UNIVERSITY OF SOUTHAMPTON
FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
School of Electronics and Computer Science

A Prototype Parallel Multi-FPGA Accelerator for SPICE CMOS Model Evaluation

by

Ahmed Maache

A thesis submitted for the degree of
Doctor of Philosophy

January 2011

ABSTRACT

Due to ever increasing complexity of circuits, EDA tools and algorithms are demanding more computational power. This made transistor-level simulation a growing bottleneck in the circuit development process. This thesis serves as a proof of concept to evaluate and quantify the cost of using multi-FPGA systems in SPICE-like simulations in terms of acceleration, throughput, area, and power. To this end, a multi-FPGA architecture is designed to exploit the inherent parallelism in the device model evaluation phase within the SPICE simulator. A code transformation flow which converts the high-level device model code to structural VHDL was also implemented. This flow showed that an automatic compiler system to design, map, and optimise SPICE-like simulations on FPGAs is feasible.

This thesis has two main contributions. The first contribution is the multi-FPGA accelerator of the device model evaluation which demonstrated speedup of 10 times over a conventional processor, while consuming six times less power. Results also showed that it is feasible to describe and optimise FPGA pipelined implementations to exploit other class of applications similar to the SPICE device model evaluation. The constant throughput of the pipelined architecture is one of the main factors for the FPGA accelerator to outperform conventional processors. The second contribution lies in the use of multi-FPGA synthesis to optimise the inter-FPGA connections through altering the process of mapping partitions to FPGA devices. A novel technique is introduced which reduces the inter-FPGA connections by an average of 18%.

The speedup and power efficiency results showed that the proposed multi-FPGA system can be used by the SPICE community to accelerate the transistor-level simulation. The experimental results also showed that it is worthwhile continuing this research further to explore the use of FPGAs to accelerate other EDA tools.

Declaration of Authorship

I, Ahmed Maache, declare that the thesis entitled: **A Prototype Parallel Multi-FPGA Accelerator for SPICE CMOS Model Evaluation**, and the work presented in it are my own, I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as listed in this thesis.

Signed: ______________________________
Date : ______________________________

Acknowledgements

First and foremost, I would like to thank both my supervisors, Dr Jeff Reeve and Professor Mark Zwolinski for their valuable help and guidance. Thanks to all members of the Electronics Systems and Devices Group at the University of Southampton for their help and support. Thanks also go to Dr Koushik Maharatna for his fruitful discussions. I would like to thank my family for their unconditional support and love.

Contents

List of Figures

List of Tables

Listings

List of Acronyms

ALAP As Late As Possible

ANSI American National Standards Institute

ASAP As Soon As Possible

ASIC Application Specific Integrated Circuit

BEE Berkeley Emulation Engine

BRAM Block Random Access Memory

BSIM Berkeley Short-channel IGFET Model

CAD Computer-Aided Design

CDFG Control-Data Flow Graph

CLB Configurable Logic Block

CMOS Complementary Metal Oxide Semiconductor

COBRA-ABS Column Oriented Butted Regular Architecture Algorithmic Behavioural Synthesis

CPU Central Processing Unit

DAG Directed Acyclic Graphs

DCT Discrete Cosine Transform

DFG Data Flow Graph

DMA Direct Memory Access

DNA Deoxyribonucleic Acid

DSP Digital Signal Processor

EDA Electronic Design Automation

FCCM FPGA Custom Computing Machines

FDS Force-Directed Scheduling

FF Flip Flop

FIFO First In First Out

FLOPS FLoating-point Operations Per Second

FM Fiduccia-Mattheyses

FP Floating-Point

FPGA Field Programmable Gate Array

FPU Floating-Point Unit

FSL Fast Simplex Link

FSM Finite State Machine

GA Genetic Algorithm

GMRES Generalized Minimal Residual Method

GPU Graphics Processing Unit

HLS High-Level Synthesis

HPC High-Performance Computing

HPRC High-Performance Reconfigurable Computing

I/O Input/Output

ICON Chipscope Integrated Controller

ILA Integrated Logic Analyser

ILP Integer Linear Programming

ILP Instruction Level Parallelism

IOB Input/Output Block

JTAG Joint Test Action Group

KCL Kirchoff's Current Law

KL Kernighan and Lin

KVL Kirchoff's Voltage Law

LU Lower/Upper Decomposition

LUT Lookup Table

MC Monte Carlo

MGT Multi-Gigabit Transceiver

MNA Modified Nodal Analysis

MOSFET Metal Oxide Semiconductor Field-Effect Transistor

MPI Message Passing Interface

NoC Networks-on-Chip

NP-hard Non-deterministic Polynomial time

NR Newton-Raphson Method

OpenMP The OpenMP API specification for parallel programming

PCB Printed Circuit Board

PLB Processor Local Bus

PSP Penn State Philips

PThreads POSIX -Portable Operating System Interface for Unix- Threads

RAMP Research Accelerator for Multiple Processors

RISC Reduced Instruction Set Computer

RTL Register Transfer Level

SA Simulated Annealing

SATA Serial Advanced Technology Attachment

SIMD Single Instruction Multiple Data

SoC System-on-Chip

SPARCS Synthesis and Partitioning for Adaptive and Reconfigurable Computer Systems

SPICE Simulation Program with Integrated Circuit Emphasis

SPO Signal Processing Object

TMD Toronto Molecular Dynamics

VHDL VHSIC Hardware Description Language

VHDL-AMS VHDL Analogue and Mixed-Signal extensions

VLIW Very Long Instruction Word

WR Waveform Relaxation techniques

Chapter 1

Introduction

1.1 FPGAs and High-Performance Computing

The term High Performance Computing (HPC) is used to describe systems with large computing capacity (teraFLOPS region), high data throughput, and complex network architecture (e.g Infiniband). HPC uses supercomputers and computer clusters to solve advanced computation problems. The supercomputing term refers to a subset within HPC which uses more powerful computers. According to the TOP500® Supercomputer Sites [1], cluster computing is now the most commonly used architecture in the highest performing systems. Cluster computing has become dominant mainly because of its cost-effectiveness. The architecture relies on standard processors from Intel and AMD, plus standard memory systems and interconnects such as Gigabit Ethernet. Applications are parallelised or partitioned into sections that can run as independent processes on multiple processors.

Regarding the wide spectrum of HPC applications, it is becoming harder for general-purpose CPUs (Central Processing Unit) to keep up with the demands for more computational power [2, 3]. They are running into memory bottlenecks and consuming increasing amounts of power, and dissipating large amounts of heat. Conventional processors also suffer from the increasing latency of using multi-layered caches. The increase in power consumption and heat dissipation is the result of increasing clock speeds of processors, which subsequently increases the costs of power usage and cooling [4].

A proposed solution to this new problem is to add Field Programmable Gate Arrays (FPGA) as built-in hardware accelerators to the clusters to boost their computational performance while reducing the power consumption significantly [5, 6, 7, 8, 9]. An FPGA is a semiconductor device that consists of an array of programmable logic elements, interconnects, and I/O (Input/Output) blocks which are user configured to implement complex digital circuits. A number of high-performance system designers have begun exploring the capabilities of FPGAs [10]. This trend highlighted the importance of using multi-FPGA systems in the domain of algorithm acceleration [11]. HPC applications are usually very large algorithms and cannot be fitted on a single FPGA, so these applications are partitioned amongst a number of FPGAs that are connected in a network [12].

FPGAs are reconfigurable hardware devices that can be finely optimised under software control to run applications efficiently. Rather than implementing applications in software, FPGAs allow the execution of applications at near ASIC (Application Specific Integrated Circuit) speeds without the extremely high cost of creating custom silicon. In addition, FPGAs have the ability to exploit the inherent parallelism in the algorithms being implemented [13, 14]. The internal FPGA architecture can be finely tuned to particularly exploit a certain application parallelism unlike conventional processors which are designed to suite broader range of applications. FPGAs also provide higher performance with their high memory bandwidth and hardware parallelism. A processor would need to execute a number of instructions before it can access the data from the memory. However, an FPGA is a hardware circuit that can be connected directly to a system bus, which gives it a direct access to the memory system.

Algorithm parallelism can be addressed at different levels. HPC applications can be structured for multi-thread execution in parallel across a cluster of processors. This level is known as coarse-grain parallelism. Coarse-grained parallelism is usually specified manually using a set of compiler directives at the input source level (e.g. threads, Message Passing Interface MPI). Another level of parallelism would be to execute a number of instructions simultaneously, which is called fine-grain parallelism. The fine-grained parallelism can be extracted automatically from the behavioural descriptions through the synthesis process. Conventional processors also support this type of parallelism; but FPGAs provide much deeper pipeline than conventional processors and can execute much larger number of instructions simultaneously [14]. Processors usually have to use their own built-in functional units to perform computations, however, FPGA designs can be finely customised and pipelined to a much deeper degree due to their internal reconfigurability features.

Benchmarks in [8] showed that an FPGA operating at a frequency of 200 MHz running the Black Scholes financial simulation [15, 16] can outperform a 3 GHz processor by an order of magnitude or more, while consuming only quarter of the power. In another example, FPGAs have shown 185 to 250 times acceleration in running sequential alignment algorithms for Deoxyribonucleic Acid (DNA) sequences over conventional processors in [17]. An FPGA based accelerator for Monte Carlo (MC) simulation have demonstrated an acceleration of nearly 25 times over the software implementation as reported in [9].

In this work, we target FPGA devices due to their capabilities to provide fine-grain parallelism and reconfigurability. One of the main strengths of FPGAs in high-performance computing lies in the ability to reorganise the internal structure of a machine to feed data-to-data processes at high speed instead of forcing them to make continual memory requests [18]. This can make FPGAs much better at sustaining performance compared to processors, as they do not have to deal with the penalties of cache misses [14].

This made FPGAs an attractive hardware acceleration solution to be used for a wide variety of computing/power-hungry applications. One of the these applications would be to enable the Computer-Aided Design (CAD) community to accelerate Electronic Design Automation (EDA) algorithms and hence reduce the design time. The SPICE (Simulation Program with Integrated Circuit Emphasis) simulator is an example EDA tool which present a growing bottleneck in the development process. This thesis outlines the use of multi-FPGA systems in accelerating the SPICE simulator.

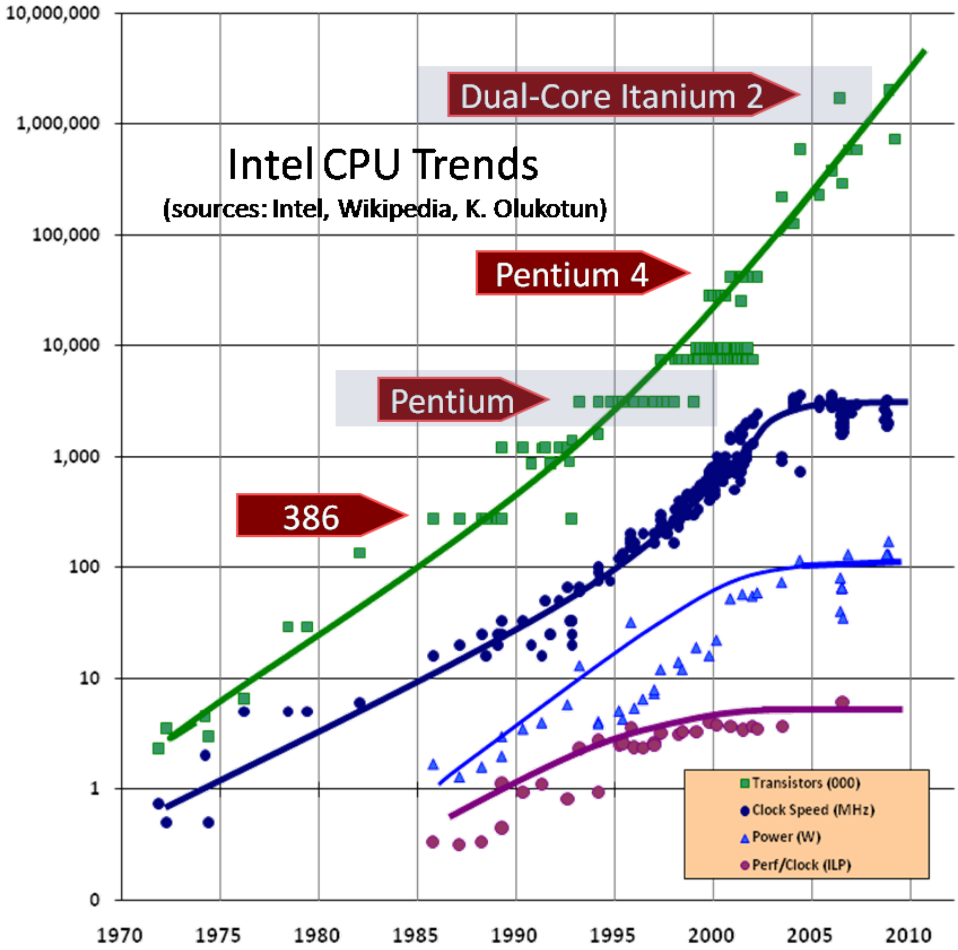


FIGURE 1.1: Intel CPU Timeline [19]

1.2 Current Technology Limitations

According to Moore's Law, the number of transistors on a chip doubles every two years. The law also states that improved clock frequency and improved architecture results in doubling of the processor performance every 18 months. This pattern is illustrated in Figure 1.1. The figure shows the Intel processors trend in terms of transistor count, clock speed (MHz), power (W), and performance/clock (ILP) [19], where Instruction-Level Parallelism (ILP) factor is a measure of the number of operations in a computer program that can be performed in parallel.

Current conventional processors largely relied on the gains leveraged by Moore's Law.

However, in recent years, these gains started to shrink as this law is expected to hit physical limitations on feature sizes in the future [20]. The other major limiting factor is the sharp flattening of clock speeds and the performance gain (just after the year 2005 as seen by the second and the third curves from the top in Figure 1.1). Another limiting factor is the wide gap between processor and memory speeds. In other words, as the number of transistors inside chips continues to rise -at least for the time being- clock speeds are flattening because faster processors would consume more power and dissipate larger amounts of heat which will increase system costs.

Current trends indicate that future computing platforms are likely to continue benefiting from the continuation of Moore's Law by relying on massive parallelism [4, 21]. One of the main current trends clearly shows a shift from single-core processors to multi-core processors, which is changing the computing arena [22]. This move is expected to achieve performance gains, given that new parallel computing tools are developed to fully exploit the available hardware parallelism. Processor vendors are already moving to a model where the number of cores available on a single chip will double with each semiconductor process generation [23, 24].

In addition to the current multi-core trend, FPGAs also have been gaining the attention of the HPC community in the last few years [5]. Recent Studies point that the peak FPGA floating-point performance is growing significantly faster than peak floating-point for CPUs. While CPU performance is doubling every 18 months according to Moore's Law, the performance of FPGAs increases by a factor of 4 every two years. For FPGAs with architectural built-in improvements such as built-in multipliers and Digital Signal Processing (DSP) blocks, the performance is estimated to be increasing by a factor of 5 every two years [25]. This rapid advances also includes the doubling in capacity of FPGAs every 18 months [24]. FPGAs now can contain approximately 330,000 logic blocks and around 1,100 I/O pins and an operating frequency of up to 1,600 MHz [26, 27].

Due to the limitation in clock speeds and Moore's Law is coming to its physical limitation, the research community have already suggested a number of ways to continue harvesting the performance gains every year. One of these approaches is to rely on performance delivered using parallelism by adding more cores onto processors and hence moving from single-core to multi-core processors. This can be seen as a fundamental turn toward concurrency in software [19]. Another research direction is to use hardware acceleration engines like FPGAs and GPUs (Graphics Processing Unit) along side conventional processors to continue the performance gain. A vision in the current research is to use heterogeneous computation elements like FPGA, GPUs, CPUs, and ASICs together in one complete system to achieve the maximum achievable speedup.

1.2.1 Efficiency of Parallel Systems

Current parallel computer systems provide a large throughput in accelerating computationally intensive tasks. Usually, the cost to develop a parallel system with N replicated processors is less than designing an N times faster single-core processor [28]. Hence, it is possible to use lower performance lower cost technology to construct higher performance parallel systems. This can also be applied to FPGAs when used to speedup applications. A number of less performant lower cost FPGAs can be used on parallel to build high performance hardware accelerators.

In order to achieve high efficiency with parallel implementation of an algorithm, one must carefully tune the application to ensure that most processors are busy throughout the execution process, while minimising the parallelisation overhead. Parallel programs are usually composed of sections of one of the following types:

- Serial Code: Sections of the code which must be executed on a single processor. Little or no useful work can be done on other processors.
- Critical Code: Sections of the code which can only be executed on a single processor at any given time. Other parallel section can be executed on other processors alongside the protected code.

- Parallel Code: Sections of the code which can be fully executed in parallel.

From Amdahl's Law [29], the ratio between the parallel execution time $T_{parallel}$ on N processors and the execution time for a single processor T_{single} is given by:

$$\frac{T_{parallel}}{T_{single}} = F_{serial} + MAX(F_{critical}, \frac{F_{critical} + F_{parallel}}{N}) \tag{1.1}$$

Where F_{serial}, $F_{critical}$, and $F_{parallel}$ are the serial, critical, and parallel fractions of the code respectively. From Equation 1.1, F_{serial} must be minimised in order not to limit the overall parallel speedup. Each processor runtime usually depends on the data being processed by that processor. If a processor requires more runtime to execute the parallel code on its data, the other processors should wait until the slow process terminates. This granularity problem reduces the gain of parallelisation.

1.3 Reconfigurable Computing

1.3.1 FPGA Architecture

FPGAs are semiconductor devices that consist of arrays of Configurable Logic Blocks (CLB), interconnects, and I/O Blocks (IOB). Xilinx and Altera are the current main FPGA vendors. Figure 1.2 shows the typical Xilinx FPGA architecture, which also contains built-in hardware such as Block Random Access Memory (BRAM) and DSP blocks [5], which are user configured to implement complex digital circuits. The basic architecture of a CLB consists of a Look-Up Table (LUT) with four -or more- inputs and a Flip Flop (FF), as seen in Figure 1.3 [30]. Each IOB provides individually selectable I/O access to one of the external pins.

Usually, ASIC systems are faster than FPGAs, consume less power, and can implement very complex designs. However, FPGAs can be reprogrammed to perform different functionalities [31]. The desired function can be described in any hardware description language and then synthesised to a technology-mapped netlist ready for reprogramming.

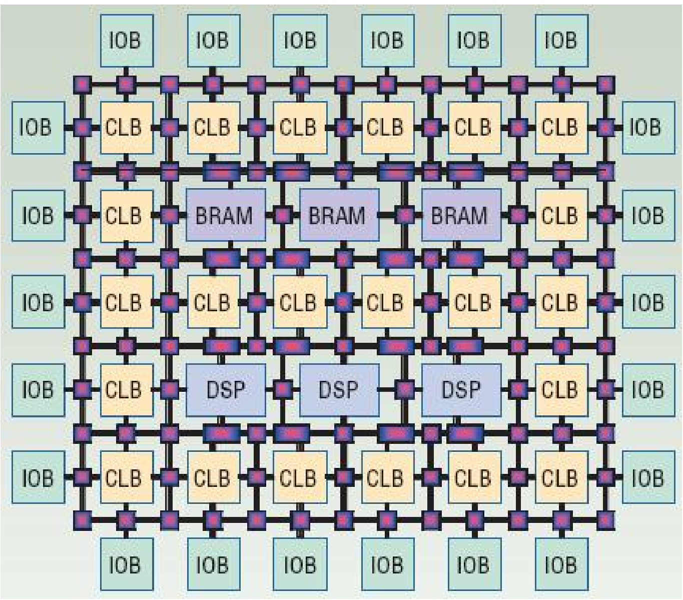


Figure 1.2: Typical Xilinx FPGA Internal Architecture [5]

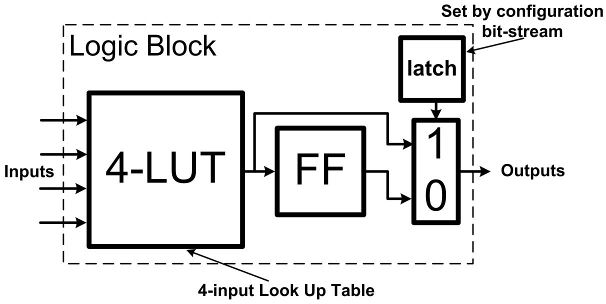


Figure 1.3: Configurable Logic Block (CLB) Basic Architecture [30]

ASIC chips follow the same design flow, however, netlists are mapped permanently on silicon and cannot be reprogrammed again. Recent advances in FPGA design flow and the increasing device capabilities have made FPGAs increasingly popular. Figure 1.4 shows the FPGA growth trend [32].

Reconfigurable Computing (RC) is a computer architecture which combines some of the flexibility of software with the high performance of reconfigurable devices like FPGAs. Figure 1.5 demonstrates a typical Multi-FPGA Reconfigurable Environment, which consists of the following two main parts:

- The Synthesis system that maps the high-level description of the application.

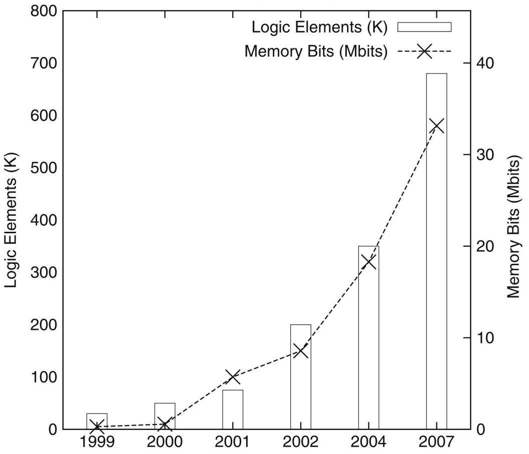


FIGURE 1.4: FPGA Growth Trend [32]

- The Multi-FPGA Reconfigurable hardware system to map applications onto.

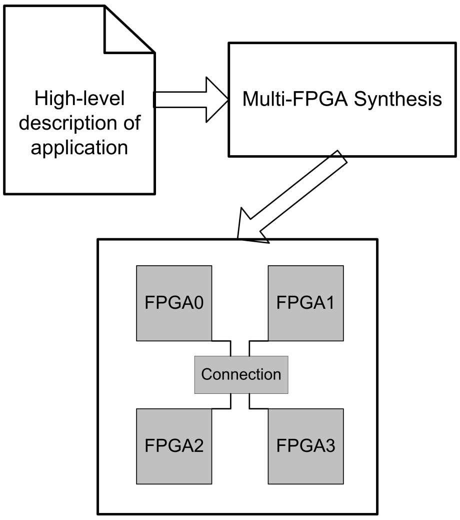


FIGURE 1.5: A Typical Multi-FPGA Reconfigurable Environment

1.3.2 Multi-FPGA Synthesis System

Multi-FPGA Synthesis design flow consists of two main processes: Synthesis and Partitioning as outlined in Figure 1.6 [33]. The input Application Specification can be expressed at three abstraction levels: High-Level (behavioural), Register Transfer Level (RTL), and Gate Level. High-Level designs are specified in the form of algorithmic descriptions; and RTL level designs are structural netlists of components; whereas Gate level designs are represented as a set of boolean equations. The size of structural details reduces as we move from Gate level up to the High-Level abstraction layer. The design flow also takes as input the RC Resources and Timing Constraints.

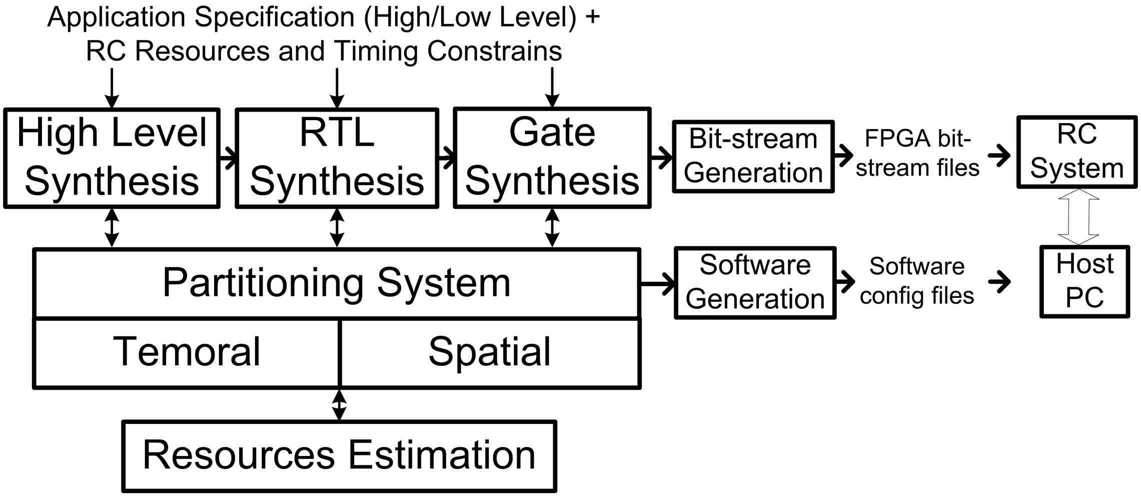


Figure 1.6: Electronics Design Automation Flow for Typical Reconfigurable Computing Systems [33]

The synthesis step processes the input specification through a number of sub-processes (scheduling, binding, and allocation) to generate a final device netlist. The input specifications are usually converted to graph-based platform independent models, with nodes denoting computations and edges denoting data and control flow. The Partitioning step uses similar model to partition the design specifications into a number of connected sub-graphs (i.e. sub-circuits). If the whole input design cannot be fitted into the RC system, the Temporal Partitioning divides this design description into a sequence of temporal segments. Each temporal segment can use all the RC resources. Dynamic programming approach is used to sequentially program the system with the different temporal portions [34, 35]. The Spatial Partitioning divides the input design into a number of spatial segments to match the number of FPGAs in the system. The system employs a number of resources estimators in order to satisfy the initial RC resources and timing constraints. Finally, the system generates the bit-stream files to configure the hardware system.

The RC Systems design flow shown in Figure 1.6 should determine the appropriate trade-offs between the overall performance of the system, resources utilised, and the inter-FGPAs communication. The synthesis process translates the behavioural/structural description of the system to a generic circuit composed of a datapath and a controller. The main goal of the partitioning process is to achieve the minimum number of signals between the different partitions due to the limited number of I/O pins in FPGAs.

The degree of acceleration that can be achieved depends heavily on the inherent parallelism available in the application itself. Furthermore, the application design flow for RC systems is still not a straightforward task. It can range from designing hardware, which is tedious and error prone, to software that requires hardware knowledge. Therefore, integrating FPGAs in HPC needs programming tools that address the whole parallel architecture.

1.3.3 Multi-FPGA Hardware System

Typical FPGA-based RC Systems have several FPGAs and memories communicating through a predefined network topology as shown in Figure 1.7. A number of approaches are used for interconnection network like direct connections, programmable interconnects, and buses. The FPGAs are connected to local memory banks or a shared global memory bank depending on the programming model used for each application. Multi-FPGA systems are typically used as co-processors connected to a host PC or as standalone computing systems [33].

RC Systems are generally statically or dynamically programmed. Static programming approach loads the configuration bit-streams onto the FPGA devices once only and the entire application is executed thereafter. Dynamic programming, however, loads partial bit-streams of the application onto the FPGAs and waits for the partial execution to finish; the host PC then re-programs the FPGA devices to perform another portion of the application. Dynamic programming approach provides virtually unlimited hardware resources for the application. However, this approach suffers from long re-programming delays [36].

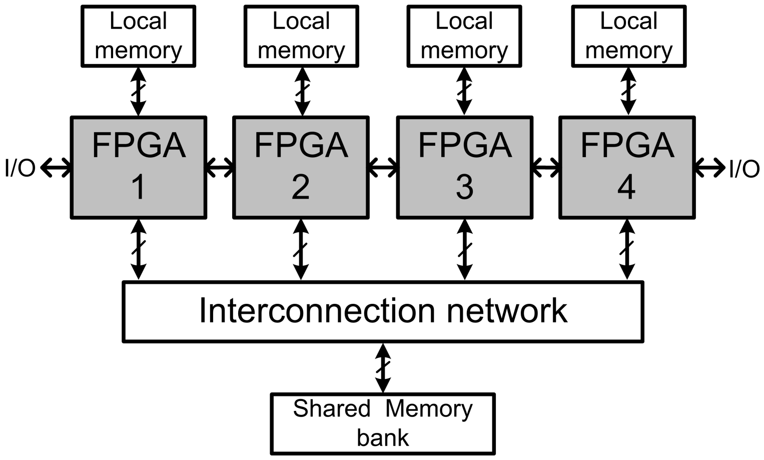


Figure 1.7: A Typical Reconfigurable Computing Architecture [33]

1.4 Research Motivations

Given the recent advances in the domain of HPRC systems, a key question to ask is whether we can use multi-FPGA systems to accelerate the SPICE simulator. This thesis explores ways to answer this question.

1.4.1 SPICE Simulation

SPICE simulation is an essential step in the design and verification of modern circuits [37, 38, 39]. The SPICE algorithm simulates the behaviour of non-linear circuits. This is done by formulating the circuit equations of the linear devices (e.g. resistors) and non-linear devices (e.g. transistors) using Kirchoff's conservation laws, also known as the Modified Nodal Analysis (MNA) [40] which is explained in Section 3.1. The MNA analysis involves the following steps:

- Formulating the circuit equations of the linear devices (e.g. resistors, capacitors) and non-linear devices (e.g. transistors) using Kirchoff's conservation laws at the different nodes of the circuit.
- Evaluating circuit conductances and current matrices from the device model equations. This phase is called the Device Model Evaluation Phase.

- Solving the circuit models using Newton-Raphson (NR) method.
- Solving the system of the linearised equations representing the circuit using methods like Lower/Upper (LU) decomposition. This phase is called the Linear Solver Phase.

1.4.2 SPICE Simulation Bottleneck

Due to the current increase in the complexity of analogue and mixed-signal chips, EDA verification tools are demanding more computational power [41]. This made the transistor-level simulation a growing bottleneck in the overall development process. SPICE simulations of large sub-micron circuits with high accuracy can often take days or weeks of runtime on current processors. SPICE simulation is typically infeasible for circuits larger than 20,000 devices [42]. Also, given the decreasing minimum feature size of devices, their numbers on a single chip has risen significantly over the last few years. The process of down-scaling transistors also impacted the electrical characteristics of devices. As a result, it became very important to run simulations on larger segments of circuits in order to validate their electrical and timing behaviours before fabrication. Hence, there is a very urgent need to accelerate circuit-level simulations without sacrificing accuracy.

The SPICE simulator has a number of components with varying degrees of inherent control and data parallelism. Hence, it is not easily parallelisable on conventional processors due to its irregular structure of computations, limited peak floating-point capacities and constraints due to limited memory bandwidth. The SPICE simulator is used as a benchmark in the SPEC92 collection which represents a set of challenging problems for CPUs [43]. Hence, the SPICE simulator is a challenging application that is worth looking at ways to accelerate through parallelism.

A number of approaches were introduced to reduce the SPICE simulator runtime by parallelisation, which met with mixed success. Attempts either compromised accuracy (which leads to convergence issues) or have employed specialised custom platforms that has been overtaken by the recent advances in general purpose and multi-core processors [44, 45, 46]. When considering the acceleration of the SPICE simulator through parallelisation, one must consider the two phases of the Newton-Raphson iteration: the Device Model Evaluation and the Linear Solver (Section 1.4.1). A number of studies explored the hardware-based acceleration of both phases, as detailed in Section 3.3.2. FPGAs and GPUs are currently under great interest in order to take advantage of their speedup in boosting the performance of current EDA tools.

1.5 Research Scope and Objectives

The area of hardware accelerated SPICE simulator is becoming more important as FPGAs and GPUs are becoming increasingly attractive to continue the performance gain. New acceleration platforms that can be used to accelerate the SPICE simulations should be based on standards in order to facilitate maintainability and portability of such applications [21]. The SPICE simulator bottleneck could be eased by exploiting the inherent hardware parallelism in FPGAs. The FPGA-based accelerators have a great potential in relieving the increasing complexities faced by current EDA tools, and hence shorten the simulation and verification times.

One of the main objectives of this project is to investigate a design methodology for a high performance, low-cost accelerator that exploits the inherent parallelism in the SPICE simulator. This involves identifying the key parts of the algorithm most suitable for FPGA implementation in addition to the design decisions related.

This thesis demonstrates how a Spatial Implementation of Device Model Evaluation phase of the SPICE circuit simulator can be designed and optimised by exploiting the inherent parallelism at different levels. The fully spacial implementation of the SPICE device model would take up the resources of a number of FPGA combined together. This will result in a number of intermediate signals being exchanged between FPGAs which have to be transferred through the I/O resources. However, FPGAs tend to be limited in terms of their available I/O pins. Hence, one of the objectives of this research project is to look at techniques to optimise the inter-FPGA connections and hence reduce the FPGA pin usage. This thesis addresses the following research questions:

- What are the different degrees of parallelism in the device model evaluation phase that can be exploited using FPGAs.
- How can the device model evaluation phase be mapped efficiently on multi-FPGA systems?
- How much acceleration can be achieved over conventional processors?
- How inter-FPGA connections can be minimised in case large device models have to be mapped on multiple FPGAs?

1.6 Thesis Structure

A multi-FPGA system was presented to perform the transistor device model evaluations in parallel. The work showed that FPGAs have a great potential to accelerate the SPICE simulator. Our study also demonstrated a code transformation flow where the device model code can be translated from a high-level description to a structural description ready for FPGA mapping.

This study also highlighted the issue of inter-device in the domain of multi-FPGA synthesis especially if a spatial implementation of large device models is considered. This would require a large number of signals to be exchanged between FPGAs in the multi-FPGA system. This brings to the surface, the optimisation of the pin usage since FPGAs are limited in terms of their resources including I/O pins. A multi-FPGA synthesis system specifically focused on inter-FPGA optimisation was designed. An optimisation approach was introduced to reduce the number of inter-FPGA signals by altering the process of mapping partitions to FPGAs.

The thesis is structured as follows:

Chapter 2 outlines the background information relating to high-level synthesis and partitioning. This Chapter also presents the state of the art advances in the field of High-Performance Reconfigurable Computing (HPRC) in the programming/hardware sides in addition to some applications employing hardware acceleration.

Chapter 3 explains the theoretical background of the SPICE simulator and also outlines the different approaches used to exploit the inherent parallelism in the algorithm. This Chapter outlines our approach to accelerate the SPICE model evaluation using a Spatial FPGA implementation.

Chapter 4 explains the design and implementation of the prototype multi-FPGA system used to accelerate the device model evaluation as proposed in Chapter 3. A single FPGA implementation is first considered in order to evaluate the acceleration and resources results of the device model. A multi-FPGA system is then prototyped using three off-the-shelf Xilinx Virtex II Pro FPGA boards to demonstrate the amount of acceleration that can be achieved through parallelism.

Chapter 5 presents the synthesis and acceleration results of the experimental work outlined in Chapter 4. The system showed that multi-FPGA systems can effectively be used to accelerate the device model evaluation, and hence SPICE simulations. The results were extended theoretically to include newer SPICE device models and to take advantage of state-of-art multi-FPGA systems.

Chapter 6 presents the design and implementation of a prototype multi-FPGA synthesis system and an optimisation technique used reduce the multi-FPGA pin-usage. This Chapter investigates the use of high-level synthesis and partitioning in the process of pin utilisation optimisation of a mesh-based topology.

Chapter 7 discusses our conclusions, research contributions, and future work. The Chapter showed that our application specific architecture can be used as a high speed co-processor attached to workstations to boost SPICE-like simulations. This Chapter also discusses the use of our device model accelerator for iterative solver based simulation.

1.7 Publications

- A. Maache, J. Reeve, and M. Zwolinski. Accelerating CMOS Device Model Evaluation Using Multi-FPGA Systems. In Fifth UK Embedded Forum 2009, Leicester, UK, pages 10–14, September 2009 [47].

- A. Maache, J. Reeve, and M. Zwolinski. Optimising Physical Wires Usage in Mesh-based Multi-FPGA Systems using Partition Swapping. In 21st International Conference on Microelectronics, ICM09, Marrakech, Morocco, pages 246–249, 19–22 December 2009 [48] .

Chapter 2

Multi-FPGA Systems Review

This Chapter outlines the background literature in the domain of algorithms acceleration using multi-FPGAs. This domain includes multi-FPGA synthesis which is composed of hardware synthesis and logic partitioning. This Chapter also presents the state of the art advances in the field of High-Performance Reconfigurable Computing. Section 2.1 gives general overview of hardware synthesis. Section 2.2 describes the different circuit partitioning methodologies and a number of existing multi-FPGA synthesis systems. Section 2.3 presents the advances in the architectural/hardware sides and some applications employing hardware acceleration in high performance computers. Section 2.3.2 outlines the current advances from a programming models' perspective. Section 2.4 summarises the design decisions taken to design our multi-FPGA system based on the materials reviewed in this chapter.

2.1 High-Level Synthesis

High-Level Synthesis (HLS) is the process of transforming an abstract specification of the system to a structural description satisfying user constraints on area, delay, and power consumption [49, 50, 51]. Figure 2.1 illustrates a generic HLS system [52]. The HLS system takes as inputs a behavioural description of the design plus user constraints.

The input description is compiled into an internal representation, usually a Control Data Flow Graph (CDFG), which is passed through the three synthesis steps: scheduling, allocation, and binding. Low level module libraries are used to guide the synthesiser through the optimisation process to meet the design objectives. The synthesis output is a mixture of structural and RTL descriptions suitable for the placement and routing tools. Example state-of-art HLS tools includes Synphony HLS from Synopsys [53], and Catapult C from Mentor Graphics [54].

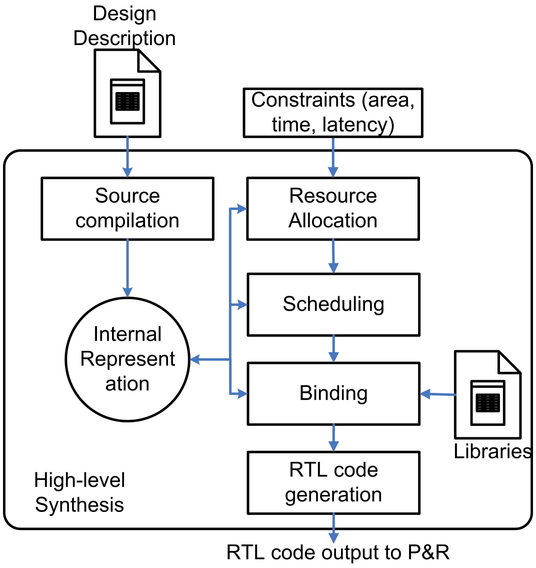


Figure 2.1: Generic High-Level Synthesis System [52]

The structural description generated by a typical HLS system consists of a datapath, a controller, and memory elements as seen in Figure 2.2 [55]. The datapath consists of a set of functional units (adders, multipliers, and shifters), storage units (registers, counters, and register files) and interconnections units (wires, multiplexers and buses). The control unit consists of a Finite State Machine (FSM) that controls the functional and storage units by steering the data in the datapath using internal control signals such as register load and multiplexer select signals.

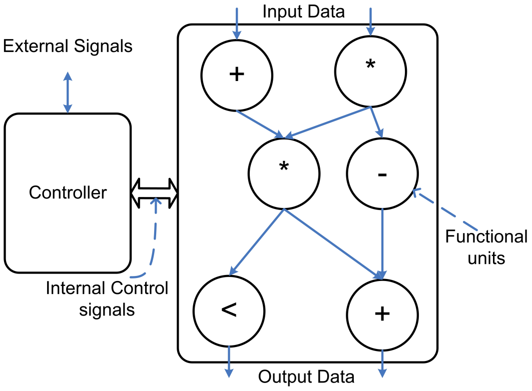


FIGURE 2.2: Generic structure of the synthesis target circuit

2.1.1 Scheduling

The Scheduling step assigns each operation in the internal representation (CDFG) to a particular time step. Schedules are usually optimised to achieve the user constraints in terms of timing and area. Scheduling algorithms can be generalised into two main categories: *constructive* and *transformational scheduling*. Constructive algorithms are called so because they construct a solution without performing any backtracking. Transformational algorithms, however, work on improving an initial schedule by applying a set of transformations [49, 56]. *As Soon As Possible* (ASAP) and *As Late As Possible* (ALAP) are the simplest constructive schedules. ASAP schedules operations in the earliest possible time step permitted by data dependencies, while ALAP assigns operations to the latest possible time step. Figure 2.3 shows an ASAP and ALAP schedules of the equation: $O = ((A - B) + (C + D))/(E * F) + (G + H)^2$. Operations are treated equally with no priority given to the more critical ones.

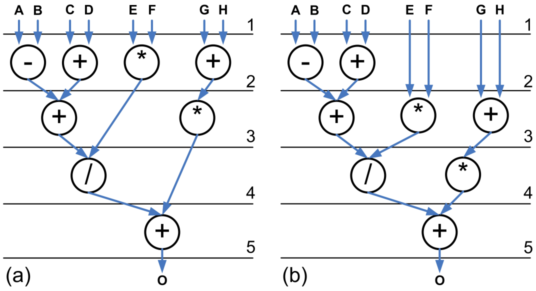


FIGURE 2.3: Example of ASAP (a) and ALAP (b) schedules

List Scheduling assigns operations to time steps based on a pre-defined priority function. Operations are scheduled sequentially as long as the required resource is available, otherwise, operations are postponed according to their priority. The Force-Directed Scheduling (FDS) is a constructive algorithm that makes decisions based on a global analysis of operations and control steps [57]. The main issue with the above algorithms is that decisions are made upon local considerations, which might not necessarily produce an optimum schedule.

The transformational approach starts with an initial schedule, generally ASAP or ALAP, and iteratively applies a set of local transformations to improve the schedule towards the user specified constraints. One important advantage of this type of algorithms is that a complete schedule exits after each iteration and hence an accurate estimates of time and area can be obtained. Integer Linear Programming (ILP) is a mathematical method to solve the scheduling problem under resource constraints which provides an exact analytical solution to the scheduling problem. However, ILP formulation and solving processes are computationally expensive and limited by the number of variables used [49, 58].

2.1.2 Allocation

Allocation is the process of determining the type and quantity of resources used in the design. It also determines the clocking scheme, pipelining style, and memory hierarchy. The selection process utilises a set of component libraries, which may contain multiple implementations of functional units, each with different properties (size, delay, and power). The main goal of the allocation phase is to perform the optimum trade-off between the design performance and cost. Designs with inherent parallelism can be assigned more hardware resources which results in better performance; but it also increases the area cost. However, allocating less hardware resources reduces the area cost, but results in poorer performance [50].

2.1.3 Binding

Binding is the process of assigning the already allocated datapath units from a list of technology-dependent cell/module libraries. The latter may contain one or more implementations of the same functional unit, in which a decision would be made based on the user objectives. Area and time estimates of the library components are also used to guide the scheduling/allocation processes [51]. Binding involves assigning the variables and instructions into one of the following types:

- Functional binding assigns each operation in the schedule to a functional unit such as adders, shifters and multipliers.
- Storage binding assigns variables to storage units such as registers, register files and memory units.
- Interconnect binding assigns an interconnection unit to a multiplexer or a bus, where each interconnect represents a data transfer between functional and storage units.

2.2 Multi-FPGA Logic Partitioning

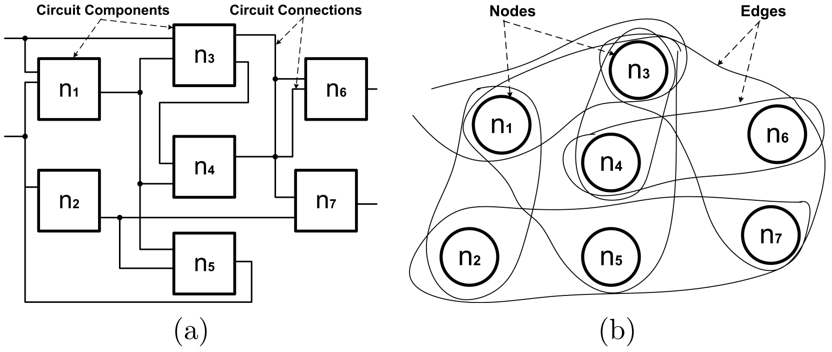
Logic Partitioning for multi-FPGAs systems consists of splitting an internal design representation into a number of balanced partitions. Each partition is programmed to a particular FPGA. A common design representation for partitioning would be a graph based model like Data Flow Graphs [59], Control and Data Flow Graphs [33], or Module Call Graphs [52, 60]. The design representation is partitioned with the aim to satisfy the optimisation criteria and the user constraints in terms of area, speed, power, and I/O pin-usage.

2.2.1 Partitioning Methodologies

Partitioning algorithms are categorised into two main categories: *constructive* and *iterative* algorithms [61]. Constructive algorithms finds partitions from a graph representation of the circuit. In other words, partitions are constructed from the original graph in an incremental fashion. The iterative approach, however, works to improve an existing solution. One of the most well-know iterative algorithms is the Kernighan and Lin (KL algorithm) [62] and its variant, the Fiduccia-Mattheyses (FM) heuristic [63]. The constructive methods include the Simulated Annealing (SA) and Genetic Algorithms (GA) [61, 64].

2.2.1.1 Partitioning Problem Formulation

The general graph partitioning consists of dividing a set of components and a netlist of connections between these components into a number of balanced partitions. A graph is a set of nodes (nodes) linked together with a set of edges. Each edge connects exactly two nodes. A *Hypergraph* is a generalisation of a graph, where an edge can connect multiple nodes. A hypergraph is particularly useful for representing typical circuit netlists because connections can be made between multiple components. Figure 2.4 shows an example circuit and its representation using a hypergraph, where the Circiuit Components $\{n_1, n_2, ..., n_7\}$ are represented as Nodes and the Circuit Connections between these components are modelled as Edges.



(a) (b)

FIGURE 2.4: (a) Example circuit, (b) Hypergraph representation

The input design is usually modelled as a hypergraph $G(V, E)$, where V is the set of nodes $i(n_i)$ corresponding to the components $\{n_1, n_2, ..., n_7\}$ as seen in Figure 2.4, and E represent the signal nets that interconnects the components. A *signal net* is a simple connection between two or more components in the hypergraph, which is represented by the set of groups in Figure 2.4(b). Dividing V into a set of K disjoint partitions is called *multi-way partitioning* when $K > 2$ and *bi-partitioning* when $K = 2$. Hence, given the K-way partition $\{V\} \rightarrow \{V_1, V_2, ..., V_K\}$, the objective of the partitioning algorithm is to minimise:

$$cutsize = \sum_{i,j} k_{ij}, \quad where \quad i \neq j. \tag{2.1}$$

Where k_{ij} is a *signal cut net* which represents a signal between the partitions V_i and V_j, and *cutsize* is the total number of these inter-partition signals (i.e. the total number of *signal cut nets*).

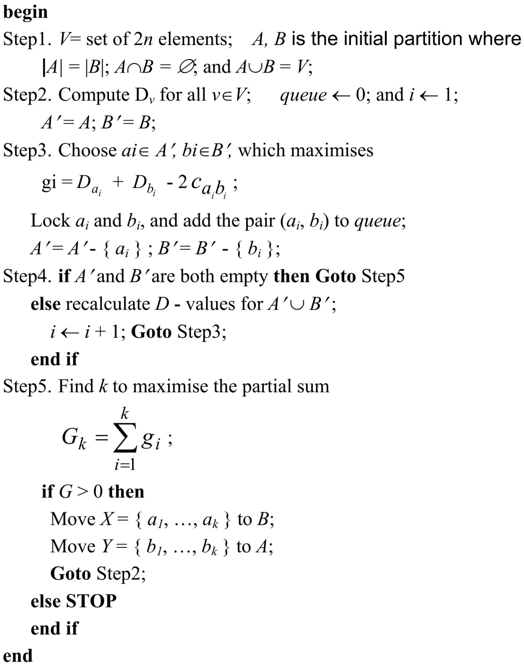
The cost constraint of each partition r is specified by:

$$cost(r) = \sum_{n_i \in V_r} c_i \leq C_r, \quad where \quad 1 \leq r \leq k. \tag{2.2}$$

Where c_i is the cost associated with the node $i(n_i)$ and C_r is the cost constrain on the partition r. C_r must be greater or equal to the sum of all the costs associated with the nodes in partition r. This model aims to minimize the number of signals required between partitions (Equation 2.1) under the cost constraints stated in Equation 2.2. The constraints can denote the area usage of each partition, or may be the power consumed by the individual partitions. Each node in the hypergraph is assigned a weight that corresponds to a particular attribute to be considered like area or power. These weights are used to balance partitions and to calculate overheads.

