Simulator for the Versat Reconfigurable Processor

João César Martins Moutoso Ratinho

Introduction to the Research in

Electrical and Computer Engineering

Supervisor(s): Prof. José João Henriques Teixeira de Sousa

January 2019

Abstract

Versat is a Coarse-Grain Reconfigurable Array architecture (CGRA), which implements self and partial reconfiguration by using a simple controller unit. This report studies the current state of the art in HDL and CGRA simulation, providing a basis to the development of a simulation environment for Versat. The main objective of this environment is to provide a faster way to develop and debug software without the use of prototyping hardware. Therefore, the two types of HDL simulators, event-driven and cycle-accurate, their advantages and disadvantages are studied, along with a performance comparison between them. A study of high-level implementations for CGRA simulation is also presented.

Keywords: Versat, coarse-grain reconfigurable arrays, HDL simulation, CGRA simulation, high-level simulation

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

During the last few years reconfigurable architectures have evolved greatly, as a result of the research that has been carried out in this topic. This has created a vast market for this type of architectures due to the shorter time to market and lower non-recurring engineering costs, when compared with other solutions.

Some solutions use standard fine-grained reconfigurable architectures like commercial FPGAs. However, they are often too large and power hungry to be used as embedded cores. In this context, the Coarse Grain Reconfigurable Array (CGRA) is a suitable alternative. Essentially, a CGRA consists of a collection of functional units and memories, interconnected by programmable switches for forming hardware datapaths that accelerate computations.

In CGRAs, like with any other digital circuit, simulation is fundamental to verify if the circuit is working properly in the different stages of its development. However, neither FPGAs nor CGRAs have good high-level simulation tools. This means that they have to be simulated using simulators already available in the market or, in alternative, using simulators developed just for the architecture in consideration.

1.2 Topic Overview

In digital electronics, frequent simulation is required during the different phases of project development. There are multiple tools, proprietary or open source, that can be used. Each of these tools present different characteristics, that can make them more or less adequate, depending not only on the circuit that is going to be simulated, but also on the speed and accuracy of the desired simulation.

In a CGRA, the reconfigurable interconnects between the functional units allow building different datapaths during the run-time. This means that event-driven simulators using HDL input are usually applied for generic simulation of applications mapped on CGRAs, resulting in long simulation times, since the simulator needs to consider all the events generated by the changing signals inside the architecture.

Consequently, finding a valid alternative to simulate CGRAs (in this particular case, the Versat architecture) could save an important amount of time and money during the development of applications for this type of architectures.

1.3 Objectives

The main objective of this work is to study the current state of the art in high level simulation of CGRAs. This implies studying the current HDL Simulator Technology, including the different types of simulators, and also the Versat architecture, a CGRA that is suitable for low-cost and low-power devices. This way, is possible to understand what types of simulators are more suitable for the Versat architecture.

Finally, this work is intended to serve as a basis to the thesis dissertation, where a simulation environment for the Versat reconfigurable processor will be developed.

1.4 Author's Work

The candidate developed this work during an internship at IObundle, Lda, the company that developed the Versat architecture (and is sponsoring the candidate's thesis). This allowed the candidate to gain a deeper knowledge of the Versat architecture, mainly by participating in a project with an international client where the Versat architecture was used to accelerate the front end of an MP3 encoder (as described in section 2.4).

1.5 Report Outline

This report is composed by 3 more chapters. In the second chapter the Versat architecture is presented. In the third chapter the current HDL simulation state-of-the-art is analysed. Finally, the fourth chapter discusses the best strategy for the problem addressed, with two different approaches to CGRA simulation being analysed.

Chapter 2

The Versat architecture

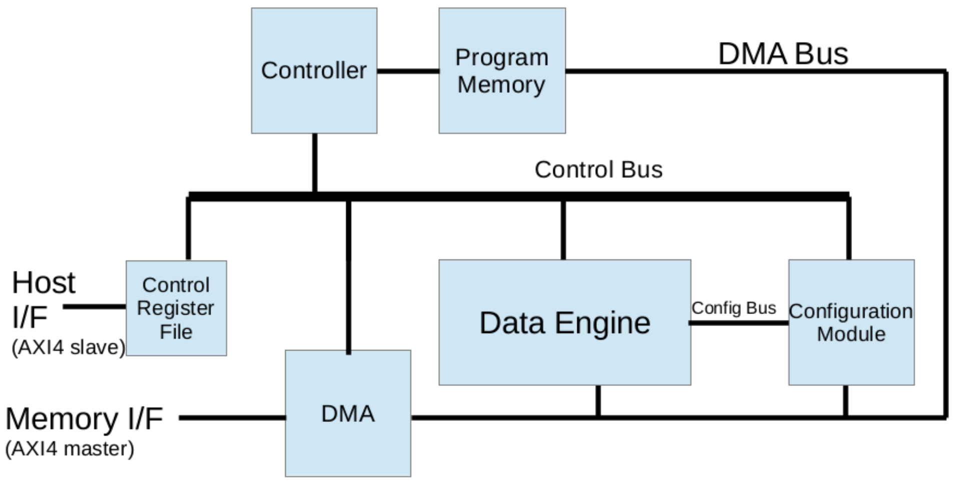

Figure 2.1: Versat top-level entity.

The Versat architecture [1–4] is shown in Figure 2.1, and it consists of the following modules: Controller, Program Memory, Control Register File, DMA, Data Engine and Configuration Module. The Controller can access the various modules in the system via the Control Bus, and it executes programs stored in the Program Memory (9kB = 1kB boot ROM + 8kB RAM). The user programs are loaded in the RAM to execute algorithms which involve managing Data Engine (DE) reconfigurations and DMA data transfers.

Versat user programs can use the DE to carry out data intensive computations. To perform these computations, the Controller writes DE configurations to the Configuration Module (CM) or simply restores configurations previously stored in the CM. The Controller can also load the DE with data to be processed or save the processed data back in the external memory using the DMA engine. The DMA engine can also be used to initially load the Versat program or to move CGRA configurations between the core and the external memory.

The Versat core has a host and a memory interface. Both of them use ARM's Advanced eXtensible Interface (AXI), a standard for busing. The host interface (AXI slave) is used by a host system to instruct

Versat to load and execute programs. The host and the Controller communicate using the Control Register File (CRF), an unit that is also used by Versat programs as a general purpose (1-cycle access) register file. The memory interface (AXI master) is used to access data from an external memory using the DMA.

2.1 Data Engine

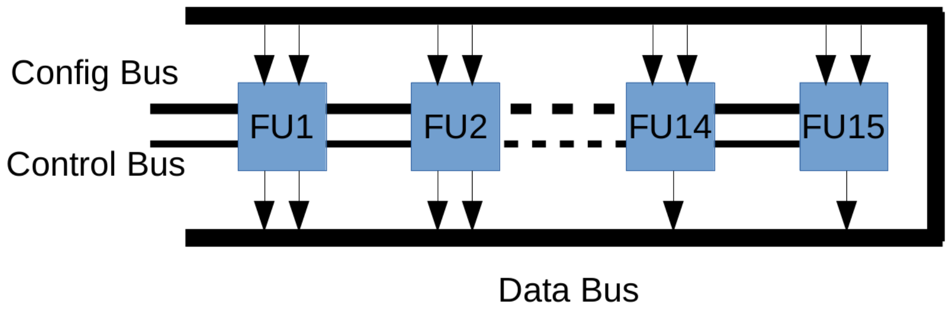

Figure 2.2: Versat data engine.

