,10}$
...	...	...	...	...
$L_{10,0}$	$L_{10,1}$	$L_{10,2}$	...	$L_{10,10}$

(b) Corresponding address

0	1	2	...	10
11	12	13	...	21
22	23	24	...	32
...	...	...	...	...
110	111	112	...	120

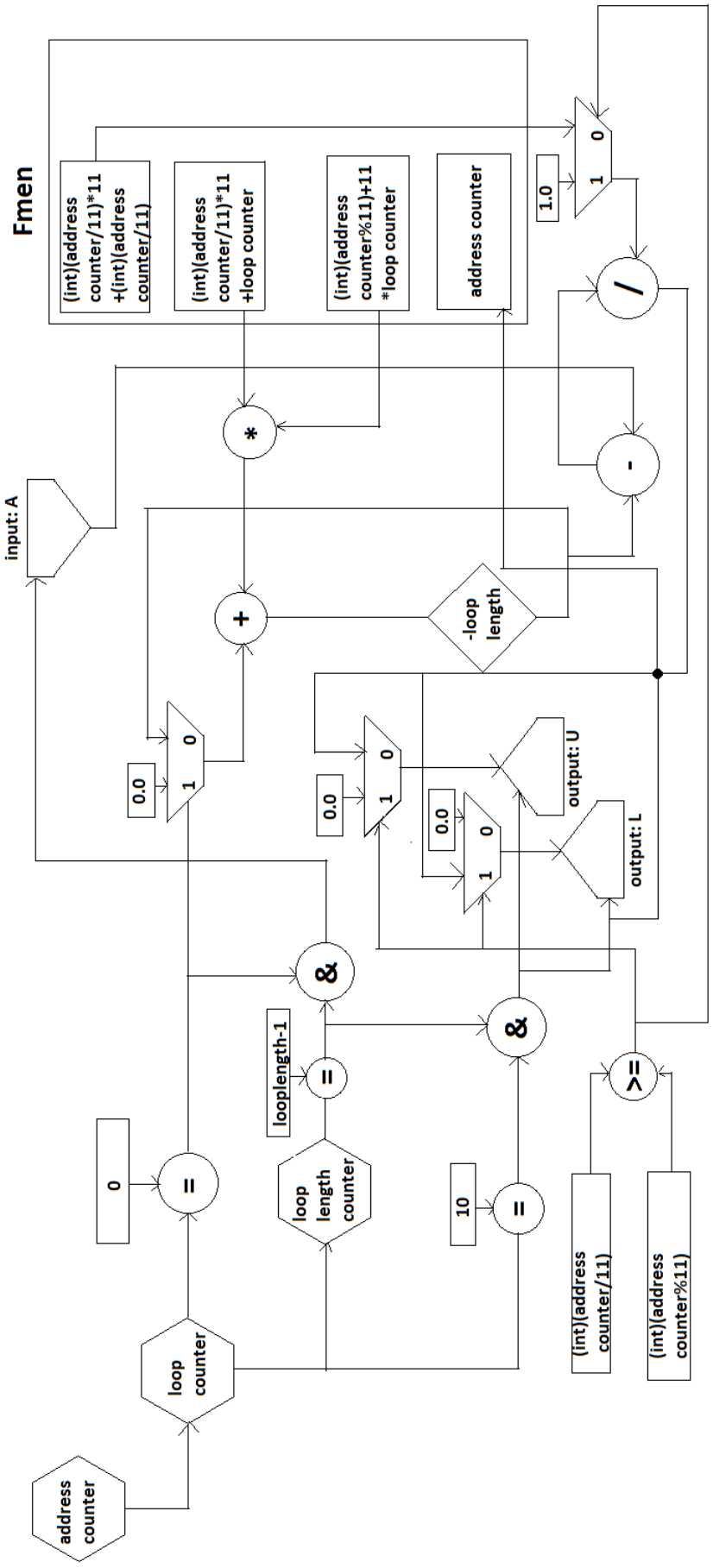
means the totally 11 loops have finished, then we need to decide the result should belong to **l** or **u**, and whether the result should be divided by **l**[**x**][**x**], and this can be achieved by comparing **x** and **y**, if **x**>=**y**, then the result belongs to lower triangle, otherwise the result belongs to upper triangle and should be firstly divided by **l**[**x**][**x**]. For now, a whole 11 loops is finished, and then the **addressCounter** will increments by 1 and a new element in matrix A should be read in and processed. After all the 121 elements have been processed then the decomposition is finished.

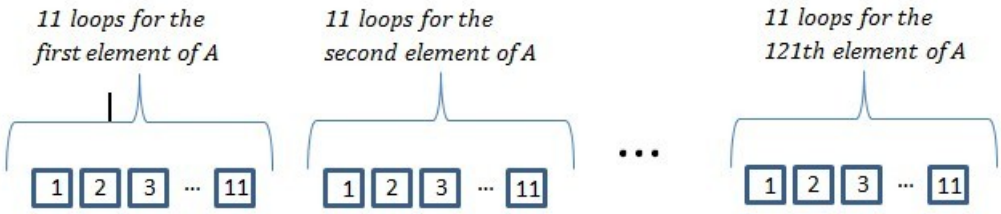
*As a result, we achieved a total ticks of 121*11*128 for implementing a LU decomposition using a dataflow kernel. Where 121 is the size of the matrix A to be decomposed, 11 is the 11 loops for processing each element in A, and 128 is the loopLength, namely, the ticks required for executing one loop.*

5.2.3 Pipeline Implementation (LU Pipeline 1)

Although the multi-tick implementation works correctly, it still can be improved a lot. Specifically, it did not make use of the strength of Maxeler which is in a dataflow engine processing pipeline every dataflow core(arithmetic unit)computes simultaneously on neighboring data items in a stream. It is because for the previous implementation, the computations look like follows:

Figure 5.4: Kernel Design





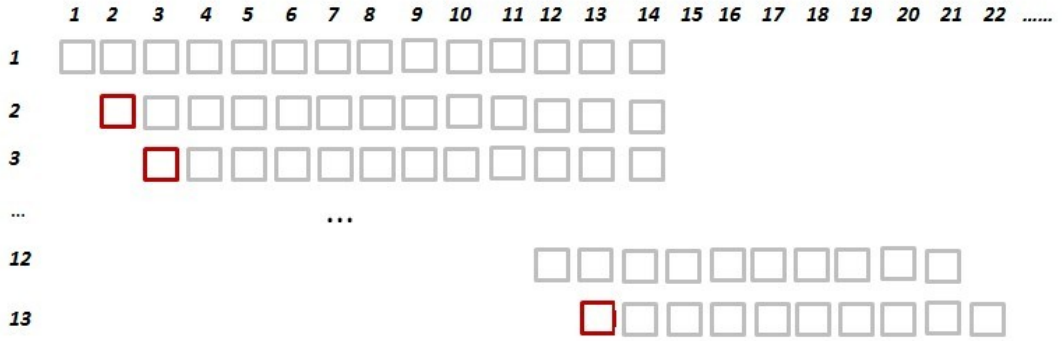
No computations are performed simultaneously. In addition, due to 11 is the minimum loop times for all the elements to be processed correctly, for most of the elements the required loop times are less than 11, which means there are a lot of loops are unnecessary and doing meaning-less computations.
In order to make the non-pipelined computations pipelined, a new implementation design was proposed, which is to read in and start processing an element of matrix A every $1 * loopLength$ instead of the previously $11 * loopLength$, which means a new element was read in every loop but not every 11 loops. The feasibility of this design can be shown from the follows:

Figure 5.5: Feasibility Analysis

(a) Required Loop Number

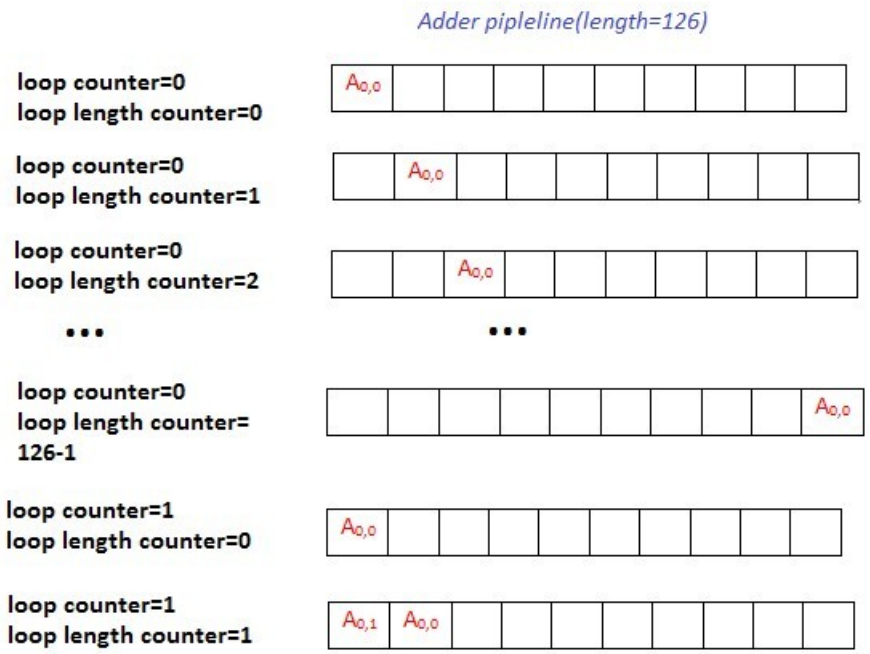
0	0	0	...	0
0	1	1	...	1
0	1	2	...	2
...	...	...	...	...
0	1	2	...	10

(b) Dependency Analysis



From `Figure 3(a)` we can know that the required loop number for processing an element is increasing(not strictly increasing) in a row, we always have $loop\_no[x][y] <= loop\_no[x][y+1]$. Refer to `Figure 3(b)`, this is a diagram showing the dependency during the computation of each element. We use gray squares to denote that the element has been processed, in other words, the element has been decomposed, and we use red square to denote that there are dependencies exist during the computation for this element. In addition, the horizontal axis is the total loop numbers and the vertical axis is the index of the element. Take some elements' computations as examples to show what are the dependencies, for the `2nd` element, a dependency exists during the first loop is due to `u[x][y]=sum[a]/l[x][x]`, where l[x][x] in this case is `l[0][0]`. Therefore, due to the non-pipelined algorithm, `l[0][0]` should have already been generated, which means we can only perform this computation when the computation for `l[0][0]` has been finished. And dependency also occurs for another reason, and take the the `13th` element as an example, due to its `x = 1` and `y = 1`, its `z also equals 1`, which means sum accumulation is involved in decomposing the

13th element, see sum[a]=sum[a]+l[x][i]*u[i][y]. In this case x = 1, y = 1, i = 0 so that only when l[1][0] and u[0][1] have already been calculated by decomposing the original matrix, performing this calculation can get the correct result.
After checking Figure 3(b), all the calculations on which the later computations depends finish before the later computations, which means if the computations are in performed in the above way then still correct results can be generated but in a pipelined way.
To realise this in Maxeler, we can make modifications on the previous multi-ticks implementation. And in details, we need to change the input behavior from reading in an element from matrix A every 11 loops to reading in an element from matrix from A every 1 loop. Using the same data dictionary Table 1, this can be achieved by changing the input condition from loopCounter.eq(0)& loopLengthCounter.eq(loopLengthVal - 1) into loopCounter.eq(loopLengthCounter). We can find this works from the following illustration:



Another necessary modification is that, due to this is a pipelined implementation, the results should be written and output once they've finished their required computations, and this can be achieved by adding one more variable:

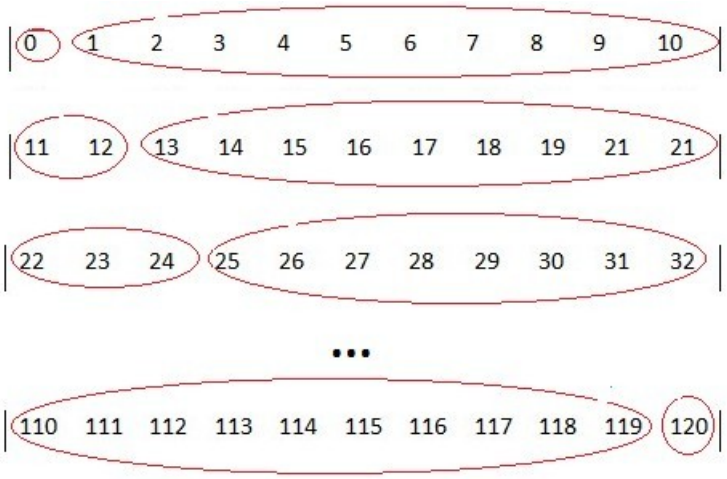
Table 5.2: Data Dictionary

Variable Name	Data Type	Usage
i	dfeUInt(7)	This variable is used for determining whether the required computations are finished

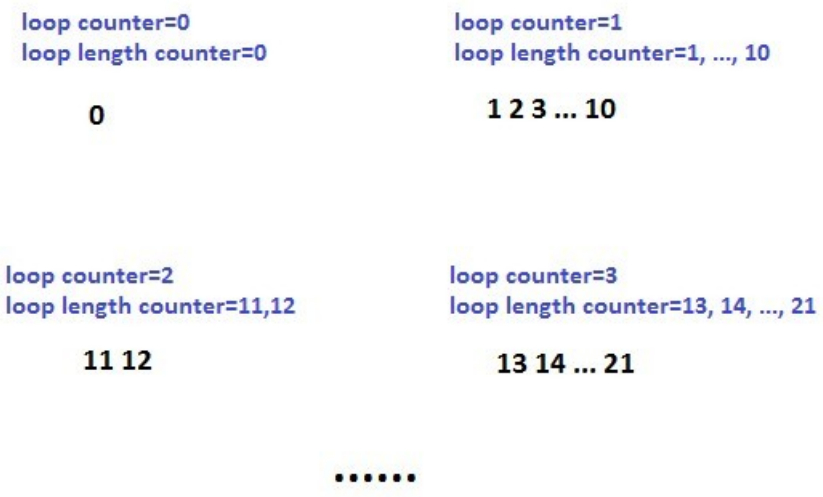
Once z equals i, the result should be written into the corresponding FMem so that the later computation won't be int.