2.2.1.2 Kernighan-Lin Algorithm

The Kernighan-Lin algorithm is an iterative algorithm that starts with two random initial partitions of the input graph $G(V, E)$. The algorithm then improves the solution by swapping pairs of nodes to reduce the number of cut nets between partitions (cutsize). The *gain* of moving a node from its current partition to another partition is the difference between the external and the internal nets. Each swap operation is made so that the highest gain value is achieved. After each pair swapping operation, the resulting gain is stored and the swapped nodes are locked and cannot be considered for swapping again. The process continues until all nodes are evaluated and locked. The algorithm terminates when the best gain found in an iteration is less than or equal to zero; in other words, it is no longer possible to achieve any more improvements from pair swapping. KL pseudo code is shown in Figure 2.5.



```
begin
Step1. $V$= set of $2n$ elements;   $A, B$ is the initial partition where
    $|A| = |B|$; $A \cap B = \varnothing$; and $A \cup B = V$;
Step2. Compute $D_v$ for all $v \in V$;   $queue \leftarrow 0$; and $i \leftarrow 1$;
    $A' = A$; $B' = B$;
Step3. Choose $ai \in A'$, $bi \in B'$, which maximises
    $gi = D_{a_i} + D_{b_i} - 2c_{a_i b_i}$;
    Lock $a_i$ and $b_i$, and add the pair ($a_i$, $b_i$) to $queue$;
    $A' = A'$ - { $a_i$ } ; $B' = B'$ - { $b_i$ };
Step4. if $A'$ and $B'$ are both empty then Goto Step5
    else recalculate $D$ - values for $A' \cup B'$;
        $i \leftarrow i + 1$; Goto Step3;
    end if
Step5. Find $k$ to maximise the partial sum
        $G_k = \sum_{i=1}^{k} g_i$;
     if $G > 0$ then
        Move $X$ = { $a_1$, …, $a_k$ } to $B$;
        Move $Y$ = { $b_1$, …, $b_k$ } to $A$;
        Goto Step2;
    else STOP
    end if
end
```

FIGURE 2.5: Kernighan-Lin Algorithm Pseudo Code

2.2.1.3 Fiduccia-Mattheyses Algorithm

The Fiduccia-Mattheyses algorithm is an extension to the KL algorithm which reduces the time per iteration to a linear time with respect to the size of the netlist [63]. One of the main concepts introduced in the FM algorithm is the *Gain Bucket Data Structure* shown in Figure 2.6, which is a list of the nodes to be moved sorted according to the gain of each move. The list is ordered from maximal to minimal gain, where a positive gain improves the overall solution and a negative gain degrades it. The grey nodes in Figure 2.6 are the ones that have already been moved from their original partition and cannot be selected to be moved again. The free nodes are sorted in the bucket ready for selection. The *Max Gain* index always points to the highest gain value in the bucket data structure.

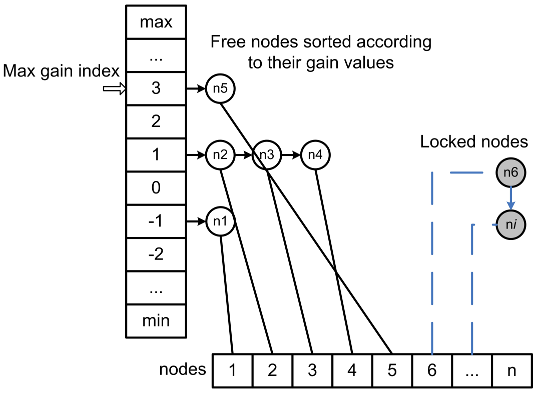


FIGURE 2.6: The Gain Bucket Data Structure used in the FM Algorithm

The FM algorithm starts with a random initial partition, and iteratively improves the solution by applying a number of *moves* within a single *pass*. The node with the highest gain is selected and moved to the other partition and subsequently locked to prevent the algorithm from selecting and moving it again. In order to prevent all nodes migrating to one partition, a balance criteria must be satisfied before any move can be made. The user specifies a balance factor r called *ratio*, $0 < r < 1$, which satisfies the criteria $r = |A|/(|A| + |B|)$, where $|A|$ and $|B|$ are the sizes of the partitions A and B. The condition for the partition $\{V\} \rightarrow \{A, B\}$ to be balanced is given by:

$$(r * |V| - s_{max}) \leq |A| \leq (r * |V| + s_{max}) \tag{2.3}$$

Where $|A| + |B| = |V|$, s_{max} is the size of the largest node in the hypergraph. After a move is performed, the gain values of the neighbouring nodes affected by the move are updated. The gain bucket data structure is also updated with the new gain values especially the *Max Gain* index. This process continues until no unlocked nodes can be moved without violating the balance criteria; which marks the end of a pass. The algorithm then uses the best intermediate solution as a starting point for the next pass. Before any pass begins, locked nodes are freed and all nodes are re-sorted again into the gain bucket data structure according to their new gain values. The process then stops when a pass fails to improve the overall solution.

2.2.1.4 Structural and Behavioural Partitioning

Partitioning can be performed at different abstraction levels such as behavioural, RTL or gate-level (structural). The behavioural approach performs the partitioning of the design description before the synthesis process takes place. As a result of this partitioning phase, the design is broken down into a set of sub-designs connected with inter-partition signals. The synthesis process then takes each sub-design and generates a corresponding datapath and a controller to implement that particular sub-functionality. Structural partitioning, however, is applied to the datapath and controller of the already synthesised design. The result of this partitioning approach is a set of segments of the synthesised design distributed over multiple devices. Figure 2.7 shows the difference between the different approaches [65]. Behavioural partitioning, however, needs a number of high level estimators, which estimates the area, timing, I/O and power attributes of the design. These estimators have to be efficient and fast in order to cover the large search space. Several studies demonstrated the superiority of the behavioural over the RTL partitioning [66, 67, 68].

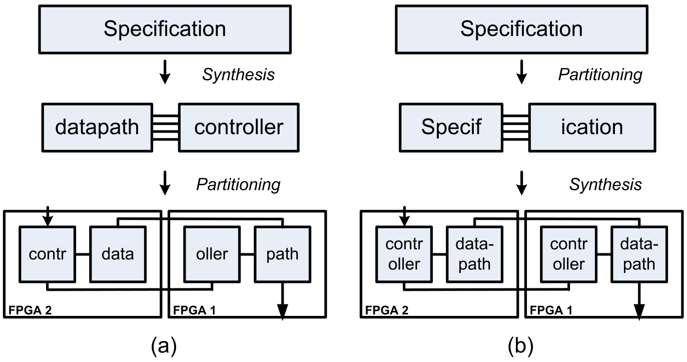


Figure 2.7: Structural (a) and Behavioural (b) logic partitioning [65]

2.2.2 Existing Multi-FPGA Synthesis Systems

A number of multi-FPGA synthesis systems were proposed with varying methodologies and algorithms. Examples of commercial synthesis tools include Auspy Partition System II [69] and Certify [70]. These tools perform the partitioning process at the structural level. A number of academic tools are also available like: COBRA-ABS [71, 72], SPARCS [67, 33], ISyn [73], and CADDY-II [59, 74].

The COBRA-ABS (Column Oriented Butted Regular Architecture Algorithmic Behavioural Synthesis) is a high-level synthesis tool designed for synthesising DSP algorithms in C, targeting Multi-FPGA Custom Computing Machines (FCCMs). The system is based on a Very Long Instruction Word (VLIW [75]) architecture, where each partition contains an independent RISC-like (Reduced Instruction Set Computer) processor and a set of functional units all connected with a bus-based architecture. The design and the target architecture files are compiled into a control-flow block graph. The synthesis and partitioning tasks are performed in a single integrated step. However, high exploration run times of more than ten hours were reported because of the complex model of the integrated problem [36].

The SPARCS system (Synthesis and Partitioning for Adaptive and Reconfigurable Computer Systems) is an integrated partitioning and synthesis framework targeted at Multi-FPGA systems [36, 76, 77]. It provides a tight integration of partitioning and synthesis tasks at different levels of abstractions. Designs are represented by *Behavioural Block*

Graph (BBG) which is a set of blocks and edges representing both control and data flow. During partitioning, each block is viewed as atomic entity that cannot be split. This approach, however, dictates that the designer should have good understanding of the behavioural specification and its effects on partitioning, which is not always the case. SPARCS considers only the direct pin-to-pin connection model and does not cope with the time-multiplexed I/O model. Also, it does not explore performance trade-off with the number of inter-FPGA connections [52].

The COBRA-ABS system focused mainly on synthesising DSP algorithms to FPGAs, whereas the SPARCS system focused on combining synthesis and partitioning into a single process by performing design space exploration. However, none of the synthesis tools mentioned above considered directly exploring and optimising the inter-FPGA connections. The is an important issue to explore because FPGAs are limited in terms of their I/O connectivity. And as the domain of applications requiring multi-FPGAs is growing noticeably over the last few years, the inter-FPGA communication requirement also increases. This means that more FPGA I/O pins are used to route the inter-FPGA signals. Hence, there is a need to further investigate and optimise the FPGA resources usage especially I/O pins.

2.3 High-Performance Reconfigurable Computing

Reconfigurable computing showed a trend of speed advantages over traditional microprocessors in a wide range of applications. For example, [78] presented a multi-FPGA accelerator of a Fourier Integral Operator kernel used in signal processing. The study in [79] tried to understand and quantify the components of the FPGA's speedup in image processing applications. The authors showed that the instruction execution efficiency of the FPGA is an important factor in addition to loading and storing data to/from memory or I/O. Similar conclusions were demonstrated for floating-point arithmetic kernels in [80, 25, 81]. Hence, FPGAs have the potential to boost the high-performance computing since scientific computing is mostly based on floating-point arithmetic.

Typically, FPGA designs are highly optimised and finely tuned, which tend to reduce the power consumption if compared to general purpose processors. It was shown in [82] that by just moving the performance critical software loops to reconfigurable hardware resulted in an average energy saving of about 35% to 70% depending on the FPGA used.

High-**P**erformance **R**econfigurable **C**omputing (HPRC) is a new approach to speedup HPC applications based on conventional processors and FPGAs. It demonstrated orders-of-magnitude improvement in the overall performance over conventional high-performance computers [83, 5]. A number of survey papers [84, 85, 86, 87, 88, 89, 90, 91] outline the modern reconfigurable architectures and design methods.

2.3.1 HPRC Hardware Platforms

Berkeley Emulation Engine (BEE) is a general purpose and scalable framework for designing high-end performance reconfigurable computers. The system can provide acceleration throughput of about 10 times more than a DSP-based system with similar power consumption and cost, and over 100 times than the throughput of general-purpose processors [92]. The system uses DSP programming models (Simulink and Xilinx System Generator) for automatic mapping from high-level block diagrams and state machines to FPGAs.

Research Accelerator for Multiple Processors (RAMP) is a conventional RTL implementation of a message-passing machine to realise a multi-core cluster based on embedded processors on multi-FPGA platforms. It relies on massive parallelism to gain performance improvements. It consists of 768–1008 MicroBlaze cores in 64–84 Virtex-II Pro 70 FPGAs on 16-21 BEE2 boards connected using point-to-point channels and switches [93]. Programming relies on GCC and uClinux running off-the-shelf scientific applications [94, 95].

Toronto Molecular Dynamics (TMD) is an MPI-based programming model for Multiprocessor System-on-Chip implemented onto Multi-FPGA system [96, 97, 98]. Figure 2.8 shows the architecture hierarchy of the TMD system [99]. The system uses point-to-point channels between the system components controlled by an MPI protocol implementation. Each compute FPGA consists of both, MicroBlaze embedded processors and custom application hardware. The communication system uses a light weight MPI implementations, a software implementation to be used on the MicroBlaze and a hardware implementation to interface with the local custom hardware. Recently, the system evolved to include multiple Intel X86 processors alongside the FPGAs, which are all connected to the main system bus [100, 101].

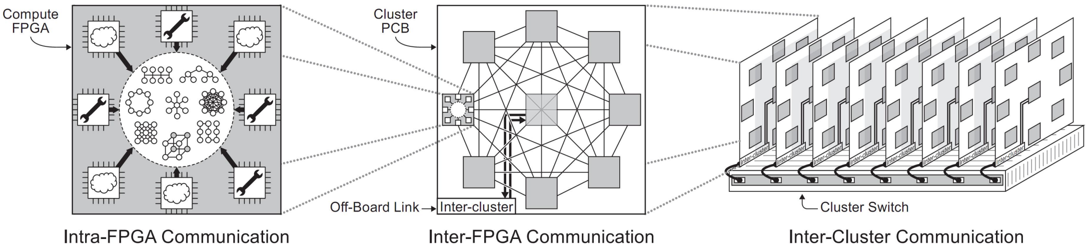


Figure 2.8: TMD Architecture Block Diagram [99]

Maxwell Supercomputer is a general-purpose computer with 64 FPGAs presented in [102]. All FPGAs are wired together directly in a two-dimensional torus using Multi-Gigabit Transceiver (MGT) Rocket I/O connectors. The system uses the Parallel Toolkit (PTK) that was designed to make it as HPC-system-like as possible [103]

XtremeData developed an FPGA coprocessor XD1000 that is socket-compatible with the AMD Opteron processor [104]. This allows it to plug directly into standard HPC systems to replace one of the motherboard's CPUs as in Figure 2.9 [105]. It also have a built-in memory controller to access the high speed memory as well as the fast connection to the host processor via HyperTransport™ [8]. The design flow uses the Impulse-C framework to program the XD1000, using standard C with an API for FPGA-specific functions. Integrated libraries provide single and double-precision support. Impulse-C is used to describe the performance-critical sections, which is then compiled, synthesised, and place-and-routed onto the XD1000.

Nallatech developed both software and hardware HPC systems with FPGA acceleration. The H100 system provides integrated solutions for the IBM™ BladeCenters™

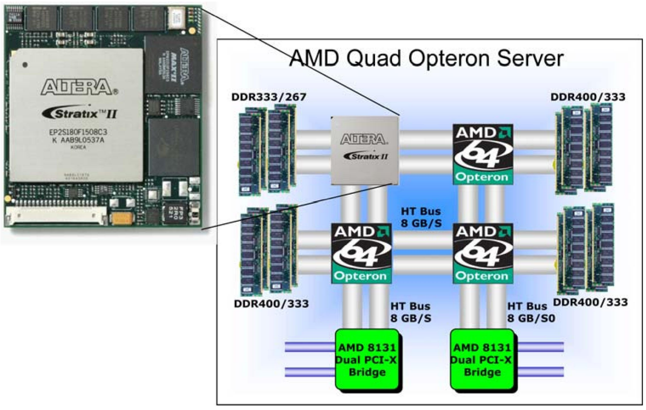


Figure 2.9: XD1000 ALTERA-based Coprocessor [105]

and expansion cards suitable for most cluster systems, which were used in the Maxwell Supercomputer [106, 102].

DINI FPGA Systems designed large FPGA boards for high-performance computing including the DN7020k10 emulation system containing 20 Altera Stratix, and the DNDPB_S327 containing 27 Altera Cyclone 3 FPGAs hosted via Ethernet [107].

2.3.2 HPRC Programming Environments

Hardware Description Languages VHDL and Verilog are well established hardware languages designed to describe high-level algorithms and low-level optimisations. This is currently the dominant approach to program FPGAs [108]. SystemC is a set of library routines and macros implemented in C++ that provides hardware-oriented constructs, which allows it to simulate concurrent processes. It is used for design and verification of hardware and software [109].

Handel-C is a C-like hardware language designed to express parallelism in algorithms. It provides platform support libraries with the design suite. Handel-C does not support floating-point entities as native data types [110].

Nallatech DIME-C is a subset of ANSI-C (American National Standards Institute C), which gives it a number of advantages over Handel-C. The compiler allows the user to create a network of hardware components on an FPGA [111, 112]. Components can be wired together to generate an FPGA bit stream.

SPARK Compiler is a C-to-VHDL high-level synthesis framework. It takes input C code, schedules it using speculative code motions and loop transformations, runs an interconnect-minimizing resource binding pass and generates a finite state machine for the scheduled design graph [113].

Trident Compiler is a compiler for floating-point C algorithms, producing reconfigurable logic that exploit parallelism in such applications [114, 115]. It extracts parallelism from the input code and pipelines loop bodies. It takes designs with float and double types and converts them to synthesisable VHDL.

2.3.3 HPRC Systems Review

The Reconfigurable Systems shown in Section 2.3.1 require programming frameworks in order to take advantage of their hardware capabilities. However, current HPC software developers are unlikely to have the necessary knowledge of the hardware development flow using Verilog and VHDL. Hence, a pure software-oriented development flow would be beneficial to the future of HPRC [116].

The fundamental difference between hardware and software is the execution model. Software is sequential by nature and follows a memory based execution, whereas hardware is concurrent [117]. Generally, software does not assume anything on timing, however, meeting performance target under timing, power, and area costs is one of the fundamental requirements for hardware. Hence, synthesis from C-like languages would require mechanisms for specifying and achieving timing constraints [23].

The degree of acceleration that can be achieved depends heavily on the inherent parallelism available in the application itself. The application design flow for HPRC systems is still not a straightforward task. It can range from designing hardware, which is tedious and error prone, to software that requires hardware knowledge. Therefore, integrating

FPGAs in HPC needs programming tools that address the whole parallel architecture [83].

As current FPGAs are getting bigger and faster, they can support more memory interfaces which means that HPRC system are requiring more memory bandwidth. The performance of memory systems, however, is not growing with the same pace as the performance of the computing systems. This means that more work is needed to improve the performance of memory architectures to keep-up with HPRC and HPC demands. However, the low-power consumption of RC systems is still making HPRC an attractive solution [83].

2.3.4 Floating-Point Operators and FPGAs

Scientific and financial applications tend to make heavy use of real numbers. This does not pose much complications to conventional clusters in terms of hardware resources. Software tends to be flexible when it comes to changing the number representation system. Reconfigurable Computing systems, however, are limited in terms of hardware resources. Appendix F.1 outlines the three main real data type representations used in scientific and financial algorithms and their hardware requirements.

A number of FPGA floating-point libraries are available some of which are examined in Appendix F. FPU100 in [118] is a IEEE-754 fully compliant core. FPLibrary is a VHDL parametrisable library of hardware operators for the floating-point and logarithmic number systems [119]. It provides extra packages supporting logarithm and exponential functions [120]. Also, [121, 122] proposed designs of parametrised FPGAs floating-point libraries. Xilinx also provides a Floating-Point Operator in [123].

The synthesis results in Appendix F.2 shows that FPLibrary is the most efficient library in terms of area utilisation and clock frequency. FPLibrary provides both combinational and pipelined versions of the floating-point operators with an easy-to-use interface. In addition, FPLibrary is fully parametrised, in which the operands' width can be specified using generics. This allows the user to synthesise the operators to a specific precision.

FPU100 supports single-precision numbers only which limits its usage. The IEEE proposed float and fixed packages provide many useful operators and conversion functions. However, these operators are all combinational blocks, which means very large area usage.

2.3.5 Serial Communication and FPGAs

A number of FPGA systems uses parallel buses for communication. However, these buses have to be routed carefully at the Printed Circuit Board (PCB) level in order to operate them at high clock frequencies. This makes serial communication an attractive alternative to the bus approach. Most current FPGAs support gigabit-rate serial I/O interfaces, one of which is the MGT hardware that implements the underlying physical link between FPGAs. Aurora is a light-weight, link-layer protocol from Xilinx used to move data across serial links through the MGT hardware. Figure 2.10 shows the functional diagram of an Aurora-based system [124]. The user application can send/receive data through the Aurora interface at speeds of up to 3.125 Gbps using two modes: streaming and framing modes [125]. An Aurora lane is a high-speed serial connection between MGTs capable of sending a 16-bit word each transfer cycle. Any number of lanes combined together are called Aurora channel. When the channel is not sending or receiving data, it is filled with a random idle sequence (Appendix C).

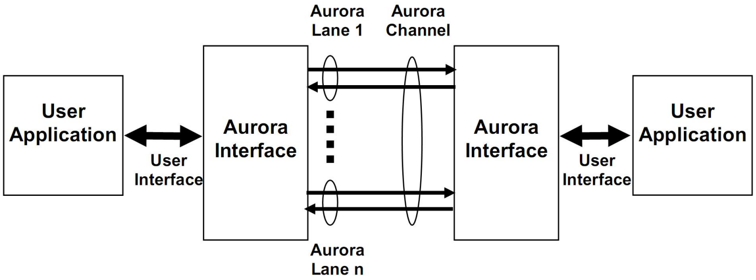


Figure 2.10: Functional diagram of an Aurora-based system

2.4 Summary

As demonstrated by a number of studies, FPGAs have shown large speedup/power gains over conventional processors up to orders of magnitude. Applications, however, require different mapping, design, memory, and computation requirement. Hence, results are very dependant on the amount of parallelism in the application and how this is mapped efficiently on FPGAs.

As explained in Section 1.4 the SPICE simulator is a challenging application that can take days or weeks of runtime. This is mainly because of the rise in the number of devices in circuits and the decreasing feature sizes in addition to the increasing complexity of device models [42, 43]. All these factors increase the complexity of circuits and hence increase the computational load on the SPICE simulator. Furthermore, current conventional processors are running into a number of technological limitations as explained in Section 1.2, which limits their ability to speedup the SPICE simulations further.

Current state of the art FPGAs contain thousands of reconfigurable logic elements, hundreds of distributed high-bandwidth on-chip memories, built-in DSP blocks, and fast interconnects. These advances allowed FPGAs to support double-precision floating-point computations with the ability to realise custom floating-point datapaths. Hence, they provide an attractive architectural solution for accelerating the SPICE simulations as it relies heavily on floating-point computations. The main challenge is to investigate whether FPGAs can deliver a respectable speedup in the case of the SPICE simulator. This involves looking into ways to exploit multi-FPGA systems in this domain. Part of this thesis focuses on exploiting the inherent parallelism in the SPICE simulator using a multi-FPGA system.

The experimental validation of the multi-FPGA system must take into account the fact that the SPICE simulator relies heavily on floating-point computations. Hence, other number representations are not suitable for accelerating this simulator. In addition, the minimum usable precision by the SPICE simulation is the single-precision floating-point [126]. As a result, single-precision floating-point numbers are used to overcome the precision limitation despite the FPGA area and the performance overhead over fixed-point numbers. The synthesis results showed that FPLibrary is the most efficient library in terms of area utilisation and clock frequency.

The proposed multi-FPGA system requires a communication interface to exchange data between FPGAs. A number of approaches are in use to transfer data amongst FPGAs in HPRC systems. The Aurora serial interface will be used in our multi-FPGA system to transfer data between FPGAs due to its high-speed and light-weight size as detailed in Section 2.3.5.

The fully spacial FPGA implementation for large SPICE device models would take up the resources of a number of FPGAs combined together. These device models can be partitioned and mapped on multi-FPGA platforms like BEE3. The intermediate signals between FPGAs have to be exchanged through the I/O resources. However, current FPGAs tend to be limited in terms of their available I/O pin. Hence, one of the goals of this research project is to look at techniques to optimise the number of inter-FPGA communication links. Hence, there is a need to further investigate the multi-FPGA systems in the domain of partitioning, synthesis and resources optimisation. This investigation will focus on tackling the main challenge of the partitioning process which is minimising the number of inter-partition signals.

Chapter 3

Parallel Device Model Evaluation

This Chapter explains the theoretical background of the SPICE simulation process (Section 3.1). The different approaches used to exploit the inherent parallelism in the algorithm are outlined in Section 3.2.1. The existing literature and previous work done in the area of paralleling the SPICE simulator are reviewed in Section 3.3. This Chapter also explains why the device model evaluation phase is chosen for acceleration. Our proposed approach to exploit the inherent parallelism in the simulator using a pipelined FPGA implementation is explained in Section 3.4.3.

3.1 SPICE Simulation Background

SPICE simulation is an essential step in the design and verification of modern integrated circuits [37, 38, 39]. The algorithm uses a matrix representation of the circuit to find the nodal voltages over a period of time. A typical SPICE simulation flow is shown in Figure 3.1, which demonstrates the following key steps:

- The simulation starts by defining an initial trial DC operating point (step 1 in Figure 3.1). At the core of the SPICE simulation flow is the Modified Nodal Analysis. This is accomplished by formulating the Nodal Matrix (steps 1, 2, and

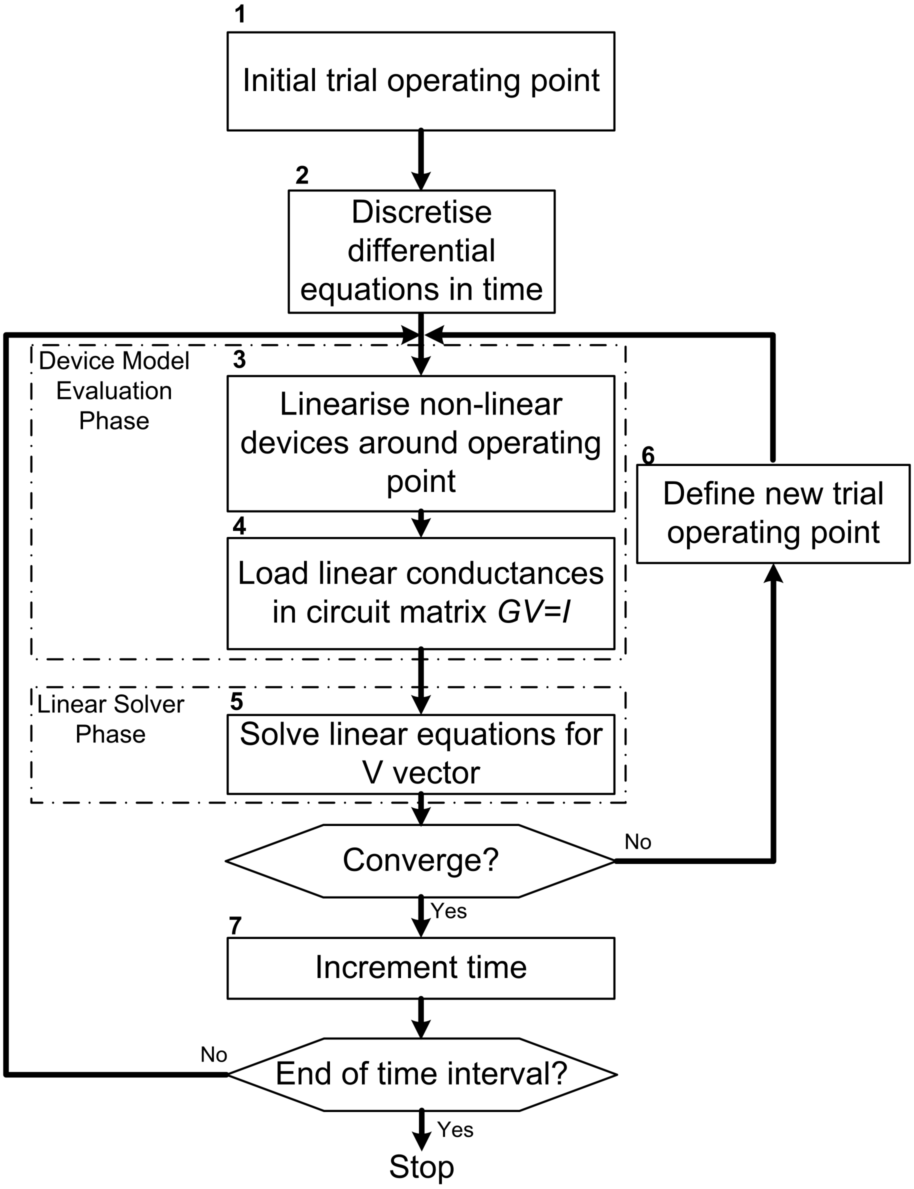

1
Initial trial operating point
2
Discretise differential equations in time
Device Model Evaluation Phase
3
Linearise non-linear devices around operating point
4
Load linear conductances in circuit matrix *GV=I*
6
Define new trial operating point
Linear Solver Phase
5
Solve linear equations for V vector
Converge?
No
Yes
7
Increment time
No
End of time interval?
Yes
Stop

FIGURE 3.1: Typical SPICE Simulation Flow

3) and solving the linearised nodal equations for the circuit voltages (step 5) that is given by:

$$GV = I \tag{3.1}$$

where G is the conductance matrix of the circuit, V is the unknown node voltage vector, and I is the current vector. The main aim of the SPICE algorithm is to find V.

- The inner loop finds the solution for non-linear circuits (steps 3, 4, 5, and 6). Non-linear devices are replaced by equivalent linear models. The linearisation stage uses methods like the Newton-Raphson (NR) to find solutions, which may take several iterations before the calculations converge. After each iteration, a new trial operating point is defined to start the next iteration (step 6).

- The linearisation process includes the **Device Model Evaluation Phase** (steps 3 and 4). This phase evaluates the non-linear device models such as transistors and diodes to obtain the electrical current I flowing through them. The current values are then loaded into the Nodal Matrix which is then solved in step 5 to find the unknown nodal voltages. The SPICE simulator uses built-in non-linear mathematical equations to model the behaviour of the physical devices. For Complementary Metal Oxide Semiconductor (CMOS) transistors, a number of models exist which have a number of physical and empirical parameters [40]. Accelerating this phase using FPGAs is the main focus of this thesis.

- The **Linear Solver Phase** in step 5 solves the Nodal Matrix to find the unknown nodal voltages. Figure 3.1 shows the two main phases of the simulator which are the Device Model Evaluation and the Linear Solver.

- The outer loop (step 7), together with the inner loop, performs a Transient Analysis for energy-storage components (capacitors, inductors, etc.) and incrementing the time-step.

3.1.1 Newton-Raphson Method

One of the well known methods to successively approximate the roots of a real-valued function is the Newton-Raphson (NR) method. Equation 3.2 demonstrates the basic formula used by this method for general root finding tasks:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{3.2}$$

where $f(x)$ is the function which we want to approximate the roots for and $f'(x)$ is its first derivative. The method starts with an inital value x_0 and uses the above formula for a better approximation x_1 and so on until convergence. This method converges quadratically when a sufficiently close initial operating point is chosen. For an efficient implementation of the NR method, the user should provide routines to evaluate both $f(x)$ and its first derivative $f'(x)$ at the point x_i [127, 128]. This present the main drawback of the NR method, as the first derivative cannot always be calculated or may be very expensive to evaluate.

Typical circuit simulator solves the non-linear equations using the Newton-Raphson (NR) [41]. The circuit is represented according to the Kirchoff's Current Law (KCL):

$$F(V) = 0 \tag{3.3}$$

Where F is the sum of the current flowing into each node in the circuit and V is the vector of the nodal voltages. Both vectors have dimension of N which is the number of nodes in the circuit. By applying the NR method to find the nodal voltages using Equation 3.2 on Equation 3.3, it yields the linear matrices system in Equation 3.4:

$$V^{i+1} = V^i - J^{-1}(V^i)F(V^i) \tag{3.4}$$

Where: $J(V^i)$ is the Jacobian computed with V^i, which is the solution at the NR i^{th} iteration, $F(V^i) = \begin{bmatrix} F_1(V^1) \\ F_2(V^2) \\ \cdots \\ F_N(V^N) \end{bmatrix}$, and $J(V^i) = \begin{bmatrix} \frac{\partial F_1}{\partial V^1} & \frac{\partial F_1}{\partial V^2} & \cdots & \frac{\partial F_1}{\partial V^N} \\ \cdots & \cdots & \ddots & \cdots \\ \frac{\partial F_N}{\partial V^1} & \frac{\partial F_N}{\partial V^2} & \cdots & \frac{\partial F_N}{\partial V^N} \end{bmatrix}$

For each NR iteration i, the voltage V^i is used to compute the next iteration's voltage V^{i+1}. This process continues until the difference between the current/previous voltages is less than or equal a pre-set threshold (i.e. $V^{i+1} \approx V^i$). In the SPICE simulator, the current change in each circuit branch is also required to be below certain threshold in order for convergence to be reached.

Transient analysis is performed using the same method. For each time step, the NR method is used to reach convergence. For a typical circuit, the non-linear equations is replaced by a set of linear system at each iteration as shown in Equation 3.5:

$$J(V^i)V^{i+1} = J(V^i)V^i - F(V^i) \tag{3.5}$$

The conductance and current parameters are calculated according to the model equations built into the simulator's device loading routines. The conductance contributes to the entries in the matrix used in the linear system, while the current values are entered into the right-hand-side of the same linear system. The process of calculating the conductance and current values $J(V^i)$, $F(V^i)$) is called the **Device Model Evaluation Phase**. Equation 3.5 is usually expressed as a generic system of linear equations as follows:

$$Ax = b \tag{3.6}$$

where A is an $m \times n$ matrix, x is a column vector with n unknowns, and b is a column vector with m entries. This system can be solved using an LU decomposition (Lower/Upper) linear solver which transforms the circuit matrix A into a lower L, and an upper U, triangular matrix. This process is called the **Linear Solver Phase**. Equation 3.6 is then rewritten as:

$$LUx = b \tag{3.7}$$

After factorising A into L and U, the unknown vector x is then given by:

$$x = U^{-1}L^{-1}b \tag{3.8}$$

The solution to the system in Equation 3.6 is done in two steps:

- First, solve the equation $Ly = b$ for y, where y is a column vector with n entries,
- Then, use backward substitution to solve the equation $Ux = y$ for x.

For a more detailed discussion of the SPICE simulation algorithm, refer to [40, 37, 129].

3.1.2 Example Circuit

In order to illustrate how the simulation process works, a simple circuit is used as an example. Figure 3.2 shows a circuit that contains a current source, two resistors, and a diode. The circuit equations are as follow:

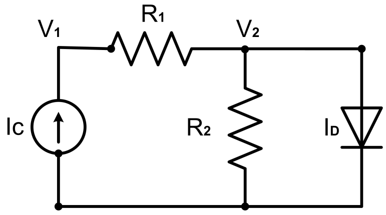


FIGURE 3.2: Example Circuit

$$\text{Node } 1 : I_C + (V_2 - V_1)/R_1 = 0$$
$$\text{Node } 2 : (V_2 - V_1)/R_1 + V_2/R_2 + I_D = 0$$

Where the diode current is given by the device model equation as follows:

$$I_D = I_S(e^{V_2/V_j} - 1) \tag{3.9}$$

where I_S is the Saturation current and V_j is the Junction potential [40], I_C is the input current of the current source. The circuit equations are then aligned into the matrix form $Ax = b$ as follows:

$$\begin{bmatrix} 1/R_1 & -1/R_1 \\ 1/R_1 & -(R_1+R_2)/R_1R_2 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} I_C \\ I_D \end{bmatrix}$$

Gaussian elimination is then used to form the following linear system:

$$\begin{bmatrix} 1/R_1 & -1/R_1 \\ 0 & -1/R_2 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} I_C \\ I_D - I_C \end{bmatrix}$$

The NR method is first used to solve the Equation 3.10 for V_2 using the Equation 3.11 before back substitution to find V_1.

$$F(V_2) = \frac{1}{R_2}V_2 + I_S(e^{V_2/V_j} - 1) - I_C \tag{3.10}$$

Therefore we have:

$$V_{2_{n+1}} = V_{2_n} - \frac{F(V_{2_n})}{F'(V_{2_n})} = V_{2_n} - \frac{\frac{1}{R_2}V_{2_n} + [\mathbf{I_S(e^{V_{2n}/V_j} - 1)}] - I_C}{\frac{1}{R_2} + [\mathbf{\frac{I_S}{V_j}(e^{V_{2n}/V_j})}]} \tag{3.11}$$

The **Device Model Evaluation Phase** evaluates the equations that represent the current and the conductance of the diode. These are shown in bold in Equation 3.11. The aim of this research project is to evaluate these device model equations on FPGAs.

3.2 SPICE Simulator Parallel Execution

3.2.1 Parallelisation Approaches

SPICE simulation has two main phases where computations can be parallelised. The first phase is the Device Model Evaluation, in which non-linear device models are evaluated (e.g. diodes, transistors). The other phase is the Linear Solver using the common LU decomposition as detailed in [130, 131] and Figure 3.1. A number of approaches have been introduced to parallelise the SPICE circuit simulation, each exploiting a different level of granularity. In the model evaluation phase, device models and their corresponding derivatives are computed. These computations are performed independently from each other, which allow them to run on different processes in parallel. The parallelism approaches are categorised into two main streams: Direct and Iterative Methods [44].

Direct Methods present one of the main approaches known in literature to parallelise the key phases in the SPICE algorithm. Direct linear solvers typically find the final solution via computationally intensive matrix factorization [132], whereas iterative methods refine a solution with each iteration [133]. These two approaches are mainly concerned with parallelising the sparse linear solver, as the device model evaluation is always performed to form the linear system $Ax = b$ as explained in Section 3.1.

Direct Methods are usually more robust in solving linear systems than Iterative solvers. However, they suffer from fill-in which results in longer execution times with higher memory requirements which effect its scalability in parallel environment. Pivoting and reordering techniques are introduced to overcome such issues. Iterative methods, even though less robust, they requires far less memory and the execution time can be lower than direct methods if convergence is achieved in relatively few iterations [134].

In [135, 136], Waveform Relaxation techniques (WR) were proposed on supercomputers, in which parts of the circuit are solved independently. These techniques are not widely used for typical designs as WR converges slowly, i.e. their convergence properties are limited [137]. Relaxation-based simulation may converge slower and simulation time can be larger than direct methods [138].

The approach proposed in [139] is based on the domain decomposition of the SPICE algorithm, in which the linear system of the Differential Algebraic equations is decoupled into smaller linear systems which can be executed on separate processes. The efficiency of the method is strongly based on the type of application, which significantly limits its use. Domain decomposition works for small systems in which the number of communication nodes should be small. In other words, the efficiency drops as the number of interface nodes increases [139, 137]. A modified overlapping domain decomposition techniques was proposed in [137] which uses the Schwarz method.

WavePipe approach in [140] extends the classical time-integration methods in the SPICE simulator. It takes advantage of both multi-threading fine-grained parallelism at the numerical discretisation level and the coarse-grained application parallelism. This is done via simultaneously computing circuit solutions at multiple adjacent time points.

A Multi-Algorithm Parallel Circuit Simulation (MAPS) approach is proposed in [141], where different simulation algorithms are started simultaneously on different threads for a single simulation task. Threads are synchronised dynamically to pick the best performing algorithm at every time point. In other words, this approach tries to find the best result for each time point by trying a number of algorithms in parallel. This is likely to add to the computational requirement of the SPICE simulation as some threads will not be considered because of their low performance in certain time points.

The techniques discussed above have either limited applicability or slow convergence, which limit their use for circuit simulation acceleration [134]. The direct methods are more robust in parallelising the linear solve and can take advantage of the hardware acceleration to match or exceed the performance of the other techniques. This emphasises the importance of speeding up the device evaluation phase as it is the first phase that have to be performed before applying any of the linear solve approaches discussed above.

3.2.2 SPICE Simulator Execution Profile

Device model evaluation is an important phase of the SPICE circuit simulator. It is characterised by large irregular floating-point compute graphs. These graphs represent the mathematical equations of the device models which make heavy use of floating-point operations. The device evaluation runtime grows linearly with the number of nodes in the circuit $O(n)$. On the other hand, the complexity of the linear solver phase ranges from $O(n^{1.2})$ to $O(n^{1.5})$, where n is the size of the matrix, when efficient sparse matrix techniques are used [44].

The time consumed by the simulator is typically divided between the two simulator phases, in other words, most of the simulation runtime is spent performing these two tasks. The runtime of the matrix solvers, however, does not scale well with the number of processors used, as demonstrated in [142, 143]. For small circuits, the device evaluation phase dominates the runtime and grows linearly with circuit size (n). The matrix solver, however, dominates the runtime as the circuit size grows, because the solver time increases approximately to the power of 2 of the number of circuit nodes [28, 144].

In [42], the profiling experiments using the Berkeley Short-channel IGFET Model (BSIM3) [145] models showed that on average 75% of the SPICE simulator runtime is spent evaluating device models. Device evaluations are generally performed for each device in the circuit and for each time step, until convergence is reached by the NR solver. At this point, the number of device models evaluated can reach an enormous figure. The profiling results in [42] showed that a benchmark design containing 324 CMOS devices requires 1.86×10^7 BSIM3 device model evaluations over the whole simulation. Hence, the speed of calculating these device evaluations can have significant impact on the overall performance of the whole simulation flow [146]. In [147], an analysis of a test suite of 27 circuits based on the BSIM4 transistor model code showed that nearly 66% of the transient runtime is spent evaluating the transistor models.

Furthermore, the profiling results presented in [43] showed that for circuit dominated by non-linear transistor devices with no parasitics, in which the Spice3f5 simulator can spend almost half its runtime evaluating device models. For circuits dominated by

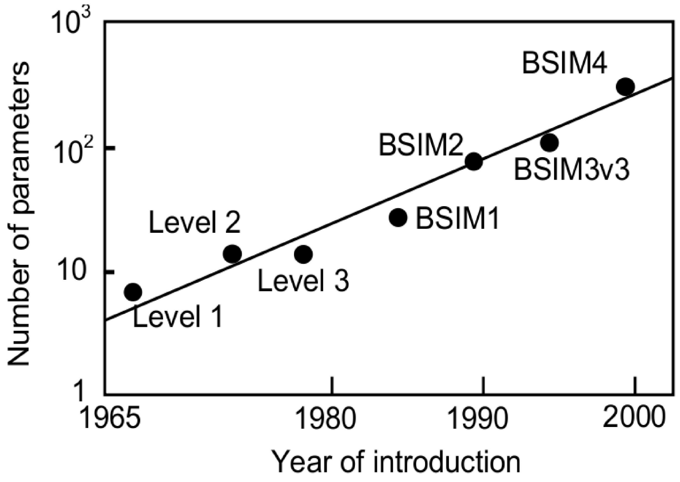


FIGURE 3.3: The Increase of MOSFET Models' Parameters [149]

linear parasitics simulation runtime may be dominated by the linear solve phase. It is, however, still important to quantify the amount of runtime reduction when accelerating the device evaluations through parallelisation, even though it is not the main time-consuming phase.

In addition to the increasing number of devices in circuits according to Moore's Law (Figure 1.1), the mathematical models themselves are becoming larger and more complicated. With the device process being scaled down, the complexity of device models grows over time in order to simulate the electrical and physical behaviours of devices accurately. This is represented in a complexity increase of about 4 to 5 times that of classical BSIM3 model, as estimated in [43, 148]. Study [43] estimated the increase in complexity by observing the increase in the number of the model parameters introduced into each new model. Figure 3.3 illustrates the increase in the number of parameters of the most known MOSFET (Metal Oxide Semiconductor Field-Eect Transistor) models [149].

This means that the device evaluation's computational requirements increase with time according to the increasing complexity of the device models. These findings, if coupled with the increasing number of devices per chip, dictates the need to accelerate the device evaluation to cope with future circuits.

While direct methods used in solving sparse linear systems found in circuits are more robust than iterative methods, our research on parallel device model evaluation becomes also important when iterative methods are used. This is because iterative algorithms are easier to parallelise than direct methods and hence a significant computation time is spent evaluating device models [150]. In other words, the iterative methods are used to solve the linear system through parallelisation which reduces the overall execution time of the SPICE simulator. Therefore it is important to accelerate the device model evaluation to reduce the simulator runtime further.

The work presented in this thesis is based on parallelising the device model evaluation phase. More precisely, performing the device evaluation of the CMOS transistor model in parallel on a number of FPGAs. All the non-linear equations representing CMOS devices will be computed in parallel using a multi-FPGA system. It is assumed that the number of transistors in a digital circuit is approximately the same number of nodes N. Hence, the hardware acceleration would perform N number of device evaluations per iteration per time step in a transient analysis. The overall number of device evaluations performed can reach enormous figures by the end of the simulation (e.g. 1.86×10^7 [42]).

3.3 SPICE Simulator Acceleration Related Work

A number of studies have explored the use of both conventional multiprocessors and hardware accelerators (FPGAs and GPUs) to implement the parallelisation approaches outlined in Section 3.2, some of which are detailed here.

3.3.1 Multiprocessor Paradigm

One of the earliest attempts to parallelise the circuit simulation was proposed in [46], which used the Alliant FX/8 shared-memory multiprocessor system with six processors. The main focus was to accelerate the transient analysis. PARASPICE simulator in [151] used a similar shared-memory multiprocessor system to accelerate the device load and linear solve phases.

In [150], accelerating device model evaluation using the PACE distributed memory multiprocessor system, with a four-processor cluster, was proposed. The aim was to accelerate the transient analysis of the AT&T ADVICE circuit simulator. The experimental results showed an acceleration of about 3.6 times for small example circuits.

A highly-parallel electronic simulator called Xyce is presented in [152, 153]. It is specifically designed for supercomputers using a message passing parallel implementation. The simulator uses weighted graphs and graph decomposition heuristics to partition the circuit graph to facilitate load-balancing between processors and reduce communications costs. The simulator showed an acceleration of about 24 times on 40 processors solving a transmission line problem with up to 140 thousand elements [152].

The study of [144] used multi-threaded implementation based on pthreads (pthreads is a POSIX -Portable Operating System Interface for Unix- standard API for creating and manipulating threads) and reported a speedup figure of about 5 times on 8 processors. The study showed a scaling trend with the number of processors without loosing accuracy. However, pthreads requires major code rewriting and porting effort as it is a low-level programming model which is particularly useful for task parallelism. The modified code would be difficult to maintain taking into account the large number of calls to pthreads library functions and explicit coding of parallelism [154].

OpenMP parallelisation approach was used in [154] to parallelise the existing the SPICE device evaluation code in SPICE3. The implementation demonstrated speedup and scaling figures, where the acceleration saturates at 2–3 times with 4 processors. The main issue with this implementation is the significant fork-join overhead incurred as the number of threads increases. Also it was not possible to perform much parallelism without great modifications of the code.

WavePipe approach introduced in [140] exploits the coarser-grain parallelism of the time-domain transient analysis by simultaneously evaluating circuit solutions at multiple time steps (i.e. parallelism along the time axis). The speedup reported was approximately 3 times using 8 processors. The approach does not focus specifically on accelerating the individual steps like the device evaluation.

Table 3.1 summarised the recent work done in the area of the SPICE simulator acceleration through parallelisation. The table shows the number of processors used in each system and the speedup results. The table also shows the speedup per processor figures (4^{th} column), which is considered to be low if compared to the speedup-per-chip results reported for FPGAs. Hence, FPGAs have a great potential to deliver respectable speedups per-chip in accelerating the SPICE simulation.