The Data Engine (DE) has a flexible topology, in which the user can configure the amount of functional units (FUs) and their respective type. In Figure 2.2 is shown a DE example with 15 FUs. The DE is a 32-bit architecture with the following configurable FUs: Arithmetic and Logic Unit (ALU), Multiplier Accumulator (MAC), Barrel Shifter (BS) and dual-port 16kB embedded memories (MEM). The output registers of the FUs are read/write accessible by the Controller via the Control Bus.

The FUs are interconnected by a wide bus called the Data Bus. This bus is the concatenation of all FU outputs, with each FU contributing with a 32-bit section to the Data Bus. An embedded memory contributes 2 sections to the Data Bus, since it has 2 ports. These sections can be selected by each FU, according to the configurations that they receive from the respective configuration registers in the CM, whose outputs are concatenated in another wide bus called the Config Bus.

The DE has a full mesh topology, meaning that each FU can select any FU output as one of its inputs. This kind of structure may seem unnecessary but it greatly simplifies the compiler design as it avoids expensive place and route algorithms [2].

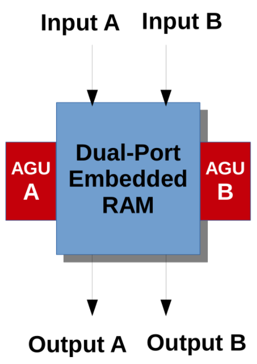

Figure 2.3: Versat dual-port embedded memory with AGUs.

Each one of the dual-port memories used in Versat have a data input, a data output and an address input, as shown in Figure 2.3. Also, each port has an Address Generation Unit (AGU), that can be programmed to generate the address sequence used to access data from the memory port during the execution of a program loop in the DE. The AGUs support two levels of nested loops and can start execution with a programmable delay, so that circuit paths with different latencies can be synchronized.

2.2 Configuration Module

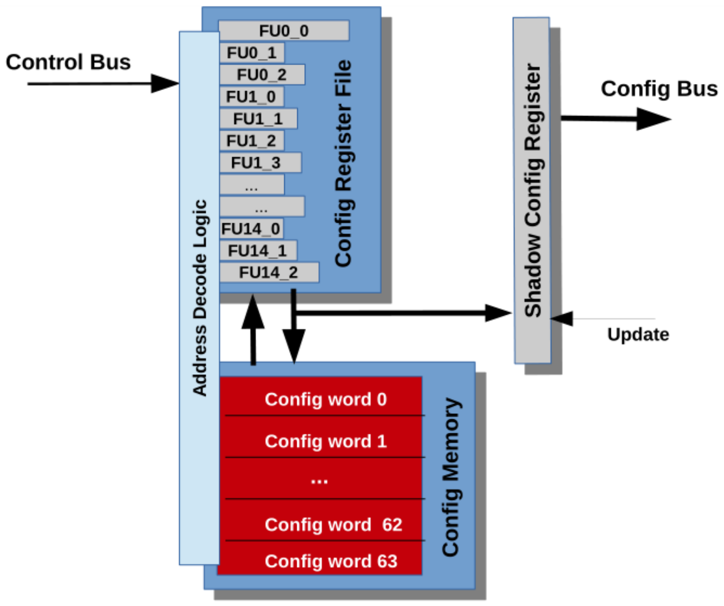

Figure 2.4: Versat configuration module example.

In Versat, the configuration bits are organized in configuration spaces, one for each FU. Each configuration space comprises multiple fields, which are memory mapped from the Controller point of view. Thus, the Controller is able to change a single configuration field of an FU by writing to the respective address. This implements partial reconfiguration.

A Configuration Module (CM) example is illustrated in Figure 2.4, with a reduced number of configuration spaces and fields for simplicity. It contains a register file with a variable length (the length depends on the FUs used), a shadow register and a memory. The shadow register holds the current configuration of the DE, which is copied from the main configuration register whenever the Update signal is activated. This means that the configuration register can be changed in the main register while the DE is running.

When the CM is addressed by the Controller, the decode logic checks if the configuration register file or the configuration memory is being addressed. The configuration register file accepts write requests and ignores read requests. On the other hand, the configuration memory deals with read and write requests in the following way: a read request causes the addressed contents of the configuration memory to be transferred into the configuration register file, while a write request causes the contents of the configuration register file to be stored into the addressed position of the configuration memory. This is a mechanism for saving and loading entire configurations in a single clock cycle because all the data transfers are internal.

2.3 Controller

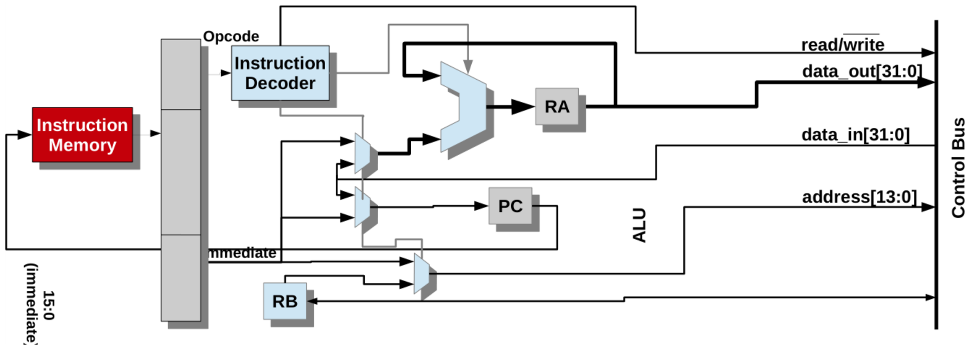

Figure 2.5: Versat controller.

The controller used in the Versat architecture is just used for reconfiguration, data transfer and simple algorithmic control. This controller is not meant to replace a more general host processor, which can run complex applications while using Versat as an accelerator.

The controller has an accumulator architecture, using a 2 stage pipeline, as shown in Figure 2.5. In this architecture 3 main registers are used: RA, RB and PC. The RA is the accumulator, the RB is the address register, used for indirect loads and stores, and the PC is the register that holds the value of the program counter.

The controller is the master of a simple bus called the Control Bus, which can be accessed using the 4 signals shown in Figure 2.5. Register RB can be accessed using the Control Bus as if it were a peripheral of the Control Bus.

Table 2.1: Versat instruction set.

