TÉCNICO LISBOA

Simulator for the RV32-Versat Architecture

João César Martins Moutoso Ratinho

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

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

Examination Committee

Chairperson: Prof. Francisco André Corrêa Alegria
Supervisor: Prof. José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Marcelino Bicho dos Santos

November 2019

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I want to thank my supervisor, Professor José Teixeira de Sousa, for the opportunity to develop this work and for his guidance and support during that process. His help was fundamental to overcome the multiple obstacles that I faced during this work.

I also want to acknowledge Professor Horácio Neto for providing a simple Convolutional Neural Network application, used as a basis for the application developed for the RV32-Versat architecture.

A special acknowledgement goes to my friends, for their continuous support, and Válter, that is developing a multi-layer architecture for RV32-Versat. When everything seemed to be doomed he always had a miraculous solution.

Finally, I want to express my sincere gratitude to my family for giving me all the support and encouragement that I needed throughout my years of study and through the process of researching and writing this thesis. They are also part of this work.

Thank you.

Resumo

Esta tese apresenta um novo ambiente de simulação para a arquitectura RV32-Versat baseado na ferramenta de simulação Verilator. A arquitectura RV32-Versat consiste no processador PicoRV32, com arquitectura RISC-V, ligado ao Versat. Este novo ambiente de simulação apresenta vantagens significativas quando comparado com os ambientes de simulação mais tradicionais que usam simuladores baseados em eventos. A primeira vantagem é a rapidez: o novo ambiente é significativamente mais rápido, poupando assim tempo no processo de desenvolvimento de novas aplicações para a arquitectura RV32-Versat. A segunda vantagem é o custo: o Verilator é disponibilizado com uma licença gratuita, enquanto os simuladores baseados em eventos típicos necessitam de licenças caras, difíceis de justificar para pequenas empresas e projectos. A terceira e última vantagem é o suporte directo para a co-simulação de *software* e *hardware*. Os simuladores baseados em eventos falham neste ponto, mas o novo ambiente baseado no Verilator resolve este problema, permitindo uma boa integração através do uso de C++ ou SystemC. Este novo ambiente de simulação é o resultado de um estudo detalhado, por um lado, dos diferentes tipos de simuladores existentes, e por outro lado, da arquitectura RV32-Versat, também apresentada nesta tese.

Palavras-chave: Matrizes Reconfiguráveis de Grão Grosso, Ambiente de Simulação, Verilator, Simulação de Matrizes Reconfiguráveis de Grão Grosso, Simulação de Alto Nível, Co-Simulação

Abstract

This thesis presents a new simulation environment for the RV32-Versat system using the Verilator simulation framework. The RV32-Versat system consists of the PicoRV32 RISC-V processor connected to the Versat Coarse-Grain Reconfigurable Array (CGRA) architecture. This new simulation environment presents significant advantages over the typical simulation environments using commercial event-driven simulators. The first advantage is speed: the new environment is considerably faster, therefore saving valuable time when developing applications for the RV32-Versat architecture. The second advantage is cost: Verilator is available under a free open-source licence, while the typical event-driven simulators require very expensive licences, hard to justify for small companies and projects. The third and last advantage is the easy support for hardware and software co-simulation. The event-driven simulators are lacking in this field, but the new environment using Verilator solves this problem, allowing a seamless integration through C++ or SystemC. This new simulation environment is the result of a detailed study of the state of the art of Central Processing Unit (CPU) and CGRA simulation and the RV32-Versat architecture, also presented in this thesis.

Keywords: Coarse-Grain Reconfigurable Arrays, Simulation Environment, Verilator, CGRA Simulation, High-Level Simulation, Co-Simulation