Table 3.1: Multiprocessor based SPICE simulator acceleration (Previous work)

Year	System	Proc	Speedup/Proc	Approach
2001 [152]	SGI Origin 2000 (MIPS)	40	0.6	Xyce Simulator
2002 [144]	Hitachi N4000 (PA-RISC 8600)	5	0.625	PThreads
2007 [154]	Sun Fire V880 (Ultrasparc-III)	4	0.75	OpenMP Pragmas
2008 [140]	High-end workstation	8	0.375	WavePipe, pthreads

3.3.2 Hardware Accelerator Paradigm

In recent years, hardware acceleration based on FPGAs and GPUs have been used to accelerate EDA tools [32, 155]. In general, FPGAs are highly customisable while providing a good expectation of performance, flexibility and low overhead. GPUs, however, tend to provide massive parallel execution resources and high memory bandwidth, while being easier to program and require less hardware resources [156, 157, 158]. Each hardware platform (FPGA and GPU) is suitable for specific application depending on its architectural requirements as explored in [159, 157].

3.3.2.1 GPU Accelerators

GPUs have also been used to accelerate the device model evaluation. Double-precision implementation in [160] showed speedup figures of 10-50 times over a quad-core AMD CPU when using an AMD Firestream 9170 GPU which contains 512 processors. [42] implemented a single-precision device evaluation accelerator which demonstrated speedup figures of 32-40 times over a quad-core Intel CPU when using an NVIDIA 8800 GTX GPU with 128 processors.

Recently, Nascentric announced the first SPICE simulator hardware accelerator, OmigaSim GX, with a speedup factor of nearly 10 times the performance of current simulators [161]. The accelerator is based on the NVIDIA Tesla C-870 PCI express add-in GPU Card [3]. The implementation is based on the fast-SPICE methodology which uses lookup tables for device evaluation.

The GPU platform used in [161] provides a massively parallel multi-threaded architecture with 128 to 512 cores depending on the system configuration. The speedup improvement is achieved by dispatching the transistors models evaluations to the GPU cores, which performs them in a fraction of the time that would be taken by a CPU core. Nascentric claims that when running the SPICE simulator in its most accurate mode, nearly ninety percent of the CPU time during simulation is spent evaluating transistors, as these evaluations are computationally intensive [161]. In [147], circuit simulation is sped up by a factor of 3 to 6, by performing the transistor model evaluation on the NVIDIA GTX 280 GPU.

3.3.2.2 FPGA Accelerators

A VLIW architecture was proposed in [162, 138, 163] to accelerate the device model evaluation process on FPGAs. The Awsim-3 architecture in [138] used lookup tables to perform the model evaluations in order to reduce the resources usage, and hence make the implementation feasible. The FPGA implementation of the Awsim-3 architecture of the device evaluation in [162] is called TINA which uses the Marc-1 reconfigurable board that contains 9 XC4005 FPGAs. The TINA system used table lookup for device evaluation in order to trades-off accuracy for resources utilisation. The study did not provide information about the speedup figures.

An FPGA-based implementation of an NMOS LEVEL 1 model using Signal-Processing Object (SPO) is presented in [41]. A design methodology was proposed which uses

Simulink to design the SPOs first then to map the model onto the FPGA based on fixed-point operations. This approach reduced the cost of resources utilised. However, the use of fixed-point operations would add to the the accuracy convergence issues faced by the SPICE simulator. Hence, this system compromises accuracy to reduce the area requirements.

A pipelined VLIW-scheduled architecture was proposed in [43, 155] to accelerate the device evaluation step using a single FPGA implementation. This demonstrated an acceleration of 2-18 times over a dual-core 3GHz Intel Xeon 5160 when using a Xilinx Virtex 5 LX330T for a variety of SPICE device models. The study reported a speedup figure of approximately 10 times for the MOS3 model which is the same model used in our system (Section 4.3). A similar system was used by the authors to parallelise the sparse matrix solver as shown in [164].

Our approach is based on a spatial implementation the CMOS LEVEL 3 model on single FPGA. This is then mapped on a number of FPGAs in parallel to perform the device evaluation process. The spacial implementation is deeply pipelined to provide the maximum throughput. The system proposed in this thesis exploits a different parallelisation approach based on the larger FPGAs currently available today to provide performance improvements without any loss in accuracy.

3.4 Parallel Device Model Evaluation

The process of parallelising the SPICE circuit simulator based on direct methods essentially reduces to designing parallel algorithms for the two most time-consuming tasks: device evaluation and linear solve. This thesis focuses on the parallelisation of the device evaluation phase. The linear solve phase was the focus of Tarek Nechma in [165].

3.4.1 Explicit Parallelism Approach

Figure 3.4 shows the basic diagram of a single Newton-Raphson iteration, where the device evaluation and the linear matrix solution are executed in parallel on different threads (T1 to T6). The device evaluation phase has to be completed before the linear solver finds the solution to the Nodal Matrix. The linear solver phase must terminate before the device evaluation from the next NR iteration can start. These two barriers limit the amount of parallelism that can be performed. Each thread of the device evaluation and the linear matrix solution can be executed on a separate FPGA. Each thread in the device model evaluation phase simultaneously evaluates a number of device models in the circuit as there is no interaction between devices.

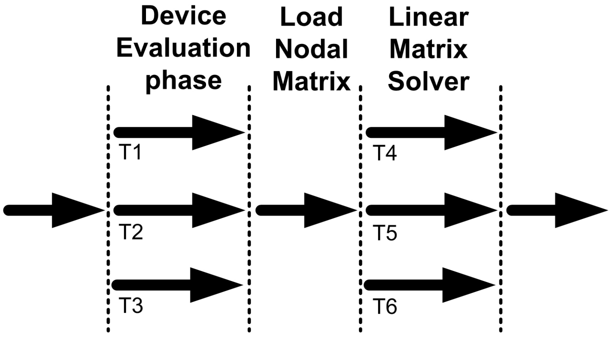


Figure 3.4: Parallel Execution of a Newton-Raphson Iteration in the SPICE simulator

Each Newton-Raphson iteration requires the evaluation of both the CMOS model and its derivative. This poses an issue as FPGAs are usually limited in terms of resources, and both the model and its derivative might not fit into a single FPGA. In this case, Secant method may be worth investigating as it does not require calculating derivatives. This is further discussed in Section 5.2.4.

The LU decomposition method can also be performed in parallel. Due to the nature of electronic circuits, the nodal matrices tend to be very sparse. In other words, many of the off-diagonal elements in the linear equation are zeros. This is because the off-diagonal terms are generated by conductances connected between pairs of nodes, and nodes in general are connected to only two or three other nodes in the circuit [40, 126]. This topic is out of the focus of this research project. For further details, refer to [131, 130, 2, 166, 165, 167].

3.4.2 Pipelined Approach

Figure 3.4 showed the parallel execution of the SPICE simulation iteration based on a of parallel threads. Each of these threads would be executing on a single FPGA. In order to improve the performance further, pipelined designed can be used to provide constant throughput. Figure 3.5 demonstrated the pipelined approach to perform the NR iteration.

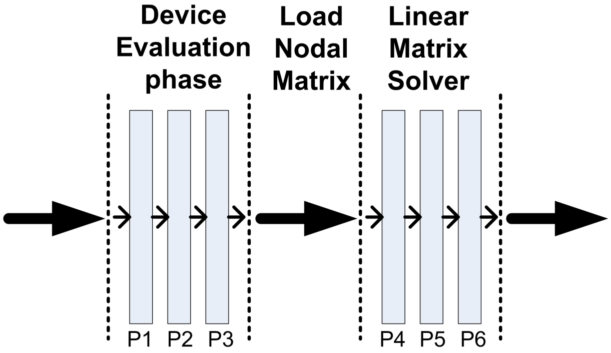


Figure 3.5: Pipelined Execution of a Newton-Raphson Iteration in the SPICE simulator

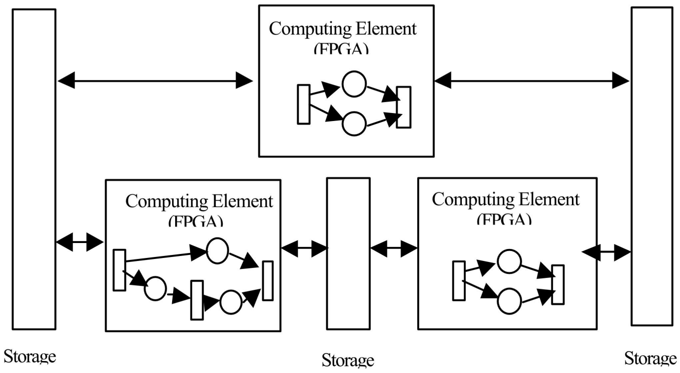


Figure 3.6: Pipelined Configuration of Multi-FPGA Systems [168]

Due to the limited resources of FPGAs, it might not possible to fit a single execution thread (i.e. the models or their derivatives) into a single FPGA. These models could be partitioned and mapped onto a number of FPGAs and pipelined to improve the overall performance as seen in Figure 3.6 [168]. Each partition shown in the Figure 3.5 is assigned to one FPGA (P1 to P5 partitions). A number of FPGAs have to be used to realise this approach. This will provide a constant throughput, but the inter-FPGA communications are expected to have heavy impact on the overall performance of the system.

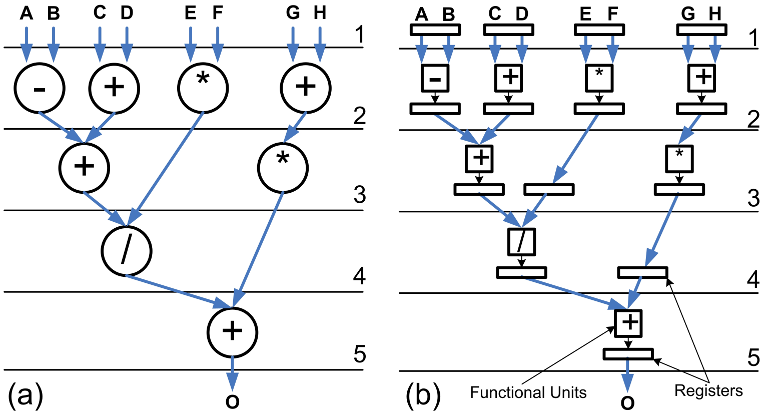


Figure 3.7: (a) The ASAP schedule (Section 2.1.1) (b) a pipelined implementation of the schedule

Each instance of the device model can be pipelined in such a way that results are produced every clock cycle. This approach would take the device model in the form of a DFG graph and adds registers between operations to allows us to start a device evaluation every clock cycle. Figure 3.7 shows the pipelined DFG with registers inserted to buffer intermediate results. In this approach, all the operation have to be mapped on hardware spatially and cannot be re-used. This is because computations are being initiated every clock cycle. This is what is known as Spatial Implementation.

3.4.3 Proposed FPGA Accelerator

Our proposed architecture is based on exploiting three degrees of parallelism embedded within the device model evaluation phase. The first degree is the data parallelism explored in Section 3.4.1 as device model evaluations can be performed independently from each other. We can map multiple instances of the same model mapped onto a number of FPGAs to have a parallel architecture. The second degree to be exploited is the pipeline parallelism approach explored in Section 3.4.2. The third degree is the basic fine-grain instruction level parallelism at the functional units level. Figure 3.8 shows the different degrees of parallelism inherent in the device model evaluation. Our multi-FPGA discussed in this thesis exploits this parallelism to maximise the FPGA acceleration.

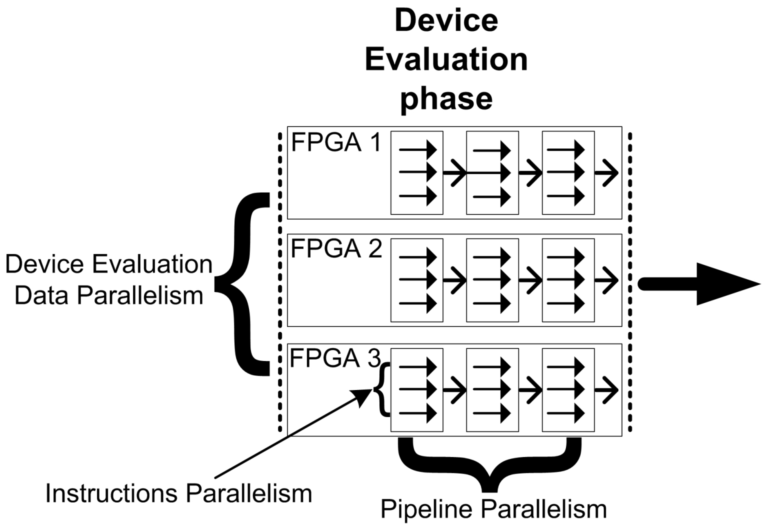


Figure 3.8: The proposed approach to exploit the inherent parallelism in the device model evaluation phase

Our application specific architecture can be used as a high speed co-processor attached to workstations to boost SPICE-like simulations. Figure 3.9 shows how the outcome of this thesis can be used to accelerate the SPICE simulator. The Host PC sends the bulk computations to the FPGAs to perform the Device Model Evaluation and the Linear Solver phases. After computations are completed, the FPGAs would send the resulted nodal voltages back to the Host PC. This process continues until the end of simulation is reached. It is shown in the figure that each simulator phase is mapped to one FPGA only. This is just to illustrate the concept of an FPGA accelerator for the SPICE simulator. More FPGAs can be used to implement any of the SPICE simulator phases if required.

3.5 Summary

The SPICE simulator has two main phases where computations can be parallelised: the device model evaluation and the linear solver. Profiling results in Section 3.2.2 showed that considerable amount of the simulator time is spent in evaluating the device models. Device evaluations are generally performed for each device in the circuit and for each time

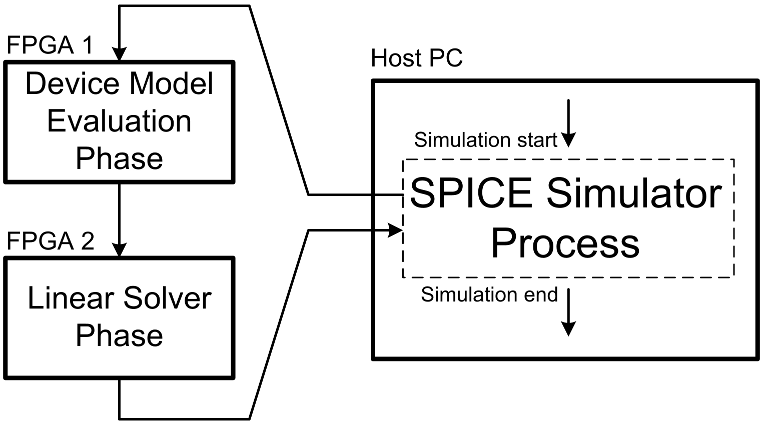


Figure 3.9: A suggested FPGA coprocessor to accelerate the SPICE simulator

step, until convergence is reached. At this point, the number of device models evaluated can reach enormous figures (e.g. 1.86×10^7 [42]). Hence, the speed of calculating these device evaluations can have significant impact on the overall performance of the whole simulation flow. In addition, the increasing complexity of device models and the increasing number of devices per chip dictates the need to accelerate the device model evaluation to cope with future circuits. It is, however, still important to quantify the amount of runtime reduction when accelerating the device evaluation phase through parallelisation, even though it is not the main time-consuming phase.

As device model evaluations can be performed independently from each other, it would be sensible to have multiple instances of the same model mapped onto a number of FPGAs to have a Single Instruction Multiple Data (SIMD) architecture. Furthermore, each instance can be pipelined in such a way that results are produced every clock cycle. This approach would take the device model in the form of a DFG graph and adds registers between operations to allows us to start a device evaluation every clock cycle as seen in Figure 3.7. This architecture allows us to exploit the fine-grained parallelism nature of FPGAs in two ways. The first way is to allow multiple instructions to run in parallel and the second way by pipelining each operation to start new computation every clock cycle. In short, our proposed approach exploits the inherent data parallelism in the SPICE device model evaluation, in addition to the instructions and pipeline parallelism. This proposed architecture is one of the main contributions of this thesis.

While we used the direct method for solving the matrix equations, the research on parallel evaluation becomes more meaningful for relaxation-based algorithms. This is because such iterative algorithms are simpler to parallelise which leaves the bulk computation time in the model evaluation phase.

Chapter 4

Multi-FPGA Device Model Accelerator

Chapter 3 briefly explained the SPICE simulation process and the different parallelism approaches that can be exploited in acceleration. It also outlined our approach to exploit such parallelism in the simulator at different levels. This Chapter presents the system architecture to implement our approach based on a multi-FPGA system built using the off-the-shelf Xilinx Virtex-II Pro FPGA boards. A single FPGA implementation is first considered in order to evaluate the acceleration and resources results of the CMOS device model. A multi-FPGA system is then prototyped to demonstrate the amount of acceleration that can be achieved through parallelism.

Section 4.1 outlines the FPGA design considerations that have to be taken into account when implementing the SPICE device evaluation on FPGAs. The mathematical description of the CMOS LEVEL 3 model used in the multi-FPGA accelerator is presented in Section 4.2 and Section 4.3. The transformation flow to transform the device model from a high-level description to a synthesisable design is described in Section 4.3.2. A single-FPGA accelerator was tested in Section 4.4. Sections 4.5 and 4.6 present the design, implementation, and experimental setup of the multi-FPGA system.

4.1 FPGA Design Considerations

4.1.1 Pipelined Architecture

Generally, the CMOS models used in current SPICE simulators are inherently sequential. Hence, extracting the low level parallelism from these models will not result in the acceleration aimed for. Therefore, in order to exploit the hardware acceleration capabilities embedded in FPGAs, a pipelined version of the models has to be designed. Computations in a pipelined architecture are cascaded together with registers inserted in between to hold the intermediate values after each clock cycle as explained in Section 3.4.2.

This would produce constant throughput after a certain start-up delay [168]. However, the pipelined implementation usually utilises a large amount of logic and interconnect on FPGAs. In addition, the CMOS models tend to be large in terms of the number of floating-point operations to be performed. Furthermore, the floating-point operators themselves would have to be pipelined in order to ensure large throughput. Therefore, a trade-off between performance and resources usage is necessary.

4.1.2 SPICE Model Parameters

In order to reduce the amount of logic utilised, it is assumed that the model parameters are pre-calculated. In other words, the circuit simulator uses a single transistor model during simulation. This assumption is based on the fact that, in most cases, the same parameters are used to describe all the transistors in the same chip. In addition, CMOS model parameters are independent of the nodal voltage values; which means that they stay fixed during the SPICE simulation process. Typically, the SPICE simulator pre-calculate the models given the pre-set parameters, and use the resulting simplified models during the rest of the iterations (i.e. transient analysis or NR iterations).

If different manufacturing process is to be targeted, the model parameters are changed accordingly. The new parameters are then pre-calculated to produce a new transistor model ready for FPGA implementation. The given assumption allows the synthesis system to apply compiler optimisation techniques like constant propagation at the scheduling level. At this level, operations involving constant operands are performed by the synthesis system and their results are substituted in subsequent operations. The parameters can be changed at the synthesis level to produce the required transistor model needed for simulation.

4.1.3 Data Word Length Considerations

Generally, SPICE simulation suffers from a number of accuracy problems. These issues are classified into either topological or numerical [40, 169]. The topological problems are due to the nature of circuits such as zero diagonal terms. These are usually solved by preordering and pivoting techniques.

The numerical aspect of the problem occurs due to the limited finite precision that computers utilise to represent the nodal matrix terms. This can lead to the loss of the significance of a term in the matrix to another during the solution of the linear equations. This can also introduce some stability issues for example when dividing by near-zero values. Hence, a large dynamic range is required in circuit simulation in order to avoid such numerical issues. Given the accuracy requirements of the SPICE simulation, single-precision floating-point would be the minimum usable precision [126].

As seen in Appendix F.1, the fixed-point number system has a limited dynamic range if compared to the floating-point format. This is the main issue with the FPGA-based MOS accelerator presented in [41], as fixed-point operations were used. Although, the resources utilisation is reduced, the system compromised accuracy to reduce the resources requirements.

Software implementations of the SPICE simulator usually employs double precision floating-point arithmetic. Single-precision based SPICE simulators are also used. FPGAs, on the other hand, are still constrained in terms of logic resources when it comes to performing high-precision floating-point computations. The implementation of floating-point applications on FPGAs is a challenging task since the basic operators require a significant amount of resources as explained in Section 2.3.4 and Appendix F.

Typically, the FPGA resources cost increases linearly with precision for adders and quadratically for multipliers [130]. A similar or higher cost increase in resources requirements would apply to the other floating-point operations like division, exponential, and logarithm [120]. In order to reduce the cost of hardware resources and to improve performance, some operators can be partially mapped using either combinational blocks or efficient built-in circuits like the on-chip 18×18 multipliers and the DSP48E blocks for Xilinx FPGAs. However, this approach is still limited and the implementation of the CMOS model on FPGAs with higher precision cannot always be implemented.

Using single-precision computations is expected to introduce a small loss in simulation accuracy over higher precisions. In [155], it was shown that convergence can still be reached at single-precision accuracy by relaxing the simulator's default tolerance parameters `reltol` (relative tolerance), `abstol` (absolute current tolerance), and `vntol` (absolute voltage tolerance). The parameter `reltol` was relaxed from $1e^{-3}$ to $1e^{-2}$ (accuracy of 1 part in 100), `abstol` was relaxed from $1e^{-12}$ to $1e^{-11}$ (accuracy of 10 picoAmperes), and `vntol` was relaxed from $1e^{-6}$ to $1e^{-3}$ (accuracy of 1 milliVolt). Although a 10% increase in the SPICE simulator iterations was observed, it is expected that single-precision model evaluation runs faster with a slight loss in the quality of results.

4.1.4 Inter-FPGA Serial Communication

In the context of multi-FPGA systems, synchronisation between FPGAs is a very important aspect affecting system performance. A number of approaches are in use to synchronise multi-FPGA communication operations. The high-speed serial communication is preferred in this context due to its simplicity of design and relatively high-bandwidth. From a system point of view, the off-chip serial communications have to be transparent, fast, and easy to integrate with the on-chip communication sub-system. An example serial interface is Aurora described in Section 2.3.5.

4.2 CMOS LEVEL 3 Model

SPICE simulation of analogue circuits uses built-in non-linear mathematical equations to model the behaviour of the physical devices. For CMOS transistors, a number of models exist which take a number of physical and empirical parameters. The LEVEL 3 model is a semi-empirical model described by a number of parameters which are defined by curve-fitting approach rather than physical background [40].

This model is one of the fundamental well known CMOS models. Although it is an old model, however, it was chosen because of its wide acceptance in the EDA community in addition to its relative simplicity to be implemented in our prototype system. One of the features of this model is that it simplifies a complex equation with many parameters into a simpler equation with fewer parameters under specific bias conditions. This led to the high acceptance and long life of this model [170].

The basic drain current equation for the CMOS LEVEL 3 model when the transistor is operating in the linear region is given by:

$$I_{DS} = \beta(V_{GS} - V_{TH} - \frac{1+F_B}{2}V_{DS})V_{DS} \tag{4.1}$$

and:

$$\beta = \mu_{eff} C_{ox} \frac{W}{L_{eff}} \tag{4.2}$$

Where: F_B is the coefficient of bulk charge, V_{TH} is the threshold voltage, V_{DS} is the drain source voltage, V_{GS} is the gate source voltage. In Equation 4.2, μ_{eff} is the charge-carrier effective mobility, W is the gate width, L_{eff} is the effective gate length and C_{ox} is the gate oxide capacitance per unit area.

When V_{DS} is small compared to the value of $(V_{GS} - V_{TH})$, the I_{DS} equation is given by:

$$I_{DS} = \beta(V_{GS} - V_{TH})V_{DS} \tag{4.3}$$

The CMOS model takes into account a large number of parameters. The physical parameters of LEVEL 1 reappear, as well as new parameters. Some of these parameters are of an empirical nature and others have a physical origin. The accuracy of the model depends heavily on the values of the input parameters. These input parameters are related to the particular process used at each manufacturing site [170].

Generally, this transistor model takes three voltage values V_{DS}, V_{GS}, and V_{BS}. The three voltages are used to calculate the nodal currents I_{DS}, which are used by the Newton-Raphson algorithm to form the nodal matrix given by Equation 3.5. The latter is then solved using methods like LU factorisation. The mathematical equations presented in Section 4.3 covers the drain-current calculation only $(F(V^i))$. The full mathematical description of the model is presented in [40].

4.3 CMOS LEVEL 3 Model FPGA Implementation

4.3.1 CMOS LEVEL 3 Parameters

The device model used in this section is based on the CMOS LEVEL 3 model as part of the Southampton VHDL-AMS Validation Suite in [171]. The VHDL-AMS code of the model is shown in Appendix A. Due to the resources limitation of FPGAs, the CMOS LEVEL 3 model implementation cost should be within the available hardware resources. In order to achieve this objective, the model implementation was carried out in two steps:

1. The full CMOS LEVEL 3 model in [172] is implemented, where all model parameters are supplied as inputs to the design each time the model is executed. The

same full model is used for all device evaluations in every iteration until the end of simulation.

2. The parameters are fixed to the default parameters as shown in Table 4.1 on page 67 [40]. Once the parameters are set, the simplified CMOS model is used to perform the model evaluations. This step uses constant propagation to pre-calculate all the operations involving constant-only operands. Once all parameters are set, only the nodal voltage values are supplied as inputs. The resulted CMOS LEVEL 3 model code -after fixing all the parameters- is shown in Figure 4.1 on page 68, where all intermediate variables were calculated. This algorithm actually represents a transistor model that corresponds to the technology parameters shown in Table 4.1. This was explained in Section 4.1.2.

After pre-calculating all the device model parameters, the resulted device model code needs to be mapped onto our multi-FPGA system. Due to the custom nature of this task, there is no tool that can be used readily off-the-shelf. Hence, a manual device model code transformation has to be considered. This flow is explained in the next section (Section 4.3.2).

4.3.2 Device Model Code Transformation Flow

A manual transformation flow was embarked due to the large complexity and the long time scale needed to develop a compiler to transform the high-level device model code to a synthesisable code. The transformation flow involves a number of steps which are shown in Figure 4.2 on page 69.

The device model parameters in the high-level VHDL-AMS code are first assigned to their default values shown in Table 4.1. The resulted formulae after fixing the model parameters are listed in the code shown in Figure 4.1, where the intermediate variables were already calculated and their values are shown. Figure 4.1 shows an irregular floating-point DFG as seen in Figure 4.3 on page 70. The resulted model code represents a device model that correspondences to a specific transistor technology. This is basically

Table 4.1: CMOS LEVEL 3 Parameters and default values set at the synthesis level [40]

Name	Parameter	Value	Unit
WIDTH	Width	1.0e-4	m
LENGTH	Length	1.0e-4	m
CHANNEL	Channel type	1.0	-
VTO	Threshold voltage	$-\infty$	V
KP	Transconductance parameter	2.0e-5	A/V^2
GAMMA	Bulk threshold parameter	0.0	$V^{1/2}$
PHI	Surface potential	0.6	V
TOX	Thin-oxide thickness	1.0e-7	m
NSUB	Substrate doping	0.0	cm^{-3}
NSS	Surface state density	0.0	cm^{-2}
NFS	Fast surface state density	0.0	cm^{-2}
TPG	Type of gate material	1.0	-
XJ	Metallurgical junction depth	0.0	m
LD	Lateral diffusion	0.0	m
UO	Surface mobility	600.0	$cm^2/V.s$
VMAX	Maximum drift velocity of carriers	0.0	m/s
XQC	Thin-oxide capacitance model flag and channel charge share for drain coefficient	1.0	-
KF	kf	0.0	-
AF	af	1.0	-
FC	Forward Bias Non-Ideal Junction Capacitance Coefficient	0.5	-
DELTA	Width effect on threshold voltage	0.0	-
THETA	Mobility modulation	0.0	V^{-1}
ETA	Static feedback coefficient	0.0	-
KAPPA	Saturation feild factor	0.2	-
NGATE	Poly Si-gate doping concentration	1.5e19	cm^{-3}
TEMP	Temperature	300.0	K

a DFG with is a set of floating-point operations that are executed on the input nodal voltages (V_{ds}, V_{gs} and V_{bs}) and return currents and charges values (I_{ds}, Q_b and Q_c).

The DFG shown in Figure 4.1 is then statically scheduled using the TORSCHE Scheduling Toolbox for Matlab in [173]. This tool schedules the operations in algorithm shown in Figure 4.1 using the ASAP scheduling approach, by assuaging each operation to a specific control step. In other words, each operation in the model is assigned both a start and a finish times as seen in Section 2.1.1. This tool was used to schedule the operations automatically as this process is tedious and error prone to be conducted manually.

```
**Inputs:** $V_{ds}$, $V_{gs}$ and $V_{bs}$

$V_{fb} = -0.1175$
$V_{gstos} = V_{gs} - V_{fb}$
$V_{gst} = max(V_{gstos}, 0)$
$V_{th} = V_{fb} = -0.1175$
$beta = kp = 2.0e - 5$

**if** $(V_{gs} \geq V_{th})$ **then**
    $V_{pp} = min(V_{ds}, V_{gst})$
    $I_t = V_{gst} - V_{pp} * 0.5$
    $I_{ds} = beta * V_{pp} * I_t$
**else**
    $\{Cutoff\ mode\}$
    $V_{pp} = 0$
    $I_t = 0$
    $I_{ds} = 0$
**end if**

$cox = 3.4531e - 12$

**if** $(V_{gs} \leq V_{th})$ **then**
    $Q_g = 0$
    $Q_b = 0$
    $Q_c = 0$
**else**
    $\{Depletion\ mode\}$
    $R = V_{pp} * V_{pp}/(12.0 * I_t)$
    $Q_g = cox(V_{gstos} - V_{pp} * 0.5 + R)$
    $Q_c = -cox(V_{gst} + (R - V_{pp} * 0.5))$
    $Q_b = -(Q_c + Q_g)$
**end if**
**return** $I_{ds}, Q_b$ and $Q_c$
```

Figure 4.1: LEVEL 3 CMOS Model with Parameter Pre-calculation

The scheduling/timing information of the operations are then used to manually create the structural VHDL implementation by instantiating and cascading the floating-point operations together according to their data dependency and their start/finish times. Intermediate registers are added next to hold the output values between operation after each clock cycle. This allows the operators to perform a new computation every clock cycle. Computations are performed using the single-precision deeply pipelined operators in FPLibrary [119] (Appendix F.2.3). The device model is implemented spatially, in which the floating-point operators were not shared amongst different operations. The resulted structural VHDL is then simulated using ModelSim and the output is compared

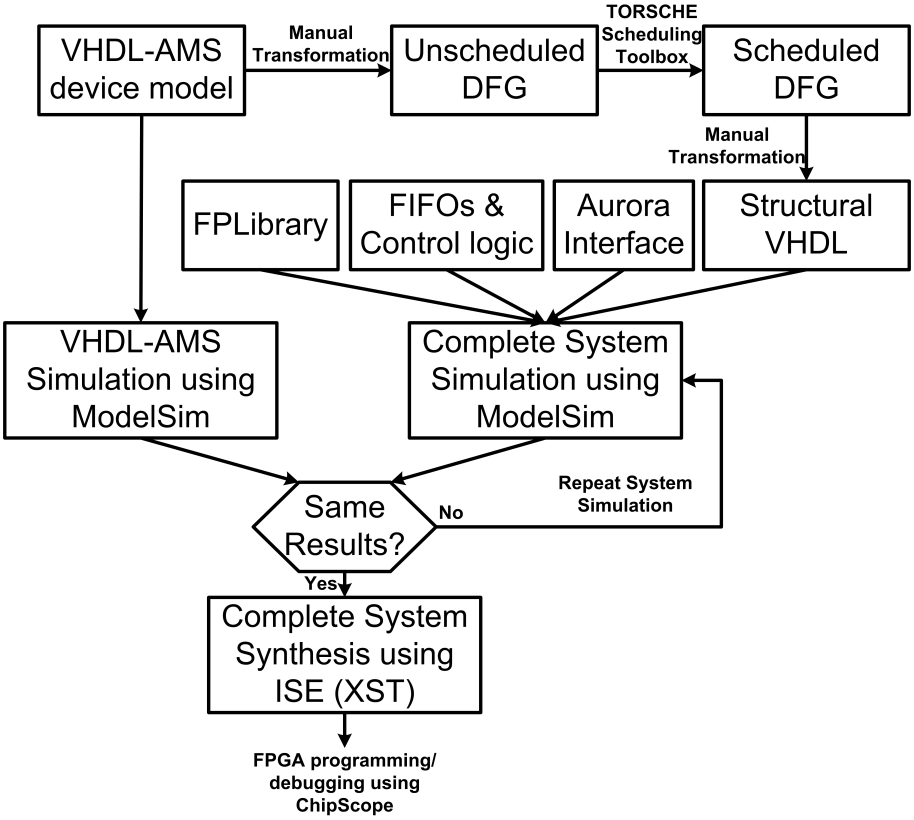


Figure 4.2: Transformation Flow of the VHDL-AMS high-level device model code to a Structural VHDL design

to the one for the VHDL-AMS high-level model. This structural VHDL code is then synthesised using the Xilinx ISE synthesis tool (XST). The synthesisable VHDL code -that implements the pipelined CMOS LEVEL Model in Figure 4.3- and its VHDL testbench are listed in Appendix A.2.

For the multi-FPGA accelerator to be discussed in Section 4.5, there are a number of more steps to simulate/synthesise the device model code. The complete system shown in Figure 4.9 and Section 4.5.2 is assembled by connecting the different system blocks together which are: the CMOS LEVEL 3 device model, the Aurora serial interface, the local FIFOs (First In First Out), the FPLibrary, and the control logic. The complete system is first simulated with ModelSim using the experimental input data described in Section 4.6.2. The input nodal voltages are sent from the FIFOs in the Host FPGA to the Slave FPGA through the Aurora interface. The CMOS device model accelerator

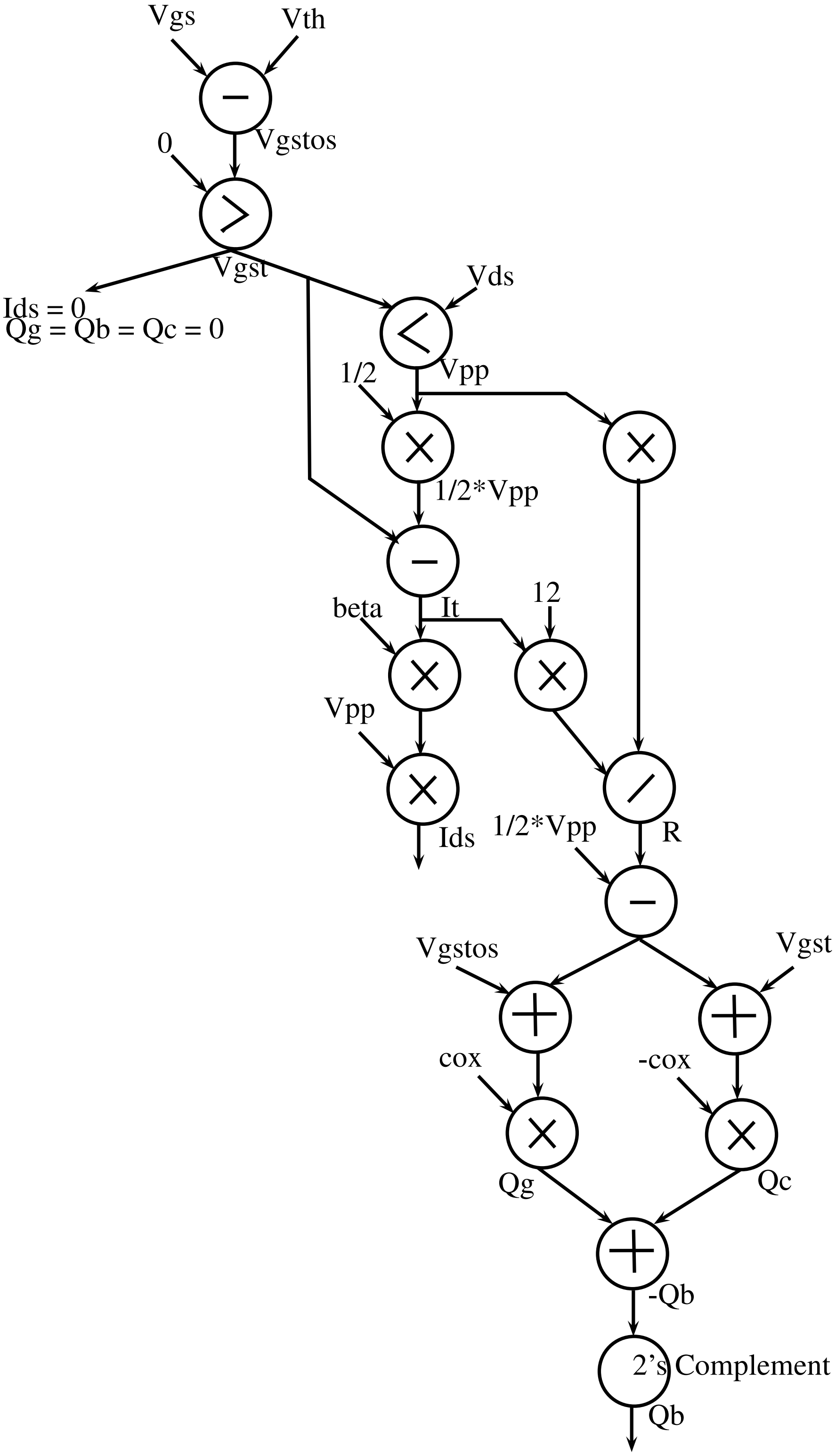


Figure 4.3: The Control-Data Flow Graph of the CMOS LEVEL 3 Model code shown in Figure 4.1

calculates the output currents/charges and returns them back to the Host FPGA through the same Aurora interface.

4.3.3 Software Implementation for Comparison

In order to measure the performance of the FPGA implementation described in Section 4.3.1, the CMOS LEVEL 3 was implemented in software running on a conventional processor. The software runtime is then compared to hardware runtime in order to quantify the acceleration offered by the FPGA system compared to a processor.

The software implementation used for comparison is written in C running on an Intel 2.0 GHz Due Core 2 processor with 2.5 GB of RAM. The software used is a direct translation of the same device model from high-level VHDL-AMS code in [172] to a C function to allow direct comparison. The software was parallelised using the OpenMP library to take advantage of the dual core processor. Device evaluations were divided on the two Intel cores using the ***#pragma omp parallel for*** directive as seen in Listing 4.1. The software execution time taken to evaluate the input device evaluations is denoted by $T_{software}$, which is measured according to the method described in Appendix E.2.

```
#pragma omp parallel for
for( index = 0; index < NumberOfDevices ; index++)
{
  deviceModel();
}
```

Listing 4.1: OpenMP parallelisation of device evaluation

4.3.4 FPGA Acceleration Calculation

In order to measure the FPGA acceleration of the FPGA implementation over the software counterpart detailed in Section 4.3.3, we examined two experiments:

- The first experiment measures the FPGA acceleration using a Single-FPGA Implementation as detailed in Section 4.4.
- In the second experiment, we measure the FPGA acceleration using a multi-FPGA implementation by using three identical FPGAs connected to a Host FPGA controller as detailed in Section 4.5.

In both experiments, the hardware timing information is obtained using the ChipScope debugging tool [174]. The ChipScope Integrated Logic Analyser (ILA) was used to count the number of clock cycles required to perform the device evaluation process as seen in Appendix E.1. The hardware execution time taken by the FPGA accelerator to perform the device evaluations is denoted by $T_{experiment}$. The hardware execution time figures are compared to the software execution times $T_{software}$.

The *FPGA Acceleration* of the device model accelerator is calculated as the ratio between the software execution times and the hardware execution times as follows:

$$FPGA\ Acceleration = \frac{T_{software}}{T_{experiment}} \tag{4.4}$$

This equation is used throughout the thesis to calculate the FPGA acceleration.

4.4 Single-FPGA Accelerator System

4.4.1 System Architecture

In order to evaluate the Single-FPGA acceleration of the CMOS LEVEL 3 model (Section 4.3), we considered two experimental cases:

- The first case utilises the embedded MicroBlaze as a controller of the accelerator core as seen in Section 4.4.1.1.

- The second case we used a dedicated controller to manage the accelerator core as seen in Section 4.4.1.2.

The purpose of conducting these two experiments is to measure the added overhead from using the MicroBlaze and the external memory. The experiments also measure the acceleration that can be achieved when the built-in BRAM blocks are used. The MicroBlaze experiment demonstrates how the accelerator core can be called from a C routine -running on the MicroBlaze- that mimics a SPICE simulator process.

4.4.1.1 With MicroBlaze

The CMOS model was implemented in VHDL and deeply pipelined (as seen in Section 4.3) and used in the FPGA system as demonstrated in Figure 4.4. The architecture consists of a single FPGA that contains a MicroBlaze processor [175] which runs a C routine that sends the data (i.e. nodal voltage values) to the hardware accelerator and reads the results back to memory. The CMOS model accelerator connects to the PLB (Processor Local Bus) system bus through two Read/Write FIFOs.

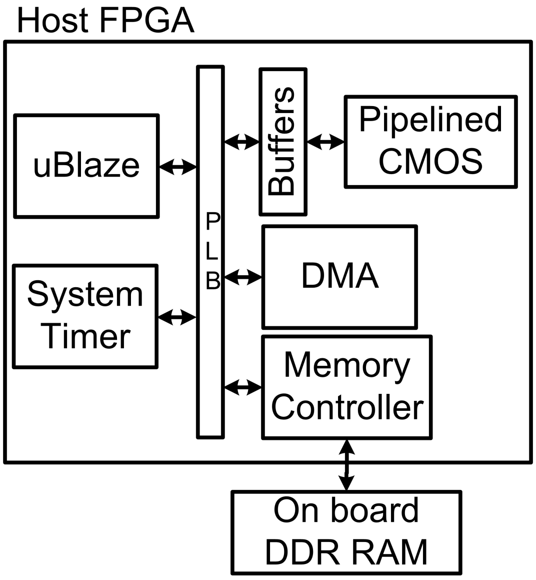


FIGURE 4.4: Block Diagram of the Single FPGA CMOS Accelerator with MicroBlaze

The MicroBlaze processor was used in order to assess the performance effect of calling the accelerator core from a C routine. The system is designed to use a Direct Memory Access (DMA) controller which transfers data from the external DRAM to the CMOS accelerator through Read/Write FIFOs. It was found through a number of experiments that using the DMA controller is the fastest way to transfer data to and from memory if compared to FSLs (Fast Simplex Link) or direct Read/Write FIFOs accesses.

4.4.1.2 Without MicroBlaze

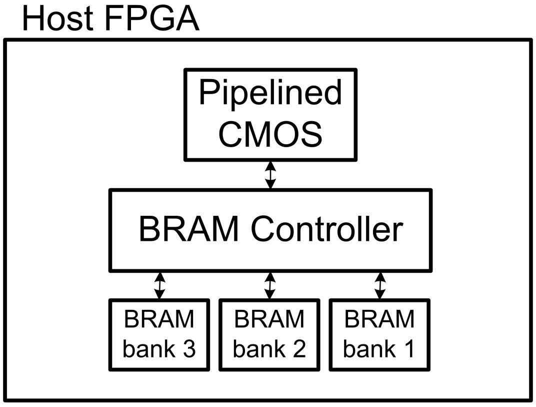


Figure 4.5: Block Diagram of the Single FPGA CMOS Accelerator without MicroBlaze

The system was also tested without the MicroBlaze as shown in Figure 4.5. A dedicated controller was implemented to read data from the local BRAM and supplies it to the device model accelerator. The controller is a State Machine which reads the data from the BRAM at every clock cycle and supplies it to the pipelined CMOS model. The controller also reads the results back from the accelerator core every clock cycle and stores them in the local BRAM. The input data is pre-loaded into the BRAM blocks when programming the FPGA.

4.4.2 Experimental Results

4.4.2.1 Software Implementation Runtime

In order to calculate the speedup offered by the FPGA implementation, we need first to quantify the runtime of the software version described in Section 4.3.3. Firstly, the OpenMP-parallelised software implementation is compared to the sequential version in order to assess the experimental setup. The dual core software implementation was found to be up to about 1.53 times faster than the single core version. This is a reasonable speedup taking into account Amdahl's Law and the underlying OS/applications limitations.

The software implementation was executed on the Intel processor system for a number of test cases where each test case corresponds to specific number of device evaluations N as shown in Table 4.2. The number of device model evaluations increases from one test case to the next to assess the variation of the hardware acceleration results for large number of model evaluations N. The latter does not correspond to any specific circuits or represent any particular pattern, they are used for evaluation purposes only as our main focus is to quantify the system speedup. The software execution runtime versus the number of device evaluations is shown in Figure 4.6. The graph indicates that the software times changes almost linearly with the number of device evaluations performed. The X-axis represents the number of device evaluations (N) for each test case as detailed in Table 4.2.

The software execution times shown in Figure 4.6 are used throughout the thesis to calculate the FPGA Acceleration using Equation 4.4 for both the Single- and Multi-FPGA Accelerators. These results are used in the next section (Section 4.4.2.2) to calculate the speedup for the Single-FPGA accelerator with- and without MicroBlaze.

Table 4.2: The number of device model evaluations N per test case

Test Case	N
1	3.0E+02
2	3.0E+03
3	1.5E+04
4	3.0E+04
5	6.0E+04
6	1.5E+05
7	3.0E+05
8	6.0E+05
9	9.0E+05
10	1.2E+06
11	1.5E+06
12	1.8E+06
13	2.1E+06

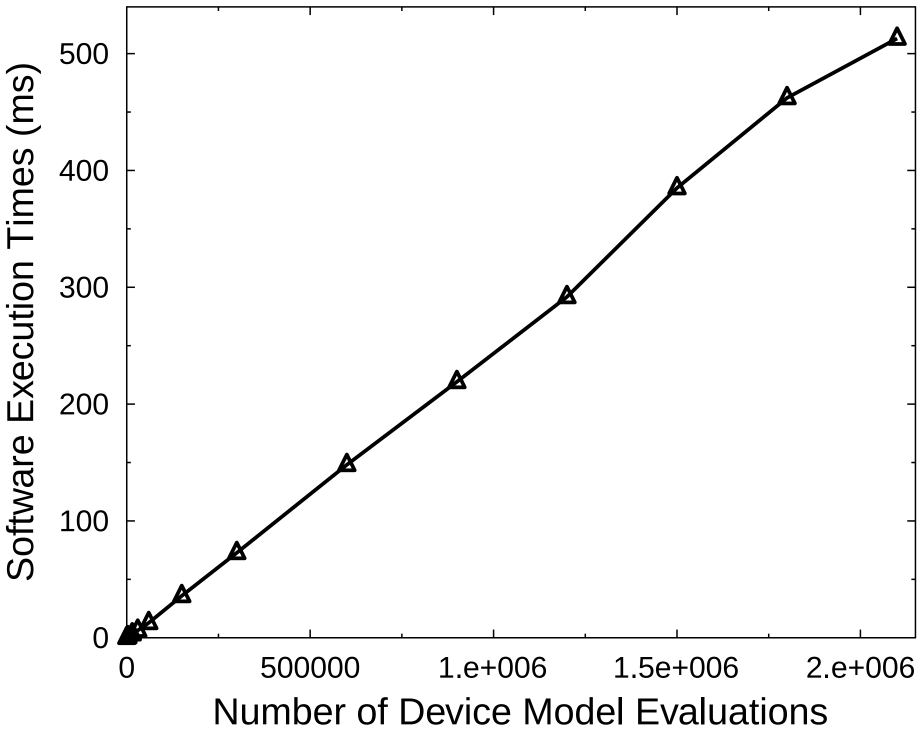


Figure 4.6: Change of the software execution times with the number of device evaluations in Table 4.2

4.4.2.2 Single-FPGA Accelerator Runtime

Both cases outlined in Section 4.4 are implemented on the Xilinx Virtex-II Pro board. For the case where the MicroBlaze is used, the system bus is clocked at 100 MHz. The hardware execution timing information is obtained using the on-chip timer. The input data is a set of nodal voltage values stored in the external memory. These values are sent to the accelerator in packets of three nodal voltages for each device evaluation. The data used in the evaluation was based on random voltage values, as the experiments are mainly concerned with quantifying the achievable hardware acceleration.