Mnemonic	Description
rdw	RA = (Imm)
wrw	(Imm) = RA
rdwb	RA = (RB)
wrwb	(RB) = RA
ldi	RA = Imm
ldih	RA[31:16] = Imm
beqi	RA == 0? PC = Imm: PC += 1; RA = RA-1
beq	RA == 0? PC = (Imm): PC += 1; RA = RA-1
bneqi	RA != 0? PC = Imm: PC += 1; RA = RA-1
bneq	RA != 0? PC = (Imm): PC += 1; RA = RA-1
add	RA = RA + (Imm)
addi	RA = RA + Imm
sub	RA = RA - (Imm)
shft	RA = (Imm < 0)? RA=RA<<1: RA=RA>>1
and	RA = RA & (Imm)
xor	RA = RA ˆ(Imm)

The Versat controller Instruction Set Architecture (ISA) has 16 instructions, as shown in Table 2.1. Currently, it can only be programmed in assembly language as it does not yet have a C compiler which is being developed by another student working in this project.

2.4 Application Example

During the work carried out for this report an application for an international client using the Versat architecture was developed. It consists of an MP3 encoder, shown in Figure 2.6, where Versat is used to accelerate the front end of the algorithm, denoted MP3-FE in Figure 2.6).

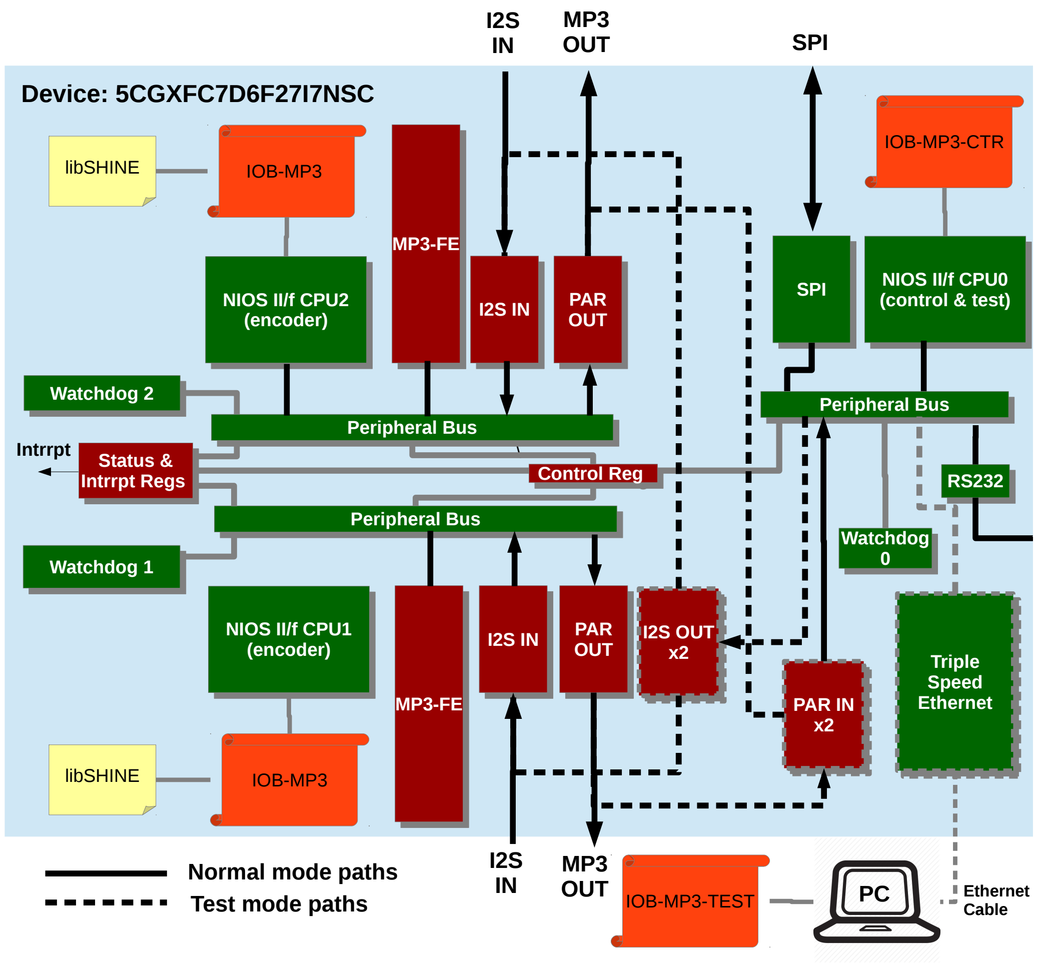

Figure 2.6: Schematic of the application where Versat was used.

The MP3 algorithm is based in Shine [5], a fast fixed-point MP3 encoding library. The front end runs sub-band filtering of the audio signal and applies the Modified Discrete Cosine Transform to the result. These functions are the most computationally expensive functions in the MP3 algorithm and normally require DSP extensions to the ISA of the processor used. In the approach taken, a more modest processor was used (Intel's NIOS2) which used Versat as a peripheral acceleration core.

To run this algorithm the Versat architecture was configured with 4 dual-port memories and 3 FUs: a Multiply Accumulate Unit (MAC), a simple Arithmetic and Logic Unit (ALU) and a Barrel Shifter Unit

(BS). The MP3 front-end was written in the Versat Assembly language, since Versat compiler does not support yet the use of a higher level programming language (like C or C++).

The FE was intensively simulated before synthesis and implementation in the FPGA. Initially, the simulations were performed using Icarus Verilog [6], an open-source Verilog simulator. However, as the complexity of the core increased, the simulation times also increased dramatically, since Icarus Verilog is a fairly slow simulator, as will be seen in section 3.3, where the performance of the different HDL simulators is analysed.

As a result, at a certain point in the project, the simulator was changed to Cadence NCSim [7] to speed up the simulations. Despite NCsim being considerably faster than Icarus Verilog, the simulations still took a considerable amount of time. This reinforces the need for a faster simulation environment for the Versat architecture.

Chapter 3

HDL Simulators

As mentioned before, HDL simulators play a fundamental role during the different phases of circuit development, and there are multiple simulation tools that can be used. However, despite all these tools having more or less the same purpose (provide a way to validate the circuit being tested), they do not work in the same way.

Typically, before testing a circuit, a test bench is created. The test bench is a program, written in a HDL or in a programming language (like SystemC, for example), that comprises three modules [8]: stimuli generator, golden response generator and response analyser, as shown in Figure 3.1. The stimuli generator module is responsible for generating the signals needed to make the circuit work properly. On the other hand, the golden response generator computes the expected circuit response, based on the inputs generated by the stimuli generator. Finally, the response analyzer compares the circuit output signals with the ones generated by the golden response generator. During simulation, if both signs are equal, it means that the circuit is working as intended.

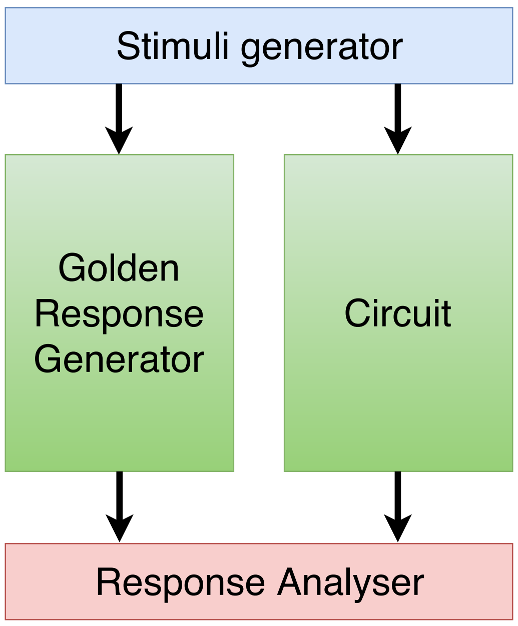

Figure 3.1: Test bench diagram.