5.2.4 Another Pipeline Implementation (LU Pipeline 2)

For now we've had a pipelined implementation. However, after consideration, the implementation can be further pipelined.For example, we may group this way, here elements in a same group will be processed in parallel, see the following illustration:



The procedure is as follows: For each group, start a new loop and all the members will be read in during the loop every tick. Therefore, for the above case, `0` will be read in and processed in `loop 1`, `1,2,3, ..., 11` will be read in `loop 2` and so on.
The above procedure works if the first column of the matrix is given. We may conduct the following analysis to show this. For group{0}, there is no dependency, for group{1, 2, 3, ..., 10}, as long as `element 0`'s computation has finished, they can be processed in parallel. For goup{11, 12}, `element 12` have to wait until `element 11`'s computation is finished. However, as long as the first column of the matrix is given so that the decomposed `element 11` is known due to there is no computation needed for the first column of matrix to be decomposed, then the group{11,12} can be computed in parallel. Here `element 12` may also need `element 1`'s computation to be finished, and this is true due to grounp{1,2,3, .., 10} start their computation at loop 2, due to they need 0 loop to finish the decomposition, when the time comes to loop 3, the group{1, 2, 3, ..., 10} have already finished decomposition and thus group{11,12} can perform their decompositon without waiting. And if we check exhaustively, we can find this applies to all the other groups.
Therefore, in order to achieve this implementation, we need the first column of the matrix and group the elements in the above way. The elements should be read in in at the following times:

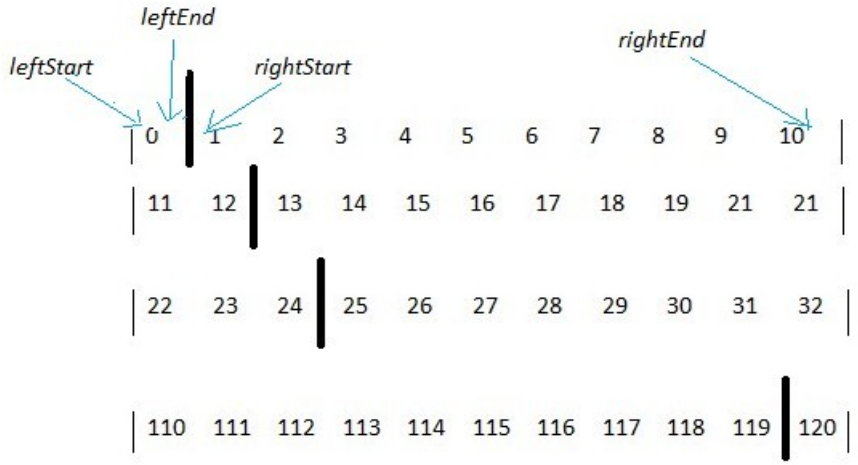


To achieve this, we need to add few more variables:

Table 5.3: Data Dictionary

Variable Name	Data Type	Usage
Ceven	dfeUInt(1)	If the value of the loop counter is even then this variable is true
Codd	dfeUInt(1)	If the value of the loop counter is odd then this variable is true
leftStart	dfeUInt(8)	Please refer to `Figure 4`
leftEnd	dfeUInt(8)	Please refer to `Figure 4`
rowCounter	dfeUInt(8)	This counter's value denotes the current row.

Figure 5.6: Illustration about start point and end point



After some trials, I found that the the value for the start point and end point for the left half can be calculated using the following formulae:

When loop_counter is even

```
left_start_point=loop_counter/2*11
left_end_point=loop_counter/2*12+1
```

And the start and end points for the right half follows:

When loop_counter is odd

```
right_start_point=(loop_counter-1)/2*12
right_end_point=(loop_counter+1)/2*11
```

Therefore, when the `loop counter` and `loop length counter` satisfies that,

$$Ceven\&leftStart <= loop\_counter\&loop\_counter < leftEnd)|(Codd\&rightStart < loop\_counter\&loop\_counter < rightEnd)$$

then an element of matrix A should be read in.
In order to make the first column of the matrix be known, we can directly pass them into the kernel by using these two variables, see `Table 4`:

As a result, there are totally 30 loops(3(n-1), and here n =11), therefore we achieved a total latency of 30*126+121.*

Table 5.4: Data Dictionary

Variable Name	Data Type	Usage
firstColumnR	Memory	The real value of the first column of the matrix is mapped directly to the kernel
firstColumnI	Memory	The imaginary value of the first column of the matrix is mapped directly to the kernel

5.2.5 Machine Parallelism

There is one more part in our program that needs acceleration. And the code has following structure:

```
for(int it=0;it<=3;it+=1){
          //Delta
          delta[1][it] = delta[0][it]+(omega[0][it] - Omega_synch)
          * stepSize;
         //Eq_dash

          eq_dash[1][it] = eq_dash[0][it]+(
          (-1) * eq_dash[0][it] - (X_d[it] - X_d_p[it])*(
          (-1) * I_d[0][it] - (X_d_p[it] - X_d_dp[it]) /
          (X_d_p[it] - X_ls[it])/
         (X_d_p[it] - X_ls[it])*(
          psid[0][it] - (X_d_p[it] - X_ls[it]) * I_d[0][it]
          - eq_dash[0][it]
          )
         ) + efd[0][it]
         ) / T_do_p[it] * stepSize;
          //Psid

          psid[1][it] = psid[0][it]+((-1) * psid[0][it] + eq_dash[0][it]
        + (X_d_p[it] - X_ls[it])
        * I_d[0][it]) / T_do_dp[it] * stepSize;//checked
          //Ed_dash

         ed_dash[1][it] = ed_dash[0][it]+(ed_dash[0][it] +
          (X_q[it] - X_q_p[it])*
          (
          I_q[0][it] - (X_q_p[it] - X_q_dp[it]) / (X_q_p[it] - X_ls[it])/
         (X_q_p[it] - X_ls[it])*(
          (-1) * psiq[0][it] + I_q[0][it] * (X_q_p[it] - X_ls[it])
          - ed_dash[0][it]
          )
          )
         ) / (-1) / T_qo_p[it] * stepSize;
}
```

After referring to the maxcompiler-loops tutorial, we found that this part of code has the same structure as the following code:

```
for (int count=0; ; count += 1) {
B[count] = A[count] + 1;
}
```

The only difference is that the computations are different. Therefore, we may implement this part in the same way. However, there is a restriction that the input stream cannot over 8 streams, but we have totally 28 inputs to be streamed. In order to solve the problem, DFEArrayType was utilized. And as a reuslt, we just need one string with length of 28. In addition, the code looks like:

```
DFEComplexType t=new DFEComplexType(dfeFloat(11,53));
                    DFEArrayType<DFEComplex> at=new
                            DFEArrayType<DFEComplex>(t,28);
                    DFEArray<DFEComplex> input=io.input("input",at);
                    DFEComplex deltaOut=input[11].getReal()>0?
                    input[0]+(input[1]-377)*0.001:
                    input[0];
                    DFEComplex eq_dashOut=input[11].getReal()>0?input[2]+(
                            (-1) * input[2] - (input[3] - input[4])*(
                            (-1) * input[5] - (input[4] - input[6]) /
                    (input[4] - input[7])/(input[4] - input[7])*(
                            input[8] - (input[4] - input[7]) * input[5]
                     - input[2])
                            ) + input[9]
                            ) / input[10] * 0.001:input[2];
```

5.3 Summary

In Summary, this chapter firstly show the profiling result which determined the area of code to be accelerated by FPGA (LU Decomposition and Machine update operations). Next, in order to design the DFE kernel for LU decomposition, we solved some problems such that, determine Input and output streams and memory pattern. Most importantly, we solved the problem of representing complex numbers in Maxeler. After these, we gave 3 design of LU decomposition and one design for machines pipeline so that the number of machines can do update operations synchronously.