The FPGA acceleration versus the number of device evaluations is shown in Figure 4.7. The graph shows the FPGA acceleration results, which is the ratio between the software execution times and the hardware execution times. The graph demonstrates two streams of results for the two test configurations: with- and without- Microblaze and external DRAM.

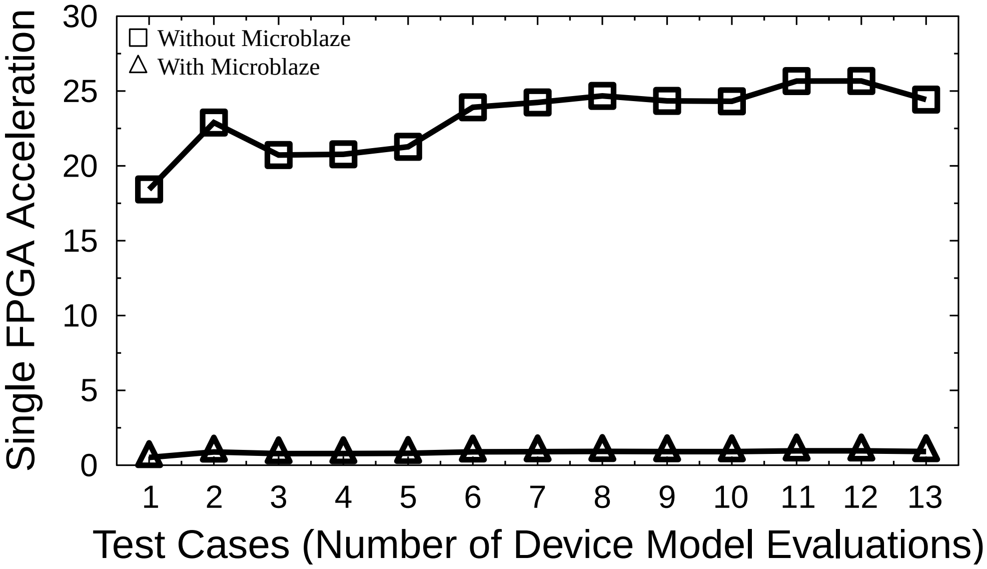


Figure 4.7: Single FPGA Acceleration with the number of device model evaluations in Table 4.2

Figure 4.7 shows no speedup over the software version (Section 4.4.2.1) for the case where the MicroBlaze is used as a controller. This result takes into account the time to send and receive data to and from the accelerator and the external memory through the system bus. This indicates that the external memory access is the bottleneck of this design as nearly most of the time was spent sending/receiving data from memory to the accelerator core.

Figure 4.7 shows that using a single instance to accelerate the CMOS model demonstrated a speedup of up to 25 times over the software execution results. This test assumed that all inputs are stored in local BRAM buffers and are provided at each clock cycle after a fixed start-up delay. This test does not include the overheads introduced by the system bus and the external memory.

The hardware execution runtime versus the number of device evaluations is shown in Figure 4.8. The graph demonstrates two streams of results for the two cases: with- and without- Microblaze and external DRAM. The graph indicates that the hardware execution times changes linearly with the number of device evaluations performed.

4.4.3 Discussion

The device model used in our system is based on the CMOS LEVEL 3 model from the Southampton VHDL-AMS Validation Suite in [171], in which the main model parameters are fixed (Section 4.3). A single FPGA pipelined implementation of the model was analysed without the use of the Microblaze and the External Memory. The results showed that using a single instance to accelerate the CMOS model demonstrated a speedup of up to 25 times. However, this result is reduced dramatically because of the slow memory interface of the XUPV2 board.

The pipelined design has a constant throughput as results are provided every a fixed number of clock cycles. Hence, this explain the nearly constant acceleration of the single-FPGA system over the software implementation. This is demonstrated by the graph in Figure 4.8. The graphs in Figure 4.7 show nearly the same pattern in terms of hardware

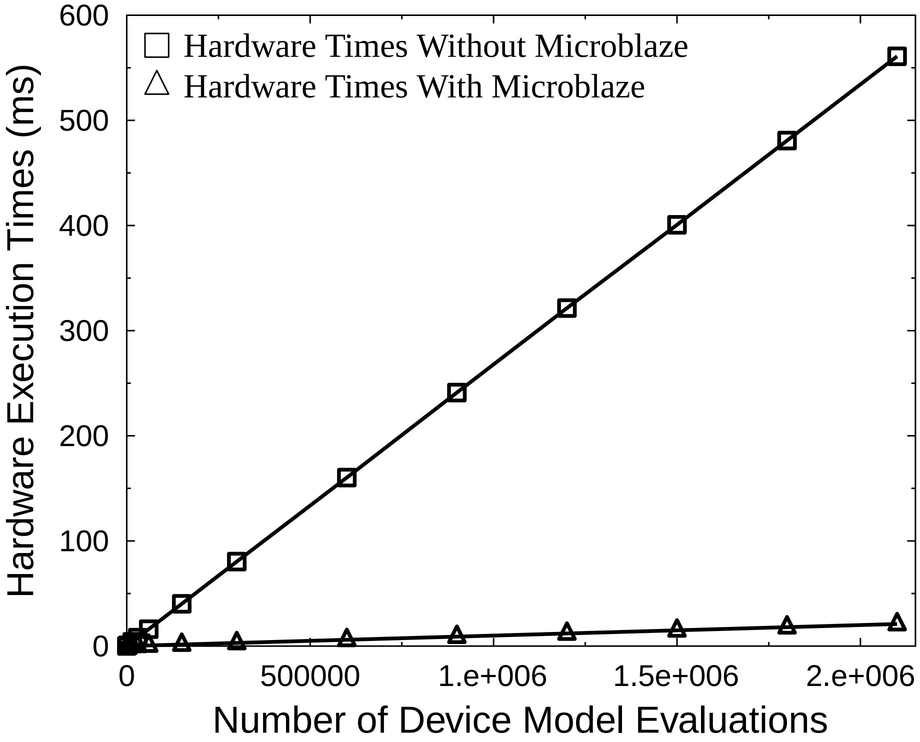


Figure 4.8: Change of the hardware execution times with the number of device evaluations in Table 4.2

acceleration change with the number of device evaluations. This is mainly because both the software and hardware times are changing almost linearly with the number of device evaluations as shown in Figure 4.6 and Figure 4.8.

The acceleration figure for the first test case is lower than the rest of the cases. This can be due to the good performance of the software implementation as the number of the device counts are small and hence data can be cached within the processor. Acceleration then increases to reach a steady state in the rest of the cases. This is due to the constant throughput of the hardware pipeline.

The MicroBlaze bus and the external memory access present the bottleneck in the Single-FPGA accelerator design. The memory interface of the XUPV2 board is slow, which limits the maximum acceleration that can be achieved. The memory bottleneck can be eased by using a better BRAM scheduler to buffer data from memory, or use an FPGA system with a higher memory bandwidth. The multi-FPGA parallel architecture to be discussed in Section 4.5 is expected to offer better acceleration as demonstrated in the next section (Section 4.5).

4.5 Multi-FPGA Accelerator System

4.5.1 System Architecture

The previous section evaluated a single-FPGA accelerator of the CMOS LEVEL 3 model. This section extends the system to perform the device model evaluations on FPGAs in a SIMD execution model. In order to have a SIMD-like architecture, all FPGAs must have the same identical code which is executed on a different simulation data. The different instances of the model should not have any data or control dependency amongst them. FPGA internal memory banks can be accessed in parallel, which means that multiple model instances can execute on the same device if hardware area permits. Hence, the SIMD execution model is an attractive approach to perform a large number of device model evaluations in parallel using FPGAs as explained in Section 3.4.

The aim of this section is to investigate a parallel architecture composed of a number of FPGAs used to perform the device model evaluation phase simultaneously. The host (master) FPGA outsources the computations to three FPGAs to perform the device evaluations in parallel. The data to and from the accelerators are buffered in local FIFOs in the Host FPGA and in each slave FPGA.

Each slave FPGA is executing an instance of the same CMOS model using the incoming data from the host as model inputs. Given a pipelined design of the CMOS model, the slave FPGAs will start to produce results at a constant rate (constant throughput). The latter assumption is only satisfied when the pipeline is full. The calculated results are sent back to the host FPGAs and used to form the nodal matrix as seen in Equation 3.5.

The prototype acceleration system computes the device evaluation of the CMOS LEVEL 3 model in the vector $F(V^i)$ of Equation 3.4 in parallel on a number of FPGAs. The system will be based on a SIMD architecture in which the same model resides on the slave FPGAs to process different data sets as seen in Figure 4.9.

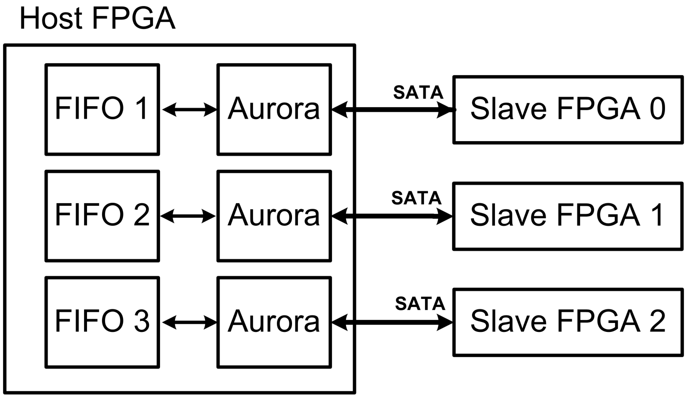


Figure 4.9: The Architecture of the multi-FPGA CMOS Accelerator

For our experiments, it is assumed that the number of transistors in a circuit is approximately the same as the number of nodes N. Hence, the hardware acceleration would perform N device evaluations per Newton Raphson iteration per time step in a transient analysis. This number of device evaluations is divided between the number of available execution threads. Therefore, each slave FPGA of the multi-FPGA system would perform approximately N_{FPGA} device evaluations during each iteration, which is given by:

$$N_{FPGA} = \frac{N}{Number\ of\ FPGAs} \tag{4.5}$$

It can be seen that the more number of FPGAs used as computation nodes, the faster the overall device evaluation step is performed, but with extra costs. However, a trade-off between the number of FPGAs used and the cost of the system must be found in order to achieve optimum acceleration results. Acceleration is also limited by the sequential sections and the synchronisation barriers in the simulation process as set by Amdahl's Law [29] and explained in Sections 1.2.1 and 3.4.

4.5.2 Accelerator Prototype

The Multi-FPGA Accelerator prototype is based on three off-the-shelf XUP V2-Pro Xilinx boards connected together using serial links as shown in Figure 4.9. The serial links are controlled by the Aurora serial interface from Xilinx (see Section 2.3.5). The host FPGA outsources the computations to three FPGAs to perform the device evaluations in a parallel SIMD fashion. The data to and from the accelerator cores are buffered in local FIFOs in the Host FPGA and in each slave FPGA. The architecture is limited to three slaves due to the limited number of available on-board serial links, as each board has only three usable serial connections.

The nodal voltages data is stored at the host FPGA and pushed into the Send FIFOs. The data is then sent to the computing slave FPGAs in parallel. The resulting currents are received back from the slaves and saved into the Receive FIFOs. The FIFOs are implemented using the on-chip BRAM. Each block of BRAM provides the data to one FPGA at an aggregate throughput of one single-precision word per clock cycle. All three blocks of BRAM used in the design are accessed independently from each other, hence, provide the maximum data throughput to the FPGAs.

The use of BRAM in the design allows the system to be modular, as other memory system hierarchy can be built on top of the BRAM. For example, a memory controller can be used to map an external memory module to the local memory blocks. Also, the BRAM blocks are addressable simultaneously, this allows the data to be easily sent to the computing FPGAs in parallel. This allows the acceleration results to be obtained regardless of the limited memory bandwidth of the board used.

4.5.3 Theoretical Hardware Execution Time

Estimating the total execution time of the hardware accelerator (in Section 4.5.1) depends on a number of factors which include the following: the number of data transfers, the latency of a single transfer, time to evaluate the device models, and the speed of the inter-FPGA serial links. The Aurora communication interface uses the streaming mode clocked at 75 MHz to send a packet of three 32-bits words containing the three voltage values to each slave FPGA. The overall hardware execution time of the system in Figure 4.9 takes into account the following times:

- The time taken for the system to initialise and the interface Aurora be ready to transmit data, this is denoted by T_{init}.
- The initial transfer time of the Aurora core to send the first packet of data from the Host to the Slave FPGA of data. This is denoted by T_a , which is equal to 38 clock cycles according to Aurora Datasheet in [176]. The Host FPGA should wait for: $2T_a$ cylces, which includes the time to send the first data inputs to the Slave FPGA and the time to receive the first output data back from the Slave FPGA to the Host.
- The time taken to process the first set of inputs nodal voltages by the device model accelerator, this is denoted by T_s as seen in Figure 4.10.

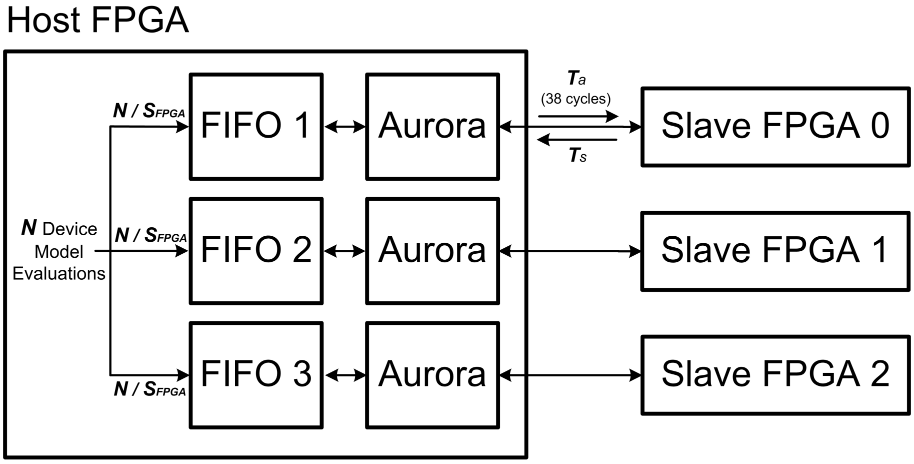


Figure 4.10: Hardware execution time estimates annotated on the architecture of the multi-FPGA Accelerator in Figure 4.9

- The main hardware execution time is spent processing an N number of device model evaluations. Each device evaluation requires three input voltage values (V_{ds}, V_{gs}, and V_{bs}) and produces three output values which are the nodal currents/charges (I_{ds}, Q_b, and Q_c). Each Slave FPGA processes a number of device evaluations which is given by: $\frac{N}{S_{FPGA}}$, where: S_{FPGA} is the number of Slave FPGAs used for computations (as also seen in Equation 4.5 and illustrated in Figure 4.10). In the case of our system we have three slave FPGAs which means that $S_{FPGA} = 3$.

After the hardware pipeline is filled with the input nodal voltages data, the accelerator starts producing results at a constant number of clock cycles. The latter is equal to the time to transfer the inputs and receive the outputs. Each input voltage value is a single precision floating-point number (32-bit) which requires two clock cycles to be transferred across the serial link as the Aurora standard uses a 16-bit transfer interface packet [125].

Aurora can also be configured to use more serial links to send higher width data transfers than the default 16-bit. This can be done by adding more serial connections between the Host FPGA and the Slave FPGA. The number of serial links between the Host and the Slave FPGA is denoted by: S_{links}. Therefore the time taken to send the three voltage values would be: $3\left(\frac{2}{S_{links}}\right)$. In the case of our system we have one single serial connection between each Slave FPGA and Host FPGA, which mean that $S_{links} = 1$. The Aurora protocol allows data to be sent and received using the same serial link simultaneously. Hence, the time taken to receive the output results is not considered. Aurora clock compensation delay which is a result of the clock synchronisation is not taken into account as it is negligible [176].

The total theoretical execution time of the hardware accelerator T_{theory}, in clock cycles, is given by the sum of all the time portions detailed above as follows:

$$T_{theory} \approx T_{init} + 2T_a + \left(\frac{N}{S_{FPGA}}\right) \times 3\left(\frac{2}{S_{links}}\right) + T_s \tag{4.6}$$

In the case of our system in Figure 4.10, we have $S_{links} = 1$ and $S_{FPGA} = 3$, hence, Equation 4.6 is reduced to:

$$T_{theory} \approx T_{init} + 2T_a + (2N) + T_s \tag{4.7}$$

4.6 Experimental Work

This section outlines the experimental setup used to implement the multi-FPGA accelerator in Section 4.5.

4.6.1 Experimental Setup

The system in Figure 4.9 was implemented using one host FPGA and three slave FPGAs. The hardware timing information is obtained using the ChipScope debugging tool [174] as seen in Appendix E. The hardware timing figures $T_{experiment}$ obtained using ChipScope are compared to their software $T_{software}$ counterparts as seen in Equation 4.4. The number of device evaluations performed by the hardware accelerator was increased after each run as seen in Table 4.2. This is to demonstrate how the acceleration changes in accordance with the number of device evaluations performed.

Figure 4.11 shows the rack of FPGAs connected together using Serial Advanced Technology Attachment (SATA) links (top four FPGAs only, see Appendix C). The top FPGA is used as the host controller, and the subsequent three FPGAs are used as computing slaves as seen in the Figure 4.9. The rest of the FPGAs are not connected, and hence not used in our system due to the limited serial connectivity of the boards (Appendix D).

4.6.2 SPICE Simulation Data

In order to test the FPGA accelerator described in Section 4.5.2, a suitable benchmark input data has to be provided. The input data to the hardware accelerator consists of the nodal voltage values for each device evaluation. The LEVEL 3 model takes the three nodal voltages: V_{ds}, V_{gs}, and V_{bs} to calculate the next iteration's voltage. The data returned from the FPGA includes the nodal currents and charges I_{ds}, Q_b, and Q_c.

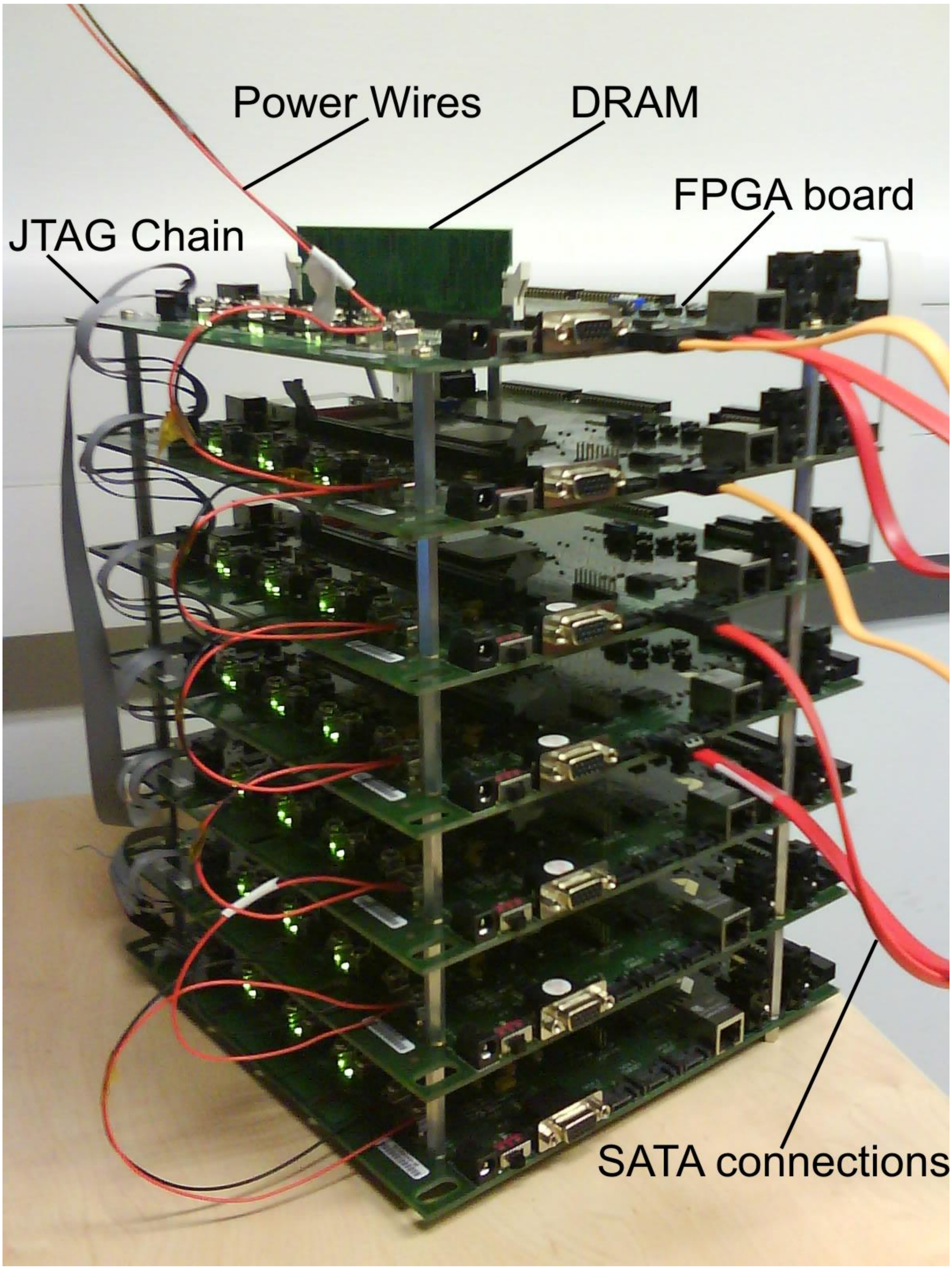


Figure 4.11: The Prototype Multi-FPGA System Designed to Accelerate the Device Model Evaluation Phase

Because the experiments in this section are mainly concerned with the maximum achievable hardware acceleration, the data used in the evaluation process is based on repeated set of 100 sample voltage values for V_{ds} changing from 0 to $5.0V$; and taking $V_{gs} = V_{bs} = 2.0V$ as constants. This test data is repeatedly used as input data to the device model in order to facilitate the comparison between the software and the experimental hardware outputs. The input voltage values (V_{ds},V_{gs}, and V_{bs}) are selected to be the same values used in [172]. This is to allow us to easily compare our drain current outputs (I_{ds}) to the VHDL-AMS results quoted in [172]. This comparison will be illustrated in the next Chapter in Section 5.1.1.

4.7 Summary

This Chapter described the experimental validation of the FPGA accelerator proposed in Section 3.4. The device model used in the multi-FPGA system is based on the CMOS LEVEL 3 model from the Southampton VHDL-AMS Validation Suite in [171], in which the main model parameters are fixed (Section 4.3). The resulted device model code after the parameters pre-calculation needs to be mapped on FPGAs. However, there is no tool that can be used readily off-the-shelf due to the custom nature of this task. Hence, a manual device model code transformation have to be considered in Section 4.3.2.

In order to implement the device model evaluation on FPGAs, two cases were considered. Firstly, A single FPGA pipelined implementation of the model was analysed without the use of the Microblaze and the External Memory. The results showed that using a single instance to accelerate the CMOS model showed speedup of up to 25 times. However, this result is reduced dramatically because of the slow memory interface of the evaluation board used. This memory bottleneck can be eased by using a BRAM scheduler to buffer data from memory, or using a higher memory bandwidth.

Secondly, a parallel architecture is designed so that each FPGA can execute one instance of the device model which would exploit the inherent parallelism in the SPICE simulator. The FPGA spatial implementation is expected to deliver high acceleration as it exploits the instructions and the pipeline parallelism approaches as explained in Section 3.5. This is due to the highly customisable FPGA architecture which allows the realisation of custom pipelined model computations. This is one of the factors which allows FPGAs to deliver respectable speedup figures over conventional processors. The main contribution of this Chapter lies in the proposed architecture to exploit the inherent parallelism in the device mode evaluation using multi-FPGA systems.

Chapter 5

Multi-FPGA Acceleration Results

This Chapter presents the experimental results of the multi-FPGA accelerator described in the previous Chapter in Section 4.5. The experimental results include the FPGA acceleration, the resources usage, and the power consumption estimates. Section 5.2 extends the experimental and theoretical results from this Chapter and the previous Chapter to include other newer device models and more advanced reconfigurable systems.

5.1 Experimental Results

5.1.1 Acceleration Results

Before calculating the FPGA acceleration achieved by our multi-FPGA accelerator, the drain current output of the multi-FPGA system is first compared to results of simulating the VHDL-AMS device model in [172]. Both experiments used the input nodal voltages data as detailed in Section 4.6.2. Figure 5.1 shows the drain current I_d versus the drain-to-source voltage V_{ds} using the simulation VHDL-AMS model shown in Listing A.1 [172] and our synthesisable VHDL code in Listing A.2 (Appendix A). The figure shows that the current values I_d calculated using our synthesisable pipelined VHDL design is aligned with the VHDL-AMS simulation results in [172]. The ModelSim simulation and ChipScope waveforms of the multi-FPGA system are shown in Appendix A.3 and A.4.

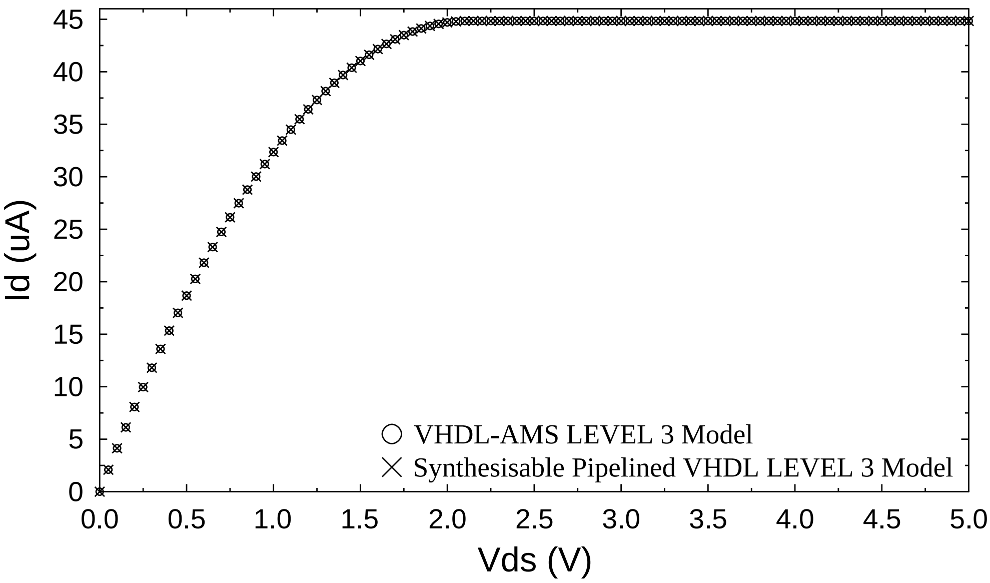


Figure 5.1: The drain current I_d vs. the drain-source voltage V_{ds}

The FPGA Acceleration of our multi-FPGA system is calculated according to Equation 4.4 (Section 4.3.4), where the hardware execution time is compared to the software execution time shown in Section 4.4.2.1. The FPGA acceleration is shown in Figure 5.2. The graph demonstrates three streams of results for three test configurations versus the number of device evaluations (test cases in Table 4.2). The three graphs correspond to the hardware acceleration using three, two, and one FPGA(s) as computation nodes (S_{FPGA}) as seen in Figure 5.3. Table 5.1 shows the average acceleration of the different set-ups.

The X-axis represents the number of device evaluations (N) for each test case as detailed in Table 4.2. The table shows that the number of device evaluations increases from one test case to the next, this is to assess the variation of the hardware acceleration results for large number of device model evaluations.

From a pipelined hardware point of view, results are produced at constant throughput regardless of the number of operations to be performed. Hence, the hardware times

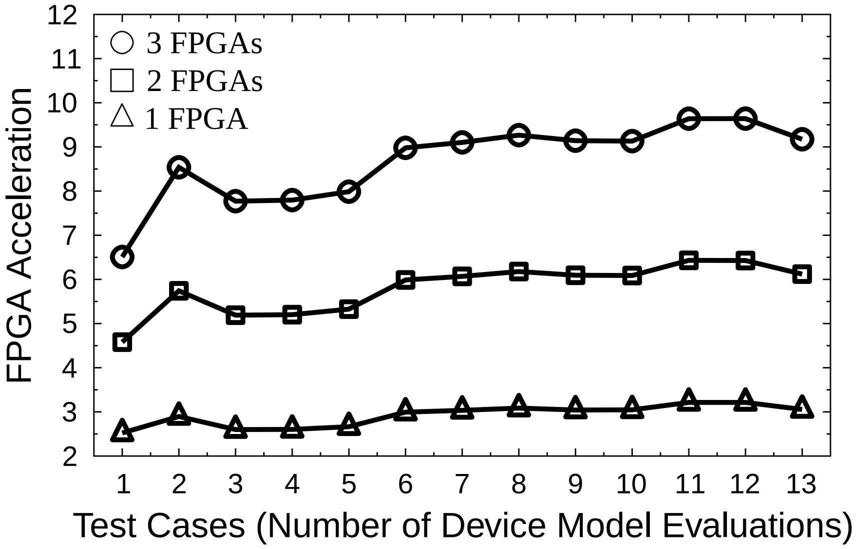


FIGURE 5.2: FPGA Acceleration for the Three Test Configurations in Figure 5.3

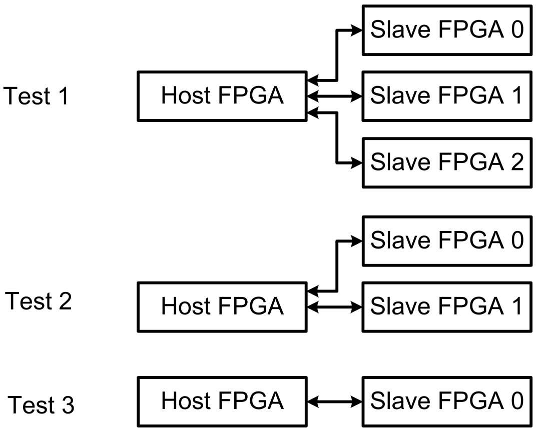


FIGURE 5.3: The Three Test Configurations of the Multi-FPGA System

change linearly with respect to the number of device evaluations. The graphs show nearly the same pattern in terms of hardware acceleration change with the number of device evaluations. This is mainly due to the fact that the software times are influenced by a number of factors like the operating system scheduling and other concurrent processes. This is confirmed by the graph shown in the Figure 4.6. The graph shows that the software times changes almost linearly with the number of device evaluations performed.

Table 5.1: Average acceleration using the multi-FPGA system

	3 FPGAs	2 FPGAs	1 FPGA
Average Acceleration	8.67	5.80	2.92

Usually, the number of transistors in typical circuit simulations does not reach the maximum number shown in Table 4.2, however, it was used to assess the effectiveness of the system in evaluating large transistor counts. Generally, SPICE simulation of circuits larger than 20,000 devices is not feasible [42].

The acceleration figure for the first test case is lower than the rest of the cases. This can be due to the good performance of the software implementation as the number of the device model evaluations are small and hence data can be cached within the processor. FPGA Acceleration then increases to reach a steady value in the rest of the cases. This is due to the constant throughput of the hardware pipeline. In addition, the main reason for this fairly constant speedup is that the system accelerates the device evaluation phase as an isolated task. In other words, other limiting (i.e. non-parallelised) tasks in the SPICE simulator like matrix solve, error truncation and transient loop are not considered.

The acceleration results are limited by the speed and the number of the available serial communication links in the hardware platform used. The parallel architecture is expected to offer larger throughput if larger FPGAs and faster serial links are used.

Figure 5.4 shows the FPGA acceleration results plotted against the number of FPGAs used. This shows a linear speedup increase when adding more FPGAs for almost all the test cases. It can be seen that the graphs do not show any noticeable saturation which would indicate when performance degrades as more FPGAs are added. Based on these results, higher performance (speedup) is expected if more FPGAs are added to the system. However, due to the limitation of the current system serial I/O resources, further study should be conducted to estimate when the speedup curve saturates.

In order to assess the accuracy of the theoretical execution time estimation T_{theory} in Equation 4.6, the experimental hardware times $T_{experiment}$ are compared to the theoretical estimates. The percentage difference between T_{theory} and $T_{experiment}$ is shown in

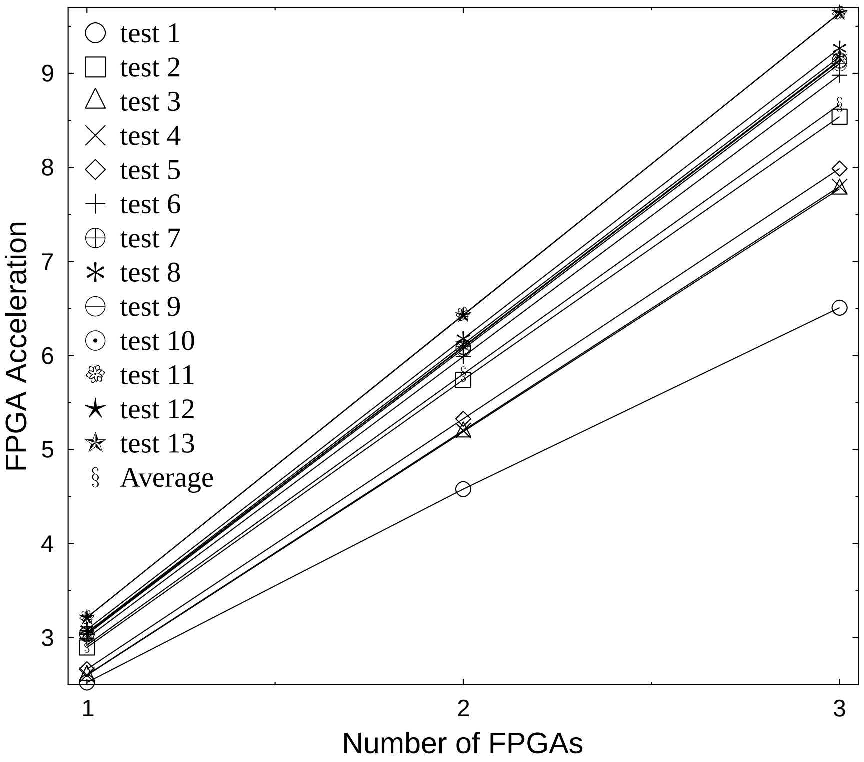


Figure 5.4: FPGA Acceleration change with the Number of FPGAs

Figure 5.5 for the three test configurations. The maximum percentage difference is approximately 5%. The graph shows that the difference between the theoretical estimation and the practical hardware times decreases with the number of device model evaluation increase for the three test configurations. This is due to the improved hardware throughput of the pipelined model accelerator for large number of model evaluations. Hence, the difference becomes more negligible when the number of device evaluations grows larger.

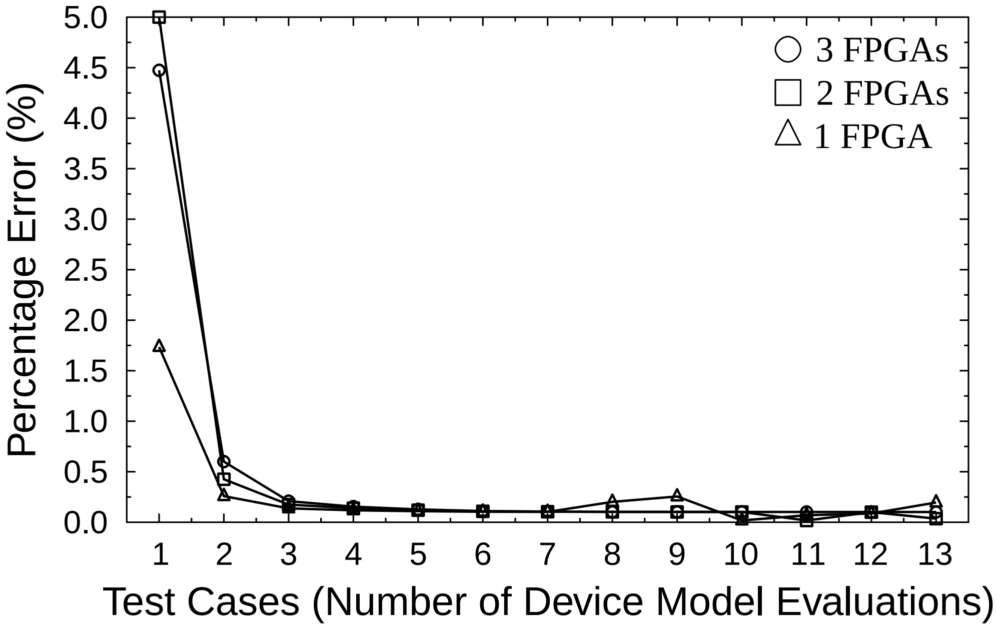


Figure 5.5: Percentage difference between the theoretical estimation T_{theory} and the practical hardware times $T_{experiment}$

5.1.2 Resources Utilisation

5.1.2.1 Slave FPGA

As explained in Section 4.3, the implementation of the CMOS LEVEL 3 model on the multi-FPGA system was performed in two steps:

1. **Without Parameter Pre-calculation**

   In the first step, the generic design of the CMOS LEVEL 3 model was implemented in VHDL using single-precision FPLibrary floating-point operations. The model was implemented without including derivatives and all the transistor parameters are not pre-calculated.

   The area utilisation ratio of the full CMOS model (32-bit operations) was estimated by the synthesis tool to be about 438% of the Virtex-II Pro FPGA. This implemenetation is a pipelined version which includes all the model parameters as inputs. Table 5.2 shows the resource utilisation of the model, based on single-precision floating-point operators. The maximum clcok frequency of this design was estimated to be around 60 MHz.

The design overutilises the FPGA resources and could not be mapped. The design requires more than four times the resources of one FPGA. The double-precision design could not be synthesised using the ISE synthesis tool due to the large design size. However, the resulting area usage would be significantly higher than that of the single-precision version found earlier.

TABLE 5.2: Resources Utilisation of the Full CMOS model Without parameter pre-calculation on the Virtex-II Pro FPGA

Logic Resources	Available	Used	Usage (%)
Slice Flip Flops	27,392	55,450	202
Slice LUTs	27,392	912,995	333
MULT18X18s	136	136	100
Block RAMs	136	4	2

The resources usage in Table 5.2 shows that the full model could be partitioned into a number of sub-designs that can be mapped onto FPGAs. These FPGAs can be cascaded together in a pipelined fashion as shown in Figure 5.6 (Section 3.4.2). This has not been evaluated experimentally, however, the number of inter-FPGA signals that have to be communicated across the serial links would be considerable. Hence, the bottleneck in this case becomes the serial communication link. The communication bandwidth between the FPGAs cannot cope with the large amount of intermediate signals (e.g. voltages, currents, and other parameters) that need to be communicated. Hence, the problem of multi-FPGA inter-device signals and the limited bandwidth of the serial communication, emphasise the need to optimise the number of signals between the different FPGAs partitions resulted from the partitioning phase in the multi-FPGA synthesis system. This problem is further investigated in the next Chapter in Section 6.3 where a technique is introduced to optimise the inter-FPGA interconnections. This technique uses the CMOS LEVEL 3 model as a benchmark to measure its effectiveness.

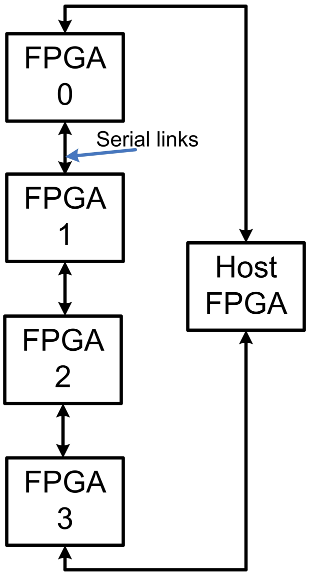


Figure 5.6: FPGAs Connected in a Pipelined Ring

2. **With Parameter Pre-calculation**

For the second step, the generic design was synthesised by setting the parameters to match the default values demonstrated in Table 4.1. The synthesis estimates of the area usage is nearly 57% of the FPGA resources. The maximum estimated frequency of the design is approximately 121 MHz.

Table 5.3 shows the resource utilisation of the model, based on single/double-precision floating-point operators, respectively. The results show that the design uses nearly the third of the LUTs slices in the FPGA and consumed half of the built-in multipliers for single-precision implementation. The double-precision design used 155% of the FPGA with a maximum frequency of about 97 MHz, and hence, cannot be fitted.

Table 5.3: Resources Utilisation of the CMOS model With parameter pre-calculation on the Virtex-II Pro FPGA

Logic Resources	Available	Single-Precision		Double-Precision	
		Used	Usage (%)	Used	Usage (%)
Slice Flip Flops	27,392	6,357	23	17,663	64
Slice LUTs	27,392	8,349	30	25,052	91
MULT18X18s	136	64	47	135	99

Table 5.4 gives the whole slave FPGA resources estimations, which includes the CMOS model plus the control and communication logic.

Table 5.4: Resources Utilisation of the Slave FPGA Design

Logic Resources	Available	Used	Usage (%)
Slice Flip Flops	27,392	9,956	36
Slice LUTs	27,392	8,701	31
MULT18X18s	136	64	47
Block RAMs	136	23	16

5.1.2.2 Host FPGA

Table 5.5 shows the resource utilisation of the host FPGA design, which consumed 15% of the host FPGA area and the maximum estimated frequency was about 134 MHz. This design consumes mainly the Block RAM (BRAM) resources (62%) as they are used for the data FIFOs. The nodal voltages data is being stored at the host FPGA and pushed into the Send FIFOs as shown in Section 4.6.2. The data is then sent to the computing slave FPGAs in parallel. The resulting currents are received back from the slaves and saved into the Receive FIFOs.

The FIFOs are implemented using the on-chip BRAM. By using a pipelined state machine, the BRAM can provide data every clock cycle, which produces the best performance using the current system. Each block of BRAM is accessed independently, which provide the maximum data parallelism.

Table 5.5: Resources Utilisation of the Host FPGA Design

Logic Resources	Available	Used	Usage (%)
Slice Flip Flops	27,392	2,263	8
Slice LUTs	27,392	3,271	11
Block RAMs	136	85	62

5.1.3 Power Consumption Estimation

Table 5.6 demonstrates the average power consumption of the different configurations of the FPGA system (3, 2, and 1 slave FPGAs) connected to one host FPGA as seen in

Figure 5.3. The power usage is approximated by experimentally measuring the current flowing into the multi-FPGA system from a 5 volts power supply. The electrical current drawn from the source by the multi-FPGA Rack (shown in Figure 4.11) is used as the power consumption estimates. The measurements are taken while the device model evaluation process is running. The measured figures are just used to approximate the power ratio between the processor and the FPGA. Hence, these estimates are for evaluation purposes and not for specifically comparing the power usage.

The Power Ratio shown in Table 5.6 represents the ratio of the power consumed by the Intel processor to the multi-FPGA system power. The maximum power consumption according to the processor specifications is approximately 65 Watts [177]. The maximum processor power consumption figure was used in our comparison as it is not feasible to estimate the power consumed by the software alone. The results shows a power consumption reduction ratio of approximately six, which means that the multi-FPGA system consumes one sixth the power consumed by the Intel Processor. The table also shows the energy estimates for both the FPGA accelerator and the Intel processor system. The last row in the table shows the Energy Ratio between the two energy figures.

Table 5.6: Average power consumption of the multi-FPGA system with different configurations

	3 FPGAs	2 FPGAs	1 FPGA
Current (Amps)	2.21	1.73	1.25
Power (Watt)	11.05	8.65	6.25
Power Ratio (CPU/FPGA)	5.88	7.51	10.40
FPGA Energy (mJ)	617.90	725.06	1049.43
CPU Energy (mJ)	33343.10	33343.10	33343.10
Energy Ratio (CPU/FPGA)	53.96	45.99	31.77

5.1.4 Results Comparison

Although the FPGA boards used in our prototype system are limited in terms of their hardware capabilities, it is worth noting that it achieved about 10 times speedup over an Intel 2.0 GHz Due core 2 processor with 2.5 GB of RAM. The system also consumes less power that the processors as demonstrated in Section 5.1.1. Our system gave a quantitative analysis of the amount of acceleration that can be achieved using current available FPGA boards given the fast technology growth curve of FPGA fabric [178]. Acceleration results showed little or no sign of saturation with increasing number of FPGAs. Other studies [144, 28, 154] also showed very little saturation of device evaluation acceleration for small number of processors, which confirms our findings.

From a design point-of-view, our design relied on a spatial deeply pipelined implementation of the model whereas the design in [43] works by compiling a high-level Verilog-AMS description to a statically-scheduled custom VLIW architecture. The result clearly depends on the number of computing FPGAs, hence any comparison must take into consideration the number of parallel chips/threads, like-with-like. The work presented in [43] is the main recent study that focused on accelerating device mode evaluations on FPGAs. Authors have used a custom VLIW processor running the device models on a single Virtex 5 FPGA. This demonstrated an acceleration of 2-18 times over a dual-core 3GHz Intel Xeon 5160 for a number of device models. The study reported a speedup figure of approximately 7 times for the MOS3 model which is the same model used in our system.

In order to compare our results to the ones in [43], the target FPGA used in our system is changed from the Virtex-II Pro to the Virtex 5 V5LX330T FPGA used in [43]. The acceleration results and resources utilisation of the new system are then estimated and therefore used for comparison. The comparison is performed in two steps. Firstly, the resources utilisation of our FPGA implementation on the Virtex 5 V5LX330T FPGA are outlined in Section 5.1.4.1. No resource usage comparison is possible because the study [43] did not quote any resources utilisation estimates. The second step involves comparing our acceleration results to the ones reported in [43] as seen in Section 5.1.4.2.

5.1.4.1 Resources Usage Comparison

This section presents the resources utilisation of the CMOS LEVEL 3 model on the V5LX330T used in [43]. As explained in Section 4.3, the experimental validation of the

CMOS model implementation on the multi-FPGA system was performed in two steps:

1. **Without Parameter Pre-calculation**

   The model was implemented with all the transistor parameters as inputs (not pre-calculated). The design was synthesised targeting the Virtex 5 V5LX330T FPGA. The area utilisation ratio of the full CMOS model (32-bit operations) was estimated by the synthesis tool to be 52% of the Virtex 5 FPGA. This implementation is a pipelined version which includes all the model parameters as inputs. Table 5.7 shows the resource utilisation of the model, based on single-precision floating-point operators. The maximum clock frequency of this design was estimated to be around 96 MHz. The double-precision design could not be synthesised using the ISE synthesis tool due to the large design size.

Table 5.7: Resources Utilisation of the Full CMOS model Without parameter pre-calculation on the V5LX330T FPGA

Logic Resources	Available	Used	Usage (%)
Slice Registers	207,360	52,129	25
Slice LUTs	207,360	76,497	36
DSP48Es	192	159	82
Block RAM/FIFO	324	3	1

2. **With Parameter Pre-calculation**

   For the second step, the generic design was synthesised by setting the parameters to match the default values demonstrated in Table 4.1. The synthesis estimates of the resource usage of the model is nearly 7% of the FPGA resources. The maximum estimated frequency of the design is approximately 192 MHz. Table 5.8 shows the resource utilisation of the model, based on single/double-precision floating-point operators. The double-precision design used 22% of the FPGA with a maximum frequency of about 87 MHz.

   Table 5.9 gives the whole slave FPGA resources estimations, which includes the CMOS model plus the control and communication logic. It can be seen that the device model occupied a small portion of the FPGA due to its large hardware size.