The results provided by the test-bench might depend on the chosen simulator. This happens because all the simulators work in different ways: while some focus on obtaining the most complete results (including simulation timings), sacrificing the speed of the simulations, others do the opposite. In this perspective, the simulators can be divided into two main categories [8, 9]: event-driven or cycle-accurate.

3.1 Event-Driven Simulators

The first category of simulators are the event-driven ones [8–10]. This simulators work by taking events sequentially, propagating them through the circuit until it reaches a steady state.

The events are generated each time that a circuit input is changed, being stored in a queue, ordered chronologically to allow the correct execution of the events. When an event is evaluated, only the circuit nodes that have their input changed by that event are evaluated. After evaluation, the event is removed from the queue, with new events resultant from the output changes being added. This means that the same element might be evaluated multiple times during the same time step due to the feedback from some signals.

It's important to mention that during the simulation process there's a timer that is used to keep track of the events timings. This leads to one of the main advantages of event-driven simulation, which is the accurate simulation results, with detailed timing information, allowing the identification of timing problems in the tested circuit.

Despite this important advantage, this type of simulation also brings some disadvantages, mainly related with its speed. Due to their complex algorithms used for event scheduling and timing evaluation, event-driven simulators are slow. While for relatively small circuits this might not be a significant problem, for large circuits this is an important disadvantage, because their increased complexity will increase significantly the simulation duration.

This type of simulators can be divided into 2 main categories: software or hardware based, each one with different subcategories. These categories are analysed in the following subsections.

3.1.1 Software-based Simulators

The software-based simulators are the most common type of simulators, including simulators like the Cadence NCSim [7], the Synopsys VCS [11], the Mentor Graphics ModelSim [12] or the Icarus Verilog [6]. Usually, they run on a general purpose computer, being divided into three categories, according to their algorithms: compiled-code, interpreter and gate level.

An interpreter software simulator reads the HDL code to simulate and interprets it, translating the original code to a set of instructions accepted by the simulator program. This translation process occurs during runtime and implies the creation of data structures to store the data taken from the HDL file, that will be used afterwards to create the simulation. This simulators are somewhat inefficient, due to the resultant overhead of the code translation. This typically results in the execution of a considerable number of instructions per element evaluation, of which only a few perform logic model evaluation [13].

On the other hand, a compiled-code simulator works by transforming the HDL circuit description, including its testbench, into an equivalent C code (or some similar programming language). The generated code is then compiled by a generic complier (like gcc, for example), resulting in an executable file, that will the be executed to run the simulation. This type of simulators are more efficient than the interpreter ones, since they eliminate the overhead of traversing the network data structures [13]. The most used simulators, like Cadence NCSim, Synopsys VCS or Icarus Verilog belong to this category of simulators.

Although the gate level simulators are either of the interpreted or compiled-code type, they differ from the simulators referred in those categories [8]. This happens because, while those simulators have full Verilog compliance (supporting also gate level simulations), the gate level simulators just support a small subset of Verilog.

RTL simulation is the most used method for circuit verification due to its reasonable accuracy [14]. However, in the last few years there has been a rising trend in the industry to run gate level simulations [15]. This happens mainly due to the more complex timing checks required by modern process nodes. As a result, despite gate level simulation being more time consuming than RTL simulation, it greatly improves the verification results.

Usually, gate level simulation is used before going into the last stages of circuit production. As shown in Figure 3.2,the circuit is synthesized to a gate level netlist only after the RTL description of the circuit is working properly. Then, the gate level netlist is simulated, with its results being compared with the ones obtained with the RTL description. It they are equal, then the circuit is working as intended.

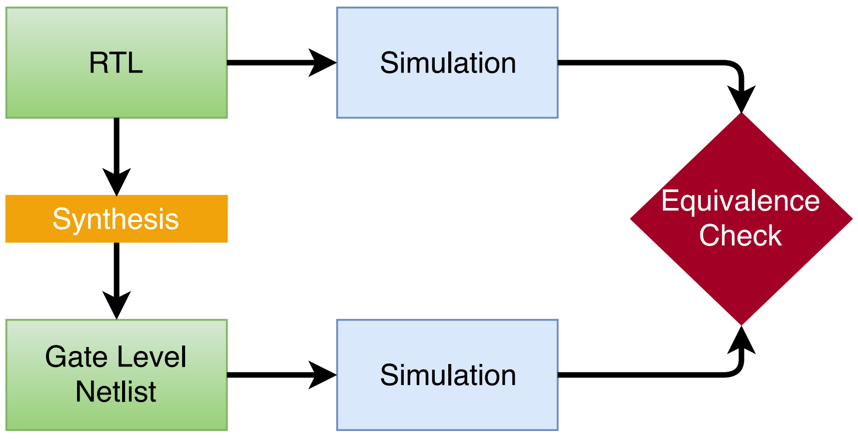

Figure 3.2: Gate level design flow diagram.

3.1.2 Hardware-based Simulators

As the name indicates, hardware-based simulators are a type of simulators that rely on configurable hardware to do the digital circuit verification. When compared with the software-based simulators, they have the advantage of being a few orders of magnitude faster [8]. However they also have some disadvantages: the hardware can be costly sometimes (depending on its specifications) and it requires long compilation times, which makes them needless for smaller designs. These simulators also require proprietary hardware platforms to perform the desired simulations, with the hardware setup depending on the platforms used, being different on each platform. As a result, these type of simulators have a steep learning curve.

In this simulation type, the Verilog design is mapped onto a reconfigurable piece of hardware with the same logical behavior as the netlist. The simulation is divided between the software simulator, which simulates all the Verilog code that is not synthesizable, and the hardware accelerator, which simulates everything that is synthesizable [15]. The design is then run on the hardware, producing the simulation results. The results, like in a Software-based simulator, must be checked in order to assess if the circuit is working properly.

There are two variants of Hardware-based simulators: FPGA-based or emulator-based. On one hand, the FPGA-based simulators, as the name indicates, rely on FPGAs. A FPGA (Field Programmable Gate Array) is an integrated circuit designed to be configured multiple times, according to the user needs. It comprises an array of programmable logic blocks, memory elements, arithmetic functions, etc.

The FPGA-based simulators follow the flow shown in Figure 3.3. The original Verilog description of the code is transformed into an intermediate representation, independent of the target platform. Here, the synthesizable portions of the code are mapped into the FPGA, while the non synthesizable portions (namely intendend for verification purposes) are run as software in the host machine (software/hardware co-simulation). The simulator and the FPGA interact with each other to produce the results.