Chapter 6

Evaluation

6.1 Expected Performance

The first component in estimating performance in dataflow computation is the bandwidth in and out of the dataflow graph. For data in DFE memory, we simply look up the bandwidth of the particular device and memory storing the data. The second component is the speed at which the dataflow pipeline is moving the data forward. A unit of time in a DFE is called a tick or (a cycle), and the speed of movement through a dataflow pipeline is given in [ticks/second].

Bandwidth can be thought of as the numbers per second that can be read into or written out from the DFE chip. The bandwidth of DFEs can be between 200-1000 million numbers per second depending on the size of the numbers and the speed of the interconnect (LMEM, PCIe, Infiniband, or MaxRing).The clock frequency is how many ticks per second the kernels can run at. The frequency is between 100-300 million ticks per second as determined and displayed during the DFE compilation process,

In our implementation, We did not use any LMem or MaxRing. We used FMem for storing values and computation which provides up to 21TB/s of memory bandwidth within in chip. We should also consider the speed of the interconnection PCIe for transferring data from CPU to FPGA. We denote the time of transferring data between the CPU and FPGAs as T_{PCIE}, and the time for reading/writing to FMem as T_{FMEM}. However, this term is usually negligible as accessing to the on-chip FMem is very fast compared to other memory patterns. For the computation time on DFEs $T_{COMPUTE}$, we have:

$$T_{COMPUTE} = \frac{Ticks}{Clockfrequncy} \tag{6.1}$$

If T_{ACCEL} is the total time used of the accelerated program,

$$T_{ACCEL} = T_{INIT} + T_{STREAM} \tag{6.2}$$

where $T_{STREAM} = MAX(T_{COMPUTE}, T_{PCIE})$, and T_{INIT} includes SW time, and also time to set scalar inputs, ROMs, prepare streams etc. Assume the data size of transfer (between CPU and FPGAs) is depend on the number of the buses N.

$$T_{PCIE} = \frac{BYTES\_IN_{PCIE} + BYTES\_OUT_{PCIE}}{BW_{PCIE}} \tag{6.3}$$

Where PCIe bandwidth depends on transfer size that can sustain >2GB/s.

6.1.1 LU Pipeline 1

In our first version of pipelining,the Input Data to Kernel is the N*N Matrix A of complex numbers (16 bytes) and a N*N matrix in of double numbers(8 bytes). And the outputs are two N*N matrixes L and U contain complex numbers. So:

$$\begin{aligned} T_{PCIE} &= \frac{(16+8)N^2 + 2*16*N^2}{BW_{PCIE}} \\ &= \frac{56N^2}{BW_{PCIE}} \end{aligned} \tag{6.4}$$

And

$$Ticks = [(N^2-1)+(N-1)] * looplength + N \tag{6.5}$$

Where $looplength$ is the number of ticks for one loop's calculation in our kernel.

6.1.2 LU Pipeline 2

In our second version of pipelining,the Input Data to Kernel is not only the N*N Matrix A of complex numbers (16 bytes) and a N*N matrix in of double numbers(8 bytes) but also the first column of matrix A which is of size N. And the outputs are the same. So:

$$\begin{aligned} T_{PCIE} &= \frac{(16+8)N^2 + 16*N + 2*16*N^2}{BW_{PCIE}} \\ &= \frac{56N^2 + 16N}{BW_{PCIE}} \end{aligned} \tag{6.6}$$

And

$$Ticks = 3(N-1)*128 + N^2 \tag{6.7}$$

6.1.3 Machine Parallelism

As we know that, the very important factor of speed is the operation volume within the kernel and the data transfer time. In the machine parallelism, the input and output streams are dependent on how many parameters to be updated. Let say, the number of parameters to be imported into the kernel and D_{in} and D_{OUT} for output. And number of the update operations all together is O_p. As we assume the data type of the parameters are all complex numbers. So,

$$T_{PCIE} = \frac{16*(D_{in}+D_{out})}{BW_{PCIE}} \tag{6.8}$$

And

$$Ticks = O_P \tag{6.9}$$

6.2 Experimental Evaluation

6.2.1 General Settings

We use the MPC-C500 reconfigurable accelerator system from Maxeler Technologies for our evaluation. The system has 4 MAX3434A cards, each of which has a Virtex-6 XC6VSX475T FPGA. The cards are connected to 2 Intel Xeon X5650 CPUs and each card communicates with the CPUs via a PCI Express gen2 x8 link. The CPUs have 12 physical cores and are clocked at 2.66 GHz. We develop the FPGA kernels using the MaxCompiler which adopts a streaming programming model and its supports customisable floatingpoint data formats. And the clock frequency is set to 100.

6.2.2 Test the LU Decomposition Kernel

The data we use is from the .m files provided by Dr Chaudhuri, e.g., The matrix size is corresponds to the number of the buses. The name of the matrix is Ybus. For lxample, if we have 11 buses the matrix is 11*11. so N=11.

We firstly tested the running time of the pure C code using on Maxeler Super Computer *maxnode2*. And of course, we only cut out the time running on LU decomposition process. For 20 seconds' simulation, it only took **0.134s** on average to run.

```
The processing_time is 10 microseconds
The processing_time is 9 microseconds
The processing_time is 10 microseconds
The processing_time is 11 microseconds
The processing_time is 5 microseconds
The processing_time is 6 microseconds
The processing_time is 6 microseconds
The processing_time is 5 microseconds
The processing_time is 5 microseconds
The processing_time is 6 microseconds
```

Figure 6.1: Runing time for LU decomposition in C (listed by each iteration)

As we can see from Fig. 6.1, which lists the running time of LU Decomposition in each iteration. So that we can approximate the running time of doing LU Decomposition once in C code (T_c) by taking an average value:

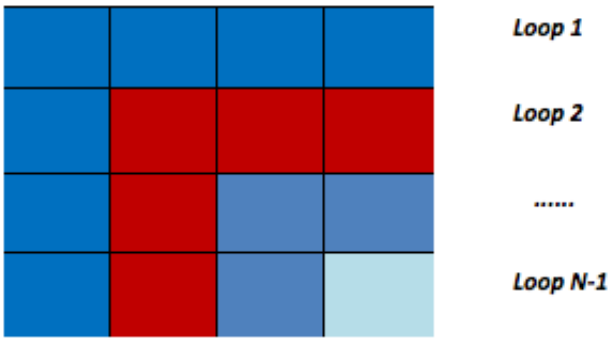
$$T_c = 7 \text{ microseconds (ms)}$$

As we know, the original LU decomposition algorithm has a time complexity $O(N^3)$. To estimate the speed up. We should know exactly how is the computation time of our C code depends on the size of N. However, as we do not have data other than the used $Ybus$ data provided, we can not do more experiment. Fortunately we can estimate this relation by derive the relation of the number of operation steps and the size of N.
As shown in the figure above. The LU decomposition algorithm processes the matrix in a sub-matrix by sub-matrix way. That is it will first do calculates on the elements on left-most layer, say, loop 1 in the figure. we denotes the number of operations of the matrix of size N $O_p(N)$, we have,

$$O_p(N) - O_p(N-1) = (N-1) + (N-1)^2 + (N-1)^2$$

$$O_p(N-1) - O_p(N-2) = (N-2) + (N-2)^2 + (N-2)^2$$

...



$$O_p(2) - O_p(1) = 3$$

Where $(N-1) + (N-1)^2$ is number of multiplications and $(N-1)^2$ is for subtraction. So when we add them together, we can get $O_p(N)$. By the principle of arithmetic progression, and we know $1^2 + 2^2 + 3^2 ... + n^2 = n(n+1)(2n+1)/6$, we can get,

$$O_p(N) = \frac{(N-1)^2}{2} + \frac{(N-1)(N)(2N-1)}{3}$$

And then, assuming the one subtraction and one multiplication use the same time in CPU, we can estimation the time of doing each operation, we call it T_{unit}, by using the experimental result of our power system. As we has 11 buses, So Op=820. And as the total computation time is 7 ms. We can average it and get $T_{unit} = 0.085ms$. Therefore, the computing time for the C program, T_C can be estimated as:

$$T_C = T_{unit} * Op = 0.0085 * \frac{(N-1)^2}{2} + \frac{(N-1)(N)(2N-1)}{3}$$

LU Pipeline 1

By only recorded the time speeding on LU decomposition. Our pipelined version of code using the kernel we firstly designed takes average **50s** to simulation 20s' calculation. Compared to the **0.13s** used by the pure C code, it is really a very bad result. As it slows down the program by about 500 times. However, actually this result is not in-apprehensible and it makes senses:

Firstly,we know the design of my kernel takes $[(N^2-1)+(N-1)]*looplength+n$ ticks, where N=11, $looplength$=128, so it is $T_{COMPUTE}$ = 130*128+121/100MHz=16761/100MHz = 0.00016 seconds = 160ms. And for a simulation of 20 seconds which means 20000 times' calls to the *LUSolver* function, The theoretical computing time on the LU Solver kernel should be: 0.000016*20000= 3.2s.

And $T_{PCIE} = \frac{56N^2}{BW_{PCIE}}$= 3231ms, by taking BW_{PCIE}=2GB/s as the worst case. This value is much bigger than the $T_{COMPUTE}$ = 160ms which means that is the Data transfer time dominates the computation time. So $T_{STREAM} = T_{PCIE}$. Let us take a look at the running time for each iteration in Fig. 6.2:

It is surprised to see that the first iteration takes much longer time than the remaining iterations and is also much(about 10 times) bigger than T_{STREAM}=3231ms. This is because that for the fist call of the Kernel, the maxeler system should do some initialization work. And we know T_{ACCEL}=T_{INIT}+T_{STREAM}, we can only assume T_{STREAM} dominates for large data volumes. However, the data we use now has too small data size so that the initialisation dominates. And another problem is that, the calculation volume is too small (only 16761 ticks), either. As we can see if the $T_{COMPUTE}$ = 160ms, the data transfer time $T_{PCIE} \approx 850ms$ is much bigger.

```
The processing_time is 31408 microseconds
The processing_time is 1023 microseconds
The processing_time is 981 microseconds
The processing_time is 954 microseconds
The processing_time is 851 microseconds
The processing_time is 757 microseconds
The processing_time is 892 microseconds
The processing_time is 926 microseconds
The processing_time is 961 microseconds
The processing_time is 864 microseconds
```

Figure 6.2: Runing time for LU decomposition in Kernel (listed by each iteration

As we know that in order to use Maxeler kernels to perform calculations, we need to transfer data from CPU to kernels and this will cause a big overhead, which means if the scale of the computation on the transferred data is not large enough then the data transfer overhead will become very obvious and even makes the program slower. Unfortunately, our program is just the case where the scale of computation is not large enough.While FPGAs would speed up the matrix decomposition, they would need to wait for the inputs to be refreshed, and the additional latency over PCIe would become apparent over a million iterations' run.

LU Pipeline 2

The second implementation takes average **50s** to simulation 20s' calculation.This design still can not give any acceleration is because

Firstly,we know the design of my kernel takes $3(N-1) * looplength + n^2$ ticks, where N=11, $looplength$=128, so it is $T_{COMPUTE}$ = 130*128+121/100MHz=3961/100MHz = 0.00004 seconds = 40ms. The computation volume shrinks as we further pipeline the kernel.

And $T_{PCIE} = \frac{56N^2+16N}{BW_{PCIE}}$= 3692ms, by taking BW_{PCIE}=2GB/s as the worst case. This value is much bigger than the one in the previous implementation since we input an extra column of matrix A to FPGA every time call the LU Solver. That is to say, with the computation volume shrink but even more Data to transfer. This won't give any acceleration.

6.2.3 Machine Parallelism

The results for only running the Machine Parallelism kernel is shown below, and the total time on the kernel for 20 seconds' simulation is about 22s.

For our system, because the update process is interdicted by the calling of LU Solver function.

```
The processing_time is 28840 microseconds
The processing_time is 669 microseconds
The processing_time is 726 microseconds
The processing_time is 718 microseconds
The processing_time is 660 microseconds
The processing_time is 595 microseconds
The processing_time is 568 microseconds
The processing_time is 476 microseconds
The processing_time is 491 microseconds
The processing time is 571 microseconds
```

we have to pipe line the upper and lower parts of operations separately. For the first part kernel design, the input Data to Kernel is an Array Type of Complex numbers with a length of 28. an

Array Type of Complex numbers with a length of 6 is the output stream. So:

$$T_{PCIE} = \frac{(28+6)*16}{BW_{PCIE}} = \frac{544}{BW_{PCIE}} \tag{6.10}$$

if we take BW_{PCIE}= 2GB/s, T_{PCIE}=544/1024/1024/2*1000000=519ms.
And as tested from the maxeler system, we know

$$Ticks = 25 \tag{6.11}$$

$T_{COMPUTE}$ = 25/100MHz=0.00000025 seconds = 0.25ms.

It is also clear to see, the time is mostly occupied by the data transfer time.