Table 5.8: Resources Utilisation of the CMOS model With parameter pre-calculation on the V5LX330T FPGA

Logic Resources	Available	Single-Precision		Double-Precision	
		Used	Usage (%)	Used	Usage (%)
Slice Registers	207,360	5,931	2	18,308	8
Slice LUTs	207,360	7,352	3	20,476	9
DSP48Es	192	21	10	282	146

Table 5.9: Resources Utilisation of the Slave FPGA Design

Logic Resources	Available	Used	Usage (%)
Slice Registers	207,360	6,084	2
Slice LUTs	207,360	7,223	3
DSP48Es	192	21	10
Block RAM/FIFO	324	1	0.3

5.1.4.2 Acceleration Comparison

The study in [43] used a VLIW custom architecture to run the device model evaluation on the Virtex 5 V5LX330T FPGA. The V5LX330T device is far superior than the Virtex-II Pro FPGA used in our work in terms of capacity, frequency, and communication bandwidth. The V5LX330T contains 207,360 logic elements and supports an operating frequency of up to 550 MHz. The Aurora interface on the V5LX330T can have a bandwidth of up to 3.7 Gb/s and a frequency of up to 300 MHz.

The simulation and synthesis estimates showed that by using the Virtex 5 in our system, the Aurora serial communication can run at 156.25 MHz (3.125Gb/s) which is twice the speed of the current system (75 MHz). In Section 5.1.4.1, the synthesis tool estimated that the single-precision device model with parameters pre-calculation can run at 192 MHz. This means that the design can run at about twice the speed of the current system. Hence, theoretically, the new system that uses the Virtex 5 FPGA will have an approximate speedup of about 20 times over the software implementation discussed in Section 4.3. If the Virtex 5 FPGA support more than one serial links between the Host and the Slave FPGAs, the acceleration would be higher than this estimate as more data will be sent/received in parallel to/from the Host to the Slave FPGAs.

The acceleration figure reported in [43] was measured against a dual-core 3.0 GHz Intel Xeon 5160, which is faster than the one used in our system (i.e. Intel Due core 2.0 GHz). For simplicity of comparison, it is assumed that the Intel Xeon is roughly one and half times faster than the Due Core 2 (by taking the ratio between both processors' speeds). Hence, in order to make the comparison as accurate as possible, the acceleration figure in [43] is multiplied by the ratio between both processors' speeds. Hence, the new acceleration figure for [43] is about 10.5 which is measured against the Intel Due core 2 processor. Hence, it can be concluded that our system would achieve an acceleration of about 20 times which is nearly twice that for [43] if the Virtex 5 FPGA is used instead.

5.1.5 Discussion of Results

A prototype multi-FPGA system has been presented to accelerate the CMOS model device evaluation step in the SPICE simulator. The architecture demonstrated a speedup of up to 10 times over a C software implementation parallelised using the OpenMP library running on an Intel 2.0 GHz Due core 2 processor with 2.5 GB of RAM. Also, the system consumed six times less power than the processor system as seen in Section 5.1.1.

This application specific architecture to accelerate the CMOS model evaluation process can be used as a high speed co-processor attached to workstations to boost performance of SPICE-like simulations as seen in Section 3.4.3. Although the prototype system implementation is based on the CMOS LEVEL 3 model, it has been demonstrated that multi-FPGA systems can effectively be employed to accelerate the CMOS device evaluation process, and hence the SPICE simulation.

In order to reduce the complexity of the prototype system, the BRAM blocks were used to map the local FIFOs. This also adds to the modularity of the memory system, as data can be easily mapped from external memory systems. The BRAM can be addressed in parallel, through the FIFOs abstraction layer, which allows the data to be sent/received simultaneously to/from the computing nodes.

The acceleration result outlined here clearly depends on the number and the hardware capabilities of the computing FPGAs used. As FPGAs are on a fast technology growth curve, our system gives an idea about the amount of acceleration that can be achieved using current state-of-art FPGA boards.

In addition, the OpenMP-based software implementation was compiled using high optimisation level using commercial compilers. However, the FPGA code was not as optimised, as it begins with implementing a textbook algorithm, with little optimisation. Hence, an acceleration figure of 10 times over an optimised software illustrates the potential of reconfigurable systems.

Only the device evaluation phase was considered in this work. The acceleration behaviour showed very little saturation with three FPGAs, which means that more speedup could be achieved using more FPGAs in parallel. This becomes possible especially as large multi-FPGA systems [107, 179] with several FPGAs and superior hardware capabilities are available off-the-shelf as seen in Section 2.3.1. However, a trade-off between the system cost and the achievable acceleration must be identified.

5.2 Results Extension

The SPICE model and the FPGAs used in the system (Section 4.2) do not reflect current state-of-art technologies. Hence, the results shown do not reflect the true amount of acceleration that can be achieved. This section estimates the acceleration and resources usage for current systems and models by theoretically extending the results shown previously. Section 5.2.1 investigates the feasibility to include current SPICE CMOS models in the device evaluation process. Section 5.2.2 estimates the speedup results when the BEE3 multi-FPGA system is used.

5.2.1 Results Extension to Current SPICE CMOS Models

The CMOS LEVEL 3 model used in the evaluation process is an old SPICE model. It is used in the system due to its relative simplicity, code size, and its high acceptance in the SPICE simulator community. Section 4.1 showed that by pre-evaluating the constant parameters in the CMOS model, large amount of computations are reduced as they are done beforehand. This was based on the assumption that most circuit simulation uses one transistor technology for its transistors (i.e. pre-calculating parameters). Model parameters are changed to target new transistor technology.

However, in order to make the obtained results useful to current SPICE simulation environments, the generalisation to accelerate other current sophisticated models such as the BSIM4 [180] and the Penn State Philips (PSP103) [181] is needed. This requires further work to determine the feasibility of mapping these models on FPGAs in terms of resources and timing.

5.2.1.1 Resources Estimation Reference Point

The resources usage for the BSIM4 and PSP models can be estimated using the results obtained so far for the CMOS LEVEL 3 model. Since one of the major factors that affect the FPGA area usage is the number of floating-point operations to be implemented on fabric. Hence, the percentage decrease in the number of floating-point operations in the LEVEL 3 model (after pre-calculating all the transistor parameters) is used as a reference point to calculate the number of floating-point operations for both models. The percentage reduction in the number of floating-point operations is denoted by $Op_r(\%)$.

For experimental purposes, the Verilog-A codes for the BSIM4 [182] and the PSP [183] models were used to approximate the number of floating point operations ($*$, $+$, $/$, *sqrt*, *exp*, *log*, and *pow*), before any parameters substitutions are performed. These estimates are used to approximate the number of floating-points operations after all the parameters are pre-calculated, by applying the percentage reduction Op_r obtained earlier. The resulted number of floating-point operations after pre-calculating all the parameters is used to estimate the number of LUTs to implement both models in the FPGA fabric.

5.2.1.2 Resources Estimation Results

Table 5.10 summarises the results of resources estimation. The first column represents the different floating-point operations, with their corresponding area usage (Op_{LUT}) in the second column. The FPGA used for resource usage assessment is the Xilinx Virtex-5 (V5-LX110T device contains 17,280 slices and 64 DSP48E). The three sub-columns under the LEVEL 3 column represent respectively:

- Op_i is the initial number of the floating-point operations, in the model before the parameters pre-calculation procedure,
- Op_f is the final number of floating-point operations after the parameters pre-calculation procedure,
- $Op_r(\%)$ is the percentage reduction in the number of floating-point operations after the parameters pre-calculation procedure.

TABLE 5.10: Resource Utilisation Estimates for the BSIM4 and the PSP models

Operations	Op_{LUT}	LEVEL 3			BSIM4		PSP	
		Op_i	Op_f	$Op_r(\%)$	Op_i	Op_f	Op_i	Op_f
$*$	504	112	7	6.25	742	47	839	53
/	827	36	1	2.78	217	7	133	4
$+, -$	551	59	6	10.17	579	59	448	46
sqrt	477	11	0	0.00	49	4	33	3
exp	1,755	1	0	0.00	44	3	7	1
log	2,460	2	0	0.00	24	2	13	1
pow	4,708	1	0	0.00	0	0	33	3
Model LUTs		**7,661**			**74,079**		**75,136**	
Total LUTs		**8,581**			**82,969**		**84,152**	

The percentage reduction, Op_r, is used to calculate the final number of floating point operations (Op_f) for both the BSIM4 and the PSP models. In the case where Op_r is zero, the average reduction percentage of the non-zero values is used instead. The average reduction percentage of the non-zero values is approximately 6.4%.

The Op_f results were used to estimate the total number of LUTs used to implement the BSIM4 and the PSP models. This is calculated as the sum of each Op_f multiplied with its corresponding LUTs usage (Op_{LUT}) for each operation. The estimated Model LUTs usage for the models is demonstrated in the row before the last in Table 5.10.

In order to estimate the amount of logic utilised by the controlling state machine, routing resources, and the communication logic, we used the same method as above. The difference between the resources used by the pure CMOS model in Table 5.3 and the slave FPGA design in Table 5.4 would give an approximate percentage for the control and communication logic. The percentage is found to be be nearly 12%. The estimated Total LUTs usage for the models is demonstrated in the last row of the Table 5.10.

Current state-of-the-art FPGA device are far larger than the FPGA used in our system. For example the Virtex-5 VLX330T device contains 207,360 LUTs, and the latest Virtex-6 VLX760 contains 474,240 LUTs. Also, current FPGAs contain more built-in hardware blocks like the DSP48E slices (Each DSP48E slice contains a 25×18 multiplier, an adder, and an accumulator), and larger Block RAM blocks. Given these large hardware resources, both the BSIM4 and the PSP models can easily be fitted in such devices given the estimated total number of LUTs shown in the Table 5.10.

Furthermore, current FPGA devices are superior than the Virtex-II Pro, in terms of supported system clock and serial communication speeds (up to 6.5 Gb/s for RocketIO GTX transceiver [184, 27]), which suggests that the acceleration figures for the CMOS model will be better than the results in Section 5.1.1. This is further explored in the next section.

5.2.2 Theoretical Performance Estimation on BEE3

The acceleration result is highly dependant on the number of FPGAs used and their hardware capabilities. Current FPGAs are far more advanced and the technology growth is continuing as seen in Figure 1.4 [25]. Hence, better acceleration results are expected to be achieved using current state-of-art FPGA boards. This section outlines the theoretical performance estimation of the CMOS LEVEL 3 model on the BEE3 board, using the results demonstrated previously.

5.2.2.1 Performance Estimation of the XUPV2P System

By recalling the theoretical performance estimate of the prototype multi-FPGA system proposed in Equation 4.6, which can be reduced to Equation 4.7 by taking:

- the number of serial links between the host FPGA and each slave FPGA is one ($S_{links} = 1$),
- and the number of slave FPGAs is three ($S_{FPGA} = 3$).

The operating frequency of the system is (75 MHz), which is the same as the operating frequency of the serial communication system. This means that the hardware execution time of the prototype system in seconds, T_{hw}, is approximately given by:

$$T_{hw} \approx \frac{T_{theory}}{clk_{hw}} \tag{5.1}$$

Where clk_{hw} is the design's operating frequency (75 MHz).

5.2.2.2 Performance Estimation of the BEE3 System

The BEE3 multi-FPGA board is one of the current well known research systems [179]. Figure 5.7 outlines the architectural block diagram of this system. This system was selected for performance estimation due to its current acceptance in the research community and its advanced hardware capabilities.

The BEE3 system is the third generation of the BEE2 engine described in Section 2.3.1. The BEE3 board have four Xilinx Virtex-5 FPGAs combined with up to 64 GB of DRAM and several I/O subsystems [179], as seen in Figure 5.7. The FPGAs are connected as a ring with 72 connections between each adjacent devices. Each FPGA is connected to four memory banks, which provide superior memory bandwidth and hence faster access to data through the local BRAM buffers.

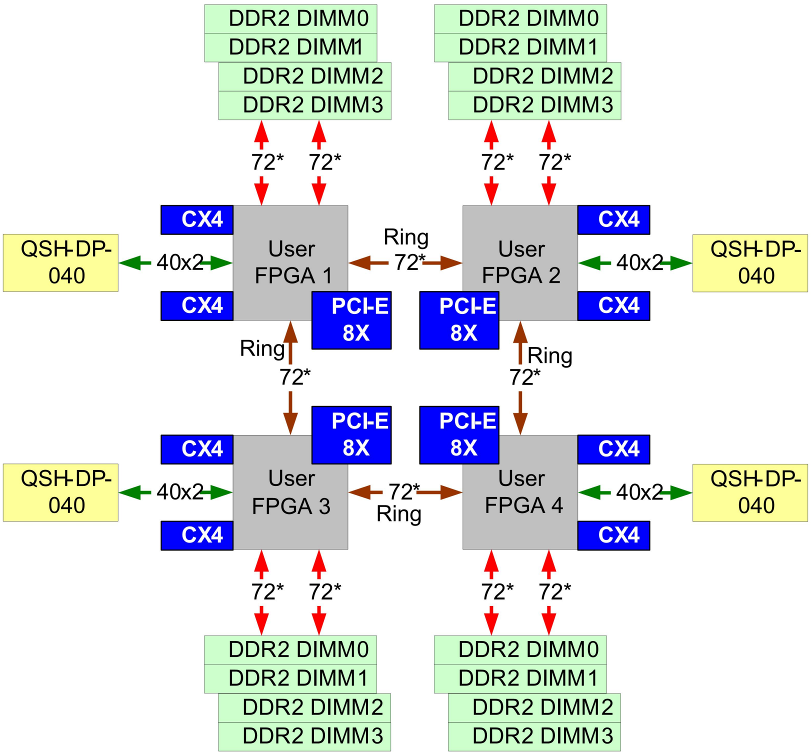


FIGURE 5.7: BEE3 Compute Module Block Diagram [179]

By mapping the system architecture shown in Figure 4.9 to the four available FPGAs in the BEE3 board (one host FPGA and three slaves), the same estimation equation as in Equation 4.6 can be used to approximate the performance figures on the BEE3 hardware, by taking:

- $S_{links} = 6$, the number of links between the host FPGA and each slave FPGA, this is taken to be the number of input words (6 words of 16-bit length) required to compute the LEVEL 3 model.
- and $S_{FPGA} = 3$ the number of slave FPGAs used for computing, as the board contains four FPGAs, three of them can be used for computing and one FPGA as a host controller.

Equation 4.6 becomes:

$$Cycles_{BEE3} \approx T_{init} + 2T_a + \frac{N}{3} + T_s \tag{5.2}$$

where $Cycles_{BEE3}$ is the estimated execution time on the BEE3 hardware in clock cycles.

Hence, the hardware execution time on the BEE3 hardware in seconds, T_{BEE3}, is given by:

$$T_{BEE3} \approx \frac{Cycles_{BEE3}}{clk_{BEE3}} \tag{5.3}$$

Where clk_{BEE3} is the design's operating frequency on the BEE3 hardware, which is estimated to be about 150 MHz.

The *Execution Time Ratio* between the BEE3 hardware execution time in Equation 5.3 and the prototype system execution time in Equation 5.1 gives an approximate estimation of the expected acceleration of the BEE3 hardware. This is given by the following:

$$Execution\ Time\ Ratio \approx \frac{T_{BEE3}}{T_{hw}} \approx \frac{2N}{2N/3} \approx 3 \tag{5.4}$$

Equation 5.4 shows that the BEE3 hardware is expected to be three times faster than the current prototype system. Hence, the theoretical results showed that the BEE3 hardware is expected to achieve an average acceleration of about 30 times than the OpenMP software implementation.

5.2.3 Discussion

Estimates of the resource usage showed that other current sophisticated models such as the BSIM4 and the PSP are feasible to be implemented on FPGAs. The system is also expected to provide larger speedup results, by taking advantage of more powerful FPGAs as demonstrated by the estimated speedup that can be achieved by the BEE3 engine.

The prototype system assumed one instance of the LEVEL 3 CMOS model per FPGA. Using BEE3 boards, Virtex-5 FPGAs can contain more instances of the model and hence have more computation threads. Therefore, the acceleration estimation reported in this section would be higher depending on the number of model instances that can be fitted on a single FPGA. Furthermore, the inter-FPGAs links between devices in the BEE3 board would enable more signals and hence more data to be exchanged in parallel. In addition, the superior memory bandwidth available will provide fast access to data through the local BRAM buffers.

5.2.4 Computing Device Model Derivatives

The prototype system discussed in Section 4.5.2 does not implement the device model derivative. For better accuracy and convergence, both continuous models and derivatives are provided. However, due to the unavailability of derivative for the LEVEL 3 model and the small size of the used FPGA, derivatives could not implemented and evaluated on the prototype system. This section looks into using the Secant Method instead of the Newton-Raphson's Method in SPICE simulations.

5.2.4.1 The Secant Method

Typically, NR method converges quadratically when a sufficiently close initial operating-point is chosen (Section 3.1.1). The simulator that uses the NR method should include routines to evaluate both the device model funtion $f(x_i)$ and its first derivative $f'(x_i)$ at the point x_i [127, 128]. The use of continues model functions and their derivatives ensures the continuity of the simulation without overflows or errors.

One of the main drawback of the NR method is that the first derivative $f'(x_i)$ cannot always be calculated or may be very expensive to evaluate. Other methods like the secant method are used in this case, which uses a difference quotient as seen in Equation 5.5 [127].

$$x_{i+1} = x_i - \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} f(x_i) \tag{5.5}$$

5.2.4.2 Accelerating the secant method

The secant method has a convergence of the order of 1.6 to 1.4 and linear at worst case, if the initial values are close to the root [185]. Hence, this method is slower in convergence than the NR Method. However, the main advantage of the secant method is that it does not require the existence of any derivatives of the main function.

In the circuit simulator in [186], device model derivatives were computed numerically using finite differences. This does not effect whether a simulation converges or not. The use of numerical differentiation, however, affects the rate of convergence of simulations. The rates of convergence drops from quadratic (NR), to approximately 1.68, which means that the error between successive iterations is reduced by a power of 1.68 instead of a power of 2. In [186], it was found that a step size (h) value of $h = 0.02V$ is suitable for computing the transconductance values for many non-linear devices [185].

Given the limited hardware area of FPGAs, the secant method would be an attractive solution to find the derivatives over Newton's method, as no separate derivative code is required. The use of difference equation in place of the derivative reduces the complexity of the hardware design and hence conserve device resources. The superior convergence rate of Newton's method can be overtook by the potential large speedup and parallelism delivered by FPGAs.

5.3 Summary

The synthesis results showed that by pre-evaluating the constant parameters in the CMOS model, large amount of computations are reduced. This is based on the assumption that most circuit simulation uses one transistor model during simulation. In other words, the device model parameters usually stay the same during the SPICE simulation.

The synthesis system can apply compiler optimisation techniques like constant propagation at the scheduling level. The parameters can be changed at the synthesis level to produce the required device model needed for a specific simulation.

Our architecture demonstrated a speedup of up to 10 times over a C software implementation, while consuming six times less power than the processor system as seen in Section 5.1.1 and Section 5.1.3. The pipelined model implementation is used to perform the device model computations on multi-FPGAs in a SIMD execution model. All FPGAs must have identical code that it is executed on different simulation data. The main contribution of this Chapter is that out prototype multi-FPGA system accelerated the device model evaluation step by up to 10 times. We expect that this approach would provide higher speedup figures if more powerful FPGAs are used.

Estimates of resource usage showed that other current models such as the BSIM4 and the PSP are feasible for implementation on FPGAs and are expected to provide similar speed-up results, by taking advantage of more powerful FPGAs. Given the small FPGAs used in our implementation process, the speed-up results may exceed the figures quoted herein.

The synthesis results in Section 5.1.2 also showed that large device models could be partitioned into a number of sub-designs that can be fitted onto multiple FPGAs. However, the number of inter-FPGA signals that have to be communicated would be considerable. These signals have to be routed through a limited number of FPGA I/O pins. Hence, the problem of exchanging the inter-FPGA signals over the limited communication bandwidth, emphasises the need to optimise the number of inter-device signals resulted from multi-FPGA partitioning. This issue is further explored in the next Chapter where an interconnection optimisation technique is proposed in Section 6.3.5.

Chapter 6

Multi-FPGA Partition Mapping

As the synthesis results in Section 5.1.2 showed that the full CMOS LEVEL 3 model used in the system in Section 4.3 could not be fitted on a single FPGA. Hence, this model can be partitioned and mapped onto a number of FPGAs, in which the number of inter-partitions connections must be minimised. This is because FPGAs are limited in terms of available I/O pins. These results emphasised the importance to optimise the inter-FPGA connections in the domain of multi-FPGA partitioning. This Chapter introduces a novel technique which optimises the process of mapping partitions to their corresponding FPGAs. Our technique exploits the topological properties of the Mesh topology using simple partition swapping operations. This technique is applied on the CMOS LEVEL 3 model in order to assess the effectiveness of the proposed technique on a SPICE simulator device model.

This Chapter investigates the use of high-level synthesis and the logic partitioning in the process of pin usage optimisation of a mesh-based topology. The design flow uses a partition mapping technique applied after partitioning to optimise the pin usage of the final design. This Chapter is organised as follows: Section 6.2 outlines the related work done in the domain of inter-FPGA communication systems. Section 6.3 explains the components of the multi-FPGA synthesis system used in the optimisation process. Section 6.4 presents the experimental setup used. Results, observations and conclusions are presented in Sections 6.5 and 6.6.

6.1 Problem Definition

The domain of applications requiring multi-FPGAs as a prototype or actual systems is growing noticeably over the last few years. Examples of such applications include emulating large System-on-Chip (SoC) designs [187, 12], evaluating large scale Networks-on-Chip (NoC) [188], and the use of increasing numbers of processors in Multi-Core systems [97, 189]. The synthesis results in Section 5.1.2 showed that the full CMOS LEVEL 3 model could not be fitted into a single FPGA and hence should be partitioned into a number of partitions. These partitions can be cascaded together in a pipelined fashion to achieve high performance, which can be mapped on multiple FPGA devices.

Speeding up such applications with Reconfigurable Computers is highly dependant on the amount of parallelism in each application and how this can be mapped efficiently on hardware. On the other hand, the complexity of reconfigurable computers is constantly increasing as well with example systems like BEE, RAMP, Maxwell, TMD, and DINI [107] as seen in Section 2.3.1. This paradigm is further encouraged by the enormous efforts of the research community to port several scientific computation kernels to the reconfigurable domain such as high-precision floating-point units and linear algebra kernels [190, 191].

These example applications are usually very large and need large amount of FPGA resources such as built-in blocks like memory, multipliers, and DSP; and require different mapping, design, memory, and computational requirements. Because of the large sizes of these applications, multi-FPGA mapping will introduce inter-FPGA connections which requires more I/O pins. Hence, there is a need to further investigate multi-FPGA system in the domain of partitioning, synthesis and resources optimisation including the I/O communication especially when fully spatial implementation is considered as this would require more inter-device connections.

6.2 Inter-FPGAs Communication Synthesis

Multi-FPGA synthesis involves both high-level synthesis and logic partitioning. The partitioning process can be performed at different abstraction levels such as behavioural, structural [192]. The design is broken down into a set of sub-designs connected with inter-partition signals synthesised over multiple devices [65, 193].

The output of the multi-FPGA design flow depends heavily on the target architecture. Inter-FPGA communication is one of the fundamental issues to be addressed when dealing with multi-FPGA systems. The communication could be done through direct pin-to-pin connections between FPGAs, or using architectures like Mesh [194, 195], Crossbar [196, 197], and Bus [198, 199].

The signals are changed in the source device, then passed on to the sink device. However, FPGAs are very limited in terms of the number of available I/O pins, and hence may not cover all inter-FPGA signals required. Poor inter-FPGA communication bandwidth, commonly limits the gate utilisation to less than 20% [200]. In order to overcome this limitation, a number of approaches have been proposed, which include the following:

6.2.1 Virtual Wires Approach

The *Virtual Wires* approach in [200] provides a way to overcome the FPGA I/O pin limitation. This approach multiplexes and pipelines the inter-FPGA logical signals into a single physical wire. Both the sender and the receiver sides use shift registers configured into shift loops. Partitioning is performed at the netlist (structural) level.

The hardware emulation platform used consists of a number of FPGA-boards each containing sixteen Xilinx XC4005 devices connected in a two-dimensional nearest-neighbour mesh. The design flow uses a commercial netlist partitioning tool and a routing algorithm that statically schedules and routes inter-FPGA communication. The routing hardware is then added to the resulted netlists.

Figure 6.1 shows an illustrating example of six logical signals allocated to six physical wires using the Hard wire interconnect approach (a). The figure also shows the Virtual Wire interconnect approach (b) in which the same six logical signals are multiplexed between two pipelined shift loops over the same single physical wire. Each register in the pipelined shift loop holds a single bit of the logical output from one FPGA to the corresponding logical input of the other FPGA.

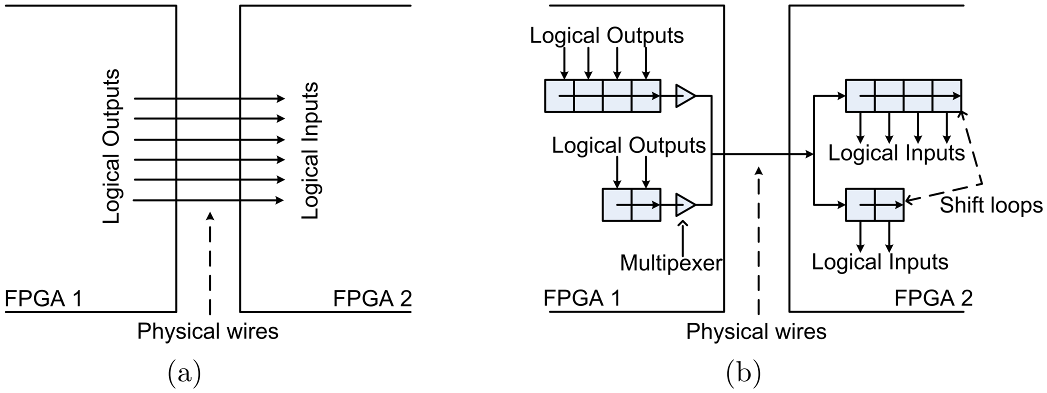


Figure 6.1: Hard wires interconnect (a), Virtual wires interconnect (b)

The design flow performs partitioning at the netlist level, which subsequently reduces the performance of the final system. Furthermore, the number of I/O signals resulted from the partitioning process is larger than the behavioural partitioning approach [201] as explained in Section 2.2.1. Multiplexing these signals via the same physical wire would hugely effect the performance of the system. In addition, the signals to be shifted in and out from one device to the other require multiplexing hardware in the receiver/sender ends. Hence, this approach traded-off area and performance with I/O usage.

6.2.2 FunctionBus Approach

Another approach to overcome the pin limitation problem was proposed in [65], where a single *FunctionBus* is used to transmit function calls between FPGAs. The bus architecture consists of a data/address bus *AD* with two bidirectional control lines *Areq* and *Dreq*, as shown in Figure 6.2 [65]. The *functional specification* of the designs were automatically partitioned rather than the structural implementation. Coarse-grain functions from the specification are considered as nodes, and data transfers between these functions are represented as edges. The bus implements these function calls by multiplexing the edges over a single bus, and multiplexing the control signal as addresses over the same bus.

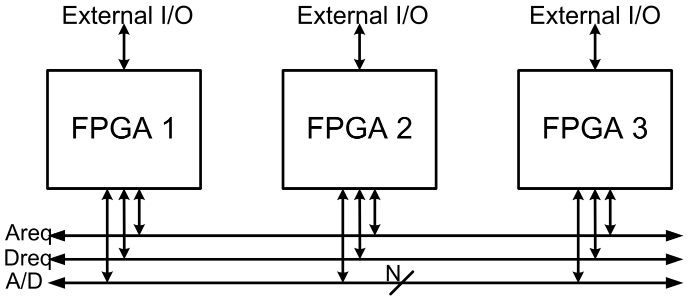


Figure 6.2: The FunctionBus Architecture

It was also shown in [65] that the system's performance can be traded-off with intercommunication by modifying the width of the bus AD. However, the bus approach would be very slow as only one FPGA is active and accessing the bus at any time. In addition, functions are executed in a sequential fashion where a caller function waits for the callee function to return and then resumes the execution. This approach underutilise the parallelism inherently embedded in FPGAs.

6.2.3 Other Approaches

Other topologies and partitioning approaches were proposed to improve upon the resource utilisation in multi-FPGA systems. A circuit partitioning algorithm, introduced in [202], uses time-multiplexed interconnection wires between FPGAs exploiting multicasting signals to reduce the pin limitation. This technique is named Time-multiplexed Off-chip Multi-casting interconnection (TOMi). The performance of this architecture can suffer degradation depending on the number of FPGAs and intercommunication signals [203]. Figure 6.3 shows the overall TOMi architecture [204].

A similar approach was proposed in [205, 206] based on time-multiplexing the I/O signals . The algorithm uses an Integer Linear Programming formulation (ILP) to find

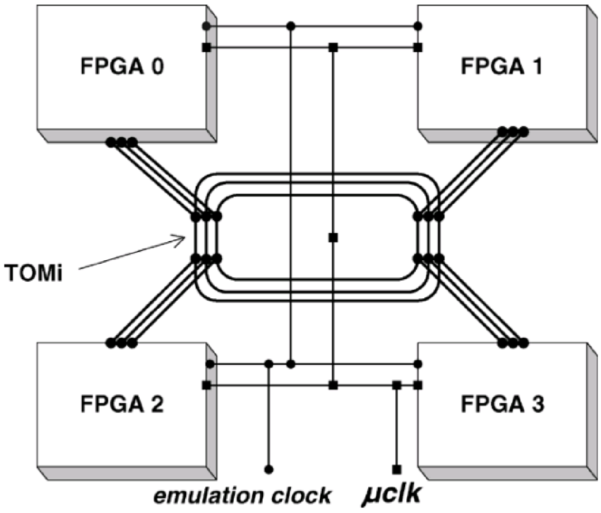


Figure 6.3: Block Diagram of the TOMi Architecture [204]

the optimum number of required I/O pins under the given time constraints by selecting the signals to be time-multiplexed. However, forming and solving ILP equations is computationally expensive as the number of variables heavily limits the solver finding a solution.

In [194], a placement algorithm called Placement&Routing-based Partitioning (PRP) was proposed. The multi-FPGA mesh topology is modelled as a single large FPGA and the borders between devices is considered as a superimposed template. The Simulated Annealing placement optimisation approach used to route inter-FPGA nets amongst fixed mesh topology.

The partitioning approaches discussed earlier focus only on the partitioning and the synthesis without taking into account the process of mapping the resulting partitions to their corresponding FPGAs. A fixed mapping of partitions to FPGAs is assumed before their corresponding optimisation process is performed. In other words, there is not consideration of which partition should be mapped to which FPGA. The technique presented here exploits the topological properties of the Mesh topology in order to reduce the physical wire count of the final design by optimizing the mapping process.

6.3 Multi-FPGA Synthesis System

A functional block digram of the proposed synthesis system is shown in Figure 6.4. This section explains in details the components of this system.

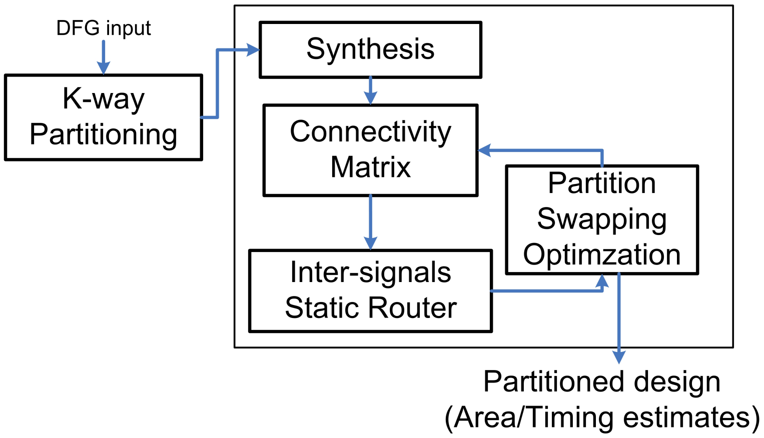

DFG input
K-way Partitioning
Synthesis
Connectivity Matrix
Partition Swapping Optimzation
Inter-signals Static Router
Partitioned design (Area/Timing estimates)

FIGURE 6.4: Block Diagram of the Multi-FPGA Synthesis System

6.3.1 Input Specification Model

The input of the system is expressed in terms of Data Flow Graphs (DFG). The DFG is a directed graph that consists of a number of *nodes* representing operations connected via a number of *edges* denoting data transfers (dependencies) between these operations. DFG is a common intermediate representation both in software and hardware which is used to model the data flow information. Hence, the DFG model appeared to be the suitable starting point for our synthesis process. The current prototype synthesis system supports DFG only, where operation are executed without any control flow changes.

6.3.2 K-way Partitioning

Logic Partitioning for multi-FPGAs systems consists of splitting the internal representation into a number of balanced partitions. Each partition is mapped to a particular FPGA. Our system uses the partitioning tool *Metis* in [207, 208]. *Metis* is a family of programs for partitioning unstructured graphs and hypergraphs. The underlying algorithms used are based on the multilevel partitioning approach that has been shown to produce high quality results and scale to very large problems. This approach starts off by coarsening (clustering) the original graph to reduce its size, then partitioning the smaller graph, and finally uncoarsening it to construct the final partition. The graph nodes can represent different design components; hence, this tool can perform partitioning at different levels of abstraction.

The DFG description of the input design is partitioned into balanced segments using Metis. The output of the partitioner consists of a number of partitions connected together using inter-partitions signals. This task is performed before the synthesis step.

6.3.3 Synthesis

High-level synthesis (HLS) is the process of transforming the DFG sub-graphs produced by the partitioner to a structural description satisfying user constraints on area, delay and power consumption. For the sake of simplicity, the synthesis task in this work mainly focuses on the scheduling of operations, as it was assumed that operations were uniquely mapped to specific resources. In other words, a spacial implementation of design is considered. The graph nodes can represent different components at different levels of granularities.

The scheduler implements a version of the resource constrained List Scheduling heuristic. The priority list labels each node with its longest path to the sink. Nodes are then ordered in a decreasing order, in other words, the most urgent operations are scheduled first [49].

6.3.3.1 Scheduling under Inter-FPGA I/O Constraints

After performing the partitioning task, inter-FPGA I/O operations are inserted in the DFG at each cut net. Each I/O operation represents the communication block between the connected FPGAs. I/O operations are then scheduled similarly to functional operations under I/O constraints. The List Scheduling processes the I/O operations according to their priority (urgency in this case) and resources availability. A constraint was added to the scheduling model to limit the number of simultaneous communication nodes in a single control step.

A simple example is illustrated in Figure 6.5. The two cut nets produced by the partitioner are replaced by two I/O operation nodes. For this example, the inter-FPGA transfer delay is assumed to be one clock cycle for each communication node. After rescheduling, the overall latency increased by one clock cycle.

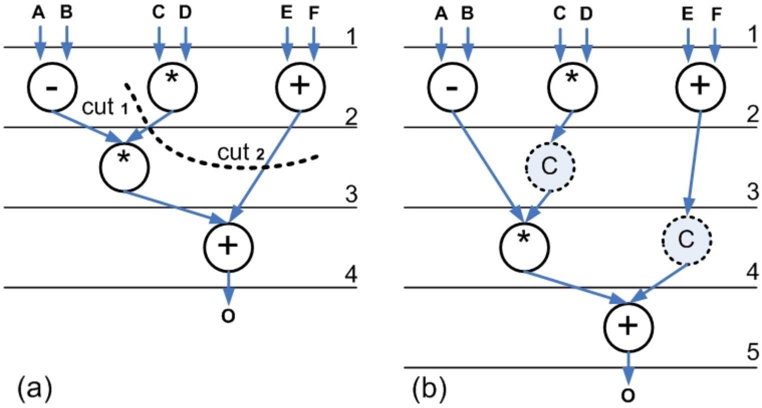


FIGURE 6.5: Inter-FPGA Communication Nodes Insertion

6.3.4 Target Architecture Models

In this section, two target model architectures were used in the evaluation process:

1. The first architecture is a classical Mesh topology with a grid of FPGAs connected together using direct physical connections.
2. The second architecture is also a Mesh based topology based on the Virtual Wires model outlined in [200]. The latter is different from the classical Mesh in the fact that the inter-FPGA signals are time-multiplexed to reduce the pin utilisation. In other word, Virtual Wires approach multiplexes and pipelines the inter-FPGA

logical signals into a single physical wire as seen in Section 6.2. For the second architecture the Virtual Wires Model is applied after the Mesh Model to reduce the inter-FPGA connections.

Figure 6.6 shows a diagram of the target Mesh topology, where p_i denotes the partition mapped to $FPGA_j$. The fully connected mesh is generally too expensive to realise unless the number of devices is small. Partially connected topology are more common where some nodes are connected to more than one other devices usually with point-to-point links. Internal signals are changed in one device and are fed to next neighbouring device until reaching the target devices. A number of multi-FPGA systems shown in Section 2.3.1 are based on varying mesh topologies, which make them interesting targets for inter-FPGA bandwidth optimisation.

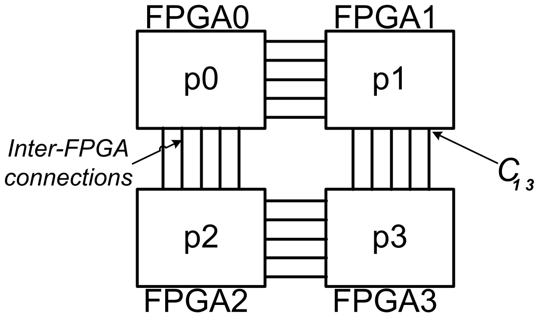


Figure 6.6: Typical Mesh Topology

6.3.4.1 Problem Formulation

FPGAs in a classical Mesh topology are connected to their imediate neighbouring FPGAs through direct I/O connections. The pin utilisation optimisation process should satisfy the constraints in Equation 6.1:

$$\text{minimise} \quad C_{total} = \sum C_i, \quad \text{where:} \quad C_i \leq C_{max} \tag{6.1}$$

where C_i is the number of pins utilised in each FPGA, C_{max} is the number of I/O pins available in each FPGA. C_{total} denotes the total number of pins utilised by each design.

In this work, we use the *wire count W* to measure the effectiveness of the proposed approach. This is defined as the number of all the allocated physical wires amongst all devices.

6.3.4.2 Connectivity Matrix

The Connectivity Matrix is an intermediate data structure used by the algorithm to keep track of the different partitions and their FPGA mapping to the actual architectural model. This structure also stores information such as the number of I/O transfers amongst FPGAs and their timings. The matrix also stores the N×N mappings of partitions to FPGAs.

The Connectivity Matrix acts like the Bucket structure in the FM algorithm (Section 2.2.1). The Connectivity Matrix, however, stores the state of the partitions and the FPGAs in the system, whereas the FM bucket stores the graph nodes and their corresponding gains.

6.3.4.3 The Inter-FPGA I/O Static Router

Once the partitioner divides the input DFG into a number of sub-graphs (N×N partitions), these are then mapped to devices in the topology. The inter-FPGA signals have to be routed from a source to a sink FPGA. The routing process is performed as follows:

- For the Mesh model, the inter-FPGA signals are first statically routed using the same approach of the Virtual Wires routing algorithms. Direct physical wires are used between adjacent FPGAs, and the feedthrough signals are routed through the neighbouring FPGAs until the sink FPGA is reached. In the routing process, no time-multiplexing technique is applied and all signals are treated separately as direct physical connections.

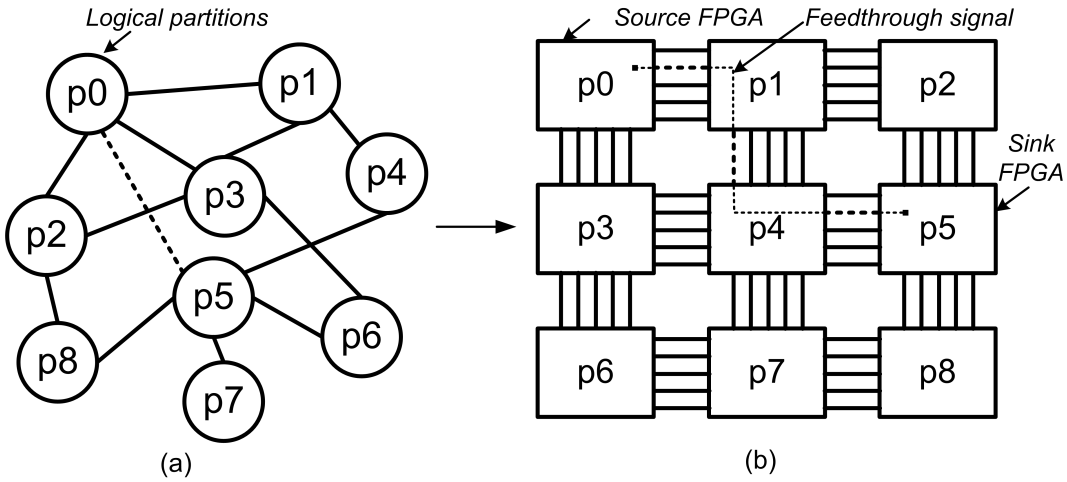


Figure 6.7: Static routing of Inter-FPGA signals using Shortest Path Algorithm, (a) Partitioned graph, (b) Partition Mapping and Signal Routing

- In the case of the Virtual Wires architecture, the *phase routing algorithm* outlined in [200] is applied to route the inter-FPGA nets. Direct connections between neighbouring devices are time-multiplexed and routed using physical wires. Connections between non-neighbouring FPGAs are routed through the nearest neighbours until reaching the target FPGA. This algorithm uses the Dijkstra's Shortest Path algorithm to find the shortest path between the source and the sink FPGAs. The process of mapping partitions to FPGAs and signal routing is shown in Figure 6.7. A signal between the source p_0 to the sink p_5 is routed through neighbouring FPGAs p_1 and p_4.

  The signals routed through intermediate FPGAs are called *feedthrough* signals. The algorithm takes into account the number of I/O pins for each device while routing the signals. If all physical connections are used, signals are scheduled to start in a later phase. One of the outputs of this procedure is a mapping of partitions to specific FPGAs with all I/O operations scheduled to particular phases. Pin utilisation of the architecture is also calculated.

6.3.5 Partition Swapping Optimisation

The output of the partitioning process consists of a number of balanced segments connected together via a number of I/O operations. These partitions are then mapped to FPGAs in both the Mesh and the Virtual Wires architectures. The phase routing procedure then statically routes the inter-FPGA signals and stores all mapping information to the Connectivity Matrix.

Assuming that there is no restriction on the flow of control or any constraint to map a particular part of the circuit to a specific FPGA; it can be suggested that any logical partition can be mapped to any FPGA in the system. For instance, a simple mapping process might assign a partition to its corresponding FPGA e.g. map partition p_1 to $FPGA_1$. This will result in a number of signals being routed through other FPGAs via feedthrough signals.

The algorithm outlined in this section explores the possibility to optimise the design by altering the *Partition* → *FPGA* mapping process in the Connectivity Matrix. The assumption that partitions are not locked for movement is important because the algorithm tries to reduce the wire usage by swapping the logical partitions to minimize the number of inter-FPGA physical connections. If there is a restriction on mapping the logical partitions to specific FPGAs, the swapping operation cannot be performed. It is also assumed that each FPGA is assigned a single logical partition only.

The novelty of our technique lies in using simple swapping operations to exploit the topological properties of the Mesh topology. Our technique looks at optimising the mapping process of partitions to FPGAs. A simple direct mapping process would assign each *Partition i* to $FPGA_i$. However, our technique tries to optimises this mapping process by swapping partitions from one FPGA to another in order to reduce the inter-FPGA connections.

The Kernighan-Lin algorithm [62] is used in minimum-cut graph partitioning to iteratively improve the solution by swapping pairs of vertices between partitions to reduce the number of cut nets between partitions (i.e. the cutsize) as seen in Section 2.2.1.

Our technique, however, is applied after the partitioning process, by swapping pairs of partitions to minimize the number of inter-FPGA connections used by exploiting the topological property of the Mesh architecture. Also, partitions are not fixed to a specific FPGA after being moved, which allow them to be swapped again as long as the move improves the current mapping.

The Partition Swapping technique starts by assigning partitions to their corresponding FPGAs, following the Direct Mapping approach in which $Partition\ i$ is assigned to $FPGA_i$. The algorithm then calculates the number of the physical wires after swapping each pair of logical partitions. The pair that results in reducing the wire count most is selected for swapping. This process is called a pass. A new pass is started and a pair of partitions is then selected for swapping again. The algorithm terminates when a pass cannot improve the design; in other words, the number of the physical wires cannot be reduced any further.

6.3.5.1 Algorithm Example

A simple example is illustrated in Figure 6.8. The example consists of four partitions p_0 to p_3 connected together using logical links. The integer numbers shown near each connection denotes the number of I/O links between the connected partitions. For this example, the I/O links are assumed to be 1 bit-sized. The figure also shows the multi-FPGA target architecture consisting of a Mesh of four FPGAs (2×2) connected together using direct connections. Two mapping techniques *(a)* and *(b)* are demonstrated as seen in the Figure.

The Direct Mapping approach (a) maps the partition p_i to the corresponding $FPGA_i$, regardless of the number of the I/O links between these partitions. In this case, the connections between p_0 and p_3 (12 links) are routed through the neighbouring ($FPGA_1$) to reach the target $FPGA_3$. This means that $FPGA_1$ would have at least 24 of its bits wasted to feedthrought signals between $FPGA_0$ and $FPGA_3$. The estimated wire count for this mapping is 36.

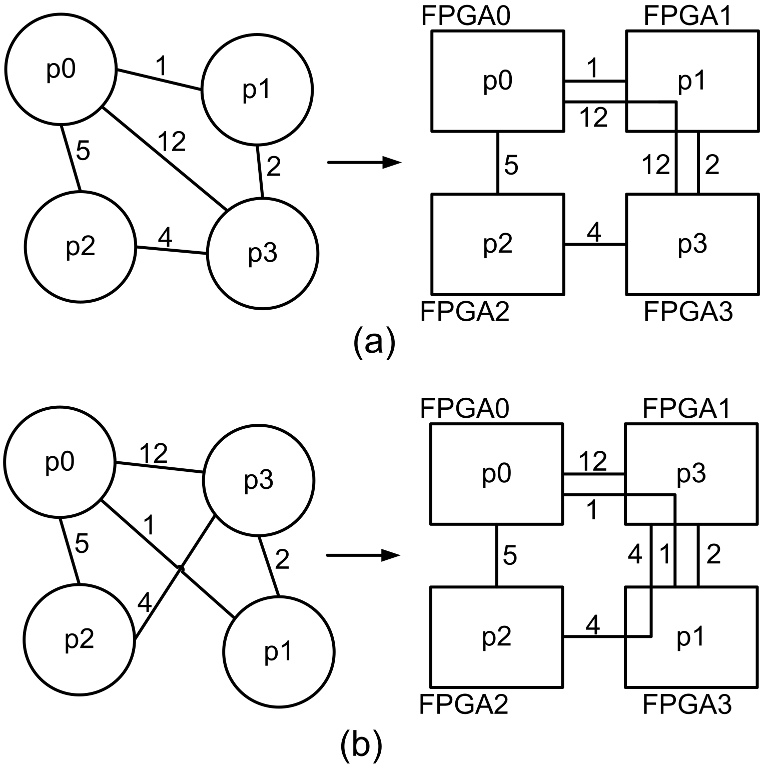


Figure 6.8: Partition Swapping Optimisation example: (a) Direct Mapping, (b) Partition Swapping

The Partition Swapping technique (b), however, improves on the Direct Mapping solution in (a). The number of physical wires required was reduced to 29, simply by swapping the partitions p_1 and p_3.

In this example, only one swap operation was identified and no more improvement is possible. In the case of much larger examples, the algorithm follows the same process to identify the best sequence of the swapping operations that leads to the minimum number of wires.