Contents

List of Tables

List of Figures

List of Acronyms

AGU Address Generation Unit

ALU Arithmetic Logic Unit

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

ASIP Application Specific Instruction Set Processor

BRAM Block Random Access Memory

CGRA Coarse-Grain Reconfigurable Array

CM Configuration Module

CNN Convolutional Neural Network

CPI Cycles Per Instruction

CPU Central Processing Unit

DE Data Engine

DMA Direct Memory Access

DSP Digital Signal Processor

FPGA Field-Programmable Gate Array

FU Functional Unit

HDL Hardware Description Language

ISA Instruction Set Architecture

LED Light Emitting Diode

LUT Lookup Table

RTL Register-Transfer Level

UART Universal Asynchronous Receiver-Transmitter

UUT Unit Under Test

VCD Value Change Dump

VPI Verilog Procedural Interface

Chapter 1

Introduction

1.1 Motivation

In any digital circuit simulation is fundamental to verify if the circuit is working properly in the different stages of its development. The motivation for this thesis became clear when I started working with Versat about a year ago, in the context of an internship in which I had the opportunity to participate in the development of an MP3 encoder. During that project I was able to understand the main limitations of the simulation environments using traditional simulators: they are slow with complex simulations, many times require costly licences and they provide a difficult integration of software / hardware co-simulation, many times leading to the use of ad hoc solutions.

These limitations gave me the motivation to look for a solution that would address them, therefore making the simulation process during a project faster, cheaper, more efficient and with an integrated support for software and hardware co-simulation. That way, industrial projects like the one I participated in could be done in a much more efficient way, which can be very important for companies with little resources.

1.2 Objectives

The main goal of this thesis is to develop a new simulation environment for the RV32-Versat system that is able to overcome the typical problems inherent to using simulation environments based on traditional simulators, as referred in the previous section. The RV32-Versat system consists of a Versat CGRA, controlled by a RISC-V PicoRV32 processor and has been developed in the context of two masters theses, including this one.

Consequently, the new simulation environment has three main objectives. The first one is being considerably faster than simulating the RV32-Versat architecture with commercial simulators. This can be particularly useful when developing new applications for this architecture, since it saves time during the debug process. The second objective is being cheaper than traditional simulators, saving resources. Finally, the third objective is to support hardware and software co-simulation in an integrated evironment rather than using special interfaces and/or ad hoc solutions.

Fulfilling these objectives required studying the current state of the art of the simulators and the RV32-Versat architecture. This way, it was possible to understand what types of simulators are more suitable for the Versat architecture. To validate the simulation environment, a convolutional neural network application for recognizing hand-written digits has been developed. This application has also been used to benchmark different commercially available simulators.

1.3 Author's Work

This work was developed during an internship at IObundle, Lda, the company that developed the Versat architecture. The internship started in September 2018 and 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.

The work presented here is the result of the work of a few people: both the PicoRV32 processor and the Versat CGRA had been previously created, and the RV32-Versat architecture was developed in collaboration with another student that is developing a multi-layer architecture for RV32-Versat. Therefore, the main contributions made by the candidate for this architecture were the removal of the controller previously used by Versat, called PicoVersat [1], replacing it by the PicoRV32 processor, the development of the application presented in Section 2.4 and the creation of a new simulation environment.

1.4 Thesis Outline

This thesis is composed of 6 more chapters. In the second chapter the RV32-Versat architecture is presented, in order to contextualize the other chapters. The state of the art of CPU and CGRA simulators is detailed in chapter 3. In the fourth chapter, a simulation environment using typical commercial simulators is presented and their problems are reported. In the fifth chapter the new simulation environment is presented, which overcomes the problems of the previous one. In the sixth chapter the results with the simulation environments are shown and, finally, in the seventh chapter the conclusions are presented.