6.3 Analysis

In conclusion, Maxeler is good at large scale and regular behavior computing, however, in our model we only have small scale computations. It seems that the model I have got now is not worth being put into Maxeler. Fortunately, our objective of this project is not for accelerating this particular power system simulation, but to explore how FPGAs can be used to accelerate power systems. In reality, the power system simulation should deal with larger scale of computation (e. g. the Smart Grid with Electric Springs). In this section we will illustrate how can we extent our design to larger systems.

The key to achieving the best performance in FPGA acceleration, while maintaining correctness, is optimization of arithmetic units and data types to suit the range/precision at each point in the computation and minimise the data transferred at each compute step as well as perform as many computations on the data as possible before moving them back to memory and minimize the number of memory accesses.

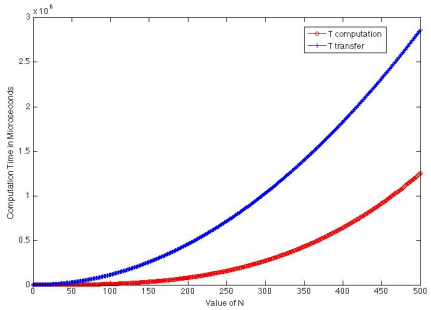
Here in our problem, the amount of computations on the data is represented by the value of $Ticks$, and the amount of data transfer depends on the value of N. We want the former as much as possible and the latter as less as possible. However, both of them depend on the value of N. When N increases, both of the computation volume and the data transfer volume will increase. We should balance these two volumes, only so the program will be accelerated. and the balancing point is when computation volume exceeds the data transfer volume. And we shall estimate the $speedup = T[C]/T[STEAM]$ with data size N for the both cases: ignore transfer time or not.

LU Pipeline 1

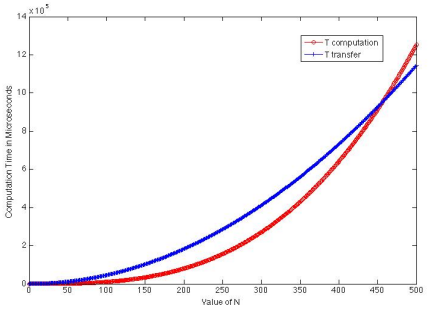
As we want the Computation time dominates the data transfer time ($T_{COMPUTE} > T_{PCIE}$), That is:

$$\{[(N^2-1)+(N-1)]*looplength+N\}/100 > 56N^2/1024/1024/BW*1000,000$$

In Fig. 6.3 (a) compares the $T_{COMPUTE}$ and T_{PCIE} when bandwidth is 2GB/s, and Fig. 6.4 (b) compares the two when bandwidth is 10GB/s. As we know bandwidth has very hauge influence on the time of data transfer. We can see that, when the bandwidth is the standard 2GB/s (which we consider), there is no possible for the computation time to exceed data transfer time. We can have a chance to ignore the overhead only if the bandwidth can reach some higher value. For example,



(a) Compare of Tcompute and Ttransfer at BW=2GB/s



(b) Compare of Tcompute and Ttransfer at BW=10GB/s

Figure 6.3: Compare of Tcompute and Ttransfer at Differet BW

when BD=10GB/s, the Kernel design will be efficient when the number of bus is larger then about 450.

Let us ignore the data transfer and only consider the design of the pipeline algorithm. By comparing the T_C and $T_{COMPUTE}$ in Fig. 6.5. we can see that when N, the data size is, as large as 225, The FPGA design should theoretically begin to show its superiority.

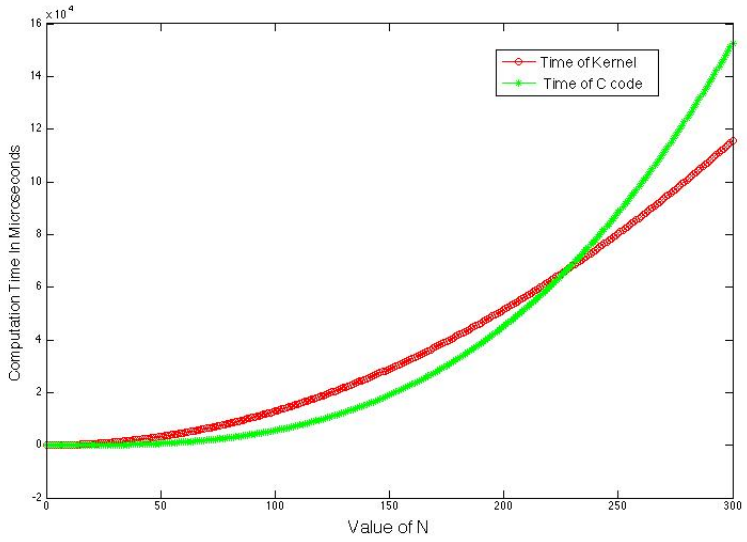
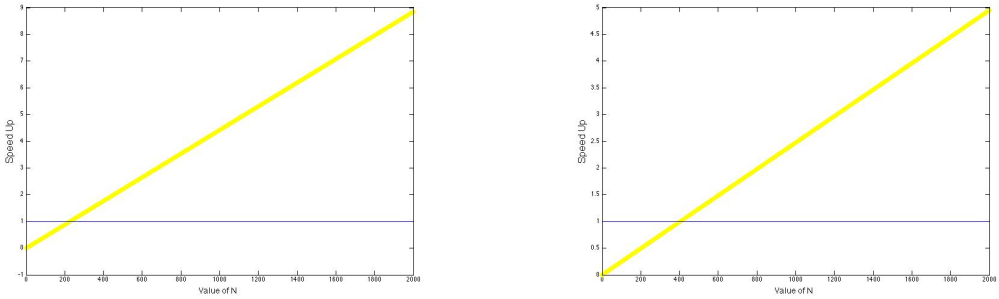


Figure 6.4: Compare of C code and Tcompute

We also show the Speedup, (T_C/T_{SREAM}), in two circumstances:1. Ignore transfer time, so that $T_{SREAM} = T_{COMPUTE}$. 2. Consider the data transfer time, so that $T_{SREAM} = T_{PCIE}$.

As we can see from Fig. 6.6, When data transfer time is ignored, the LU decomposition process can get almost 10 times speed up when the system has more than 20000 buses. And if data transfer time is considered. The performance of the FPGA design will get worse and can only be useful when data size is greater than 400, and can only obtain 5 times' speed up when data size gets to 20000.



(a) Speed Up When Transfer Time is Ignored

(b) Speed Up When Transfer Time is Considered

Figure 6.5: Speed Up with N

LU Pipeline 2

The same with the first design, if we want the Computation time dominates the data transfer time ($T_{COMPUTE} > T_{PCIE}$), That is:

$$\{3(N-1) * looplength + N^2\}/100 > (56N^2 + 16N)/1024/1024/BW * 1000,000$$

However, we can not achieve this for our second design. Although it has less computation ticks within the Kernel, It needs more data to be transferred from CPU to FPGA. And from Fig. 6.7 we can see, the T_{PCIE} rapidly increases with the N increases only by 1 and it is alway above the $T_{COMPUTE}$. Therefore, the data transformation overhead will always slow down our program. Therefore, as a conclusion, our second kernel design for LU Decomposition is not applicative to power system simulation.