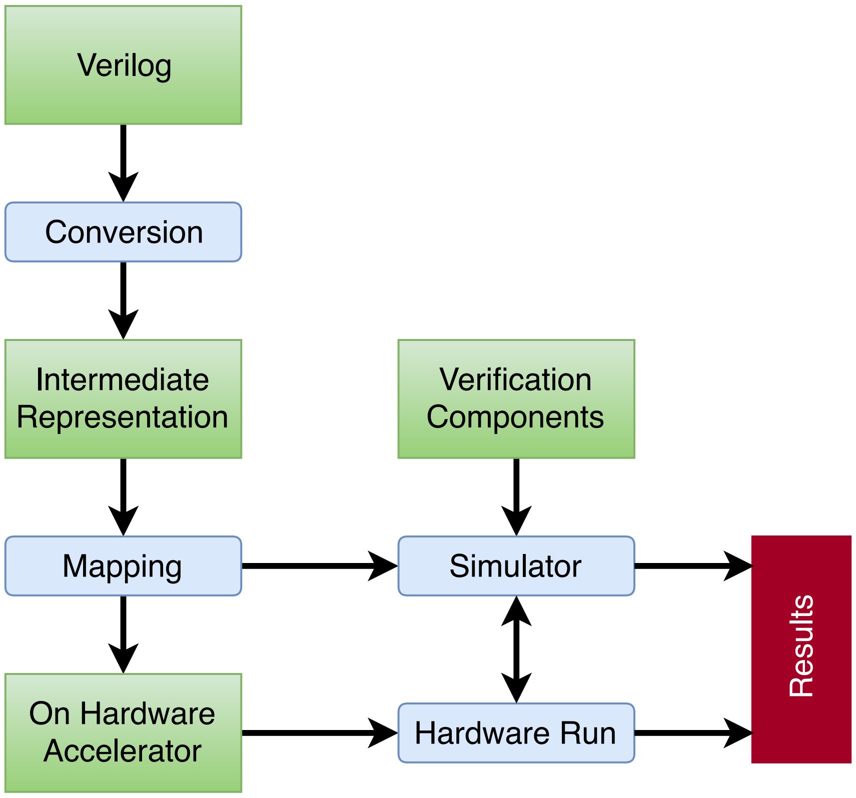

Figure 3.3: FPGA-based simulation flow [15].

On the other hand, Emulator-based simulators rely on emulators to run their simulations. An emulator is a specialized piece of hardware, that is , an Application Specific Circuit (ASIC) that, when compared to a FPGA offers limited reconfigurability, but with the advantage of a higher simulation speed. Even though, the FPGA can run the actual circuit which in many cases superseeds emulators in performance despite the lower clock frequency.

This type of simulation offers the possibility of testing software on the developed design before having it implemented on chip, in a way that the software application runs exactly as it would on the real chip in a real circuit. It also offers the possibility for testing more complex programs, that would take large amounts of time (sometimes even days) to run on other types of simulators (like the software-based ones).

The simulation flow for this type of simulators has some similarities with the FPGA-based simulators. In this case, as shown in Figure 3.4, the Verilog code is converted into an intermediate representation, that will the be mapped into the emulator. The code to be mapped must include only code that can be implemented into the emulator. This means that the non synthesizable code must be separated, being included in the software application. Both the software and hardware platform interact with each other to produce the results.

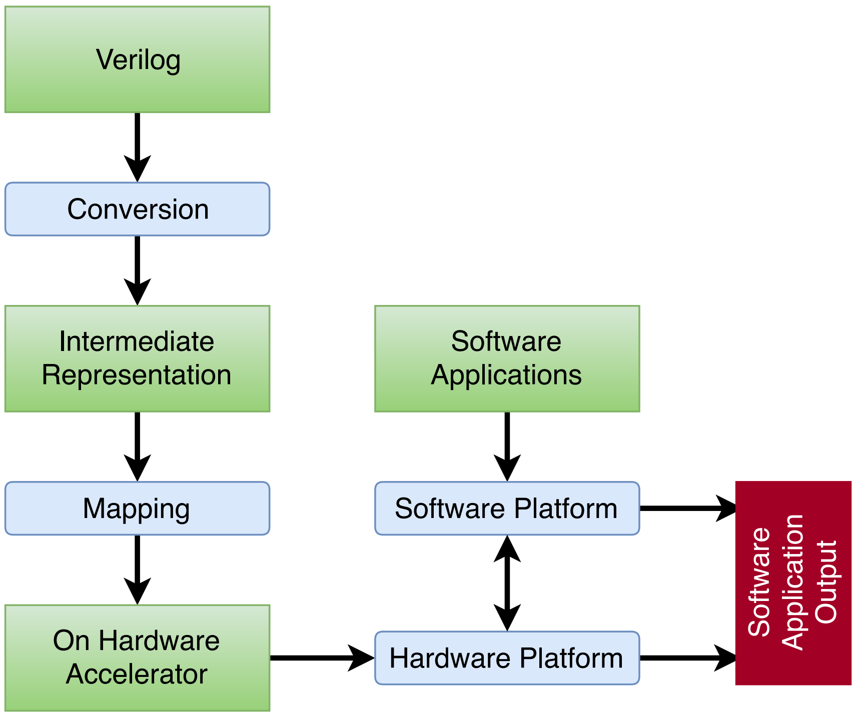

Figure 3.4: Emulator-based simulation flow [15].

3.2 Cycle-Accurate Simulators

Cycle-Accurate simulators are another important category of HDL simulators. Instead of taking events sequentially, propagating them through the circuit until it reaches a steady state, like the event-driven simulators, this type of simulators evaluate each logic element of the circuit in a clock cycle. They do this evaluation for each clock cycle, without taking into consideration the propagation times and delays within the cycle [15].

As a result, these simulators are considerably faster than the event-driven ones. However, they provide incomplete information about the circuit, since they do not evaluate the delays and propagation times when evaluating each clock cycle. So, if a circuit has timing problems, a cycle-accurate simulator will not be able to notice them, making necessary the use of an even-driven simulator at some stage to evaluate the existence of timing problems. All these characteristics make the cycle-accurate simulators best suited for large circuit simulation, like CPUs, when simulation speed is an important factor.

Most cycle-accurate simulators use a 2-state model (0 or 1) to calculate the values of the signals through the circuit. A typical event-driven simulator uses a more complex model, with more states (adding states like undefined, unknown or high-impedance) [16]. This means that cycle-accurate simulators have to make assumptions when the signals may have a value different from 0 or 1 (for example, a signal that was uninitialized). While this speeds up the simulation process, it also might be prone to produce wrong results.

From all the cycle-accurate simulators available, the most used one is probably Verilator. Verilator is an open source simulator that compiles synthesizable Verilog RTL, generating cycle accurate C++ and SystemC models. For each circuit, Verilator compiles a different model. These models are then linked to a test bench, being executed in order to generate the simulation. Verilator does not only translate Verilog code to C++ or SystemC. Instead, it compiles the code into a much faster optimized and thread-partitioned model, which is in turn wrapped inside a C++/SystemC module [17].