Chapter 2

The RV32-Versat Architecture

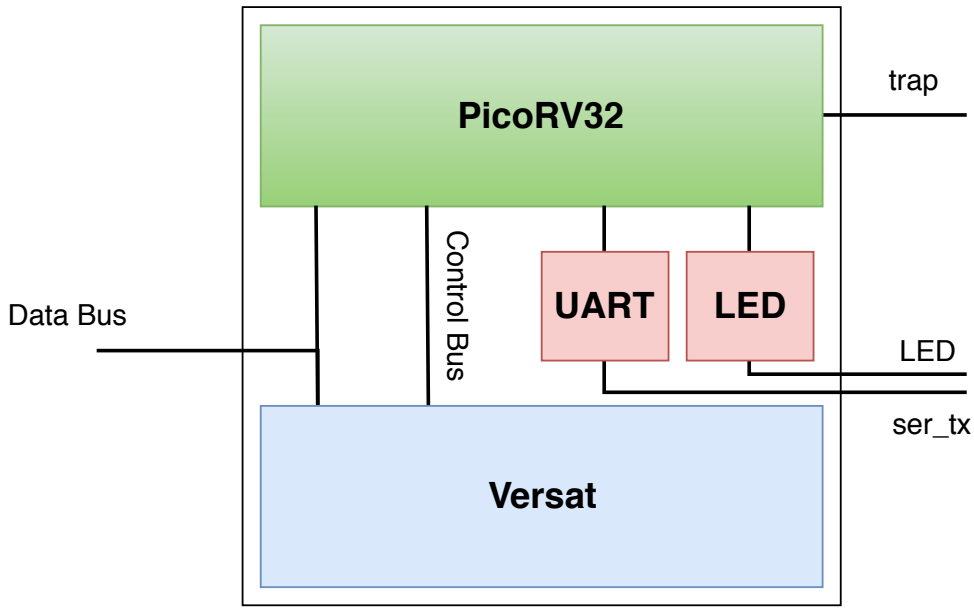

Figure 2.1: RV32-Versat top-level diagram.

The RV32-Versat architecture was developed during this thesis, by a team of 3 people including the author. The starting point was the Versat CGRA [2–6]. A Coarse-Grain Reconfigurable Array (CGRA) is a collection of programmable Functional Units (FU) and embedded memories connected by programmable interconnects, forming the reconfigurable array. For a given set of configuration bits, the reconfigurable array forms a hardware datapath that can execute an algorithm orders of magnitude faster than a conventional CPU. This type of architectures can be used as hardware co-processors to accelerate algorithms that are time/power consuming in general purpose CPUs.

The Versat CGRA could only be programmed in assembly language which, despite being great for optimising algorithms, it was a problem when creating complex and lengthy algorithms. Thus, by removing its controller (called PicoVersat) and replacing it by the PicoRV32 processor [7], with a RISC-V Instruction Set Architecture (ISA), this problem was solved since now Versat can be programmed in the C/C++ language via the PicoRV32 processor. Versat will work as a peripheral of the processor for accelerating parts of the program, using an interface of C++ classes created for this purpose, as is explained in Section 2.3.1.

The RV32-Versat architecture is shown in Figure 2.1, and it comprises two main modules: the PicoRV32 processor and the Versat CGRA. There are also two additional modules in the RV32-Versat: the Universal Asynchronous Receiver-Transmitter (UART) and the Light Emitting Diode (LED). Both modules are used for debugging. In this context the UART module is particularly useful, given that it allows to print values in the computer terminal when simulating or testing the circuit. The LED provides an even simpler debug mechanism when everything else fails.

As can be seen in the figure, the databus can be used to access the Versat's memories with values from the PicoRV32 controller, external host or Direct Memory Access (DMA) master or, in simulation, by the testbench. This bus is used to upload Versat with data to be processed and to download data already processed by Versat. In case a testbench is used, hexadecimal files containing the input and output data can be used, which is useful for debugging when developing new applications. The data bus is comprised of the signals data databus_sel (global bus select), databus_rnw (read / not write signal), databus_addr (address), databus_data_in (input data to PicoRV32) and databus_data_out (output data to PicoRV32).

In RV32-Versat, Versat and the other cores are connected as memory mapped peripherals of the PicoRV32 processor. This means that they are accessed/programmed via the PicoRV32 bus, having dedicated memory addresses, as shown in Table 2.1, where the RV32-Versat memory map is shown.

Table 2.1: RV32-Versat memory map.

Peripheral	Memory address
UART module	0x100
LED module	0x101
Versat	0x110
Versat databus	0x120

Other changes were made to Versat in order to adapt it for the RV32-Versat architecture. One of these changes was the creation of a C++ driver, as detailed in Section 2.3.1, that allows the user to program Versat using C++ classes. For that, the user just needs to include the respective `versatUI.hpp` header file and link the driver functions. This is a major improvement, since previously Versat could only be programmed in Assembly, whereas now it can be programmed using its C++ classes inside of a C or C++ code that runs in the PicoRV32 processor. This is more user-friendly, as shown in the application example of the Section 2.3.2.

As detailed in Section 2.2, the PicoRV32 processor is far from being a fast processor, since its main purpose is to be a small processor with low power consumption. However, the impact that this has in the overall performance of the system is attenuated by the use of Versat to speed up the time consuming parts of the algorithms implemented in this system. This means that the RV32-Versat is expected to have a performance similar to processors with more resources (in terms of area and power consumption).