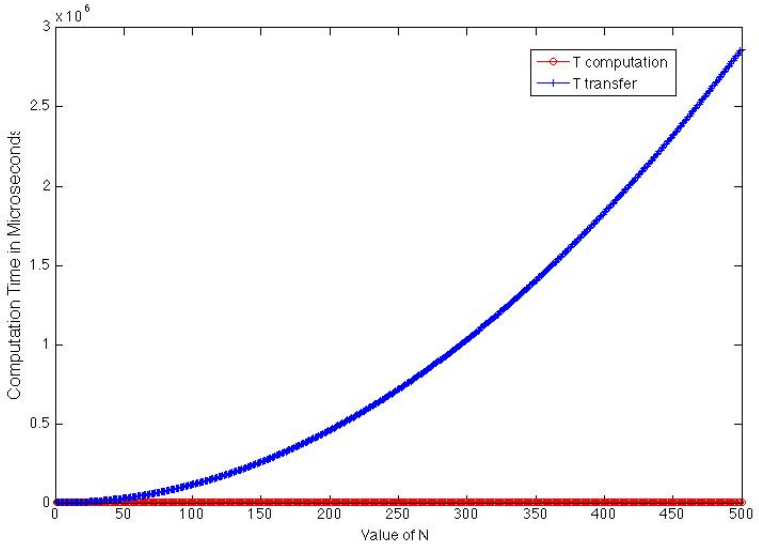
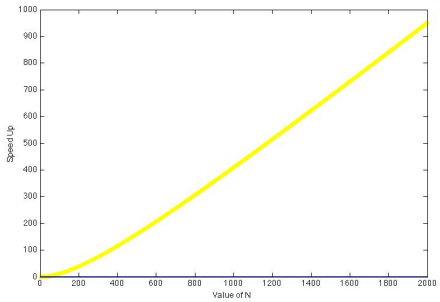
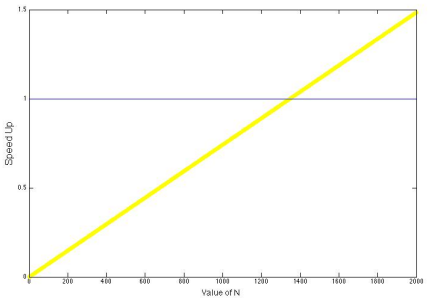


Figure 6.6: Compare of Tcompute and Ttransfer at BW=2GB/s

In a similar way, let us first ignore the data transfer and only the design of the pipeline algorithm. The speed up shown in Fig 6.8 (a) show a significant jump of acceleration with the increase of N. And it can get to 950 times speed up when data size is 20000. However, if we consider data transfer time, It can only get a slight shrink in computation time when the band width is 6GB/s and it is not adaptive to small systems. (Fig. 6.7(b))



(a) Speed Up When Transfer Time is Ignored



(b) Speed Up When Transfer Time is Considered

Figure 6.7: Speed Up with N

6.3.1 Machine Parallelism

For the Machine Parallelism design, as we can not get the relationship between the number of operations O_P and the computation volume $Ticks$, it is hard to estimate the speedup. However, this design is very promising if the operations scale gets larger.

6.4 Summary

According to the above illustration, we can get to the conclusion that although the experiments targeting Maxeler systems show that our FPGA-based design can not improve the time efficiency from the C application of the study system. We builds the relationship between the speedup of simulation and the data size of the power system which indicates that our acceleration design can give a significant acceleration to larger-scale larger power systems.

Chapter 7

Conclusion and Future Work

The object of this project is to explore how the huge computational power and memory optimizations of FPGA based hardware accelerators can be used in the dynamic simulation of power systems. And the study begins with the available power system simulation model, which deals with relatively simple structure and data size.

In detail, we firstly focused on a 4-machine, 2 -area study system for analyzing the dynamic behavior of different components in the power system provided by Dr Chaudhuri form his book Robust Control in Power Systems [25]. This is a relatively simple power system simulation without the use of electric springs but using the Multiple-model adaptive control (MMAC) approach for robust control.

With a SIMULINK system as the original version of the power system simulation, we did the follolwing things:

- Analyised the simulation model in SIMULINK. Proposed a method to trace the sorted order of different subsystems and blocks in SIMULINK. This was paramount in providing a base to begin migrating to a compiled language.

- An optimised version of the simulation in C language (chapter. We maximise the performance gains available without hardware acceleration by 500 times' faster than SIMULINK. We re-implemented everything from scratch so that have concluded an efficient scheme of converting SIMULINK diagrams to C(C++) language.

- An accelerated version of the LU decomposition solver used in the power system simulation. For this algorithm, we proposed 2 pipeline schemes with the Field Programmable Gate Array (FPGA) applied by the Maxeler Technologies. And we also tried to pipeline the calculations of the machines in the system (the 4 generators) . During this process, we successfully solved the problem of handling complex numbers in maxeler.

- Analysed the performance of the designed kernels. Although the DFE designs we proposed did not work very well on the simulation system given. We derived a relationship between the speedup of simulation and the data size of the power system which indicated that our acceleration design can give a significant acceleration to larger-scale larger power systems.

In conclusion, this project gave me a lot of challenges but I enjoyed it very much. I still have more idea of a extension of this project, but due to the limit of time. I can only list them here, and I may work then out later.

7.1 Future Trial

7.1.1 A New Design of LU Decomposition

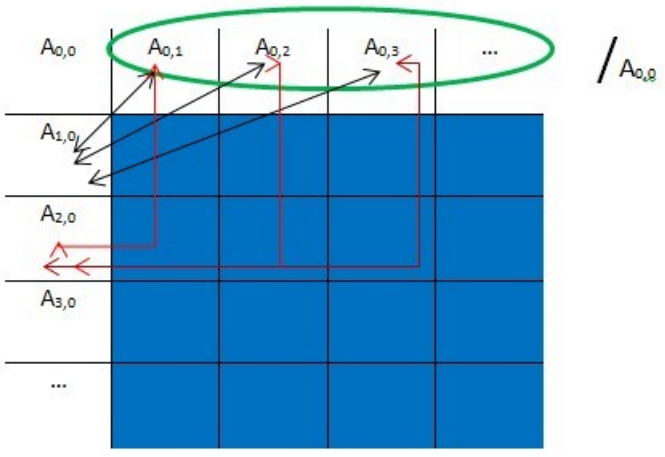
So far, our algorithms are all code driven, specifically, we analyzed the parallelism based on the code. However, the code, which in our case is written in C, is generated based on the sequentialism of normal processor. Therefore implementing algorithms based on parallelism may generate more possible speed-ups. After performing some research, in Choi's thesis [4], an algorithm was proposed:

```
Step 1: The column vector ax,1 where 2 <= x <= b is multiplied by the
reciprocal of a1,1. The resulting column vector is denoted lx,1.

Step 2: lx,1 is multiplied by the row vector a1,y(= u1,y) where
2 <= y <=b. The product lx,1*u1,y is computed and subtracted
from the submatrix ax,y where 2 <= x, y <= b.

Step 3: Step 1 and 2 are recursively applied to the new submatrix
formed in Step 2. An iteration denotes an execution of Step 1 and 2.
During the k-th iteration, the column vector lx,k and the row vector uk,
y where k + 1 <= x, y <= b are generated. The product lx,k * uk,
y is subtracted from the submatrix ax,y where k <= x, y <= b
obtained during the (k-1)th iteration.
```

The above can be illustrated as:



If the matrix is N*N, then(N-1)+ (N-1)*(N-1) multiplications are performed as shown in the above diagram, and then (N-1)*(N-1) subtractions are operated. However, all these can be computed in parallel, and then apply the same algorithm to the sub-matrix, which is the blue matrix in the above diagram. Thus the decomposition process will looks like this:

The implementation is feasible using Maxeler, and possibly further reduce the execution time required. However, due to the time constraints, we regard this as a future trial.