3.3 Performance Comparison

When comparing event-driven and cycle-accurate simulators, not only their working principles differ, but also their performance. Event-driven simulators are typically slower than cycle-accurate simulators. This happens because the algorithms used are more complex in event-driven simulators, having to implement event scheduling and timing evaluation. The events are taken sequentially, being propagated through the circuit until it reaches a steady state. New events resultant from the output changes are added, meaning that the same element might be evaluated multiple times during the same time step due to the feedback from some signals. This whole process requires a considerable amount of time (and computational power), being the main reason behind the lower speed of event-driven simulators, when compared with cycle-accurate simulators.

The cycle-accurate simulators are typically the fastest type of simulators. This happens because the simulation algorithm is simpler, evaluating each logic element of the circuit in a clock cycle only once. They do this evaluation for each clock cycle, without taking into consideration the propagation times and delays within the cycle. There are also some simplifications that help improving the simulation speed, like the use of a 2-state model (0 or 1) to calculate the values of the signals through the circuit, instead of a model with multiple states [16].

In Figure 3.5 a graphic comparing the performance of the most popular HDL simulators available in the market [18] is shown. To run these benchmarks, a slightly modified model of the Motorolla M68K processor was used. All the simulators shown were run in a general purpose computer with an AMD

Phenom 9500 2.2GHz processor, DDR2 667 Memory and running the SuSE 11.1 operating system. The benchmark measures the number of cycles that a simulator can run in a fixed amount of time, so a higher result means that the simulator has a better performance.

From the analysis of the benchmark results, it can be concluded that Verilator is considerably faster than the other tested simulators, both in 32 and 64 bits versions. Cadence NCSim is almost 2 times slower than Verilator, while Synopsis VCS is 3.5 times slower. Icarus Verilog is the slowest simulator tested, being almost 80 times slower than Verilator. The benchmark results shown should only work as reference, given their limitations: the versions of the simulators used are already outdated, the same happening with the hardware and operating systems used in the general purpose computers. Also, this benchmark only evaluates the performance in one model (Motorolla M68K processor), instead of using multiple models in order to provide more accurate results.

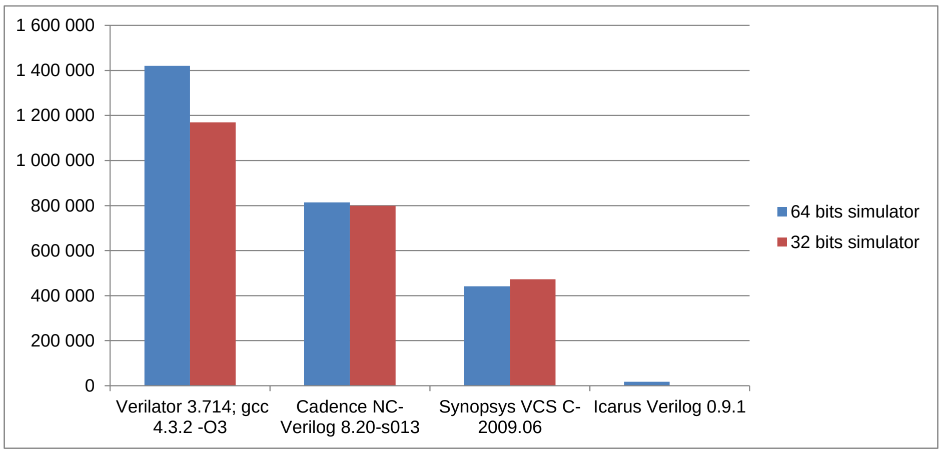

Figure 3.5: Benchmark results for different HDL simulators (higher means faster) [18].

The benchmark runs for the results shown in Figure 3.5 were made in single-threaded mode. However, most commercial simulators (including the ones analysed) support simulations in multi-threaded mode. This consists in creating multiple threads for the different processes and distributing them through multiple cores in a chip or even across multiple chips, that will be executed in parallel. The threads are usually inter-dependent, so there should be a synchronization process when transferring data between threads. This process is time consuming, since different threads have different execution times, requiring the faster threads to hold on until the slowest thread finishes [8].

Despite the concurrent nature of the Verilog statements, it isn't feasible to parallelize the totality of the statements in a model, since the number of statements is much greater than the amount of threads available, and also because this would require a great amount of synchronization processes between the different threads. Instead, the Verilog top-level code is divided in multiple subsystems, with each one being simulated as a different thread.

The parallelization of HDL simulations is the solution found to achieve considerable performance improvements in HDL simulators. This happens because through the years the algorithms used in the simulators were optimized to such a point where there were no further optimizations available, so the only viable solution was to start using parallelization techniques.

Chapter 4

CGRA Simulation

As referred before in this work, CGRA architectures have some big differences when compared with dedicated circuits. This poses difficulties in some areas, with one of these areas being the simulation of the developed architectures. Unlike a dedicated circuit, in a CGRA the reconfigurable interconnects between the functional units allow to build zillions of different datapaths at run-time. This is an overhead compared to simulating the various datapaths separately without simulating the reconfigurable infrastructure.

For FPGAs this problem is even worse as the reconfigurable infrastructure is fine grain and there are small Look Up Tables (LUTs) instead of high-level FUs like in CGRAs, and even more programmable interconnects. For this reason the FPGA fabric is never simulated, only the circuits that are mapped onto it are.

As a result, simulating CGRAs with event-driven simulators will result in lengthy simulation times. Consequently, finding a valid alternative to simulate CGRAs (in this particular case, the Versat architecture) could save an important amount of time and money during the development of applications for this type of architectures.

Using cycle-accurate simulators could be a good alternative to speed up CGRA simulations. As seen in the previous chapter, the use of a cycle-accurate simulator like Verilator could cut considerably the simulation time, reducing it, at least, by a factor of 2 (see Fig 3.5). It also has the advantage of having no additional cost, since Verilator is open source. However, the CGRA (or any other circuit) should not be tested exclusively with a cycle-accurate simulator, since their results don't have in account the propagation delays inside the CGRA, and also make some simplifications regarding the signals state. This is specially true if the CGRA is in constant development as is the case for Versat.

Another alternative is doing simulations at a high-level, instead of doing them at the RTL level. High-level simulation techniques have been applied in different types of circuits, like processors, with good results. However, for CGRA there is an additional difficulty compared to regular processors for this type of simulations because of the reconfiguration process. Two examples of approaches to high-level simulation are presented next: one that proposes a cycle-accurate simulator [19], and another one that proposes a framework for high-level simulation of CGRAs [20].

4.1 High-level cycle-accurate simulator

The high-level cycle-accurate approach proposed in [19] introduces the concept of a timing-constrained datapath. In CGRAs there is not a well defined pipeline structure due to the reconfigurable interconnects. The pipeline is formed on the fly with registers or memories being used to save the intermediate results between the different datapaths.

With this in mind, a register-centric synthesis technique is used using a timing-constrained datapath rooted at a chosen register. For the example in Figure 4.1, the datapath is rooted at the register R0, with the different paths being tracked down within the timing constraint given. These datapaths can finish either on a circuit input or on an intermediate register.