2.1 The Versat architecture

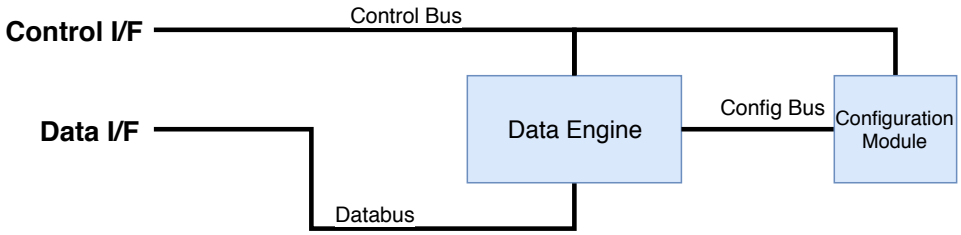

Figure 2.2: Versat top-level entity.

The Versat CGRA architecture [2–6] used in RV32-Versat is shown in Figure 2.2, and it consists of two main modules: the Data Engine and the Configuration Module. The Controller, Program Memory and Control Register File that were previously included in the Versat architecture were removed for the RV32-Versat system, given that in this new architecture the Versat works as a peripheral, being controlled by the PicoRV32 processor.

The Versat core has a Control and a Data interface. The Control interface is used by PicoRV32 processor to instruct Versat to load and execute programs. On the other hand, the Data interface is used to load and read data from Versat.

Versat user programs can use the Data Engine (DE) to carry out data intensive computations. To perform these computations, the PicoRV32 processor writes DE configurations to the Configuration Module (CM), through the host or memory interface, or simply restores configurations previously stored in the CM. The PicoRV32 can also load the DE with data to be processed or save the processed data back in the external memory using the Data interface. This interface can also be used to initially load the Versat program or to move CGRA configurations between the core and the external memory.

2.1.1 Data Engine

The DE has a flexible topology, in which the user can configure the number of FU and their respective types. In Figure 2.3 it is shown a DE example with 15 FUs. The DE is a 32-bit architecture with the following configurable FUs: Arithmetic and Logic Unit (ALULite), Multiplier Accumulator (Muladd), Barrel Shifter (BS) and dual-port 16kB embedded memories (MEM).

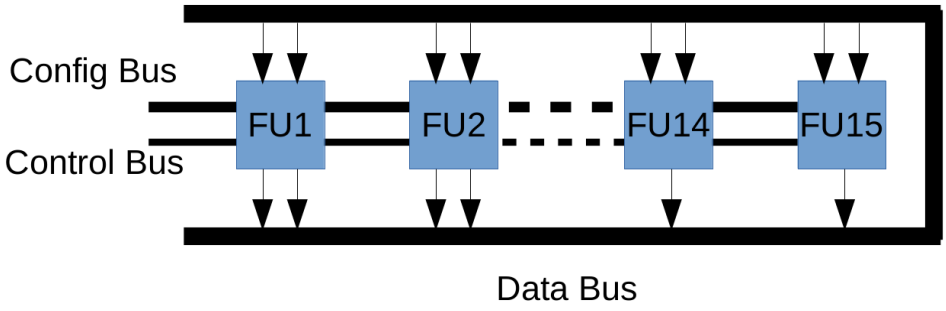

Figure 2.3: Versat data engine.

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. The Data Bus allows for a maximum of 19 sections, and 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 FU's have different latencies due to pipelining, as shown in Table 2.2 . As a result, when configuring a datapath those latencies must be taken into account and compensated by configuring the Delay parameter of the memories, as explained in Table 2.6.

Table 2.2: Latencies of the Versat FUs.

Functional Unit	Latency
Memory	1
Barrel Shifter	1
ALULite	2
Muladd	3

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 [3]. It also facilitates the configuration of the different datapaths, because the user doesn't need to remember or check what is connected to what since all the FUs are interconnected.

ALULite

The ALULite used in Versat has 2 data inputs (A and B), an output and 4 configuration bits. One of the configuration bits is used to create an internal feedback loop from the output to the input A when activated (if the bit is set to '0' there will be no feedback), meaning that an ALULite it can support up to 8 different operations. A list of this operations is shown in Table 2.3.

Table 2.3: ALULite operations.