6.3.5.2 Algorithm Pseudo Code

Our technique is applied after the partitioning process by swapping pairs of partitions to minimize the number of inter-FPGA pins used by exploiting the topological property of the Mesh topology. Partitions can be swapped again as long as the move improves the current mapping. The novel features of our technique lies in using simple swapping operations to exploit the topological properties of the Mesh topology. Our technique optimises the mapping process of partitions to FPGAs by swapping partitions from one FPGA to another in order to reduce the inter-FPGA connections.

The swapping operation reduces the feedthrough signals which are going from the source FPGA to the sink FPGA through the neighbouring FPGAs. In other words, this technique brings the FPGAs that have large number of interconnections closer to each other to avoid these connections to be routed through other neighbouring devices.

The algorithm shown in Figure 6.9 is the pseudo code for the proposed optimisation technique. The function
`routingAlgorithm()` is an implementation of the routing algorithm that corresponds to the specific target architecture (the mesh routing algorithm and the phase routing algorithm). This function estimates the number of the physical wires of the mapped design. `swap(i,j)` is the function to swap the partitions i and j. This is done by switching a partition tag in all the functional nodes belonging to these partitions.

In line 10, the partitions i and j are swapped back after the wire count value is calculated; this is performed so that the swapping operation will not effect the values calculated in the next pass. After each swap operation, the resulting `tempWireCount` - which is less than the initial wire count `initWireCount` - is stored using the function `storeTempWireCount()`. A swap operation is selected if there exist a `tempSmallestWireCount` which is less than the `tempPinCount` achieved so far.

The algorithm terminates if there is no swap operation that further reduces the `tempPinCount`. The `optWireCount` is the final wire count achieved by the optimisation algorithm.

```
initWireCount ← routingAlgorithm()
optWireCount ← initWireCount
loop
  {Start of a pass}
  tempWireCount ← optWireCount
  for $i$ = partition 0 to (n-1) do
    for $j$ = partition (i+1) to (n-1) do
      swap(i, j)
      swapWireCount ← routingAlgorithm()
      swap(j, i)
      if swapWireCount $<$ tempWireCount then
        storeTempWireCount()
        tempWireCount = swapWireCount
      end if
    end for
  end for
  tempSmallestWireCount ← findSmallestTempWireCount()
  if tempSmallestWireCount $<$ tempWireCount then
    swap(i, j)
    optWireCount ← tempSmallestWireCount
    {End of a pass}
  else
    break
    {End of algorithm}
  end if
end loop
```

Figure 6.9: The Partition Swapping Optimisation Algorithm

6.4 Experimental Setup

6.4.1 Benchmark DFGs

Table 6.1 shows a number of designs with varying sizes used as inputs to the multi-FPGA synthesis system described earlier. The input operators are 16-bit sized. The first five design were first written mathematically in a straight-line fashion (i.e. no loops or conditional) and then translated into our custom DFG format.

2D4x4DCT and **2D8x8DCT** are DFG examples for two dimensional discrete cosine transforms of 4x4 and 8x8 matrices respectively. All the *sine* values are pre-calculated and hardcoded in the code. **MatMult** are the DFG examples for $n \times n$ matrix multiplication with different lengths (4, 8 and 16).

CMOS LEVEL 3 is model used in our device model accelerator in Section 4.3. We used the full CMOS LEVEL 3 model before the parameters pre-calculation. Our technique is applied on the CMOS LEVEL 3 example in order to assess the effectiveness of the proposed technique on the SPICE simulator device model. This model could not be fitted on the Virtex-II Pro, and hence makes a suitable benchmark to test our technique.

DFG is a randomly generated DAG (Directed Acyclic Graphs) using the random generation tool described in [209]. This graph is then annotated randomly with information like operation type and area size. The randomly generated graph is used to assess the effectiveness of the system with big designs. This is due to the lack of large DFG benchmarks dedicated for synthesis and partitioning in literature. Also, the random graph is used to examine whether it can be representative of the actual DFGs in synthesis and partitioning.

Table 6.1: DFG Benchmarks

Benchmark	No. of Nodes	No. of Edges	Latency (cycles)
2D4x4DCT	226	336	9
2D8x8DCT	1922	2880	11
MatMult4	114	176	5
MatMult8	962	1472	6
MatMult16	7938	12032	8
CMOS LEVEL 3	143	246	40
DFG	15606	45878	1065

6.4.2 System Implementation

The multi-FPGA synthesis system discussed in Section 6.3 (Figure 6.4) was implemented in Java. The target architecture consists of 16 FPGAs connected together in a 4 × 4 grid topology. Inter-FPGA signals are routed using both the Mesh model and the Virtual Wires model. The system input consists of DFG graphs. These graphs are partitioned using the *Metis* tool into 16 balanced partitions. Partitions are balanced according to their size, where the total size of the all operations in each partition must be the same (Section 2.2.1.3). Each design is then routed using the phase routing algorithm. This algorithm time-multiplexes all inter-FPGA logical signals into physical wires. The optimisation algorithm then tries to reduce the wire count of the design using the procedure explained in Section 6.3.5.

The same procedure is applied to the Mesh model, but without the time-multiplexing step. The evaluation metric used to measure the improvement in this experimental work is the *wire count*. This is the number of effective physical wires between all the FPGAs. It is clear that reducing the wire count of the design implies the reduction in the pin utilisation of FPGAs.

6.5 Experimental Results

The multi-FPGA synthesis optimisation procedure was applied to the different DFG benchmarks and the results are shown in Table 6.2 for the Mesh model and Table 6.3 for the Virtual Wires model.

Table 6.2: Wire Count improvement using the Partition Swapping technique for the 4 × 4 Mesh Model

Benchmark	W_{init}	W_{final}	P	$I(\%)$
2D4x4DCT	100	97	2	3.00
2D8x8DCT	232	230	2	0.86
MatMult4	46	34	2	26.09
MatMult8	167	132	2	20.96
MatMult16	843	542	7	35.71
CMOS LEVEL 3	128	103	4	19.53
DFG	1916	1414	7	26.20
Averages			**3.71**	**18.91**

Table 6.3: Wire Count improvement using the Partition Swapping technique for the 4 × 4 Virtual Wires Model

Benchmark	W_{init}	W_{final}	P	$I(\%)$
2D4x4DCT	63	63	0	0.0
2D8x8DCT	177	163	4	7.91
MatMult4	43	33	2	23.26
MatMult8	155	121	3	21.94
MatMult16	728	473	5	35.03
CMOS LEVEL 3	61	48	5	21.31
DFG	187	149	4	20.32
Averages			**3.29**	**18.54**

Both tables include the results of applying our optimisation algorithm on the SPICE LEVEL 3 model. The results demonstrate the optimised wire counts for both architectures: the Virtual Wires and the Mesh models. The tables show the *initial wire count* W_{init} obtained using the routing algorithm that corresponds to each architecture model prior to the optimisation process. The next column of the tables show the *final wire count* W_{final} which is the minimum number of connections achieved by our optimisation algorithm. The number of passes performed by the algorithm is P, and the percentage improvement in the wire count $I(\%)$ are shown in the last two columns of the tables. This is calculated as the relative difference bewteen the initial wire count W_{init} and the final wire count W_{final} as follows:

$$I(\%) = \frac{100(W_{init} - W_{final})}{W_{init}} \tag{6.2}$$

The average improvement over the evaluated benchmarks was found to be approximately 18% for both architecture models. From both tables (6.2 and 6.3), it can also be seen that there is consistency between the wire count of the actual DFG graphs and the random DFGs. It can also be noted that there is consistency between the wire count for the large and the small graphs. The latter suggests that the algorithm is also effective when dealing with large random graphs. The average wire count is approximately the same for both architectures. The table also suggests that the algorithm can perform an average of 3 to 7 iterations to achieve the final wire count. It can be seen from the table that the concept of Virtual Wires provides an improvement over the Mesh topology, which confirms the conclusions in [200].

The results showed that our technique achieved nearly the same results for both the Mesh model and the Virtual Wires model. This is because the main factor which drives the swapping technique is minimising the interconnection between FPGAs regardless if they are time-multiplexed or not. Hence, as far as our technique is concerned, a Mesh model and a Virtual Wires model would look like a generic Mesh topology with connections between partitions.

6.5.1 Results Comparison

In order to assess the effectiveness of the optimisation approach, our results are compared with the *optimum wire count* denoted by W_{opt}. This is obtained by calculating the wire count W for each possible mapping of the partitions. However, calculating such results for the 4×4 Mesh would require phenomenal amount of time (16! permutations). Instead, only the following Mesh topologies were calculated: 3×3 and 3×2. Each of the latter arrangement was tested on both the Mesh and the Virtual Wire Models.

6.5.1.1 Mesh Model Case

Tables 6.4 and 6.5 compare the calculated wire count with the optimum values for the 3×3 and 3×2 Mesh Models. The tables show the *initial wire count* W_{init} obtained using the Mesh routing algorithm prior to the optimisation process. The next column of the table shows the *final wire count* W_{final} which is the number of connections achieved by our optimisation algorithm.

The next column shows the *optimum wire count* W_{opt}. The Optimum wire count values are obtained by calculating the wire count of all the possible mapping permutations. In other words, all the possible cases of mapping N×N partitions to N×N FPGAs, which would result in $N!$ possible mappings to be tested. Due to the time complexity of this process, only 3×2 and 3×3 mesh topologies were calculated as seen in Tables 6.4 and 6.5. The last column indicates the relative error value $Error(\%)$ between W_{final} and W_{opt} which is given by:

$$Error(\%) = \frac{100(W_{final} - W_{opt})}{W_{final}} \tag{6.3}$$

It can be seen from the table that the results obtained by our approach are the same or very close to the optimum solutions. It can also be seen that our optimisation technique is also effective for the CMOS LEVEL 3 device model used in our multi-FPGA accelerator in Section 4.3.

Table 6.4: Comparing the wire count improvement with the optimum solution for 3×3 mesh.

Benchmark	W_{init}	W_{final}	W_{opt}	$Error(\%)$
2D4x4DCT	79	49	49	0
2D8x8DCT	193	161	161	0
MatMult4	34	26	26	0
MatMult8	116	91	91	0
MatMult16	464	368	368	0
CMOS LEVEL 3	87	64	64	0
DFG	1237	967	930	3.83

Table 6.5: Comparing the wire count improvement with the optimum solution for 3×2 mesh.

Benchmark	W_{init}	W_{final}	W_{opt}	$Error(\%)$
2D4x4DCT	55	41	41	0
2D8x8DCT	224	150	150	0
MatMult4	25	22	22	0
MatMult8	107	83	83	0
MatMult16	437	334	333	0.30
CMOS LEVEL 3	60	51	51	0
DFG	842	669	669	0

6.5.1.2 Virtual Model Case

Tables 6.6 and 6.7 compare the calculated wire count with the optimum values for the 3×3 and 3×2 Virtual Wires Models. The tables show the *initial wire count* W_{init} obtained using the Virtual Wire routing algorithm prior to the optimisation process, the *final wire count* W_{final} which is the number of connections achieved by our optimisation algorithm. The *optimum wire count* W_{opt} figure is calculated the same way used in Section 6.5.1.1. The last column indicates the relative error value $Error(\%)$ between W_{final} and W_{opt} which is calculated using Equation 6.3.

The same conclusions obtained for the Mesh model can also be observed for the Virtual Wire model as the results obtained by our approach are the same or very close to the optimum solutions. It can also be seen that our optimisation technique is also effective for the CMOS LEVEL 3 device model.

Table 6.6: Comparing the wire count improvement with the optimum solution for 3×3 mesh.

Benchmark	W_{init}	W_{final}	W_{opt}	$Error(\%)$
2D4x4DCT	49	34	33	3.03
2D8x8DCT	129	109	108	0.93
MatMult4	29	20	20	0
MatMult8	109	86	86	0
MatMult16	417	326	325	0.31
CMOS LEVEL 3	30	24	22	9.09
DFG	95	85	82	3.66

Table 6.7: Comparing the wire count improvement with the optimum solution for 3×2 mesh.

Benchmark	W_{init}	W_{final}	W_{opt}	$Error(\%)$
2D4x4DCT	38	27	27	0
2D8x8DCT	148	102	102	0
MatMult4	22	18	18	0
MatMult8	97	73	73	0
MatMult16	372	287	287	0
CMOS LEVEL 3	22	18	18	0
DFG	67	51	48	6.25

6.6 Results Analysis

An effective technique was presented to reduce the inter-FPGA connections and hence the I/O pins utilisation of the Mesh-based multi-FPGA architecture. The technique achieved an average improvement of about 18% as seen in Section 6.5. The results produced were very close to the calculated optimum solution. The technique is based on optimising the process of mapping the logical partitions to their corresponding FPGAs. It has been tested on an architecture based on the Virtual Wires model and a classical Mesh model. Our technique showed the same effectiveness when applied on the LEVEL 3 SPICE device model.

In order to reduce the complexity of the implementation, a number of assumptions were made. This technique assumes that partitions can be freely swapped from one FPGA to the other. However, if there is any design restriction on a particular part that cannot be swapped, this would reduce the effectiveness of the optimisation process. The process of parsing and lexically analysing the high level descriptions using C or VHDL was not implemented due to time limits, hence DFG graphs were used instead as nodes and edges can represent systems at different abstraction levels.

The results shows that the algorithm is also effective in the case of large DFG designs. The randomly generated graph showed similar improvement results to the actual DFGs, which suggests that the random graphs are representative of actual graphs. Because the technique uses simple swapping operations, it can be applied after the partitioning step regardless of the level of granularity is was performed at.

The swapping technique does not require any extra hardware circuitry for I/O interfacing like the shift loops (serialiser, diserialiser) needed in the Virtual Wires. The proposed approach can be extended to exploit the topological properties of other architectures. It can also be integrated into current multi-FPGA EDA tools like Certify [70] at the optimisation stage to reduce the physical wire usage and hence increase the FPGA logic utilisation.

6.7 Summary

In this Chapter, a multi-FPGA synthesis system has been presented focusing on minimising the inter-FPGA connections. This system performs partitioning and synthesis on the input design specified using DFG graphs. Inter-FPGA I/O operations are inserted in the partitioned graphs and handled similarly to other functional operations, this simplifies the scheduling and the static routing tasks. The main focus of our synthesis system, however, is optimising the inter-FPGA connections which subsequently reduces the FPGA I/O pins usage. An optimisation technique is introduced to minimise the inter-FPGA connections by applying partition swapping moves. The novelty of our technique lies in using simple swapping operations to exploit the topological properties of the Mesh topology. Our technique looks at optimising the mapping process of partitions to FPGAs by bringing the FPGAs with heavy inter-communication closer together.

One of the main advantages of our technique is that it does not require any extra hardware circuitry like the ones needed in the Virtual Wires. Furthermore, it can be applied after the partitioning process regardless of abstraction level is was performed at. The technique was also shown to be effective to reduce the pin-usage when the SPICE LEVEL 3 model was used as a benchmark.

Future extension to our technique is to investigate the application of such technique in system-on-chip clustering and FPGA on-chip routing. In addition to extending the algorithm to work on mapping multiple partitions on a single FPGA.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

EDA tools and algorithms are example applications which are demanding more computational power due to the current increase in complexity of circuits. This made the transistor-level SPICE simulation a growing bottleneck in the circuit development process. The main objective of this thesis is to investigate the FPGA's potential to accelerate the SPICE simulator through parallelism. Our work demonstrated an architecture to exploit the parallelism in the SPICE model evaluation to maximum acceleration.

The first part of this thesis focuses on the design and implementation of a prototype multi-FPGA system to accelerate the device model evaluation step in the SPICE simulator. The main contribution of this part lies in the proposed architecture which exploits three degrees of inherent parallelism available in the model evaluation phase. Firstly, each device model can be evaluated independently from each other, which can easily be mapped on a number of FPGAs. Secondly, the device model is pipelined using a spacial implementation in order to maximise the throughput. The pipelined design can start a device model evaluation at constant number of clock cycles. Thirdly, instruction level parallelism is also exploited by executing multiple instructions simultaneously.

The second part of the thesis outlined the experimental validation of the proposed architecture. The experimental results demonstrated speedup of up to 10 times over a C software implementation parallelised using the OpenMP library running on an Intel 2.0 GHz Due core 2 processor with 2.5 GB of RAM. The FPGA accelerator consumed one sixth of the processor power. The main factor which allowed the FPGA accelerator to show such improvement over conventional processors is the flexibility and efficiency of FPGAs to implement custom pipelined datapaths. Estimates of resource usage showed that other current device models such as the BSIM4 and the PSP models are feasible for FPGA implementation. The FPGA accelerator is expected to provide larger speedup results by taking advantage of more powerful FPGAs. The platform is built using Xilinx development boards, which will make it much cheaper than third-party specialised hardware.

In the experimental validation part, a manual transformation flow to translate the high-level device model code to a synthesisable code was embarked due to the large complexity and the long time scale needed to develop a dedicated compiler. However, the manual flow showed that an automatic compiler system that performs this operation is feasible. The transformation flow reduces the FPGA resource usage by fixing the model parameters beforehand. This is based on the assumption that in most cases only one device model is used to model all the transistors in a circuit. The model parameters can be changed to target a different transistor model.

The third part of this thesis focused mainly on the case where the device model is large and cannot be fitted into a single FPGA. The synthesis results of the full CMOS LEVEL 3 model showed that it could not be mapped on the Virtex-II Pro FPGA. This model can be partitioned to a number of sub-designs which can be mapped on a multi-FPGA system. This approach dictates that inter-FPGA signals must be minimised as FPGAs are limited in terms of their available I/O pins. Hence, a multi-FPGA synthesis system is designed specifically to explore the problem of minimising the inter-FPGA connections. An effective optimisation technique has been presented that can be used to reduce the pin utilisation and hence the inter-FPGA connections of the Mesh topology by optimising the process of mapping the logical partitions to their corresponding FPGAs. This technique uses simple partition swapping moves to bring FPGAs with heavy inter-communication together. The technique has been tested on an architecture based a on the Mesh model and the Virtual Wires model. The technique achieved an average improvement of about 18% for a number of benchmark DFGs which include the CMOS LEVEL 3 model. The results produced were very close to the calculated optimum solution.

To conclude, the work presented in this thesis aims to contribute towards permitting the EDA community to speed up the design/verification using modern hardware platforms. The proposed system can be used as an acceleration coprocessor that can be used by the SPICE simulator community to speedup transistor-level simulations. In this thesis we demonstrated that it is worthwhile continuing this research direction further to explore the use of FPGAs to accelerate other state-of-art EDA tools.

7.2 Summary of Research Contributions

This thesis serves as a proof of concept to evaluate and quantify the cost of using multi-FPGA systems in SPICE-like simulations in terms of area, power, acceleration, and throughput. A code transformation flow which converts the high-level model code to structural VHDL was also implemented. This showed that an automatic compiler system to design, map, and optimise the SPICE-like simulation on FPGAs is feasible.

This thesis has two main contributions. The first contribution is the multi-FPGA accelerator of the device model evaluation which demonstrated a 10 times speedup over conventional processors. The second contribution lies in the use of multi-FPGA synthesis to optimise the inter-FPGA connections through altering the process of mapping partitions to FPGA devices.

An expanded list of research contributions is as follows:

- **Parallel Device Model Evaluation on Multi-FPGAs:** Our architecture exploited the medium-grained parallelism, by executing the device model evaluation on a number of FPGAs simultaneously. The pipeline and instructions parallelism levels are also exploited to maximise the FPGA acceleration. We demonstrated a speedup of up to 10 times over a C software implementation parallelised using the OpenMP library, and running on an Intel 2.0 GHz Due core 2 processor with 2.5 GB of RAM. The system consumed six times less power than the Intel processor used for comparison. The prototype system used the SPICE CMOS LEVEL 3 model [40] in the evaluation process. The system is built using off-the-shelf Xilinx development boards, which will make the system much cheaper than third-party specialised multi-FPGA hardware.

- **Multi-FPGA Partition Mapping Optimisation:** In the multi-FPGA synthesis domain, a novel optimisation technique was introduced to reduce the inter-FPGA connections and hence the pin utilisation of the Mesh-based Multi-FPGA architecture. The technique achieved an average improvement of about 18%. It has been tested on an architecture based on a classical Mesh model and the Virtual Wires model. It was also shown that our technique achieved semilar results for the CMOS LEVEL 3 device model used earlier. One of the main advantages of our optimisation technique is that it does not require any extra hardware circuitry like the ones needed in the Virtual Wires. Furthermore, it can be applied after the partitioning process regardless of abstraction level is was performed at. This technique can be integrated into current multi-FPGA EDA tools at the optimisation stage to reduce the FPGA pin usage.

7.3 Future Work

7.3.1 Multi-FPGA SPICE Accelerator

Alongside this project, the realisation of a parallel multi-FPGA system to accelerate part of the linear solve phase using the LU decomposition was carried out by Tarek Nechma. The system performs the linear solve in parallel using a Virtex-5 FPGA [165]. Acceleration figures of about 10–30 times are reported compared to a 2.4 GHz Intel Core Duo processor running the state-of-the-art sparse matrix solver UMFPACK.

A future work would be to combine both phases implemented (device evaluation and linear solve) to form a complete multi-FPGA SPICE simulator as seen in Figure 3.9 (Section 3.4.3). However, a number of challenges have to be overcome before the realisation of the complete system. The main problem would be the pre-processing task of the linear solve. In addition to all the other parts of the SPICE algorithm such as the transient analysis, DC operating point determination, and error truncation that cannot be easily parallelised or mapped on reconfigurable hardware. Also, investigation has to be carried out to find the trade-offs between the performance improvement achieved by the multi-FPGA system over the conventional simulators and the computational error due to the FP precision selection. Another challenge to investigate is to find efficient ways to incorporate the hardware accelerators into the current circuit design flow.

7.3.2 Multi-FPGA Model Evaluation Acceleration

More FPGAs are integrated into HPC systems CPU sockets or designed to reside on the same board and connected to the same standard bus like the HyperTransport in the XtremeData Module [105]. The FPGA will have direct access to the main memory and the host CPU. Hence, FPGAs can be used to accelerate SPICE as coprocessors to perform the device evaluation and linear solve on parallel while the host CPU performs the other non-parallelisable tasks. Smaller and cheaper FPGAs can also be used together in parallel to provide better performance, which would be the same as using cheaper/less performant processors in current Multi-Core systems. GPUs are also very interesting accelerators which have recently showed large speedup figures for different scientific applications including SPICE, especially after the realisation of high precision floating-point arithmetic.

Close coupling of computing devices (FPGAs, CPUs and GPUs) would reduce the communication cost and allow faster access to memory. The built-in BRAM blocks in FPGAs can be used to improve the modularity of the memory system, as data can easily be mapped from external memory system. Also, the BRAM can be addressed in parallel which allows the data to be sent/received simultaneously to/from the computing nodes. This would enable more EDA tools to be accelerated by taking advantage of such systems.

Therefore, it is possible to build heterogeneous computing systems using conventional/embedded processors and FPGA engines to accelerate both phases in SPICE. Figure 7.1 shows an example HPRC configuration which uses systems from XtremeData and Nallatech. Large multi-FPGA systems like DINI systems and the BEE3 (Section 2.3.1) can also be used by incorporating embedded processors as shown in Figure 7.2.

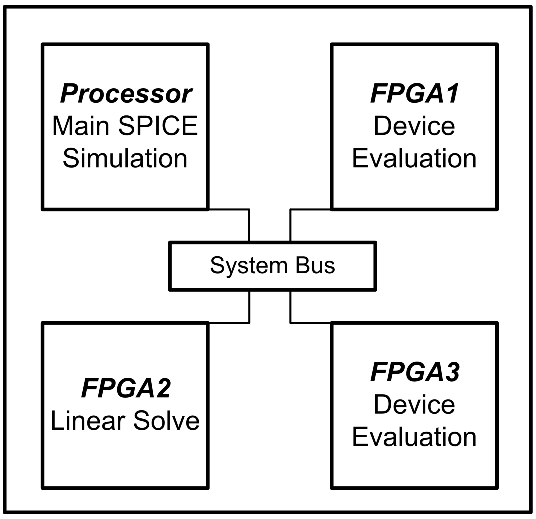


Figure 7.1: Multi-FPGA system with conventional processors like in XtremeData systems

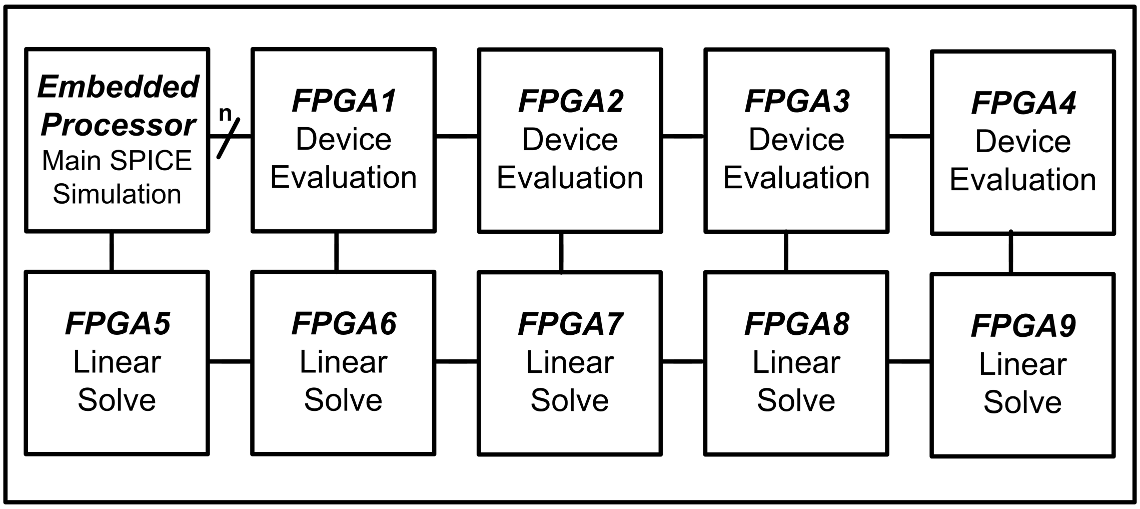


Figure 7.2: A suggested multi-FPGA system to accelerate SPICE

The multi-FPGA synthesis system can also take advantage from the reducing size of device models when parameters are fixed as seen in Section 5.1. The framework could allow the user to select the specific device models that will be used and set their possible parameters. The device models should be already built in the framework. The framework then generates the FPGA synthesisable code to implemented the required models, which will then reside on the slave FPGAs as demonstrated in Figure 7.3.

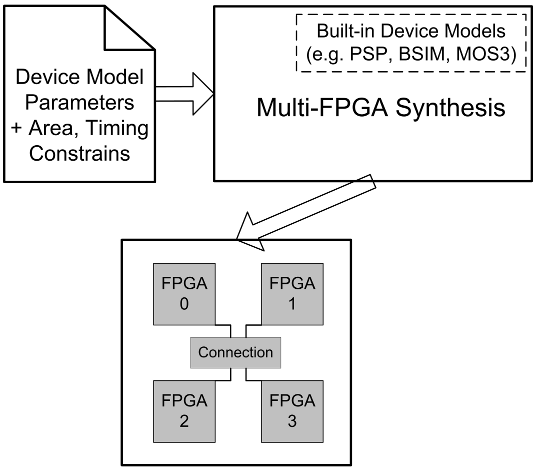


Figure 7.3: Multi-FPGA framework to synthesise built-in device models

7.3.3 Multi-FPGA Iterative Linear Solve

Direct linear solvers are usually more robust than Iterative solvers in solving sparse linear systems found in SPICE. However, they tend to have higher memory requirements, dynamic data structures, and longer execution times due to fill-ins appearing during factorisation. These factors effect the scalability of the direct liner solvers to parallel environments and to hardware platforms such as FPGAs. Pivoting and reordering techniques are introduced to overcome these issues. For FPGAs, static pivoting and pre-ordering is used to determine the positions of fill-ins.

Iterative methods, on the other hand, are less robust than the direct methods, but they tend to have small and constant memory requirements and can have significantly less execution times when convergence is reached within few iterations [210]. Although convergence is an issue for iterative methods, but it can improved by employing the appropriate pre-conditioners [211]. Therefore, iterative methods are attractive for their low memory usage, which make them suitable for FPGA implementations. In addition, a number of linear algebra kernels are recently ported to FPGAs such as LAPACKrc which is a Fast Linear Algebra Kernels/Solvers for FPGAs [190] and sparse/dense matrix-vector multiplier in [212, 213]. These advances encouraged the Reconfigurable Computing community to easily port their time-consuming applications to hardware.

A future extension to our work is to investigate how iterative linear methods -used to solve sparse matrices- can be realised on reconfigurable devices like FPGAs by exploiting the inherent parallelism. An example iterative method that can be studied is the GMRES method (Generalized Minimal Residual Method) [133, 211]. The investigation would demonstrate the suitability of the GMRES algorithm for use on FPGAs. Further research can be conducted to analyse the different design trade-offs and decisions involved in creating a Parallel Iterative Linear Solver using multi-FPGA Systems.

References

[1] TOP500 Supercomputer Sites. TOP500 project. see link: `http://www.top500.org/`, 2006.

[2] Michael Garland. Sparse Matrix Computations on Manycore GPUs. In *Computer Arithmetic, 2003. Proceedings. 16th IEEE Symposium on*, pages 2–6, June 2008.

[3] NVIDIA. NVIDIA Tesla C870 GPU Computing Processor Board. `http://www.nvidia.com/docs/IO/43395/C870-BoardSpec_BD-03399-001_v04.pdf`, July 2009.

[4] Tarek El-Ghazawi, Dave Bennett, Dan Poznanovic, Allan Cantle, Keith Underwood, Rob Pennington, Duncan Buell, Alan George, and Volodymyr Kindratenko. Is High-Performance Reconfigurable Computing the Next Supercomputing Paradigm? In *ACM/IEEE conference on Supercomputing*, page 71, New York, USA, 2006.

[5] Duncan Buell, Tarek El-Ghazawi, Kris Gaj, and Volodymyr Kindratenko. High-Performance Reconfigurable Computing. *IEEE Computer Society, Computer Magazine*, 40(3):23–27, 2007.

[6] C. Edwards. The Soft Machines. *Electronics Systems and Software*, 5(1):28–33, Feb.-March 2007.

[7] Ronald Scrofano. *Accelerating Scientific Computing Applications with Reconfigurable Hardware.* PhD thesis, University of Southern California, Los Angeles, CA, USA, 2006. Adviser-Viktor K. Prasanna.

[8] Bryce Mackin and Nathan Woods. FPGA Acceleration in HPC: A Case Study in Financial Analytics. Technical report, XtremeData, 2006.

[9] G.L. Zhang, P.H.W. Leong, C.H. Ho, K.H. Tsoi, C.C.C. Cheung, D.-U. Lee, R.C.C. Cheung, and W. Luk. Reconfigurable Acceleration for Monte Carlo Based Financial Simulation. In *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pages 215–222, Dec. 2005.

[10] Ian Kuon and Jonathan Rose. Measuring the Gap between FPGAs and ASICs. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 21–30, New York, NY, USA, 2006. ACM.

[11] Volodymyr Kindratenko and David Pointer. A Case Study In Porting A Production Scientific Supercomputing Application To A Reconfigurable Computer. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 13–22, Washington, DC, USA, 2006. IEEE Computer Society.

[12] H. Krupnova. Mapping Multi-Million Gate SoCs on FPGAs: Industrial Methodology and Experience. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume Vol.2, pages 1236 – 41, Paris, France, 2004.

[13] Maya Gokhale and Paul S. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays.* Springer Verlag, 2005.

[14] C. Edwards. Game on for Acceleration. *Engineering & Technology*, 3(11):36–38, 21 June 2008.

[15] Black Fischer and Myron Scholes. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy*, 81 (3):637–654, 1973.

[16] Sachin Tandon. A Programmable Architecture for Real-time Derivative Trading. Master's thesis, School of Informatics, University of Edinburgh, 2003.

[17] Peiheng Zhang, Guangming Tan, and Guang R. Gao. Implementation of The Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform. In *The 1st International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pages 39–48, New York, USA, 2007.

[18] Walid A. Najjar and Jason R. Villarreal. Reconfigurable Computing in the New Age of Parallelism. In Koen Bertels, Nikitas J. Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *SAMOS*, volume 5657 of *Lecture Notes in Computer Science*, pages 255–262. Springer, 2009.

[19] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs Journal*, 30(3):202–210, 2005. `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[20] S. Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4):23–29, Jul-Aug 1999.

[21] Tim Mattson and Michael Wrinn. Parallel programming: can we PLEASE get it right this time? In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 7–11, New York, NY, USA, 2008. ACM.

[22] Shekhar Borkar. Thousand Core Chips: A Technology Perspective. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 746–749, New York, NY, USA, 2007. ACM.

[23] Satnam Singh. Integrating FPGAs in High-Performance Computing: Programming Models for Parallel Systems – The Programmer's Perspective. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, pages 133–135, New York, NY, USA, 2007. ACM.

[24] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[25] Keith Underwood. FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 171–180, New York, NY, USA, 2004. ACM.

[26] Ian Kuon, Russell Tessier, and Jonathan Rose. FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electronic Design Automation*, 2:135–253, February 2008.

[27] Xilinx. *Virtex-6 Family FPGAs*, July 2009. `http://www.xilinx.com/products/virtex6/`.

[28] P.F. Cox, R.G. Burch, D.E. Hocevar, P. Yang, and B.D. Epler. Direct Circuit Simulation Algorithms for Parallel Processing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(6):714–725, Jun 1991.

[29] Gene Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings*, 30:483–485, 1967.

[30] Tutorial reports. FPGA Logic Blocks. see link: `http://www.tutorial-reports.com/computer-science/fpga/logic-block.php`, 2007.

[31] Wayne Luk. Customising Processors: Design-Time and Run-Time Opportunities. In *Embedded Computer Systems: Architectures, MOdeling, and Simulation*, volume 3133 of *Lecture Notes in Computer Science*, pages 49–58. Springer, 2004.

[32] Kanupriya Gulati and Sunil P Khatri. *Hardware Acceleration of EDA Algorithms: Custom ICs, FPGAs and GPUs.* Springer, 2010.

[33] Vinoo Natt Srinivasan. *Partitioning for FPGA-based Reconfigurable Computers.* PhD thesis, Departement of Electrical and Computer Engineering and Computer Science, University of Cincinnati, Cincinnati, OH, USA, 1999. Adviser-Ranga R. Vemuri.

[34] Esam El-Araby, Ivan Gonzalez, and Tarek El-Ghazawi. Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(4):1–23, January 2009.

[35] Karthikeya M. Gajjala Purna and Dinesh Bhatia. Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. *IEEE Trans. Computing*, 48(6):579–590, 1999.

[36] Sriram Govindarajan. *Algorithms for Design Space Exploration and High-Level Synthesis for Multi-FPGA Reconfigurable Computers.* PhD thesis, University of Cincinnati, Engineering : Computer Science and Engineering, 2000.

[37] Laurence W Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits.* University of California, Berkeley, May 1975.

[38] Laurence W. Nagel. Is it Time for SPICE4? In *Numerical Aspects of Device and Circuit Modeling Workshop, June 23–25, 2004, Santa Fe, New Mexico*, 2004.

[39] David Pescovitz. 1972: The release of SPICE, still the industry standard tool for integrated circuit design. `http://coe.berkeley.edu/labnotes/0502/history.html`, May 2002.

[40] Andrei Vladimirescu. *The SPICE Book.* John Wiley & Sons, Inc., New York, NY, USA, 1994.

[41] B.S. Deepaksubramanyan, P. Parakh, Zhenhua Chen, H. Diab, D. Marcy, and F.H. Schlereth. An FPGA-Based MOS Circuit Simulator. In *Proc. 48th Midwest Symposium on Circuits and Systems*, pages 655–658 Vol. 1, 2005.

[42] Kanupriya Gulati, John F. Croix, Sunil P. Khatr, and Rahm Shastry. Fast Circuit Simulation on Graphics Processing Units. In *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 403–408, Piscataway, NJ, USA, 2009. IEEE Press.

[43] N. Kapre and A. DeHon. Accelerating SPICE Model-Evaluation using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines, 09*, pages 37–44, Washington, DC, USA, April 2009. IEEE Computer Society.

[44] R.A. Saleh, K.A. Gallivan, M.-C. Chang, I.N. Hajj, D. Smart, and T.N. Trick. Parallel Circuit Simulation On Supercomputers. *Proceedings of the IEEE*, 77(12):1915–1931, Dec 1989.

[45] A. Vladimirescu, D. Weiss, M. Katevenis, Z. Bronstein, A. Kifir, K. Danuwidjaja, K. C. Ng., N. Jain, and S. Lass. A Vector Hardware Accelerator with Circuit Simulation Emphasis. In *DAC '87: Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 89–94, New York, NY, USA, 1987. ACM.

[46] P. Sadayappan and V. Visvanathan. Circuit Simulation on Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 37(12):1634–1642, 1988.

[47] A. Maache, J. Reeve, and M. Zwolinski. Accelerating CMOS Device Model Evaluation Using Multi-FPGA Systems. In *Fifth UK Embedded Forum 2009, Leicester, UK*, pages 10–14, September 2009.

[48] A. Maache, J. Reeve, and M. Zwolinski. Optimising Physical Wires Usage in Mesh-based Multi-FPGA Systems using Partition Swapping. In *21st International Conference on Microelectronics, ICM09, Marrakech, Morocco*, pages 246–249, 19-22 December 2009.

[49] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Higher Education, 1994.

[50] Daniel D. Gajski and Loganath Ramachandran. Introduction to High-Level Synthesis. *IEEE Design and Test*, 11(4):44–54, 1994.

[51] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, Steve Y.-L. Lin, and Steve Y.-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[52] Tack Boon Yee. *Synthesis of Multi-FPGA Systems with Asynchronous Communications.* PhD thesis, Electronics and Computer Science School, University of Southampton, 2007.

[53] Synopsys. Synphony High-Level Synthesis. `http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/default.aspx`, November 2010.

[54] Mentor Graphics. Catapult C Synthesis. `http://www.mentor.com/esl/catapult/overview`, November 2010.

[55] Arash Ahmadi. High Level Synthesis of Signal Processors. MPhil Thesis, School of Electronics and Computer Science, University of Southampton, 2005.

[56] Youn-Long Lin. Recent Developments in High-Level Synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 2(1):2–21, 1997.

[57] P.G. Paulin and J.P. Knight. Force-directed Scheduling for the Behavioral Synthesis of ASICs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(6):661–679, 1989.

[58] Jiahn-Hung Lee, Yu-Chin Hsu, and Youn-Long Lin. A New Integer Linear Programming Formulation For The Scheduling Problem In Data Path Synthesis. In *Computer-Aided Design, ICCAD-89, Digest of Technical Papers, IEEE International Conference on*, pages 20–23, Nov 1989.

[59] Oliver Bringmann, Wolfgang Rosenstiel, and Carsten Menn. Target Architecture Oriented High-Level Synthesis for Multi-FPGA Based Emulation. In *DATE '00.*, pages 326 – 32, Paris, France, 2000.

[60] Tack Boon Yee, M. Zwolinski, and A.D. Brown. Multi-FPGA Synthesis with Asynchronous Communication Subsystems. In *IFIP International Conference on Very Large Scale Integration*, pages 139 – 45, Perth, WA, Australia, 2005.

[61] Charles J. Alpert and Andrew B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration, the VLSI Journal*, 19(1-2):1–81, 1995.

[62] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(1):291–307, 1970.

[63] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *DAC '82: Proceedings of the 19th conference on Design Automation*, pages 175–181. IEEE Press, 1982.

[64] Sao-Jie Chen and Chung-Kuan Cheng. Tutorial on VLSI partitioning. *VLSI Design*, 11(3):175 – 218, 2000.

[65] Frank Vahid. I/O and Performance Tradeoffs with the FunctionBus During Multi-FPGA Partitioning. In *ACM fifth Int. Symposium on FPGAs*, pages 27–34, New York, USA, 1997.

[66] Frank Vahid, Thuy Dm Le, and Yu-Chin Hsu. Functional Partitioning Improvements over Structural Partitioning for Packaging Constraints and Synthesis: Tool Performance. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 3(2):181–208, April 1998.

[67] V. Srinivasan, S. Govindarajan, and R. Vemuri. Fine-grained and Coarse-grained Behavioral Partitioning with Effective Utilization of Memory and Design Space Exploration for Multi-FPGA Architectures. *IEEE Trans. on VLSI Systems*, 9(1):140 – 58, 2001.

[68] Wen-Jong Fang and Allen C.-H. Wu. Multiway FPGA Partitioning by Fully Exploiting Design Hierarchy. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(1):34–50, 2000.

[69] Auspy Development Inc. Auspy Partition System II. http://www.auspy.com, May 2008.

[70] Synopsys. Multi-FPGA Implementation and Partitioning. `http://www.synopsys.com/Tools/Verification/HardwareAssistedVerification/Confirma/Pages/Certify.aspx`, May 2010.

[71] Andrew A. Duncan, David C. Hendry, and Peter Gray. An Overview of the COBRA-ABS High Level Synthesis System for Multi-FPGA Systems. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 106, Washington, DC, USA, 1998. IEEE Computer Society.

[72] Andrew A. Duncan, David C. Hendry, and Peter Gray. The COBRA-ABS High-Level Synthesis System for Multi-FPGA Custom Computing Machines. *IEEE Trans on VLSI Systems*, 9(1):218–223, 2001.

[73] Wen-Jong Fang and Allen C.-H. Wu. Integrating HDL Synthesis and Partitioning for Multi-FPGA Designs. *IEEE Design and Test of Computers*, 15(2):65–72, 1998.

[74] K. Harbich and E. Barke. PuMA++: from Behavioral Specification to Multi-FPGA-Prototype. In *The 11th Int. Conf. on Field-Programmable Logic and Applications*, pages 133 – 41, Belfast, Northern Ireland, UK, 2001.

[75] J.A. Fisher. The VLIW Machine: A Multiprocessor for Compiling Scientific Code. *Computer*, 17(7):45–53, July 1984.

[76] Preetham Lakshmikanthan, Sriram Govindarajan, Vinoo Srinivasan, and Ranga Vemuri. Behavioral Partitioning with Synthesis for Multi-FPGA Architectures under Interconnect, Area, and Latency Constraints. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 924–931, London, UK, 2000. Springer-Verlag.

[77] S. Govindarajan, V. Srinivasan, P. Lakshmikanthan, and R. Vemuri. A Technique for Dynamic High-Level Exploration During Behavioral-Partitioning for Multi-Device Architectures. In *VLSI Design, 2000. Thirteenth International Conference on*, pages 212–219, 2000.

[78] Jason Lee, Lesley Shannon, Matthew J. Yedlin, and Gary F. Margrave. A Multi-FPGA Application-Specific Architecture for Accelerating a Floating Point Fourier Integral Operator. In *ASAP '08: Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors*, pages 197–202, Washington, DC, USA, 2008. IEEE Computer Society.

[79] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A Quantitative Analysis of The Speedup Factors of FPGAs over Processors. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 162–170, New York, NY, USA, 2004. ACM.

[80] Xizhen Xu, Sotirios G. Ziavras, and Tae-Gyu Chang. An FPGA-Based Parallel Accelerator for Matrix Multiplications in the Newton-Raphson Method. In *Embedded and Ubiquitous Computing, International Conference EUC 2005, Nagasaki, Japan, December 6-9, 2005. Proceedings*, pages 458–468, 2005.

[81] K.D. Underwood and K.S. Hemmert. Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 219–228, April 2004.

[82] Greg Stitt, Frank Vahid, and Shawn Nematbakhsh. Energy Savings and Speedups From Partitioning Critical Software Loops to Hardware in Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 3(1):218–232, 2004.

[83] Rob Baxter, Stephen Booth, Mark Bull, Geoff Cawood, Kenton D'Mellow, Xu Guo, Mark Parsons, James Perry, Alan Simpson, and Arthur Trew. High-Performance Reconfigurable Computing - the View from Edinburgh. In *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 273–279, Aug. 2007.

[84] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable Computing: Architectures and Design Methods. *Computers and Digital Techniques, IEE Proceedings -*, 152(2):193–207, Mar 2005.

[85] K. Bondalapati and V.K. Prasanna. Reconfigurable Computing Systems. *Proceedings of the IEEE*, 90(7):1201–1217, Jul 2002.

[86] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computuer Surveys*, 34(2):171–210, 2002.

[87] Russell Tessier and Wayne Burleson. Reconfigurable Computing for Digital Signal Processing: A Survey. *J. VLSI Signal Process. Syst.*, 28(1/2):7–27, 2001.

[88] Joao M. P. Cardoso and Mario P. Vestistias. Architectures And Compilers To Support Reconfigurable Computing. *Crossroads*, 5(3):15–22, 1999.

[89] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh. A Quick Safari Through The Reconfiguration Jungle. In *Design Automation Conference, 2001. Proceedings*, pages 172–177, 2001.

[90] T. Ramdas and G. Egan. A Survey of FPGAs for Acceleration of High Performance Computing and their Application to Computational Molecular Biology. In *TENCON 2005 2005 IEEE Region 10*, pages 1–6, Nov. 2005.

[91] S. Arash Ostadzadeh and Koen Bertels. Parallelism Utilization in Embedded Reconfigurable Computing Systems: A Survey of Recent Trends. *The Journal of VLSI Signal Processing*, 28(1-2):7–27, 2004. Springer.

[92] C. Chang, J. Wawrzynek, and R.W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *Design & Test of Computers, IEEE*, 22(2):114–125, March-April 2005.

[93] Dan Burke, John Wawrzynek, Krste Asanovic, Alex Krasnov, Andrew Schultz, Greg Gibeling, and Pierre-Yves Droz. RAMP Blue: Implementation of a Manycore 1008 Processor System. In *Proceedings of the Reconfigurable Systems Summer Institute RSSI 2008*, 2008.

[94] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz. RAMP Blue: A Message-Passing Manycore System in FPGAs. In *Proc. International Conference on Field Programmable Logic and Applications FPL 2007*, pages 54–61, 27–29 Aug. 2007.

[95] RAMP. Research Accelerator for Multiple Processors. see link: `http://ramp.eecs.berkeley.edu/index.php?index`, March 2009.

[96] M. Saldana and P. Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *Proc. International Conference on Field Programmable Logic and Applications FPL '06*, pages 1–6, 28–30 Aug. 2006.

[97] Manuel Saldana, Daniel Nunes, Emanuel Ramalho, and Paul Chow. Configuration and Programming of Heterogeneous Multiprocessors on a Multi-FPGA System Using TMD-MPI. In *Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference on*, pages 1–10, Sept. 2006.

[98] Manuel Alejandro Saldana De Fuentes. A Parallel Programming Model for A Multi-FPGA Multiprocessor Machine. Master's thesis, University of Toronto, 2006.

[99] Arun Patel, Christopher A. Madill, Manuel Saldana, Christopher Comis, Regis Pomes, and Paul Chow. A Scalable FPGA-based Multiprocessor. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 111–120, Washington, DC, USA, 2006. IEEE Computer Society.

[100] M. Saldana, A. Patel, C. Madill, D. Nunes, Danyao Wang, H. Styles, A. Putnam, R. Wittig, and P. Chow. MPI as an Abstraction for Software-Hardware Interaction for HPRCs. In *Proc. Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications HPRCTA 2008*, pages 1–10, 16–16 Nov. 2008.

[101] M. Saldana, E. Ramalho, and P. Chow. A Message-Passing Hardware/Software Co-simulation Environment to Aid in Reconfigurable Computing Design Using TMD-MPI. In *Proc. International Conference on Reconfigurable Computing and FPGAs ReConFig '08*, pages 265–270, 3–5 Dec. 2008.

[102] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest. Maxwell - a 64 FPGA Supercomputer. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pages 287–294, Aug. 2007.

[103] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest. The FPGA High-Performance Computing Alliance Parallel Toolkit. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pages 301–310, 2007.

[104] Nathan Woods. Integrating FPGAs in High-Performance Computing: the Architecture and Implementation Perspective. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 132–132, New York, NY, USA, 2007. ACM.