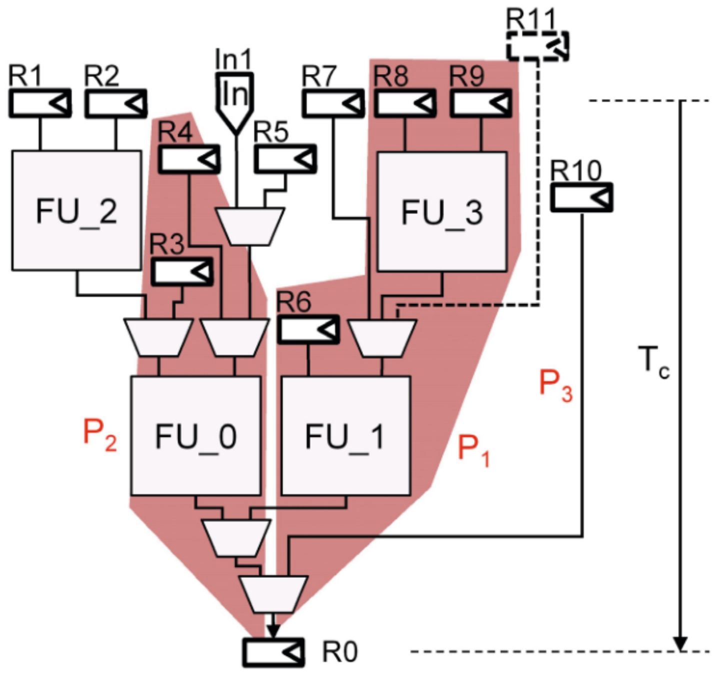

Figure 4.1: Timing-constrained datapath example [19].

The timing-constrained datapaths will be extracted and used by the compiler, being converted into a pattern graph of FUs. In this way, the simulation becomes a signal flow graph simulation, being performed via a routine that updates the value of all registers and output ports on each clock cycle. For the register/port udpate routine the obtained graph will be traversed, starting on the root register, and evaluating all the possible paths. This means that the simulator only does the necessary operations for the existent configuration bits, and also that the worst case for the simulation execution time will be proportional to the number of configuration bits available on the longest path of the flow graph.

The main simulation routine uses three main components: Input Read, Register/Port Update (described previously) and State Update. The Input Read is responsible for reading all the inputs on each clock cycle, and the Register/Port Update for updating all the values of registers that need to be updated.

To evaluate the performance of this simulator, multiple simulations were made, using two different CGRAs: CGRA-1 (modelled similarly to ADRES architecture [21]) and CGRA-2 (similar to [22]), and running six different application kernels (3 on each CGRA). Those kernels were run with different timing constraints, being noted that, while in CGRA-1 the simulation time increased linearly with the timing constraint, in CGRA-2 the simulation time did not increase so much. This happened because the CGRA-2 had a well defined pipeline architecture, along with less complex interconnections.

The simulations speed was compared with some commercially available simulators (Synopsys VCS-MX 2011.03 and Verilator 3.853), using a system with an AMD Phenom II X4 955 CPU and 8 GB of memory. To run the simulations the timing constraint was set to 3 FUs, since the performance of the kernels is the same as without timing constraint. Both the generated high-level simulator and Verilator were compiled with GCC 4.4.7. The obtained results are shown in Figure 4.2, normalized with the simulation time for the high-level simulator.

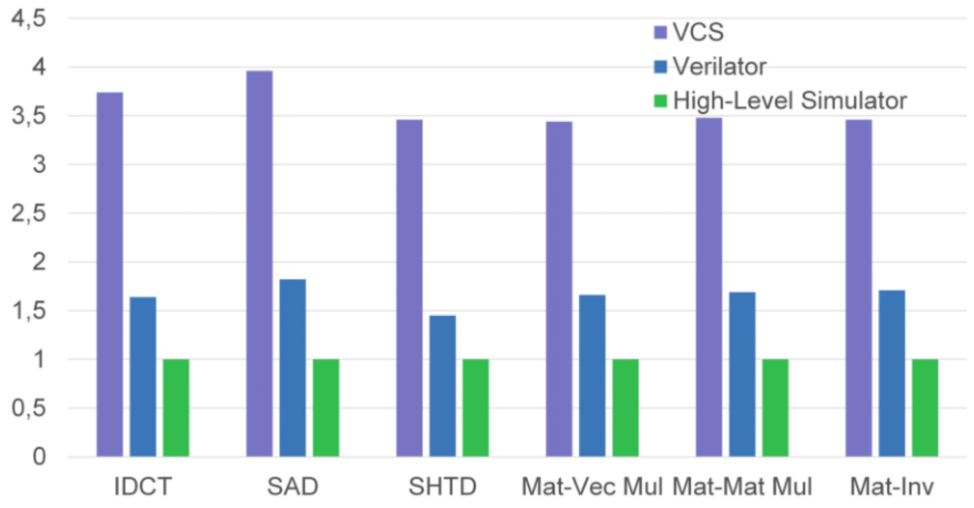

Figure 4.2: Performance comparison between the high-level and other commercially available simulators [19].

From the results in Figure 4.2 it can be seen that the high-level simulator is almost 4 times faster than Synopsys VCS and around 1.5 times faster than Verilator. This was already expected given that, while the high-level simulator only considers the values in the registers after each execution cycle, the other simulators also evaluate the intermediate signals.

4.2 CGRA high-level simulation framework

The framework presented in [20] provides a high-level simulation and optimization solution for a given mesh-based CGRA. It accepts applications written in C, generating the corresponding VHDL description for the target CGRA, and it can be divided into two main parts: C to netlist transformation and netlist to CGRA mapping.

The C to netlist transformation relies on GeCoS [23], an open-source compiler mainly used for ASIPs (Application Specific Instruction Set Processors) to compile the C code, generating a data flow graph that represents the original C code. This was made using GeCoS direct acyclic graphs generation capabilities. To transform the generated graphs into netlists for the target CGRA, a parser was created. In the resulting netlist, each ALU corresponds to a node in the original graph.

For the netlist to CGRA mapping, the previously generated netlists are placed into the target CGRA, using a simulated annealing algorithm. The placement and routing algorithms for netlist to CGRA mapping used are independent of the CGRA architecture, so they can be used for exploring different architectures. In the end an estimation of the area used is calculated.

The performance of the developed framework was only compared with FPGAs, using an open-source high-level simulation tool targeted at FPGA generation. However, instead of comparing the CGRA simulation performance with the FPGA simulation performance, it would be more useful (in the scope of this report) to compare how this new framework performs simulating a CGRA when compared with other commercial HDL simulators, using the same architecture.

Chapter 5

Conclusions

In this report, a study of the state-of-the-art of the current HDL simulation was made with the perspective of developing a new simulation environment for the Versat CGRA architecture, which is the subject of the thesis dissertation.

The Versat architecture, a minimal CGRA with self-generated partial reconfiguration, has been introduced. This architecture comprises a Data Engine and a Configuration Module to implement the runtime reconfigurable datapaths characteristic of CGRAs. Versat has a complete graph topology, which allows a high level of configurability. Additionally, Versat features a Controller, Program Memory, a Register File, a DMA, in a modular setup.