FNS	Operation	Operands
000	Addition	Y = A + B
001	Subtraction	Y = B - A
010	Signed compare	Y[31] = (A>B)
011	Multiplexer	Y = (A<0)? B: 0
100	Signed maximum	Y=max(A,B)
101	Signed minimum	Y=min(A,B)
110	Logic OR	Y = A | B
111	Logic AND	Y = A & B

When the feedback loop is activated the input A of the ALULite can be used to implement conditional statements in Addition, Subtraction, Multiplexer, Signed maximum and Signed minimum operations. While the input A is not negative the ALULite keeps accumulating the values coming to input B, stopping when the input A has a negative value.

Muladd

The Multiply-Accumulate (Muladd) unit used in Versat has 3 data inputs (A, B and O), an output and 3 configuration bits. The input O is used to control the accumulation operations: if this input is different than 0, then the Muladd keeps accumulating the multiplication results, otherwise the multiplier doesn't accumulate the results. The Table 2.4 shows the supported operations.

Table 2.4: Muladd operations.

FNS	Operation	Operands
000	MUL	Y = A $\times$ B
001	MUL_DIV2	Y = (B $\times$ A)[63:32]
010	MACC_16Q16	Y = Y + (B $\times$ A)[47:16]
011	MACC	Y = Y + B $\times$ A
100	MACC_DIV2	Y = Y + (B $\times$ A)[63:32]
101	MSUB	Y = Y - (B $\times$ A)<<1
110	MSUB_DIV2	Y = Y - B $\times$ A
111	MUL_LOW	Y = (Y + B $\times$ A)<<32

Barrel Shifter

The Barrel Shifter has 2 inputs (D for data and S for shift) and an output. This functional unit can perform left and right shits, as shown in the Table 2.5, where the supported operations are listed. Two configuration bits are used for this FU, meaning that up to four operations are available.

Table 2.5: Barrel shifter operations.

FNS	Operation	Operands
00	SHR_A	Y = D >> S
01	SHR_L	Y = D >>> S
10	SHL	Y = D << S

Memories

Each one of the dual-port memories used in Versat has a data input, a data output and an address input, as shown in Figure 2.4. The number of memory positions is configurable, with each position having a fixed size of 32 bits.

Each one of the ports 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 AGU support two levels of nested loops and can start execution with a programmable delay, so that circuit paths with different latencies can be synchronized.

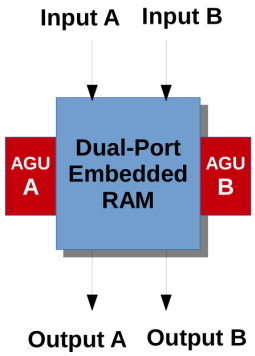

Figure 2.4: Versat dual-port embedded memory with AGU.

Each AGU has multiple configurable parameters, as detailed in Table 2.6, in order to control the address sequences generated. An example address sequence is shown in Figure 2.5 , with the AGU configured with Start=10, Per=5, Duty=3, Incr=2, and Shift=-5, Delay=2, and Reverse=0. Normally an AGU is used for generating an address sequence for the memory port. However, the port can be configured to output the generated sequence to the DE, where it can be used.

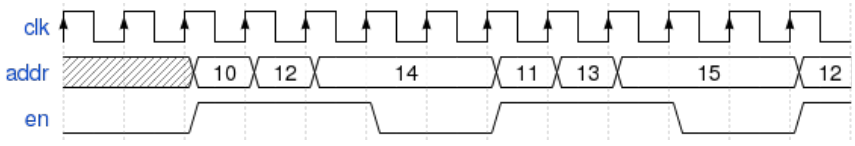

Figure 2.5: AGU output for Delay=2, Per=5, Duty=3, Start=10, Incr=2, Shift=-5 and Reverse=0.

Table 2.6: AGU parameters.

Parameter	Size (bits)	Description
Start	11	Memory start address
Per	5	Number of iterations of the inner loop, aka Period
Duty	5	Number of cycles in a period (Per) that the memory is enabled
Incr	11	Increment for the inner loop
Iter	11	Number of iterations of the outer loop
Shift	11	Additional increment in the end of each period. Note that Per+Shift is the increment of the outer loop
Delay	5	Number of clock cycles that the AGU must wait before starting to work. Used to compensate different latencies in the converging branches of the configured hardware datapaths
Reverse	1	Bit-wise reversion of the generated address

The embedded memory ports also have their own configuration fields: one for selecting/disabling the input generated sequence (Sel) and another for bypassing the AGU (Ext). When the memory input is disabled the respective port becomes enabled for reading. Otherwise, when the input is selecting a data source the port becomes enabled not only for writing but also for reading the word previously stored at that address.