[105] XtremeData. XD1000 Accelerator Module. `http://www.xtremedatainc.com/index.php?option=com_content&view=article&id=89&Itemid=140`, June 2007.

[106] Nallatech. FPGA Accelerated Computing Solutions. `http://www.nallatech.com/?node_id=1.2&request=2008update&family=1&details=true`, June 2009.

[107] DINIGroup. Big FPGA Boards: High Performance Computing. `http://www.dinigroup.com/`, April 2010.

[108] Richard Wain, Ian Bush, Martyn Guest, Miles Deegan, Igor Kozin, and Christine Kitchen. An Overview of FPGAs and FPGA Programming: Initial Experiences at Daresbury. Technical report, CCLRC Daresbury Laboratory, 2006.

[109] SystemC Community. Synthesisable Subset Document. see link: `http://www.systemc.org/`, December 2006.

[110] Agility Design Solutions. *Handel-C Language Reference Manual*, 2007. `http://www.agilityds.com/literature/HandelC_Language_Reference_Manual.pdf`.

[111] Nallatech. DIME-C, C-to-VHDL Compiler. `http://www.nallatech.com/index.php/Development-Tools/dime-c.html`, July 2009.

[112] G. Genest, R. Chamberlain, and R. Bruce. Programming an FPGA-based Super Computer Using a C-to-VHDL Compiler: DIME-C. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pages 280–286, Aug. 2007.

[113] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits.* Kluwer Academic, 2004.

[114] J. L. Tripp, K. D. Peterson, C. Ahrens, J. D. Poznanovic, and M. B. Gokhale. TRIDENT: An FPGA Compiler Framework for Floating-point Algorithms. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 317–322, 2005.

[115] J.L. Tripp, M.B. Gokhale, and K.D. Peterson. TRIDENT: From High-Level Language to Hardware Circuitry. *Computer*, 40(3):28–37, March 2007.

[116] David Andrews, Douglas Niehaus, Razali Jidin, Michael Finley, Wesley Peck, Michael Frisbie, Jorge Ortiz, Ed Komp, and Peter Ashenden. Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link. *IEEE Micro*, 24(4):42–53, 2004.

[117] Stephen A. Edwards. The Challenges of Synthesizing Hardware from C-Like Languages. *IEEE Design & Test*, 23(5):375–386, 2006.

[118] Jidan Al-Eryani. FPU. see link: `http://www.opencores.org/projects.cgi/web/fpu100/overview`, 2007.

[119] J. Detrey and F. de Dinechin. A VHDL Library of Parametrisable Floating-Point and LNS Operators for FPGA. `http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/`, Dec 2006.

[120] Jérémie Detrey and Florent de Dinechin. Parameterized Floating-Point Logarithm and Exponential Functions For FPGAs. *Microprocessors & Microsystems*, 31(8):537–545, 2007.

[121] Pavle Belanovic and Miriam Leeser. A Library of Parameterized Floating-Point Modules and Their Use. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 657–666, London, UK, 2002. Springer-Verlag.

[122] Jian Liang, R. Tessier, and O. Mencer. Floating Point Unit Generation and Evaluation for FPGAs. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 185–194, April 2003.

[123] Xilinx. Floating-Point Operator. `http://www.xilinx.com/products/ipcenter/floating_pt.htm`, March 2009.

[124] Xilinx. *Aurora Quick Start Guide XUP Virtex-2 Pro Development System.* Xilinx, Inc., 2008.

[125] Xilinx. *Xilinx University Program Virtex-II Pro Development System Hardware Reference Manual*, April 2008.

[126] Mark Zwolinski. Multi-Threaded Circuit Simulation Using OpenMP. Unpublished paper, Dec 2008.

[127] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 2 edition, October 1992.

[128] R. Raghuram. *Computer Simulation of Electronic Circuits.* Halsted Press, New York, NY, USA, 1988.

[129] V. Litovski and M. Zwolinski. *VLSI Circuit Simulation and Optimization.* Chapman and Hall, 1997.

[130] Junqing Sun, G.D. Peterson, and O.O. Storaasli. High-Performance Mixed-Precision Linear Solver for FPGAs. *Computers, IEEE Transactions on*, 57(12):1614–1623, Dec. 2008.

[131] Seth Young, Arvind Sudarsanam, Aravind Dasu, and Thomas Hauser. Memory Support Design for LU Decomposition on the Starbridge Hyper-Computer. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 157–164, Dec. 2006.

[132] Michael T. Heath, Esmond Ng, and Barry W. Peyton. Parallel Algorithms for Sparse Linear Systems. *SIAM Review*, 33(3):420–460, 1991.

[133] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[134] Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating Scientific Computations with Mixed Precision Algorithms. *Computer Physics Communications*, 180(12):2526 – 2533, 2009.

[135] E. Lelarasmee, A.E. Ruehli, and A.L. Sangiovanni-Vincentelli. The Waveform Relaxation Method for Time-Domain Analysis of Large Scale Integrated Circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1(3):131–145, July 1982.

[136] A. Lumsdaine, M.W. Reichelt, J.M. Squyres, and J.K. White. Accelerated Waveform Methods for Parallel Transient Simulation of Semiconductor Devices. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(7):716–726, Jul 1996.

[137] He Peng and Chung-Kuan Cheng. Parallel Transistor Level Circuit Simulation using Domain Decomposition Methods. In *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pages 397–402, Jan. 2009.

[138] D.M. Lewis. A Compiled-Code Hardware Accelerator for Circuit Simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 11(5):555–565, May 1992.

[139] U. Wever and Q. Zheng. Parallel Transient Analysis for Circuit Simulation. In *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on ,*, volume 1, pages 442–447 vol.1, Jan 1996.

[140] Wei Dong, Peng Li, and Xiaoji Ye. WavePipe: Parallel Transient Simulation Of Analog And Digital Circuits On Multi-Core Shared-Memory Machines. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 238–243, New York, NY, USA, 2008. ACM.

[141] Xiaoji Ye, Wei Dong, Peng Li, and S. Nassif. MAPS: Multi-Algorithm Parallel Circuit Simulation. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 73–78, Nov. 2008.

[142] H. Kotakemori, H. Hasegawa, and A. Nishida. Performance Evaluation of a Parallel Iterative Method Library using OpenMP. In *High-Performance Computing in Asia-Pacific Region, 2005. Proceedings. Eighth International Conference on*, pages 432–436, July 2005.

[143] S. Markus, S.B. Kim, K. Pantazopoulos, A.L. Ocken, E.N. Houstis, P. Wu, S. Weerawarana, and D. Maharry. Performance Evaluation of MPI Implementations and MPI Based Parallel ELLPACK Solvers. In *MPI Developer's Conference, 1996. Proceedings., Second*, pages 162–169, Jul 1996.

[144] P.M. Lee, S. Ito, T. Hashimoto, J. Sato, T. Touma, and G. Yokomizo. A Parallel and Accelerated Circuit Simulator with Precise Accuracy. In *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings.*, pages 213–218, 2002.

[145] BSIM3 MOSFET SPICE model. `http://www-device.eecs.berkeley.edu/~bsim3/`, March 2009.

[146] Lawrence Pillage. *Electronic Circuit & System Simulation Methods (SRE).* McGraw-Hill, Inc., New York, NY, USA, 1999.

[147] R.E. Poore. GPU-Accelerated Time-Domain Circuit Simulation. In *Custom Integrated Circuits Conference, 2009. CICC '09. IEEE*, pages 629–632, Sept. 2009.

[148] B. Murmann, P. Nikaeen, D.J. Connelly, and R.W. Dutton. Impact of Scaling on Analog Performance and Associated Modeling Needs. *Electron Devices, IEEE Transactions on*, 53(9):2160–2167, Sept. 2006.

[149] M. Chan and C. Hu. The Engineering of BSIM for the Nano-Technology Era and Beyond. In *Modeling and Simulation Microsistem*, pages 662–665, 2002.

[150] P. Agrawal, S. Goil, S. Liu, and J.A. Trotter. Parallel Model Evaluation for Circuit Simulation on the PACE Multiprocessor. In *VLSI Design, 1994., Proceedings of the Seventh International Conference on*, pages 45–48, Jan 1994.

[151] Gung-Chung Yang. PARASPICE: A Parallel Circuit Simulator for Shared-Memory Multiprocessors. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 400–405, Jun 1990.

[152] S. Hutchinson, E. Keiter, R Hoekstra, H. Watts, A. Waters, T. Russo, R. Schells, S. Wix, and C. Bogdan. The Xyce Parallel Electronic Simulator - An Overview. In *Parallel Computing, Advances and Current Issues. Proceedings of the International Conference, ParCo2001, Naples, Italy*, pages 165–172, London, September 2001. Imperial College Press.

[153] Xyce: Parallel Electronic Simulator. `http://xyce.sandia.gov/`, July 2009.

[154] Tien-Hsiung Weng, Ruey-Kuen Perng, and Barbara Chapman. OpenMP Implementation of SPICE3 Circuit Simulator. *Int. J. Parallel Program.*, 35(5):493–505, 2007.

[155] N. Kapre and A. DeHon. Performance Comparison of Single-Precision SPICE Model-Evaluation on FPGA, GPU, Cell, and Multi-Core Processors. In *International Conference on Field Programmable Logic and Applications, 09*, pages 65–27, September 2009.

[156] B. Cope, P.Y.K. Cheung, W. Luk, and S. Witt. Have GPUs made FPGAs redundant in the field of video processing? In *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pages 111–118, Dec. 2005.

[157] Shuai Che, Jie Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107, June 2008.

[158] Zhe Fan, Feng Qiu, A. Kaufman, and S. Yoakum-Stover. GPU Cluster for High Performance Computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 47–47, Nov. 2004.

[159] David Barrie Thomas, Lee Howes, and Wayne Luk. A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72, New York, NY, USA, 2009. ACM.

[160] Amr M. Bayoumi and Yasser Y. Hanafy. Massive Parallelization of SPICE Device Model Evaluation on GPU-Based SIMD Architectures. In *IFMT'08: Proceedings Of The 1st International Forum on Next-generation Multicore/Manycore Technologies*, pages 1–5, New York, NY, USA, 2008. ACM.

[161] Nascentric. OmegaSim GX Hardware-Accelerated SPICE Simulator. see link: `http://www.nascentric.com/omegasim_gx.html`, October 2008.

[162] Marcus van Ierssel. Circuit Simulation on a Field Programmable Accelerator. Master's thesis, University of Toronto, 1995.

[163] D.M. Lewis, M.H. van Ierssel, and D.H. Wong. A Field Programmable Accelerator for Compiled-Code Applications. In *Computer Design: VLSI in Computers and Processors, 1993. ICCD '93. Proceedings., 1993 IEEE International Conference on*, pages 491–496, Oct 1993.

[164] N. Kapre and A. DeHon. Parallelizing Sparse Matrix Solve for SPICE Circuit Simulation using FPGAs. In *IEEE International Conference on Field-Programmable Technology, FPT09*, pages 190–198, December 2009.

[165] Nechma Tarek, Zwolinski Mark, and Reeve Jeff. Parallel Sparse Matrix Solver for Direct Circuit Simulations on FPGAs. In *IEEE International Symposium on Circuits and Systems (ISCAS10)*, June 2010.

[166] J. Johnson, T. Chagnon, P. Vachranukunkiet, P. Nagvajara, and C. Nwankpa. Sparse LU Decomposition using FPGAs. In *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008.

[167] Xiaofang Wang and Sotirios G. Ziavras. Parallel LU Factorization of Sparse Matrices on FPGA-based Configurable Computing Engines. *Concurrency and Computation: Practice & Experience*, 16(4):319–343, 2004.

[168] H. Ziegler, Byoungro So, M. Hall, and P.C. Diniz. Coarse-Grain Pipelining on Multiple FPGA Architectures. In *FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 77 – 86, Napa, CA, USA, 2002.

[169] K G Nichols, T J Kazmierski, A D Brown, and M Zwolinski. Overview of SPICE-like Circuit Simulation Algorithms. *IEE Proc. Circuits, Devices and Systems*, 141(4):242–250, 1994.

[170] SIMUCAD. *ModelLib User's Manual*. SIMUCAD Design Automation, Inc., April 2010. `https://dynamic.silvaco.com/dynamicweb/jsp/downloads/EntryAction.do?action=silen-menu&key=20004&format=4`.

[171] Mark Zwolinski. Southampton VHDL-AMS Validation Suite. `http://www.syssim.ecs.soton.ac.uk/`, April 2010.

[172] Mark Zwolinski. VHDL-AMS Model of LEVEL 3 MOS Transistor. `http://www.syssim.ecs.soton.ac.uk/vhdl-ams/examples/mos.htm`, June 2005.

[173] P. Šůcha, M. Kutil, M. Sojka, and Z. Hanzálek. TORSCHE Scheduling Toolbox for Matlab. In *IEEE Computer Aided Control Systems Design Symposium (CACSD'06)*, pages 1181–1186, Munich, Germany, October 2006.

[174] Xilinx. *ChipScope Pro Tool User Guide*, April 2009. `http://www.xilinx.com/tools/cspro.htm`.

[175] Xilinx. MicroBlaze Processor. `http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm`, March 2009.

[176] Xilinx. Aurora Link-layer Protocol. `http://www.xilinx.com/products/design_resources/conn_central/grouping/aurora.htm`, March 2009.

[177] Intel. Core 2 Duo E6300 specifications. `http://ark.intel.com/product.aspx?id=27248`, July 2009.

[178] Florent de Dinechin, Jérémie Detrey, Octavian Cret, and Radu Tudoran. When FPGAs are better at floating-point than microprocessors. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 260–260, New York, NY, USA, 2008. ACM.

[179] John D. Davis, Charles P. Thacker, and Chen Chang. BEE3: Revitalizing Computer Architecture Research. Technical Report MSR-TR-2009-45, Microsoft Research, Silicon Valley Campus, April 2009.

[180] BSIM4 MOSFET SPICE model. `http://www-device.eecs.berkeley.edu/~bsim3/bsim4.html`, March 2009.

[181] G. Gildenblat, X. Li, H. Wang, W. Wu, R.Van Langevelde, A.J. Scholten, G.D.J. Smit, and D.B.M. Klaassen. Introduction to PSP MOSFET Model. In *2005 Workshop on Compact Modeling*, pages 19 – 24. Pennsylvania State University, US, 2005.

[182] SIMUCAD. Open Source Verilog-A Models. `https://dynamic.simucad.com/dynamicweb/jsp/downloads/EntryAction.do?action=smcen-menu&key=95&format=16`, July 2009.

[183] PSP Verilog-A code. `http://pspmodel.asu.edu/psp_code.htm`, July 2009.

[184] Xilinx. *Virtex-5 Family FPGAs*, July 2009. `www.xilinx.com/support/documentation/data_sheets/ds100.pdf`.

[185] R.J. McDonald. Convergence in SPICE for Advanced IC Device Modelling. In *Southeastcon '88., IEEE Conference Proceedings*, pages 349–352, Apr 1988.

[186] DA Zein. Solution of a Set of Nonlinear Algebraic Equations for General Purpose CAD Programs. *IEEE Circ. and Dev*, 1(5):7–20, 1985.

[187] Ari Kulmala, Erno Salminen, and Timo D. Hämäläinen. Evaluating Large System-on-Chip on Multi-FPGA Platform. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 7th International Workshop, SAMOS 07, Greece, Proceedings*, pages 179–189, July 16-19 2007.

[188] Abdellah-Medjadji Kouadri-Mostefaoui, Benaoumeur Senouci, and Frederic Petrot. Large Scale On-Chip Networks: An Accurate Multi-FPGA Emulation Platform. In *DSD '08: Proceedings of the 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 3–9, Washington, DC, USA, 2008. IEEE Computer Society.

[189] P.G. Del valle, D. Atienza, I. Magan, J.G. Flores, E.A. Perez, J.M. Mendias, L. Benini, and G.D. Micheli. A Complete Multi-Processor System-on-Chip FPGA-Based Emulation Framework. In *Very Large Scale Integration, 2006 IFIP International Conference on*, pages 140–145, Oct. 2006.

[190] Juan Gonzalez and Rafael C Nez. LAPACKrc: Fast Linear Algebra Kernels/Solvers for FPGA Accelerators. *Journal of Physics: Conference Series*, 180(1):012042, 2009.

[191] Jong-Ho Byun, A. Ravindran, A. Mukherjee, B. Joshi, and D. Chassin. Accelerating the Gauss-Seidel Power Flow Solver on a High Performance Reconfigurable Computer. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pages 227 –230, april 2009.

[192] S. Hauck and G. Borriello. An Evaluation of Bipartitioning Techniques. In *ARVLSI '95: Proceedings of the 16th Conference on Advanced Research in VLSI (ARVLSI'95)*, page 383, Washington, DC, USA, 1995. IEEE Computer Society.

[193] Raghu Burra and Dinesh Bhatia. Timing Driven Multi-FPGA Board Partitioning. In *VLSID '98: Proceedings of the Eleventh International Conference on VLSI Design: VLSI for Signal Processing*, page 234, Washington, DC, USA, 1998. IEEE Computer Society.

[194] Juan de Vicente, Juan Lanchares, and Romn Hermida. Placement Optimization Based on Global Routing Updating for System Partitioning onto Multi-FPGA Mesh Topologies. In *Field Programmable Logic and Applications*, volume 1673/2004 of *Lecture Notes in Computer Science*, pages 91–101. Springer Berlin / Heidelberg, 1999.

[195] Scott Hauck, Gaetano Borriello, and Carl Ebeling. Mesh Routing Topologies for Multi-FPGA Systems. In *ICCS '94: Proceedings of the1994 IEEE International Conference on Computer Design: VLSI in Computer & Processors*, pages 170–177, Washington, DC, USA, 1994. IEEE Computer Society.

[196] Sushil Chandra Jain, Shashi Kumar, and Anshul Kumar. Evaluation of Various Routing Architectures for Multi-FPGA Boards. In *VLSID '00: Proceedings of the 13th International Conference on VLSI Design*, page 262, Washington, DC, USA, 2000.

[197] Mohammed A. S. Khalid and Jonathan Rose. A Novel And Efficient Routing Architecture For Multi-FPGA Systems. *IEEE Trans. Very Large Scale Integr. Syst.*, 8(1):30–39, 2000.

[198] M. Dörfel and R. Hofmann. A Prototyping System for High Performance Communication Systems. In *RSP '98: Proceedings of the Ninth IEEE International Workshop on Rapid System Prototyping*, page 84, Washington, DC, USA, 1998. IEEE Computer Society.

[199] F. Vahid. Techniques for Minimizing and Balancing I/O During Functional Partitioning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(1):69–75, Jan 1999.

[200] J. Babb, R. Tessier, M. Dahl, S.Z. Hanono, D.M. Hoki, and A. Agarwal. Logic Emulation with Virtual Wires. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(6):609–626, Jun 1997.

[201] Vahid, Le, and Hsu. A Comparison of Functional and Structural Partitioning. In *ISSS '96: Proceedings of the 9th international symposium on System synthesis*, page 121, Washington, DC, USA, 1996. IEEE Computer Society.

[202] Young-Su Kwon and Chong-Min Kyung. ATOMi: An Algorithm for Circuit Partitioning Into Multiple FPGAs Using Time-Multiplexed, Off-Chip, Multicasting Interconnection Architecture. *IEEE Trans. on VLSI Systems*, 13(7):861–4, 2005.

[203] Young-Su Kwon and Chong-Min Kyung. Performance-Driven Event-Based Synchronization for Multi-FPGA Simulation Accelerator with Event Time-Multiplexing Bus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1444 – 56, 2005.

[204] Young-Su Kwon and Chong-Min Kyung. Scheduling driven circuit partitioning algorithm for multiple FPGAs using time-multiplexed, off-chip, multi-casting interconnection architecture. *Microprocessors & Microsystems*, 28(5-6):341–350, 2004.

[205] M. Inagi, Y. Takashima, Y. Nakamura, and A. Takahashi. ILP-Based Optimization of Time-Multiplexed I/O Assignment for Multi-FPGA Systems. In *ISCAS 2008.*, pages 1800–1803, May 2008.

[206] Masato Inagi, Yasuhiro Takashima, Yuichi Nakamura, and Yoji Kajitani. A Performance-Driven Circuit Bipartitioning Method Considering Time-Multiplexed I/Os. *IEICE Trans Fundamentals*, E90-A(5):924–931, May 2007.

[207] George Karypis and Vipin Kumar. METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System. Technical report, Department of Computer Science and Engineering, University of Minnesota, 1995.

[208] George Karypis. METIS - Family of Multilevel Partitioning Algorithms. http://glaros.dtc.umn.edu/gkhome/views/metis/, March 2008.

[209] Richard Johnsonbaugh and Martin Kalin. A Graph Generation Software Package. *SIGCSE Bull.*, 23(1):151–154, 1991.

[210] W. M. Zuberek and T. D. P. Perera. Performance Analysis of Distributed Iterative Linear Solvers. In *MMACTE'05: Proceedings of the 7th WSEAS International Conference on Mathematical Methods and Computational Techniques In Electrical Engineering*, pages 194–199, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS).

[211] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial Mathematics, 2003.

[212] G. Kuzmanov and M. Taouil. Reconfigurable Sparse/Dense Matrix-Vector Multiplier. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 483 –488, dec. 2009.

[213] Ling Zhuo and Viktor K. Prasanna. High-Performance Designs for Linear Algebra Operations on Reconfigurable Hardware. *IEEE Trans. Computers*, 57(8):1057–1071, 2008.

[214] Microsoft. QueryPerformanceCounter Function. `http://msdn.microsoft.com/en-us/library/ms644904(VS.85)29.aspx`, May 2009.

[215] IEEE. *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std 754*, 1985.

[216] IEEE. *IEEE Standard for Radix-Independent Floating-Point Arithmetic, IEEE Std 854*, 1987.

[217] Zaher Abdulkarim Baidas. *High Level Floating-Point Synthesis.* PhD thesis, Electronics and Computer Science, University of Southampton, 2000.

[218] Michael F. Cowlishaw. Decimal Floating-Point: Algorism for Computers. In *IEEE Symposium on Computer Arithmetic*, pages 104–111, 2003.

[219] Gokul Govindu, Ronald Scrofano, and Viktor K. Prasanna. A Library of Parameterizable Floating-Point Cores for FPGAs and their Application to Scientific Computing. In *In Proc. of International Conference on Engineering Reconfigurable Systems and Algorithms*, pages 137–148, 2005.

[220] IEEE. *Standard VHDL Reference Manual, IEEE Std 1076*, 1993.

[221] David Bishop. *Fixed-Point Package User Guide*, 2006. see link: `http://www.vhdl.org/vhdl-200x/vhdl-200x-ft/packages/Fixed_ug.pdf`.

[222] David Bishop. *Floating-Point Package User Guide*, 2006. see link: `http://www.vhdl.org/vhdl-200x/vhdl-200x-ft/packages/float_ug.pdf`.

[223] J. Detrey and F. de Dinechin. *FPLibrary v0.91 User Documentation*, Dec 2006. `https://lipforge.ens-lyon.fr/docman/view.php/12/1/fplib_doc.pdf`.

Appendix A

CMOS LEVEL 3 VHDL-AMS Model

A.1 Simulation Model VHDL-AMS Code

The CMOS LEVEL 3 device model from the Southampton VHDL-AMS Verification Suite used in the evaluation process is shown in Listing A.1 [172].

```
library IEEE;

use IEEE.math_real.all;
use IEEE.electrical_systems.all;

entity mos is
      generic(
    -- instance parameters
    width   : real:=1.0E-4;  -- should be global constant DEFW!
    length  : real:=1.0E-4;  -- DEFL
    channel: real :=1.0;       -- +1 for NMOS, -1  for PMOS
    -- model parameters
    vt0     : real:= real'low;
    kp      : real:= 2.0E-5;
    gamma   : real:= 0.0;
    phi     : real:= 0.6;
    tox     : real:= 1.0E-7;
    nsub    : real:= 0.0;
    nss     : real:= 0.0;
    nfs     : real:= 0.0;
    tpg     : real:= 1.0;
    xj      : real:= 0.0;
    ld      : real:= 0.0;
    u0      : real:= 600.0;
    vmax    : real:= 0.0;
    xqc     : real:= 1.0;
    kf      : real:= 0.0;
    af      : real:= 1.0;
    fc      : real:= 0.5;
    delta   : real:= 0.0;
    theta   : real:= 0.0;
    eta     : real:= 0.0;
    kappa   : real:= 0.2;
    ngate   : real:= 1.5e19;
    -- environment parameters
    Temperature : real :=300.0  -- Should be global );
    port(terminal drain, gate, source, bulk : electrical);
end entity ;

architecture mos3 of mos is

  quantity MOSquantities: real_vector(0 to 3);
  quantity Vdsq across drain to source;
  quantity Vgsq across gate to source;
  quantity Vbsq across bulk to source;
  quantity Idq through drain;
  quantity Igq through gate;
  quantity Isq through source;
  quantity Ibq through bulk;
  constant eps0 : real :=8.85418e-12;
  constant Ni : real :=1.45e16;
  constant Boltzmann : real :=1.380662e-23;
  constant echarge: real :=1.6021892e-19;
  constant epsSiO2 : real :=3.9*eps0;
  constant epsSi : real :=11.7*eps0;
  constant pi: real := 3.14159;

  Function Max(x,y: real) return real is
    variable z: real;
  begin
          if x>=y then
                z:=x;
          else
                z:=y;
          end if;
          return z;
  end function Max;

  Function Min(x,y: real) return real is
    variable z: real;
  begin
          if x<=y then
                z:=x;
          else
                z:=y;
          end if;
          return z;
  end function Min;

  Function MOSequations(vdsq,vgsq,vbsq,width,length,channel,vt0,kp,gamma,
                phi,tox,nsub,nss,nfs,tpg,xj,ld,u0,vmax,xqc,kf,af,fc,delta,theta,
                eta,kappa,ngate,temperature: real) return real_vector is

     variable Qc,Qb,Qg: real;
     variable cox,beta,vt,sigma,nsub_in,Phi_in,Gamma_in,nss_in,ngate_in,A,B,C,D,Vfb,fshort,
                  wp,wc,sqwpxj,vbulk,delv,vth,Vgstos, Vgst, eff, Tau, Vsat,Vpp,fdrain,egfet,
                  fermig,mobdeg,stfct,leff,xd,qnfscox,fn,dcrit,deltal,It,Ids,R,Vds,Vgs,Vbs,
                  forward,kTQ : real;
     variable results: real_vector(0 to 3);
  begin
      kTQ :=Boltzmann*temperature/echarge;

      if tox<=0.0 then
        cox:=epsSiO2/(1.0e-7);
      else
        cox:=epsSiO2/tox;
      end if;

      if kp = 0.0 then
        beta:=cox*u0;
      else
        beta:=kp;
      end if;

      nsub_in:= nsub * 1.0e6;  -- scale nsub to SI units

      if (phi = real'low) then
        if (nsub_in > 0.0) then
          Phi_in:=max(0.1,2.0*kTQ*log(nsub_in/Ni));
        else
          Phi_in:=0.6;
        end if;
      else
        Phi_in:=phi;
      end if; -- model.phi = undefined

      if (gamma = real'low) then
        if (nsub_in > 0.0) then
          Gamma_in:=sqrt(2.0*epsSi*echarge*nsub_in)/cox;
        else
          Gamma_in:=0.0;
        end if;
      else
        Gamma_in:=gamma;
      end if; -- gamma = undefined

      nss_in:=nss*1.0e4;                 -- Scale to SI
      ngate_in:=ngate*1.0e4;              -- Scale to SI

      if (vt0 = real'low) then
        egfet:=1.16-(7.02e-4*Temperature*Temperature)/(Temperature+1108.0);
        if tpg=0.0 then
          fermig:=0.05+egfet/2.0;
        else
          if ngate_in >0.0 then
            fermig:=tpg*channel*kTQ*log(ngate_in/Ni);
          else
            fermig:=tpg*channel*egfet/2.0;
          end if;
          vt:=-fermig+channel*(Phi_in*0.5+Gamma_in*sqrt(Phi_in))-nss_in*echarge/cox;
        end if;
      else
        vt:=vt0;
      end if; -- vt0 = undefined

      leff:=length-2.0*ld;

      if leff >0.0 then
        Sigma:=eta*8.15e-22/(cox*leff*leff*leff);
      else
        Sigma:=0.0;
      end if; -- leff>0

      if nsub_in >0.0 then -- N.B. nsub was scaled, above.
        xd:=sqrt(2.0*epsSi/(echarge*nsub_in));
      else
        xd:=0.0;
      end if; -- nsub >0

      if (nfs >0.0) and(cox >0.0) then
        qnfscox:=echarge*nfs/cox;
      else
        qnfscox:=0.0;
      end if; --nfs > 0

      if cox >0.0 then
        fn:=delta*pi*epsSi*0.5/(cox*width);
      else
        fn:=delta*pi*epsSi*0.5*tox/epsSiO2;
      end if;  -- cox > 0

      --Scale beta and convert cox from Fm^-2 to F
      beta:=beta*width/leff;
      cox:=cox*width*leff;

    Vds:=channel*Vdsq;

    if Vds>=0.0 then
        Vgs:=channel* Vgsq;
        Vbs:=channel* Vbsq;
        forward:=1.0;
    else
        Vds:=-Vds;
        Vgs:=channel* Vgsq;
        Vbs:=channel* Vbsq;
        forward:=-1.0;
    end if; -- Vds >=0

    if Vbs<=0.0 then
        A:=Phi_in-Vbs;
        D:=sqrt(A);
    else
        D:=2.0*sqrt(Phi_in)*Phi_in/(2.0*Phi_in+Vbs);
        A:=D*D;
    end if; -- Vbs <= 0

    Vfb:=Vt-Gamma_in*sqrt(Phi_in)-Sigma*Vds;

    if (xd=0.0) OR (xj=0.0) then
        fshort:=1.0;
    else
        wp:=xd*D;
        wc:=0.0631353*xj+0.8013292*wp-0.01110777*wp*wp/xj;
        sqwpxj:=sqrt(1.0-(wp*wp/((wp+xj)*(wp+xj))));
        fshort:=1.0-((ld+wc)*sqwpxj-ld)/leff;
    end if; -- xd or xj = 0

    vbulk:=Gamma_in*fshort*D+fn*A;

    if nfs=0.0 then
        delv:=0.0;
    else
        delv:=kTQ*(1.0+qnfscox+vbulk*0.5/A);
    end if; -- nfs = 0

    vth:=Vfb+vbulk;
    Vgstos:=Vgs-Vfb;
    Vgst:=max(Vgs-vth, delv);

    if (vgs>=vth) or (delv/=0.0) then
      if (Vbs<=0.0) or (Phi_in /= 0.0) then
          B:=0.5*Gamma/D+fn;
      else
          B:=fn;
      end if;
      mobdeg:=1.0/(1.0+theta*Vgst);
      if (vmax /=0.0) then
         Ueff:=u0*mobdeg;
          Tau:=Ueff/Leff*vmax;
      else
          Tau:=0.0;
      end if;

      Vsat:=Vgst/(1.0+B);
      Vsat:=Vsat*(1.0-0.5*Tau*Vsat);
      Vpp:=min(Vds,Vsat);
      fdrain:=1.0/(1.0+Tau*Vpp);

      if (Vgs<vth+delv) and (nfs>0.0) then
        stfct:=exp((Vgs-vth-delv)/delv);
      else
          stfct:=1.0;
      end if;

      if Vds>=Vsat then
        if (kappa>0.0) and (xd>0.0) then
          if vmax=0.0 then
              deltal:=sqrt(kappa*xd*xd*(Vds-Vsat));
          else
              dcrit:=(xd*xd*vmax*0.5)/(Ueff*(1.0-fdrain));
              deltal:=sqrt(kappa*xd*xd*(Vds-Vsat)+dcrit*dcrit)-dcrit;
          end if;
          if deltal<=0.5*Leff then
              C:=Leff/(Leff-deltal);
          else
              C:=4.0*deltal/Leff;
          end if;
        else --kappa=0.0 or xd=0.0
            C:=1.0;
        end if;
      else
          C:=1.0;
      end if;
      It:=Vgst-Vpp*(1.0+B)*0.5;
      Beta:=Beta*mobdeg;
      Ids:=Beta*Vpp*It*C*fdrain*stfct;
    else
        -- Cutoff
        Ids:=0.0;
    end if; -- vgs >= vth

    if Cox /= 0.0 then
        --Charges
      if Vgs<=vth then
        if Gamma_in /= 0.0 then
          if Vgstos < -A then
                Qg:=Cox*(Vgstos+A); -- Accumulation
          else
                Qg:=0.5*Gamma_in*Cox*(sqrt(4.0*(Vgstos+A)+sqrt(Gamma_in))-Gamma_in);
          end if ; -- vgstos <-A
        else-- Gamma = 0.0
            Qg:=0.0;
        end if; -- gamma /= 0
          Qb:=-Qg;
          Qc:=0.0;
      else
         -- depletion mode:
          R:=(1.0+B)*Vpp*Vpp/(12.0*It);
          Qg:=Cox*(Vgstos-Vpp*0.5+R);
          Qc:=-Cox*(Vgst+(1.0+B)*(R-Vpp*0.5));
          Qb:=-(Qc+Qg);
      end if; -- vgs<=vth
    else
        Qg:=0.0;
        Qc:=0.0;
        Qb:=0.0;
    end if; -- cox /= 0

    results(0):=channel*forward*Ids;
    results(1):=channel*xqc*Qc;
    results(2):=channel*Qg;
    results(3):=channel*Qb;

    return results;
    end function MOSequations;

begin
    -- equations for currents:
      MOSquantities == MOSequations(vdsq,vgsq,vbsq,width,length,channel,vt0,kp,gamma,
                    phi,tox,nsub,nss,nfs,tpg,xj,ld,u0,vmax,xqc,kf,af,fc,delta,theta,
                    eta,kappa,ngate,temperature);
      Idq == MOSquantities(0)+MOSquantities(1)'dot;
      Igq == MOSquantities(2)'dot;
      Ibq == MOSquantities(3)'dot;
      Isq == -Idq - Igq - Ibq;

end architecture mos3;

--        2. Testbench
library IEEE;
use IEEE.math_real.all;
use IEEE.electrical_systems.all;

entity test_mos is
end entity test_mos;

architecture test of test_mos is
        terminal d,g: electrical;
        alias ground is ELECTRICAL_REF;
begin
        vgs: entity v_constant generic map (level=>2.0) port map (pos=>g,neg=>ground);
        vds: entity v_pulse generic map (pulse=>5.0,tchange=>10sec) port map(pos=>d,neg=>ground
    );
        nmos: entity mos port map (drain=>d,gate=>g,source=>ground,bulk=>ground);
end architecture;

-- 3. Constant Voltage Source
library IEEE;
use IEEE.math_real.all;
use IEEE.electrical_systems.all;

entity v_constant is
        generic(level: voltage);
        port(terminal pos,neg:electrical);
end entity v_constant;

architecture ideal of v_constant is
        quantity v across i through pos to neg;
begin
        v == level;
end architecture;

--    4 . Pulse Voltage Source

library IEEE;
use IEEE.math_real.all;
use IEEE.electrical_systems.all;

entity v_pulse is
    generic(
            initial: real:= 0.0;
            pulse : real:= 5.0;
            tchange : time:= 10sec); -- initial to pulse [Sec]
    port(terminal pos,neg: electrical);
end entity v_pulse;

architecture behaviour of v_pulse is
    function time2real(tt : time) return real is
    begin
              return time'pos(tt) * 1.0e-15;
    end time2real;

    constant slope:real:=pulse/time2real(tchange);

    quantity v across i through pos to neg;
    -- Signal used in CreateEvent process below
    signal pulse_signal : real := initial;
begin

    v==pulse_signal'slew(slope);
    CreateEvent : process
    begin
              wait until domain = time_domain; -- Run process in Time Domain only
              pulse_signal <=pulse;
    end process CreateEvent;
end architecture behaviour;
```

Listing A.1: VHDL-AMS CMOS LEVEL 3 Model

A.2 Synthesisable Model VHDL Code

The synthesisable CMOS code, full implementation, only the simple implementation is shown in Listing A.2. The VHDL testbench is shown in Listing A.3. The simulation waveforms is shown in Appendix A.3.

```
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use std.textio.all;

library fplib;
use fplib.pkg_fplib.all;

library ieee_proposed;
use ieee_proposed.math_utility_pkg.all;
use ieee_proposed.float_pkg.all;

--use work.matrix;

entity cmos_syn_pipeline is

        generic(
          wE : positive := 8;
     wF : positive := 23 );  -- Should be global
          -- port(terminal drain, gate, source, bulk : electrical);
     port(vdsq_i, vgsq_i, vbsq_i :  in std_logic_vector(2+wE+wF downto 0);
               clk        : in std_logic;
                    start      : in std_logic;
                    done       : out std_logic;
                    result1_o  : out std_logic_vector(2+wE+wF downto 0);
                    result2_o  : out std_logic_vector(2+wE+wF downto 0);
                    result3_o  : out std_logic_vector(2+wE+wF downto 0);
                    result4_o  : out std_logic_vector(2+wE+wF downto 0);
                    reset      : in std_logic);--;
end entity ;

architecture rtl of cmos_syn_pipeline is
          type vector is array(integer range <>) of std_logic_vector(2+wE+wF downto 0);

          signal vdsq, vgsq, vbsq :  std_logic_vector(2+wE+wF downto 0);
          signal  result1 : std_logic_vector(2+wE+wF downto 0);
          signal  result2 : std_logic_vector(2+wE+wF downto 0);
          signal  result3 : std_logic_vector(2+wE+wF downto 0);
          signal  result4 : std_logic_vector(2+wE+wF downto 0);

          constant one : std_logic_vector(2+wE+wF downto 0) := "01" & to_slv(to_float(1.0, wE,
     wF)); -- "0100111111100000000000000000000000";
          constant half : std_logic_vector(2+wE+wF downto 0) := "01" & to_slv(to_float(0.5, wE,
     wF)); -- "0100111111000000000000000000000000";
          constant two : std_logic_vector(2+wE+wF downto 0) := "01" & to_slv(to_float(2.0, wE,
     wF)); -- "0100111111000000000000000000000000";
          constant zero : std_logic_vector(2+wE+wF downto 0) := "00" & to_slv(to_float(0.0, wE,
     wF)); -- "0000000000000000000000000000000000";
          constant four : std_logic_vector(2+wE+wF downto 0) := "01" & to_slv(to_float(4.0, wE,
     wF));
          signal real_low   : std_logic_vector(2+wE+wF downto 0) := "10" & to_slv(to_float(real'
     low, wE, wF));        --    11111111100000000000000000000000"; --  - Inf
          signal real_high  : std_logic_vector(2+wE+wF downto 0) := "10" & to_slv(to_float(real'
     high, wE, wF));  -- 01111111100000000000000000000000"; --     + Inf
          constant fp : positive := wE + wF;

          signal t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13, t14, t14_1, t14_2, t15,
      t16, t17, t17_1, t18, t19, t20, t21, t22, t23, t24, t25, t26, t27, t27_1, t28, t29, t30,
     t30_1, t31, t32, t33, t34, t35, t36, t37, t38, t39, t40, t41, t42, t43, t44, t45, t46, t47
     , t48, t49, t50, t51, t52, t53, t54, t55, t56, t57, t58, t59, t60, t61, t62, t63, t64, t65
     , t66, t66_1, t67, t68, t69, t70, t71, t72, t73, t74, t75, t76, t77, t78, t79, t80, t81,
     t82, t83, t84, t85, t86, t87 , t88, t89, t90, t91, t92, t93, t94, t95, t96, t97 , t98, t99
     , t100, t101, t102, t103, t104, t105, t106, t107, t108, t109, t110, t111, t112, t113, t114
     , t115 : std_logic_vector(2+wE+wF downto 0) := (others => '0');

     signal t104_in, t86_in, Vpp_in, Vgstos_in, Vgst_in2, Qc, Qc_out,  Vds_in, Vgst_out,
      Vgst_in, ids_out, Qb, Qg_out, Qg, cox,new_beta, Vt,Sigma,nsub_in, sqrtPhi_in, Gamma_in,
     nss_in,ngate_in,A,B,C,D,fshort, vth_delv, xd_2, Vds_Vsat, wp,wc,sqwpxj,vbulk,delv,Vgstos,
     Vgst, Ueff, Tau, Vsat, Vsat0,Vpp,fdrain,egfet, egfet_2, fermig,mobdeg,stfct,Leff, Leff_2,
     leff_3,xd,qnfscox,fn,dcrit,deltal,It,Ids,R,Vds,Vgs,Vbs, vtTmp, echCox : std_logic_vector
     (2+wE+wF downto 0) := (others => '0');

          signal forward : std_logic := '0';

          function minus (i : std_logic_vector) return std_logic_vector is
                  variable res : std_logic_vector(2+wE+wF downto 0) := i;
          begin
                  if(i(wE+wF) = '0') then
                          res(wE+wF) := '1';
                  else
                          res(wE+wF) := '0';
                  end if;
          return res;
          end function;

          function f (i : std_logic_vector) return  float is
                  variable ex : std_logic_vector(1 downto 0);
                  variable sign : std_logic;
                  variable s : line;
          begin
                  ex := i(2+fp downto 1+fp);   -- The execption bits
                  sign := i(fp);  -- Sign bit
                  if ( ex = "00") then              -- Zero
                          return to_float(0.0, wE, wF);
                  elsif (ex = "10") then            -- inf
                          if(sign = '1') then
                                  return to_float(real'low, wE, wF);
                          else
                                  return to_float(real'high, wE, wF);
                          end if;
                  elsif (ex = "11") then            -- NaN
                          return qnanfp(wE, wF);
                  elsif (ex = "01") then            -- Normal Fp number
                          return to_float(i (fp downto 0), wE, wF);
                  else
                          return to_float(i (fp downto 0), wE, wF);                 -- To pass the
      X dont
                  end if;
          end function;

          constant beta : std_logic_vector(2+wE+wF downto 0) := "01" & to_slv(to_float(2.0e-5,
     wE, wF));
          constant Phi_in : std_logic_vector(2+wE+wF downto 0) := "01" & to_slv(to_float(0.6, wE
     , wF));
          constant Phi_in2 : std_logic_vector(2+wE+wF downto 0) := "01" & to_slv(to_float(1.2,
     wE, wF));
          constant phi_2_sqrt : std_logic_vector(2+wE+wF downto 0) := "01" & to_slv(to_float
     (0.92951600308978, wE, wF));
          constant Vfb : std_logic_vector(2+wE+wF downto 0) := "01" & to_slv(to_float
     (-0.117562084960685, wE, wF));
          constant Vth : std_logic_vector(2+wE+wF downto 0) := "01" & to_slv(to_float
     (-0.117562084960685, wE, wF));
          constant new_cox : std_logic_vector(2+wE+wF downto 0) :=  "01" & to_slv(to_float
     (3.4531e-012, wE, wF));
          constant minus_new_cox : std_logic_vector(2+wE+wF downto 0) :=  "01" & to_slv(to_float
     (-3.4531e-012, wE, wF));
          --constant cons9 : std_logic_vector(2+wE+wF downto 0) :=  "01" & to_slv(to_float
     (0.0833333333333333333333333333333333, wE, wF)); -- 1/12
          constant cons9 : std_logic_vector(2+wE+wF downto 0) :=  "01" & to_slv(to_float(12.0,
     wE, wF)); -- 1/12

          -- Real values:
          -- synthesis translate_off
          signal vds_r, vgs_r, vbs_r, Vgst_r, Vpp_r, It_r, Ids_r, R_r, Qg_r, Qb_r, Qc_r: real :=
      0.0;
          -- synthesis translate_on

          -- Registers to pipeline the operations in the CMOS model
          signal r1 : vector(1 to 3);               -- Vds
          signal r2 : vector(1 to 29);     -- Vgst
          signal r3 : vector(1 to 29);     -- Vgstos
          signal r4 : vector(1 to 22);     -- 0.5 * Vpp
          signal r5 : vector(1 to 25);     -- Ids
          signal r6 : vector(1 to 39);     -- Vgst to output
          signal r7 : vector(1 to 11);     -- Vpp to (beta*It*Vpp)
          signal r8 : vector(1 to 4);              -- Vgst to (Vgst*0.5*Vpp)
          signal r9 : vector(1 to 7);              -- Vgst to (Vgst*0.5*Vpp)
          signal r10 : vector(1 to 3);     -- Qg
          signal r11 : vector(1 to 3);     -- Qc

begin

          in_buff : process (clk, reset) --, start, vdsq_i, vgsq_i, vbsq_i, result1, result2,
     result3, result4)
                  variable timer : integer := 0;
          begin
                  if(reset = '1') then
                          vdsq <= zero;
                          vgsq <= zero;
                          vbsq <= zero;
                          result1_o <= zero;
                          result2_o <= zero;
                          result3_o <= zero;
                          result4_o <= zero;
                          done <=  '0';
          elsif(rising_edge(clk)) then
                  if (start = '1') then
                  vdsq <= vdsq_i;
                  vgsq <= vgsq_i;
                  vbsq <= vbsq_i;
                  if(timer > 51) then
                          result1_o <= result1;
                          result2_o <= result2;
                          result3_o <= result3;
                          result4_o <= result4;
                          done <= '1';
                  end if;
                  timer := timer + 1;
                  else
                  vdsq <= zero;
                  vgsq <= zero;
                  vbsq <= zero;
                  result1_o <= zero;
                  result2_o <= zero;
                  result3_o <= zero;
                  result4_o <= zero;
                  done <=  '0';
                  end if;
          end if;
end process;


Vgs <= Vgsq;
Vbs <= Vbsq;

fsm : process(reset, clk)
begin
          if(reset = '1') then

                  Vds <= zero;
                  forward <= '0';
                  Vgst <= zero;
                  Vpp <= zero;
                  It  <= zero;
                  Ids <= zero;
                  R <= zero;
                  Qg <= zero;
                  Qc <= zero;
                  Qb <= zero;
                  result1 <= zero;
                  result2 <= zero;
                  result3 <= zero;
                  result4 <= zero;

          elsif(rising_edge(clk)) then

                  if (f(Vdsq)>=f(zero)) then
                  Vds <= Vdsq;
                  forward <= '0';
                  else
                  Vds <= minus(Vdsq);
                  forward <= '1';
                  end if; -- Vds >=0

                  Vgst <= "01" & to_slv(maximum(f(Vgstos),f(zero)));

                  -- 230
                  if (f(vgs) >= f(vth)) then      -- depletion mode:
                  Vpp <= "01" & to_slv(minimum(f(Vds_in),f(Vgst)));
                  It  <= t88; -- ;Vgst-Vpp*(1.0+B)*0.5;
                  Ids <= t94;-- Beta*Vpp*It*C*fdrain*stfct;
                  R  <= t107; -- (1.0+B)*Vpp*Vpp/(12.0*It);
                  Qg <= t110; -- new_cox*(Vgstos-Vpp*0.5+R);
                  Qc <= t113; --  -new_cox*(Vgst+(1.0+B)*(R-Vpp*0.5));
                  Qb <= minus(t114);--  -(Qc+Qg);
                  else
                  -- Cutoff
                  Vpp <= zero; -- min(Vds, Vsat);
                  It  <= zero;
                  Ids <= zero;
                  R   <= zero;
                  Qg  <= zero;
                  Qc  <= zero;
                  Qb  <= zero; --minus(Qg);
                  end if; -- vgs >= vth

                  --if (f(Vgs)<=f(vth)) then
                  --else
                  --end if; -- vgs<=vth

                  if(forward = '0') then
                  result1 <= Ids_out;
                  else
                  result1 <= minus(Ids_out);
```

```
                        end if;

                        result2 <= Qc_out;
                        result3 <= Qg_out;
                        result4 <= Qb;

                end if;

        end process;

        registers : process(clk, reset)
        begin
                if(reset = '1') then
                        r1  <= ((others=> (others=>'0')));
                        r2  <= ((others=> (others=>'0')));
                        r3  <= ((others=> (others=>'0')));
                        r4  <= ((others=> (others=>'0')));
                        r5  <= ((others=> (others=>'0')));
                        r6  <= ((others=> (others=>'0')));
                        r7  <= ((others=> (others=>'0')));
                        r8  <= ((others=> (others=>'0')));
                        r9  <= ((others=> (others=>'0')));
                        r10 <= ((others=> (others=>'0')));
                        r11 <= ((others=> (others=>'0')));

                elsif(rising_edge(clk)) then
                        -- r1
                        Vds_in <= r1(1);
                        for i in 1 to (r1'high - 1) loop
                                r1(i) <= r1(i+1);
                        end loop;
                        r1(r1'high) <= Vds;

                        -- r2
                        Vgst_in <= r2(1);
                        for i in 1 to (r2'high - 1) loop
                                r2(i) <= r2(i+1);
                        end loop;
                        r2(r2'high) <= Vgst;

                        -- r3
                        Vgstos_in <= r3(1);
                        for i in 1 to (r3'high - 1) loop
                                r3(i) <= r3(i+1);
                        end loop;
                        r3(r3'high) <= Vgstos;

                        -- r4
                        t86_in <= r4(1);
                        for i in 1 to (r4'high - 1) loop
                                r4(i) <= r4(i+1);
                        end loop;
                        r4(r4'high) <= t86;  -- Vpp*0.5

                        -- r5
                        ids_out <= r5(1);
                        for i in 1 to (r5'high - 1) loop
                                r5(i) <= r5(i+1);
                        end loop;
                        r5(r5'high) <= Ids;

                        -- r6
                        Vgst_out <= r6(1);
                        for i in 1 to (r6'high - 1) loop
                                r6(i) <= r6(i+1);
                        end loop;
                        r6(r6'high) <= Vgst;

                        -- r7
                        Vpp_in <= r7(1);
                        for i in 1 to (r7'high - 1) loop
                                r7(i) <= r7(i+1);
                        end loop;
                        r7(r7'high) <= Vpp;

                        -- r8
                        Vgst_in2 <= r8(1);
                        for i in 1 to (r8'high - 1) loop
                                r8(i) <= r8(i+1);
                        end loop;
                        r8(r8'high) <= Vgst;

                        -- r9
                        t104_in <= r9(1);
                        for i in 1 to (r9'high - 1) loop
                                r9(i) <= r9(i+1);
                        end loop;
                        r9(r9'high) <= t104;  -- Vpp*Vpp
```

```
            -- r10
            Qg_out <= r10(1);
            for i in 1 to (r10'high - 1) loop
                r10(i) <= r10(i+1);
            end loop;
            r10(r10'high) <= Qg;

            -- r11
            Qc_out <= r11(1);
            for i in 1 to (r11'high - 1) loop
                r11(i) <= r11(i+1);
            end loop;
            r11(r11'high) <= Qc;  -- Vpp*Vpp
        end if;

    end process;

    add16 : entity fplib.fpadd_clk generic map ( wE, wF) port map ( Vgs, minus(Vfb),
  Vgstos, clk);
    -- 274
    mul54 : entity fplib.fpmul_clk generic map ( wE, wF) port map ( Vpp, half, t86, clk);
    add30 : entity fplib.fpadd_clk generic map ( wE, wF) port map ( Vgst_in2, minus(t86),
  t88, clk); -- It:=Vgst-Vpp*(1.0+B)*0.5;
    -- 276
    mul56 : entity fplib.fpmul_clk generic map ( wE, wF) port map ( beta, It, t89, clk);
    mul57 : entity fplib.fpmul_clk generic map ( wE, wF) port map ( t89, Vpp_in, t94, clk)
  ;
    -- -- 298
    mul66 : entity fplib.fpmul_clk generic map ( wE, wF) port map ( Vpp, Vpp, t104, clk);
   -- Vpp*Vpp
    mul68 : entity fplib.fpmul_clk generic map ( wE, wF) port map ( It, cons9, t105, clk
  );   -- It * 12
    div20 : entity fplib.fpdiv_clk generic map ( wE, wF) port map ( t104_in, t105, t107,
  clk);    -- Vpp*Vpp  / It*12
    -- --  299
    add34 : entity fplib.fpadd_clk generic map ( wE, wF) port map ( R, minus(t86_in), t108
  , clk); -- R-Vpp*0.5
    add35 : entity fplib.fpadd_clk generic map ( wE, wF) port map ( Vgstos_in, t108, t109,
   clk);
    mul69 : entity fplib.fpmul_clk generic map ( wE, wF) port map ( new_cox, t109, t110,
  clk);
    -- -- 300
    add36 : entity fplib.fpadd_clk generic map ( wE, wF) port map ( Vgst_in, t108, t112,
  clk);
    mul71 : entity fplib.fpmul_clk generic map ( wE, wF) port map ( minus_new_cox, t112,
  t113, clk);
    -- -- 301
    add37 : entity fplib.fpadd_clk generic map ( wE, wF) port map ( Qc, Qg, t114, clk);

    counter : process(clk, reset)
        variable c : integer := 0;
    begin
        if(reset = '1') then
            c := 0;
        elsif(clk'event and clk='1') then
            c := c + 1;
        end if;
    end process;

    -- synthesis translate_off

    vds_r   <= to_real(f(vds));
    vgs_r   <= to_real(f(vgs));
    vbs_r   <= to_real(f(vbs));
    Vgst_r  <= to_real(f(Vgst));
    Vpp_r   <= to_real(f(Vpp));
    It_r    <= to_real(f(It));
    Ids_r   <= to_real(f(Ids));
    R_r             <= to_real(f(R));
    Qg_r    <= to_real(f(Qg));
    Qb_r    <= to_real(f(Qb));
    Qc_r    <= to_real(f(Qc));

    -- synthesis translate_on

end architecture;
```