7.1.2 System Extension

- Assuming we need to carry out multiple times of this computation, could we include multiple copies of your design on chip so that they can support multiple independent computation in parallel? What is the maximum Can the current external memory support this? If not, just assume that the memory is fast enough and see how fast your design would run. As to now, we did not have time to think about it, I do want to explore this later.
- Extension to a more complex power system, e.g., the smart grid with electronic springs.

Bibliography

[1] Shrirang Abhyankar and Alexander J Flueck. Real-time power system dynamics simulation using a parallel block-jacobi preconditioned newton-gmres scheme. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 299–305. IEEE, 2012.

[2] Jonathan W Babb, Matthew Frank, and Anant Agarwal. Solving graph problems with dynamic computation structures. In *Photonics East'96*, pages 225–236. International Society for Optics and Photonics, 1996.

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[4] Seonil Choi and Viktor K Prasanna. Time and energy efficient matrix factorization using fpgas. In *Field Programmable Logic and Application*, pages 507–519. Springer, 2003.

[5] C Concordia. Ieee committee report on recommended phasor diagram for synchronous machines. *IEEE Transactions on Power Apparatus and Systems*, 88(11):1593–1610, 1969.

[6] Andreas Dandalis, Alessandro Mei, and Viktor K Prasanna. Domain specific mapping for solving graph problems on reconfigurable devices. In *Parallel and Distributed Processing*, pages 652–660. Springer, 1999.

[7] Rob Dimond, Sébastien Racaniere, and Oliver Pell. Accelerating large-scale hpc applications using fpgas. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pages 191–192. IEEE, 2011.

[8] EROCT. Electrical buses and outage scheduling, 2013. [Online; accessed 5-September-2013].

[9] Joseph H Eto and Robert J Thomas. Computational needs for the next generation electric grid. *Department of Energy*, 2011.

[10] Djalma M Falcão. High performance computing in power system applications. In *Vector and Parallel ProcessingVECPAR'96*, pages 1–23. Springer, 1997.

[11] Narain G Hingorani, Laszlo Gyugyi, and Mohamed El-Hawary. *Understanding FACTS: concepts and technology of flexible AC transmission systems*, volume 1. IEEE press New York, 2000.

[12] Lee W Howes, Oliver Pell, Oskar Mencer, and Olav Beckmann. Accelerating the development of hardware accelerators. In *Proc. Workshop on Edge Computing*, 2006.

[13] Zhenyu Huang and Jarek Nieplocha. Transforming power grid operations via high performance computing. In *Power and Energy Society General Meeting-Conversion and Delivery of Electrical Energy in the 21st Century, 2008 IEEE*, pages 1–8. IEEE, 2008.

[14] PJM Interconnection. Pjm statistics, 2013. [Online; accessed 5-September-2013].

[15] M Klein, GJ Rogers, and P Kundur. A fundamental study of inter-area oscillations in power systems. *Power Systems, IEEE Transactions on*, 6(3):914–921, 1991.

[16] David Koester, Sanjay Ranka, and Geoffrey Fox. Power systems transient stability-a grand computing challenge. *Northeast Parallel Architectures Center, Syracuse, NY, Tech. Rep. SCCS*, 549, 1992.

[17] Prabha Kundur. *Power system stability and control*, volume 12. , 2001.

[18] Aaron Richard Mandle. Fpga based hardware acceleration: A case study in protein identification, 2013. [Online; accessed 5-Octobet-2013].

[19] T Mathworks. Simulink getting started guide.

[20] MatlabWork. Control and display sorted order, 2013. [Online; accessed 5-September-2013].

[21] Oskar Mencer. Asc: a stream compiler for computing with fpgas. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(9):1603–1617, 2006.

[22] Oskar Mencer, Zhining Huang, and Lorenz Huelsbergen. Hagar: Efficient multi-context graph processors. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 915–924. Springer, 2002.

[23] Felipe Morales, Hugh Rudnick, and Aldo Cipriano. Balanced decomposition for power system simulation on parallel computers.

[24] University of Malaga. Simulink basics tutorial, 2013. [Online; accessed 5-Octobet-2013].

[25] Bikash Pal and Balarko Chaudhuri. *Robust control in power systems*. Springer, 2005.

[26] Oliver Pell, Lee W Howes, Kubilay Atasu, Olav Beckmann, and Oskar Mencer. Accelerating scientific computations using fpgas. In *The Advanced Maui Optical and Space Surveillance Technologies Conference*, volume 1, page 97, 2006.

[27] Oliver Pell and Oskar Mencer. Surviving the end of frequency scaling with reconfigurable dataflow computing. *ACM SIGARCH Computer Architecture News*, 39(4):60–65, 2011.

[28] Michael Ryabtsev and Ofer Strichman. Translation validation: from simulink to c. In *Computer Aided Verification*, pages 696–701. Springer, 2009.

[29] Peter W Sauer and MA Pai. *Power system dynamics and stability*. Prentice Hall Upper Saddle River, NJ, 1998.

[30] User's Guide Simulink. Version 7. *The MathWorks Inc*, 2009.

[31] Yong Hua Song and Allan Thomas Johns. *Flexible ac transmission systems (FACTS)*, volume 30. IET, 1999.

[32] Brian Stott. Power system dynamic response calculations. *Proceedings of the IEEE*, 67(2):219–241, 1979.

[33] Gilbert Sybille, Patrice Brunelle, Hoang Le-Huy, Lo-uis A Dessaint, and Kamal Al-Haddad. Theory and applications of power system blockset, a matlab/simulink-based simulation tool for power systems. In *Power Engineering Society Winter Meeting, 2000. IEEE*, volume 1, pages 774–779. IEEE, 2000.

[34] Isa Servan Uzun, Abbes Amira, and Ahmed Bouridane. Fpga implementations of fast fourier transforms for real-time signal and image processing. In *Vision, Image and Signal Processing, IEE Proceedings-*, volume 152, pages 283–296. IET, 2005.

[35] Wikipedia. Lu decomposition — wikipedia, the free encyclopedia, 2013. [Online; accessed 5-September-2013].

[36] Changyan Zhou and Ratnesh Kumar. Semantic translation of simulink diagrams to input/output extended finite automata. *Discrete Event Dynamic Systems*, 22(2):223–247, 2012.