5.1 Achievements

In the analysis of the state-of-the-art of HDL simulators, two main types have been identified: event-driven and cycle-accurate simulators. While the cycle-accurate ones have the advantage of providing faster simulations, they have the disadvantage of not taking into account the different delays and propagation times inside a circuit, which makes them less reliable.

The application of the two main types to CGRA simulation has been analyzed. It has been concluded that event-driven simulation should be used in early stage development or simulation of individual modules but that for complete system simulation it is preferable to use cycle-accurate simulators to speed up verification.

Lastly, high-level simulation techniques that detach themselves from the above traditional techniques have been discussed and two different approaches to this topic have been presented. They bring the advantage of simulating the CGRA behaviour from a higher level than traditional RTL simulators, evaluating just the signals exchanged between the different functional units, instead of also looking to the intermediate signals.

Through this work, not only was it possible to study and understand the advantages and disadvantages of the different RTL simulators, but it was also possible to assess their performance, by analysing the results in [18, 19]. From these benchmarks it was seen that Verilator, an open source cycle-accurate simulator, has a clear advantage over the other commercially available HDL simulators.

The candidate spent a considerable time working with Versat in the scope of an industrial internship at company IObundle, Lda, and participated in a project with a real customer. In this project the candidate programmed Versat in assembly participating in the acceleration of MP3 encoding. This experience is invaluable for making correct design choices when developing the Versat simulation environment.

5.2 Future Work

The work to be developed in the thesis consists in creating a simulation environment for the Versat architecture. This simulation environment must be faster than the one that is currently used (based on NCsim), allowing for fast software development and debugging before running it on an FPGA for more exhaustive testing. This way, the environment that is going to be developed will be based in Verilator, given its performance advantage over the other simulators. The work that will be performed is shown in Table 5.1, together with the scheduling for each task.

Table 5.1: Planning of the work that will be performed in the thesis

Work Planning	**Scheduling**
Study Verilator documentation	18/02 - 22/02
Perform simulations with Verilator using simple circuits	23/02 - 04/03
Study the changes needed to the simulation environment in order to simulate Versat with Verilator	05/03 - 09/03
Apply the changes and perform the first simulations	10/03 - 10/04
Create a testbench for the simulation environment	11/04 - 29/04
Test & Debug the simulation environment	30/04 - 20/05
Write the Thesis report	21/05 - 01/07

A high-level simulator that works at the functional unit level instead of the RTL is going to be developed by another student. It will use a high-level simulator specially developed for the Versat architecture, similarly to [19], extracting the data from the memories and executing the desired high-level operations (no bit-level operations) by reading the configuration bits. This will allow for an even better performance but also has a disadvantage: architecture changes in Versat would also require changes in the simulator architecture. Note that Verilator is excellent at solving this problem as RTL changes can simply be recompiled. This kind of high-level approach does not evaluate any kind of timing in the circuit, but this is also not needed once the circuit is silicone proven and its frequency of operation is known.

Bibliography

[1] J. D. Lopes and J. T. de Sousa. Versat, a coarse-grained reconfigurable array with self-generated partial reconfiguration. Unpublished, 2018.

[2] J. D. Lopes and J. T. de Sousa. Versat, a minimal coarse-grain reconfigurable array. In D. I., C. R., B. J., and M. O., editors, *High Performance Computing for Computational Science – VECPAR 2016*, pages 174–187. Springer, 2016. doi:10.1007/978-3-319-61982-8_17.

[3] J. D. Lopes, R. Santiago, and J. T. de Sousa. Versat, a runtime partially reconfigurable coarse-grain reconfigurable array using a programmable controller. Jornadas Sarteco, 2016.

[4] R. Santiago, J. D. Lopes, and J. T. de Sousa. Compiler for the versat reconfigurable architecture. REC 2017, 2017.

[5] Shine: Super fast fixed-point mp3 encoding, 2018. URL `https://github.com/toots/shine`.

[6] Icarus verilog, 2018. URL `http://iverilog.icarus.com`.

[7] Incisive enterprise simulator, 2018. URL `https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html`.

[8] T. S. Tan and B. A. Rosdi. Verilog hdl simulator technology: A survey. *Journal of Electronic Testing*, 2014. DOI:10.1007/s10836-014-5449-5.

[9] S. Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Prentice Hall, 2003.

[10] M. Gunes, M. A. Thornton, F. Kocan, and S. A. Szygenda. A survey and comparison of digital logic simulators. In *Midwest Symposium on Circuits and Systems*. IEEE, 2005. doi:10.1109/MWSCAS.2005.1594208.

[11] Synopsys vcs, 2018. URL `https://www.synopsys.com/verification/simulation/vcs.html`.

[12] Mentor modelsim, 2018. URL `https://www.mentor.com/products/fv/modelsim/`.

[13] D. M. Lewis. A hierarchical compiled code event-driven logic simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(6):726–737, June 1991. doi:10.1109/43.137501.

[14] J. T. de Sousa, C. A. Rodrigues, N. Barreiro, and J. C. Fernandes. Building reconfigurable systems using open source components. REC 2014, 2014. doi:10.13140/2.1.3133.2483.

[15] A. Khandelwal, A. Gaur, and D. Mahajan. Gate level simulations: verification flow and challenges, March 2014. URL `https://www.edn.com/design/integrated-circuit-design/4429282/Gate-level-simulations--verification-flow-and-challenges`.

[16] J. Bennett. *High Performance SoC Modeling with Verilator*. Embecosm, February 2009.

[17] Introduction to verilator, 2018. URL `https://www.veripool.org/projects/verilator/wiki/Intro`.

[18] Verilog simulator benchmarks, 2018. URL `https://www.veripool.org/projects/veripool/wiki/Verilog_Simulator_Benchmarks`.

[19] A. Chattopadhyay and X. Chen. A timing driven cycle-accurate simulation for coarse-grained reconfigurable architectures. In K. S. et al, editor, *Applied Reconfigurable Computing*, pages 293–300. Springer, 2015. doi:10.1007/978-3-319-16214-0 24.

[20] M. A. Pasha, U. Farooq, M. Ali, and B. Siddiqui. A framework for high level simulation and optimization of coarse-grained reconfigurable architectures. In W. S., B. A., B. K., and C. L., editors, *Lecture Notes in Computer Science*. Springer, 2017. doi: 10.1007/978-3-319-56258-2_12.

[21] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins. Architecture exploration for a reconfigurable architecture template. In *IEEE Design & Test of Computers*, volume 22, pages 90–101. IEEE, 2005. doi:10.1109/MDT.2005.27.

[22] X. Chen, A. Minwegen, Y. Hassan, D. Kammler, S. Li, T. Kempf, A. Chattopadhyay, and G. Ascheid. Flexdet: Flexible, efficient multi-mode mimo detection using reconfigurable asip. In *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2012. doi:10.1109/FCCM.2012.22.

[23] L. L'Hours. Generating efficient custom fpga soft-cores for control-dominated applications. In *IEEE International Conference on Application-Specific Systems, Architecture Processors*. IEEE, 2005. doi:10.1109/ASAP.2005.37.