Listing A.2: Synthesisable VHDL CMOS LEVEL 3 Model

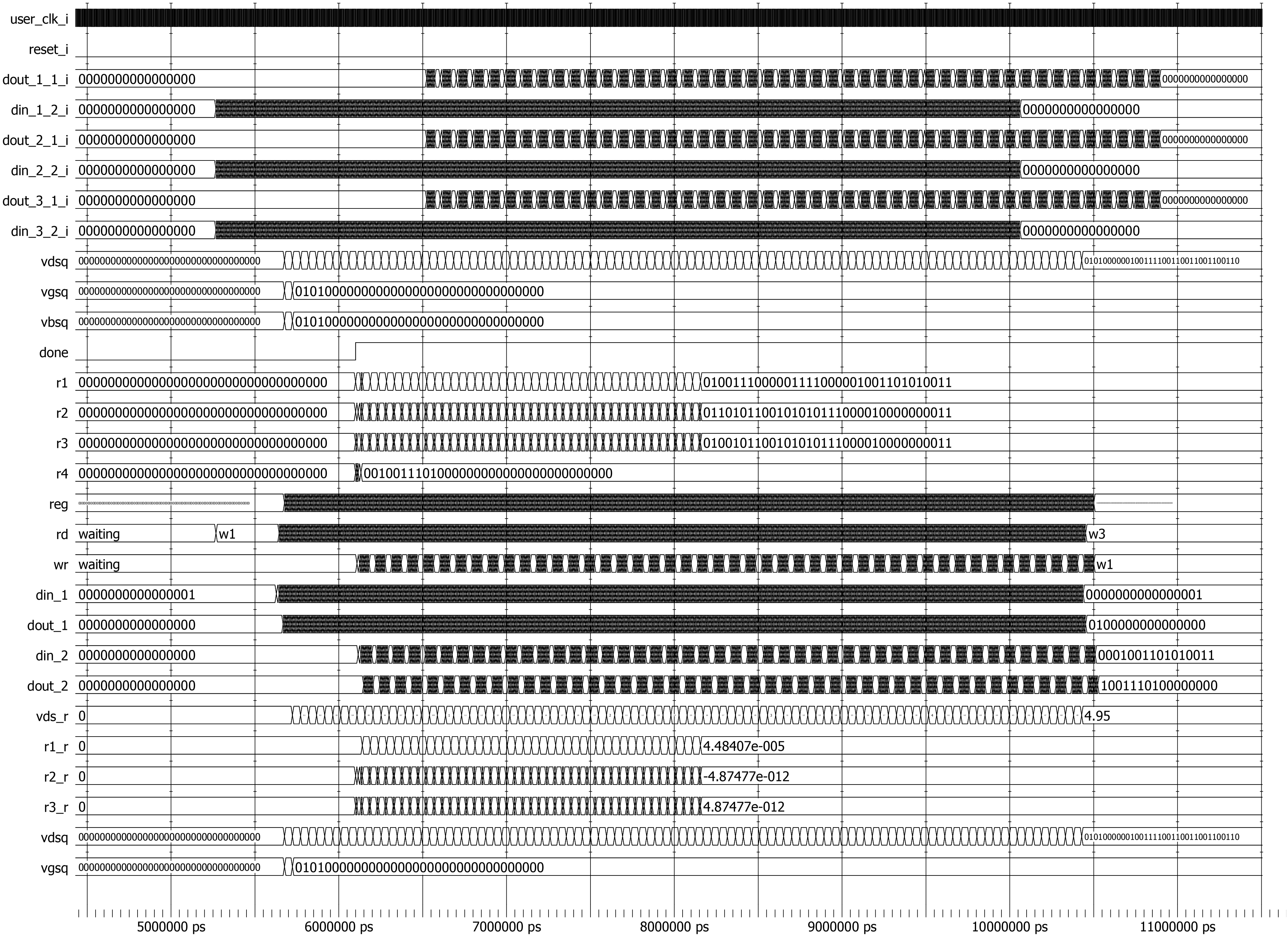
```
library ieee;

use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use ieee.math_real.all;
use std.textio.all;
```

```
library ieee_proposed;
use ieee_proposed.math_utility_pkg.all;
use ieee_proposed.float_pkg.all;

entity cmos_tb is
        --port(clk : in std_logic);
end entity cmos_tb;

architecture rtl of cmos_tb is
        -- signal Vd, N, T, I_s, res : std_logic_vector(2+wE+wF downto 0);
        signal clk, reset, res_rdy, start, done : std_logic := ’0’;
        constant wE : positive := 8;
        constant wF : positive := 23;
    signal r1_1, r1_2, r1_3, r1_4, r2_1, r2_2, r2_3, r2_4 : std_logic_vector(2+wE+wF downto 0)
    ;
        signal Id, Ig, Ib, Isq, Idq1, Igq1, Ibq1 : real;

        signal vdsq : std_logic_vector(2+wE+wF downto 0) := ”01” & to_slv(to_float(2.5, wE, wF
    ));
        signal vgsq : std_logic_vector(2+wE+wF downto 0) := ”01” & to_slv(to_float(1.5, wE, wF
    ));
        signal vbsq : std_logic_vector(2+wE+wF downto 0) := ”01” & to_slv(to_float(1.5, wE, wF
    ));

        constant one : std_logic_vector(2+wE+wF downto 0) := ”01” & to_slv(to_float(1.0, wE,
    wF));
        constant half : std_logic_vector(2+wE+wF downto 0) := ”01” & to_slv(to_float(0.5, wE,
    wF));
        constant two : std_logic_vector(2+wE+wF downto 0) := ”01” & to_slv(to_float(2.0, wE,
    wF));
        constant zero : std_logic_vector(2+wE+wF downto 0) := ”00” & to_slv(to_float(0.0, wE,
    wF));
        constant four : std_logic_vector(2+wE+wF downto 0) := ”01” & to_slv(to_float(4.0, wE,
    wF));
        signal real_low  : std_logic_vector(2+wE+wF downto 0) := ”10” & to_slv(to_float(real’
    low, wE, wF));        --  - Inf
        signal real_high : std_logic_vector(2+wE+wF downto 0) := ”10” & to_slv(to_float(real’
    high, wE, wF));  --     + Inf
        constant fp : positive := wE + wF;

        type vector is array(integer range <>) of std_logic_vector(2+wE+wF downto 0);
        signal voltages : vector (1 to 300);

begin
        clk <= not clk after 5 ns;
        reset <= ’1’, ’0’ after 10 ns;
        --start <= ’0’, ’1’ after 10 ns;

        --vdsq <= ”01” & to_slv(to_float(1.75, wE, wF));
        vgsq <= ”01” & to_slv(to_float(2.0, wE, wF));--e-5;
        vbsq <= ”01” & to_slv(to_float(2.0, wE, wF));--e-5

        Id  <= to_real(to_float(r2_1(wE + wF downto 0), wE, wF));
        Ig  <= to_real(to_float(r2_2(wE + wF downto 0), wE, wF));
        Ib  <= to_real(to_float(r2_3(wE + wF downto 0), wE, wF));
        Isq <= to_real(to_float(r2_4(wE + wF downto 0), wE, wF));

        mos_syn : entity work.cmos_syn_pipeline generic map (wE, wF) port map (vdsq, vgsq,
    vbsq, clk, start, done, r2_1, r2_2, r2_3, r2_4, reset);

        datain: process(clk)
                constant num  : real := 100.0;
                variable cycle, delay : integer := 0;
                variable vds : real := 0.0;
                variable vds_max : real := 5.0;
                variable inc : real := vds_max/num;
                variable re1, re2, re3, re4, re5, re6, re7, re8, id1, id2, error : real;
                variable s : line;
                variable i : integer := 1;

        begin
                if(reset = ’1’) then
                        vdsq <= zero;
                        start <= ’0’;
                        vds := inc;
                elsif(rising_edge(clk)) then
                        if(delay > 10) then
                        start <= ’1’;
                        --if(vds <= 5.0) then
                                --cycle := cycle + 1;
                                if(done = ’1’) then
                                        re1 := to_real(to_float(r2_1(wE + wF downto 0), wE, wF
    ));
                                        re2 := to_real(to_float(r2_2(wE + wF downto 0), wE, wF
    ));
                                        --error := re2 - Idq1;

                                        -- re3 <= to_real(to_float(r3(wE + wF downto 0), wE,
    wF));
```

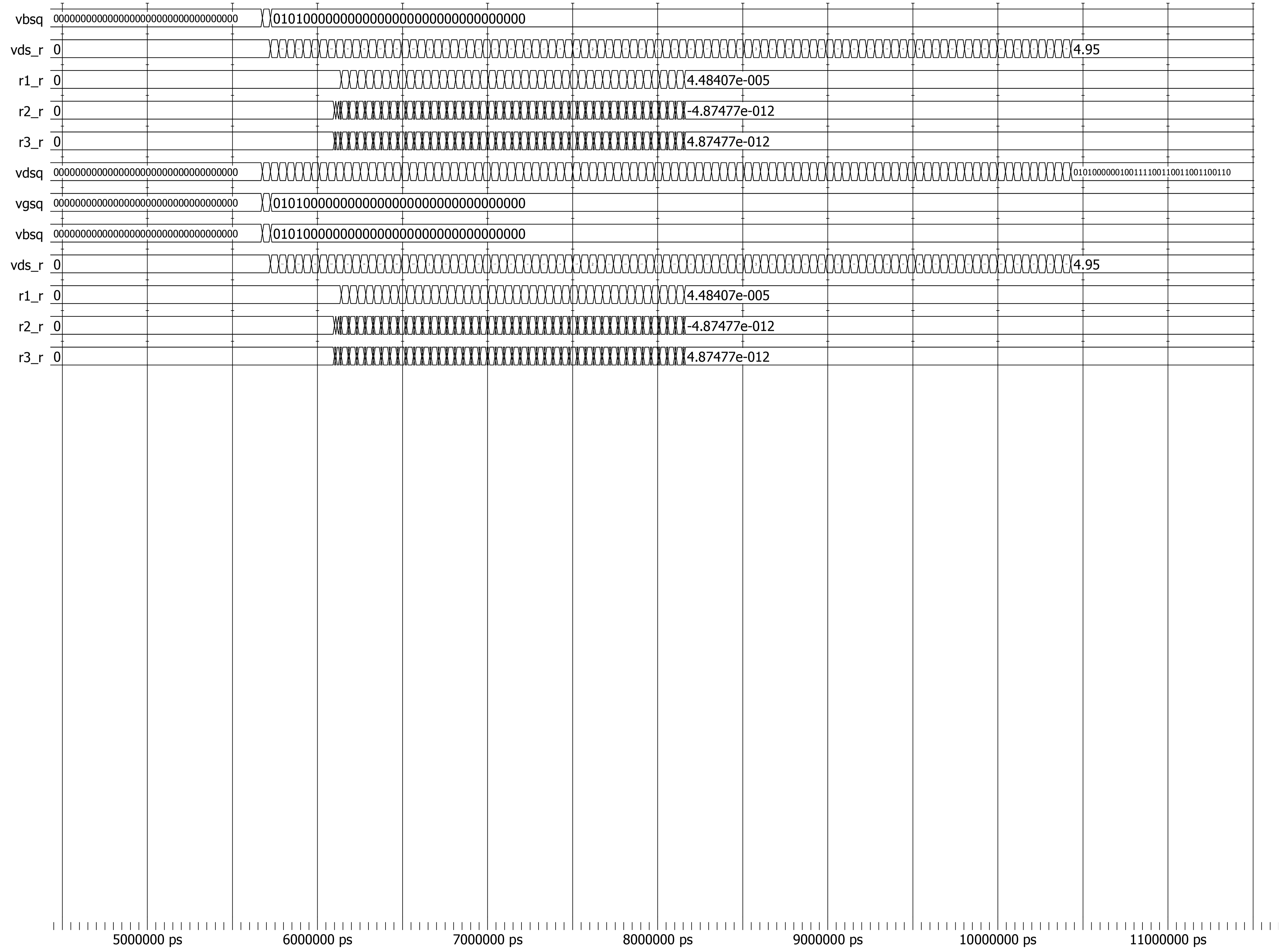
```
                    --  re4 <= to_real(to_float(r4(wE + wF downto 0), wE,
  wF));
                    id1 := re1 + re2;

                    --id2 :=

                    --write(s, vds);
                    --write(s, string'("  ,  "));
                    write(s, id1);
                    writeline(output,s);
                end if;

                if(vds <= vds_max) then
                    --cycle := 0;
                    vdsq <= ”01” & to_slv(to_float(vds, wE, wF));
                    voltages(i) <= ”01” & to_slv(to_float(vds, wE, wF));
                    i := i+1;
                    voltages(i) <= ”01” & to_slv(to_float(2.0, wE, wF));
                    i := i+1;
                    voltages(i) <= ”01” & to_slv(to_float(2.0, wE, wF));
                    i := i+1;
                    vds := vds + inc;
                end if;
            --end if;
            end if;
            delay := delay + 1;
        end if;
    end process;

    --mos3 : entity work.cmos generic map (wE, wF) port map (vdsq, vgsq, vbsq, clk, start,
   done, r1_1, r1_2, r1_3, r1_4, reset);
    --, width, length, channel, vt0, kp, gamma, phi, tox, nsub, nss, nfs, tpg, xj, ld, u0,
   vmax, xqc, delta, theta, eta, kappa, ngate, temperature);

    --mosOrig : entity work.mos port map (5.0,  2.0,  2.0, Idq1, Igq1, Ibq1);
    --mos_syn : entity work.cmos_syn generic map (wE, wF) port map (vdsq, vgsq, vbsq, clk,
   start, done, r2_1, r2_2, r2_3, r2_4, reset);

    --mos3 : entity work.cmosFull port map (vdsq, vgsq, vbsq, clk, r1, r2, r3, r4, width,
  length, channel, vt0, kp, gamma, phi, tox, nsub, nss, nfs, tpg, xj, ld, u0, vmax, xqc,
  delta, theta, eta, kappa, ngate, temperature);

    --re1 <= to_real(to_float(r1_1(wE + wF downto 0), wE, wF));
    --re2 <= to_real(to_float(r1_2(wE + wF downto 0), wE, wF));
    --re3 <= to_real(to_float(r1_3(wE + wF downto 0), wE, wF));
    --re4 <= to_real(to_float(r1_4(wE + wF downto 0), wE, wF));

end architecture;
```

Listing A.3: VHDL CMOS LEVEL 3 Model Testbench

A.3 ModelSim Simulation Waveforms for the pipelined CMOS LEVEL 3 model

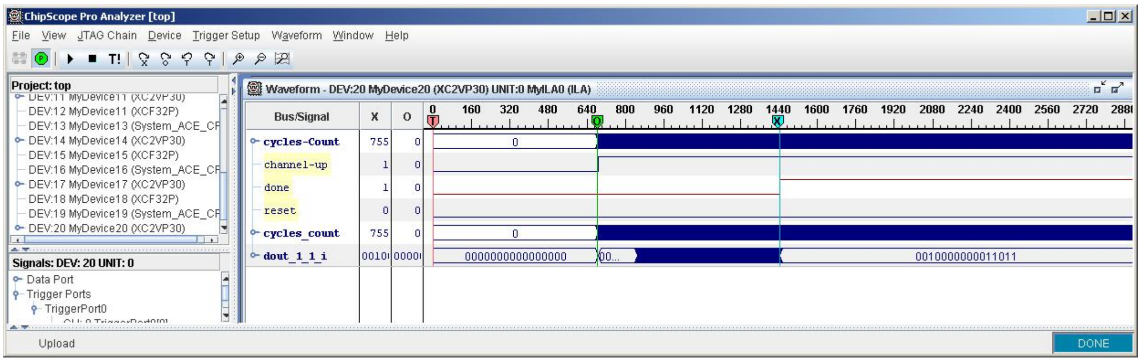


Entity:top_tb_client Architecture:mapped Date: Wed Jun 16 15:20:34 GMT Daylight Time 2010 Row: 1 Page: 1


vbsq
00000000000000000000000000000000
01010000000000000000000000000000
vds_r
0
4.95
r1_r
0
4.48407e-005
r2_r
0
-4.87477e-012
r3_r
0
4.87477e-012
vdsq
00000000000000000000000000000000
0101000000100111100110011001100110
vgsq
00000000000000000000000000000000
01010000000000000000000000000000
vbsq
00000000000000000000000000000000
01010000000000000000000000000000
vds_r
0
4.95
r1_r
0
4.48407e-005
r2_r
0
-4.87477e-012
r3_r
0
4.87477e-012
5000000 ps
6000000 ps
7000000 ps
8000000 ps
9000000 ps
10000000 ps
11000000 ps

A.4 Chipscope Waveforms for the pipelined CMOS LEVEL 3 model

Figure A.1 shows the Chipscope output waveforms used to measure the hardware times. The signal names are the same used in the ModelSim simulation waveforms in Section A.3.


ChipScope Pro Analyzer [top]
File View JTAG Chain Device Trigger Setup Waveform Window Help
T!
Project: top
DEV:11 MyDevice11 (XC2VP30)
DEV:12 MyDevice11 (XCF32P)
DEV:13 MyDevice13 (System_ACE_CF
DEV:14 MyDevice14 (XC2VP30)
DEV:15 MyDevice15 (XCF32P)
DEV:16 MyDevice16 (System_ACE_CF
DEV:17 MyDevice17 (XC2VP30)
DEV:18 MyDevice18 (XCF32P)
DEV:19 MyDevice19 (System_ACE_CF
DEV:20 MyDevice20 (XC2VP30)
Signals: DEV: 20 UNIT: 0
Data Port
Trigger Ports
TriggerPort0
Waveform - DEV:20 MyDevice20 (XC2VP30) UNIT:0 MyILA0 (ILA)
0 160 320 480 640 800 960 1120 1280 1440 1600 1760 1920 2080 2240 2400 2560 2720 288
T
O
X
0
0
00...
Upload
DONE

Bus/Signal	X	O
cycles-Count	755	0
channel-up	1	0
done	1	0
reset	0	0
cycles_count	755	0
dout_1_1_i	0010	0000

0000000000000000

0010000000011011

Figure A.1: The Chipscope output waveform similar to the simulation waveforms in Section A.3

Appendix B

Pipelined VHDL Design Synthesis Results

B.1 Synthesis Reports for both host and slave FPGAs

Table B.1 and Table B.2 show the resource usage reports for the host and slave FPGAs.

TABLE B.1: Synthesis report of the slave FPGA

Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	9,956	27,392	36%
Number of 4 input LUTs	8,019	27,392	29%
Logic Distribution			
Number of occupied Slices	7,445	13,696	54%
Number of Slices containing only related logic	7,445	7,445	100%
Number of Slices containing unrelated logic	0	7,445	0%
Total Number of 4 input LUTs	8,701	27,392	31%
Number used as logic	7,267		
Number used as a route-thru	682		
Number used as Shift registers	752		
Number of bonded IOBs			
Number of bonded	7	556	1%
IOB Master Pads	1		
IOB Slave Pads	1		
Number of RAMB16s	23	136	16%
Number of MULT18X18s	64	136	47%
Number of BUFGMUXs	2	16	12%
Number of BSCANs	1	1	100%
Number of GTs	1	8	12%

Table B.2: Synthesis report of the host FPGA

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	2,263	27,392	8%
Number of 4 input LUTs	2,755	27,392	10%
Logic Distribution			
Number of occupied Slices	2,672	13,696	19%
Number of Slices containing only related logic	2,672	2,672	100%
Number of Slices containing unrelated logic	0	2,672	0%
Total Number of 4 input LUTs	3,271	27,392	11%
Number used as logic	2,534		
Number used as a route-thru	516		
Number used as Shift registers	221		
Number of bonded IOBs			
Number of bonded	8	556	1%
IOB Master Pads	1		
IOB Slave Pads	1		
Number of RAMB16s	85	136	62%
Number of BUFGMUXs	2	16	12%
Number of BSCANs	1	1	100%
Number of GTs	3	8	37%

Appendix C

Xilinx Virtex-II Pro Development Board

Aurora is a gigabit serial communication protocol which can be customised to add communication ports to communicate with the other components in the FPGA fabric. The core utilises the build-in RocketIO MGT (Multi Gigabit Transceiver) hardware to transfer data between FPGAs in two different modes of operation: framing and streaming modes [176]. The theoretical serial communication bandwidth of the system in Figure 4.9 is approximately 3.6 Gb/s, as the aggregate bandwidth for each serial link is about 1.2 Gb/s. The architecture uses up to three slave FPGAs only because of the limitation of the number of available on-board serial links in the XUPV2Pro board [125].

Xilinx boards in Figure C.2 have built in support for the serial communication both in terms of hardware (e.g. Aurora IP) and software (e.g. configuration). However, the boards have a limitation in which only three MGTs are brought forward to SATA connectors. Serial ATA (Serial Advanced Technology Attachment) is a computer bus interface for connecting host bus adapters to mass storage devices such as hard disk drives and optical drives. The rest of the MGTs are connected to the on-board SMA connectors which require new type of connectors. The SATA channels are split into two different formats: two HOST ports and a single TARGET port. Any ports of the same format cannot be connected together. This allows two boards to be connected together, or multiple boards to be connected in a ring fashion. Figure C.1 shows the physical connections of the Target (T) and Host (H) SATA connectors for the four FPGAs used.

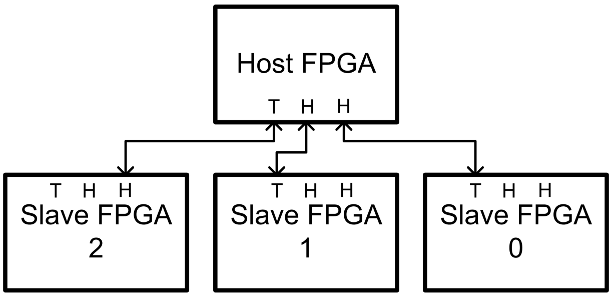


Figure C.1: The Serial Connections between the FPGAs

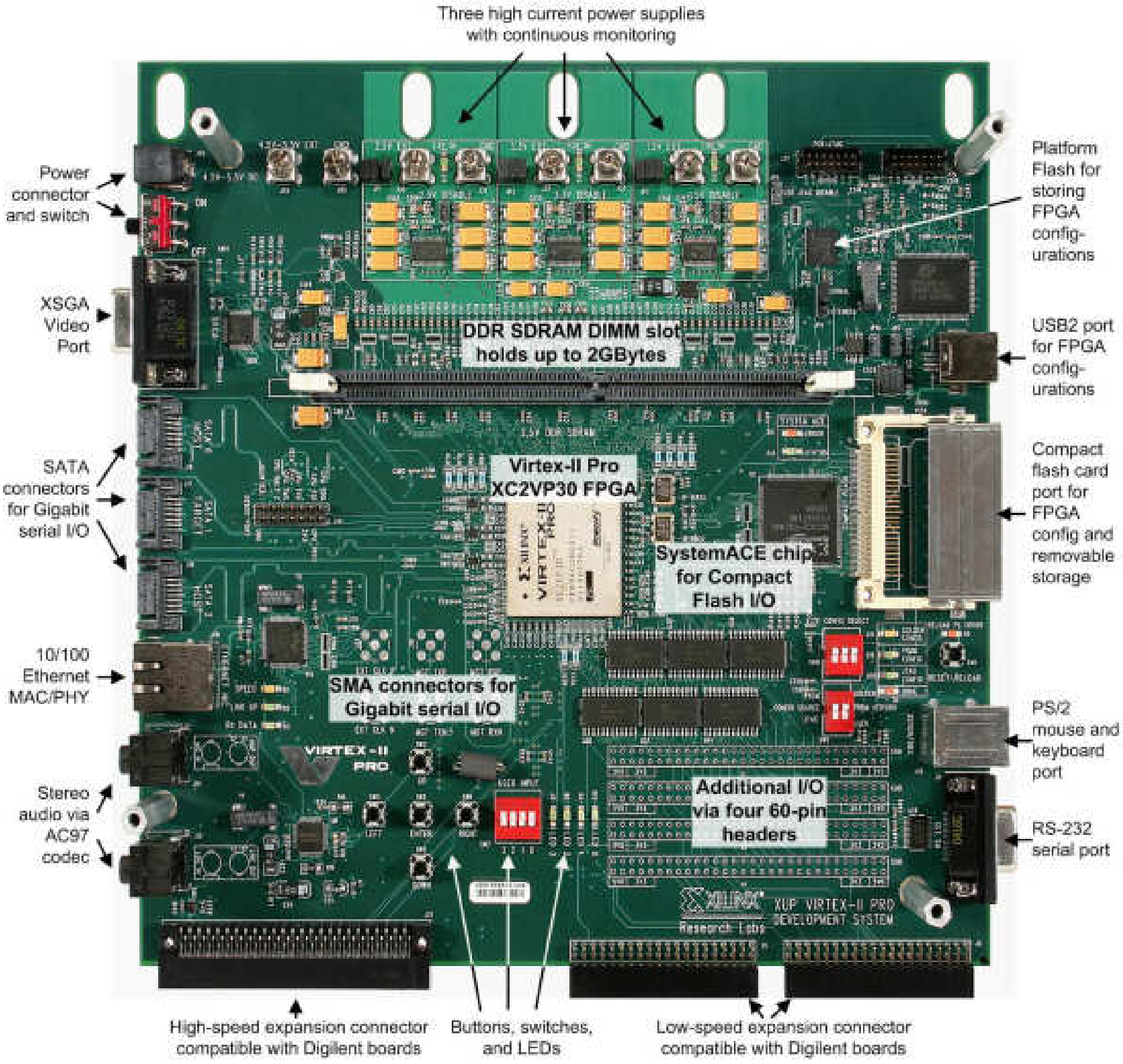

Three high current power supplies with continuous monitoring
Power connector and switch
XSGA Video Port
SATA connectors for Gigabit serial I/O
10/100 Ethernet MAC/PHY
Stereo audio via AC97 codec
DDR SDRAM DIMM slot holds up to 2GBytes
Virtex-II Pro XC2VP30 FPGA
SystemACE chip for Compact Flash I/O
SMA connectors for Gigabit serial I/O
Additional I/O via four 60-pin headers
Platform Flash for storing FPGA config-urations
USB2 port for FPGA config-urations
Compact flash card port for FPGA config and removable storage
PS/2 mouse and keyboard port
RS-232 serial port
High-speed expansion connector compatible with Digilent boards
Buttons, switches, and LEDs
Low-speed expansion connector compatible with Digilent boards

Figure C.2: Xilinx University Program Virtex-II Pro Development System

Appendix D

JTAG Configuration

The JTAG (Joint Test Action Group) scan-chain used to debug the multi-FPGA system is shown in Figure D.1. The JTAG TCK clock termination circuit is shown in Figure D.2 where $R = 54\Omega$.

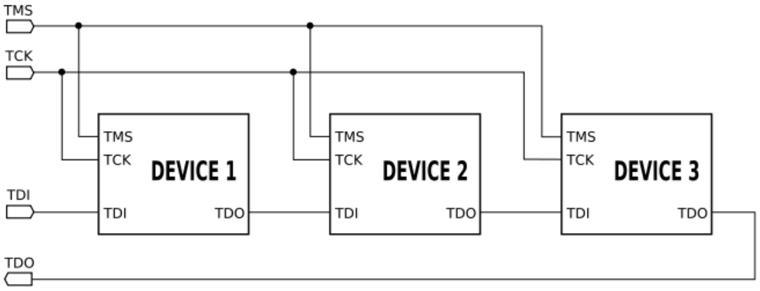


Figure D.1: JTAG Chain Arrangement

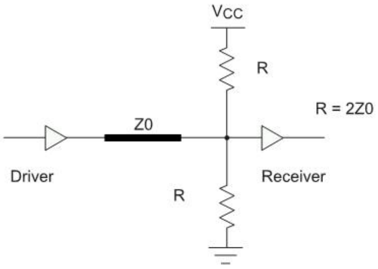


Figure D.2: Thevenin Clock Termination

The boards shown in Figure 4.11 (Section 4.6.1) are connected to a bench power supply providing 5 volts and 4 Amps power source. The boards are daisy-chained through the JTAG interface in order to facilitate batch programming as seen in Figure D.1. This chain is also used by the ChipScope debugger to retrieves the timing/state information for all the FPGAs.

Appendix E

Measuring Hardware and Software Times

E.1 Measuring Hardware Times

Measuring the hardware execution times is done using the Chipscope tool. The ICON core provides a communications path between the JTAG Boundary Scan port of the target FPGA. The ILA core is a logic analyser core that can be used to monitor any internal signal of the design. Figure E.1 shows the overall Chipscope system block diagram. The embedded ILA is triggered and stopped by a start and a stop signals which are added to the overall design for control.

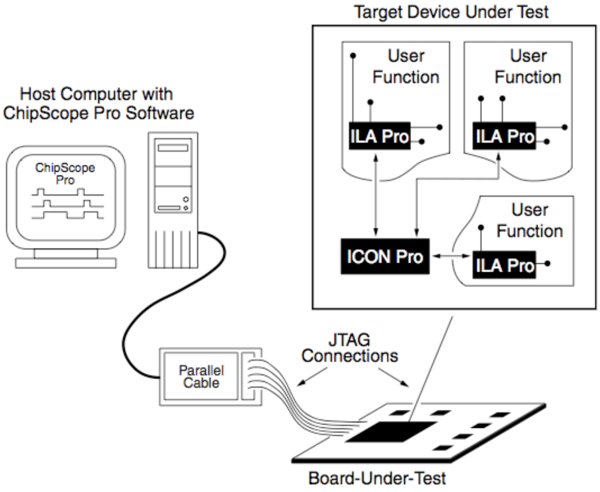


FIGURE E.1: Chipscope System Block Diagram

E.2 Measuring Software Times

For measuring the time spent in software execution of the model evaluation, the `QueryPerformanceCounter()` function from the MSDN Library was used. The function retrieves the current value of the high-resolution performance counter [214]. Example usage is shown in Listing E.1.

```
#include <windows.h>

double PCFreq = 0.0;
__int64 CounterStart = 0;

void StartCounter()
{
    LARGE_INTEGER li;
    if(!QueryPerformanceFrequency(&li))
        cout << ”QueryPerformanceFrequency failed!\n”;

    PCFreq = double(li.QuadPart)/1000.0;

    QueryPerformanceCounter(&li);
    CounterStart = li.QuadPart;
}
double GetCounter()
{
    LARGE_INTEGER li;
    QueryPerformanceCounter(&li);
    return double(li.QuadPart-CounterStart)/PCFreq;
}

int main()
{
    StartCounter();
    Sleep(1000);
    cout << GetCounter() <<”\n”;
    return 0;
}
```

Listing E.1: QueryPerformanceCounter Function usage

Appendix F

Floating-Point Operators for FPGAs

Generally, scientific and financial algorithms make heavy use of floating-point operations. Therefore, the use of FPGA in HPC acceleration requires extra hardware libraries including floating-point operations. This section presents three different floating-point implementations written in VHDL. The VHDL designs were tested using ModelSim and synthesised using Synplify Pro targeting the ALTERA Cyclone-I FPGA.

F.1 Number Representations

Number Representation is a fundamental topic in scientific algorithms and hardware design due to its impact on resources usage. Accuracy is a major requirement for many scientific applications such as the SPICE simulator, where convergence is hugely affected by the precision of quantities (voltages and currents). There are several ways to represent real numbers, however, the most widely used is the binary floating-point format. This section explains the different number systems and their applications especially for resources limited devices like FPGAs.

F.1.1 Fixed-Point System

This system represents a real data type with a fixed number of digits before and after the radix point. Fixed-point numbers are useful for representing fractional numbers in two's complement format. This can be written as $M.F$ where M is the integer part and F is the fractional part. Each integer bit represents a power of two, while each fractional bit represents an inverse power of two. This system is useful for representing fractional numbers in native two's complement format if a Floating-Point Unit (FPU) is not available. This format provides improved performance and reduce hardware complexity as most low-cost processors do not have FPUs. One of the main applications of fixed-point representation are 2D and 3D graphics engines where high throughput is gained with less complex hardware. However, information loss can occur if the results of fixed-point operations exceed the operands' length. The resulting values then have to be rounded or truncated.

F.1.2 Binary Floating-Point System

Floating-point system represents a real number with a string of digits or bits. The IEEE-754 [215] and IEEE-854 [216] standards provides full representation and arithmetic of Binary and Radix-Independent floating-point numbers. It is the most widely adopted standard with many CPU implementations and a number of FPGA FPUs Section F.2. The IEEE-754 standard specifies the binary floating-point format in which a float type is represented by a sign-magnitude form. Figure F.1 shows the single-precision floating-point format (32-bit). The most significant bit is the *Sign* bit where a negative number has a sign bit equals to '1'. The *Biased Exponent* is an unsigned integer representing a multiplicative value of power of two biased with 127. The *Fraction* field contains the 23 most significant bits of the mantissa, with an implicit leading '1' that does not appear in the fraction field. A real number R in single-precision format can be generated using Equation F.1 [217].

$$R = (-1)^S \times F \times 2^{E-127} \tag{F.1}$$

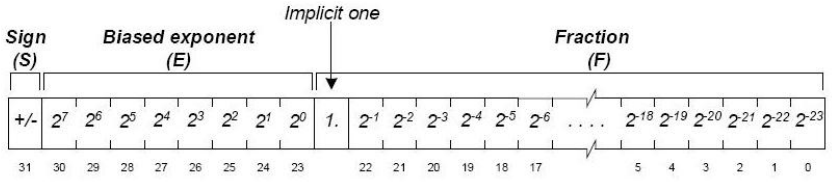


Figure F.1: IEEE single-precision floating-point format

F.1.3 Decimal Floating-Point System

Despite the fact that binary floating-point is suitable for many applications, it cannot exactly represent decimal values used in human calculations. Hence, it should not be used for financial, and commercial applications. This problem can be avoided by using decimal floating-point numbers. Initial benchmarks in [218] indicates that some applications spend 50% to 90% of their time processing decimal data, and software decimal arithmetic is around hundred times slower than hardware implementation.

F.1.4 Comparison

Fixed-point representation has the advantage of being very efficient if terms of performance and hardware area requirement as a fixed-point number has a defined width and decimal point location. This is fine for many applications as long as the number is within the range to give enough precision. Decimal floating-point solves the problem of representing decimal values accurately. A common use of this representation is for storing monetary values, where the inexact values of floating-point numbers are often a liability. But, the hardware needed to implement this system is rather complex.

Recently, there have been an increasing demand for the extended dynamic range and precision in floating-point arithmetic by several applications such as signal processing, advanced wireless communication, and imaging applications. Floating-point numbers are used to overcome this precision limitation. However, these operators often take up around three times the hardware area of fixed-point on FPGAs.

High numerical precisions are a big problem for both FPGAs and ASICs as these designs tend to consume significant area and require deep pipelining [157]. For example, double precision multipliers require about 20 pipeline stages and 30-40 stages for the square root operator as shown in [219].

F.2 Floating-Point FPGAs Libraries

F.2.1 New IEEE VHDL Standard Revision

IEEE is currently undergoing a revision to the VHDL standard [220]. The new standard IEEE 1076-2006 contains some improvements to the old packages plus two new math packages : ‘`fixed_pkg`’ and ‘`float_pkg`’. These packages have been designed for use in VHDL-2006 and will be part of the IEEE library. A compatibility version of the proposed packages is provided which is fully synthesisable and has no dependencies on the other new packages.

F.2.1.1 Fixed-Point Package ‘fixed_pkg’

This package defines two new types : ‘`ufixed`’ which is the unsigned fixed-point type and ‘`sfixed`’ is the signed fixed-point type [221]. The following VHDL listing shows the usage model of this package.

```
type ufixed is array (INTEGER range <>) of STD_LOGIC;
type sfixed is array (INTEGER range <>) of STD_LOGIC;
...
use ieee.math_utility_pkg.all; -- ieee_proposed for VHDL-93 version
use ieee.fixed_pkg.all; -- ieee_proposed for compatibility version
...
signal a, b : sfixed (7 downto -6);
signal c: sfixed (8 downto -6);
begin
...
c <= a + b;
```

Listing F.1: ‘`fixed_pkg`’ package usage model

The location of the decimal point is assumed to be between the 0 and −1 indices. The package provides most of the standard functions available in the `numeric_std` and `std_logic_1164` packages such as add “+”, subtract “−”, multiply “∗”, divide “/”, modulo “mod”, and remainder “rem”. All fixed-point operators defined in this package are purely combinational. Conversion operators are also available to convert between standard types and the new fixed-point types.

F.2.1.2 Floating-Point Package ‘float_pkg’

The floating-point numbers are defined by the standards IEEE-754 [215] and IEEE-854 [216]. The floating-point package provides full implementation of these specifications. The base package defines three floating-point types [222]:

‘Float32’ : 32-bit IEEE 754 single precision floating point
‘Float64’ : 64-bit IEEE 754 double precision floating point
‘Float128’ : 128-bit IEEE 854 extended precision floating point

The package also allows custom floating-point widths to be specified by bounding the ‘float’ type as shown in the VHDL listing bellow. A negative index is used to distinguish between the exponent and the fraction fields. The package defines the operators for all the standard math, conversion, and compare operations specified in the IEEE floating-point standard. All of these operators are purely combinational. The following VHDL listing shows the package usage model.

```
use ieee.float_pkg.all; -- use ieee_proposed for VHDL-93 version

variable x, y, z : float (5 downto -10);
begin
y := to_float (3.1415, y); -- Uses ‘‘y’’ for the sizing only.
z := ‘‘0011101010101010’’;-- 1/3
x := z + y;
```

Listing F.2: ‘float_pkg’ package usage model

All operators implemented in the new packages are combinational, which means very high area usage and low frequencies. However, both packages contain several useful conversion and comparison operators that can be used to handle real data type. Table F.1 shows the operations' area usage both single and double-precision types.

Table F.1: Float package synthesis results (Synplify Pro)

Operation	**Single-precision FP**		**Double-precision FP**	
	CLBs	% of EP1C12	CLBs	% of EP1C12
Addition	1167	9	2745	22
Multiplication	1535	12	7467	61
Division	2322	19	8313	68
Square root	27605	228	204454	1695

F.2.2 OpenCores FPU

This is a free 32-bit floating-point arithmetic implementation fully compliant with the IEEE-754 Standard [118]. It supports addition, subtraction, multiplication, division, and square root. For each operation four rounding modes are supported: round up, round down, round to nearest even, and round to zero. These operators are sequential, designed to achieve high operating frequency with less hardware area. Table F.2 shows the number of clock cycles needed to perform each operation. Synthesis process showed 37% area usage and 70 MHz frequency on Cyclone-I EP1C12. However, one of the main constraints of this library is that it supports single-precision floating-point numbers only (32-bit).

F.2.3 FPLibrary

FPLibrary is a parametrisable library of hardware operators for the floating-point and logarithmic number systems, developed in the Arénaire project at ENS, University of Lyon [119]. This is an open source library written in VHDL, mainly targeted for FPGAs. All FP and LNS operators

TABLE F.2: Number of clock cycles needed for each operation

Operation	**Number of clock cycles**
Addition	7
Subtraction	7
Multiplication	12
Division	35
Square root	35

are parametrisable in terms of precision of operands and results. They are also available in both combinatorial and pipelined versions. The library provides extra packages supporting floating-point logarithm and exponential functions [120]. The floating-point format used in FPLibrary is slightly different from the IEEE-754 Standard. This representation is parametrised by two bit-widths w_E (exponent width) and w_F (fraction width) [223]. A FP number X is represented as a vector of $w_E + w_F + 3$ bits as shown in figure F.2.

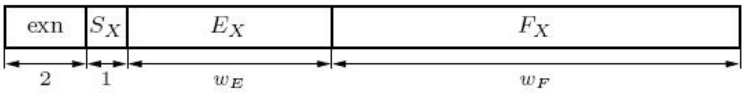


FIGURE F.2: FPLibrary FP number format

- exn (2 bits): the exception tag
- S_X (1 bit): the sign bit
- E_X (w_E bits): the exponent biased by $E_0 = 2^{w_E-1} - 1$
- F_X (w_F bits): the fraction

The exception tag controls the value of X as follows:

TABLE F.3: Value of X According to Exception Flag

exn	**Value of X**
00	0
01	$(-1)^{S_X} \times 1.F_X \times 2^{E_X-E_0}$
10	$(-1)^{S_X} \times \infty$
11	NaNs

Table F.4 shows the number of clock cycles needed to perform each single-precision operation. Table F.5 demonstrates the synthesis results of the FPLibrary floating-point operations. It shows the estimated frequency and area usage of each operation for single and double-precision FP formats (pipelined designs only). It can be seen from these results that FPLibrary uses larger number of logic units than the OpenCores FPU, but with very high frequencies which means higher throughput.

Table F.4: Number of clock cycles needed for each operation

Operation	**Number of clock cycles**
Addition	3
Multiplication	4
Division	15
Square root	14
Logarithm	11
Exponential	14

Table F.5: FPLibrary Synthesis Results (Synplify Pro)

Operation	**Single-precision FP**			**Double-precision FP**		
	f(MHz)	CLBs	% of EP1C12	f(MHz)	CLBs	% of EP1C12
Addition	113.3	897	7	78.8	1794	14
Multiplication	139.7	1283	10	110.4	6133	50
Division	146.9	1993	16	111.1	8463	70
Square root	153.5	950	7	121.8	4472	37
Logarithm	71.57	3122	26	n/a	n/a	n/a
Exponential	80.9	2811	23	n/a	n/a	n/a

F.2.4 Floating-point Libraries Comparison

The synthesis results shows that FPLibrary is the most efficient library in terms of area utilisation and clock frequency. FPLibrary provides both combinational and pipelined versions of the floating-point operators with an easy-to-use interface. In addition, FPLibrary in fully parametrised, in which the operands' width can be specified using generics. This allows the user to synthesise the operators to a specific floating-point precision of choice. FPU supports single-precision numbers only which limits its usage. IEEE proposed float and fixed packages provide many useful operators and conversion functions. However, these operators are all combinational blocks, which means very large area usage. The IEEE fixed and float packages provide very useful functions that are not provided in the other two libraries. This includes data types conversion functions, logical operators, compare operations, and text I/O functions.