The configuration for bypassing the AGU (Ext=1) enables the memory port to use as address values computed by datapath. In this case the port address is input by the other port of the same memory. This feature provides Versat with the capability of working with pointers.

2.1.2 Configuration Module

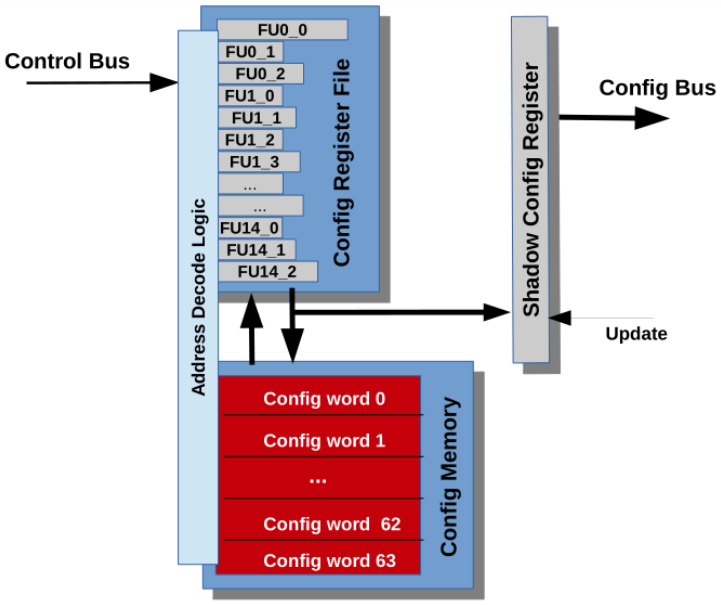

Figure 2.6: Versat configuration module example.

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 (PicoRV32) 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 CM example is illustrated in Figure 2.6, 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 FU 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.

2.2 The PicoRV32 Architecture

The CPU architecture used in the RV32-Versat system is the PicoRV32 architecture [7]. It implements the RISC-V RV32IMC instruction set and is publicly available under a free and open hardware licence. This processor has a small size (750-2000 Lookup Tables (LUT) in 7-Series Xilinx Architecture), and was originally meant to be used as an auxiliary processor in Field-Programmable Gate Array (FPGA) designs and Application-Specific Integrated Circuit (ASIC). Also, its high maximum frequency of operation (250-450 MHz on 7-Series Xilinx FPGAs) means that it can be integrated in most existing designs without crossing clock domains.

The Cycles Per Instruction (CPI) numbers for the individual instructions can be found in Table 2.7. It can be seen that the average CPI is approximately 4, depending on the mix of instructions in the code, meaning that the PicoRV32 is far from being a fast processor. However, this was already expected, since the main purpose of this processor is to have a small area and power consumption. Also, the high maximum clock frequency allowed by this processor reduces the performance impact of the high CPI figure.

2.3 Programming RV32-Versat

As said before, the RV32-Versat can be programmed using C/C++, and the code can be compiled using the GNU g++ compiler. To correctly configure Versat for accelerating parts of the program, the C++ classes forming the Versat C++ driver, detailed in the next sub-section, must be included and used.

Table 2.7: PicoRV32 CPI for different instructions.

Instruction	CPI
direct jump (jal)	3
ALU reg + immediate	3
ALU reg + reg	3
branch (not taken)	3
memory load	5
memory store	5
branch (taken)	5
indirect jump (jalr)	6
shift operations	4-14
multiplication	40
division	40
multiplication (MULH)	70

2.3.1 C++ Driver

In order to allow the user to program Versat in an intuitive way using PicoRV32 as a controller, a driver was created. It consists of a C++ header file called `versatUI.hpp`, a C++ file file called `versat_func.cpp` and a python script called `xdictgen`. The `versatUI.hpp` file, included in the `versat_func.cpp` file, contains multiple C++ classes, one for each type of functional unit, that contain constructors that allow configuring the different functional units used in Versat, as shown in Figure 2.7. Each class was implemented with multiple functions, this way allowing the configuration of only the necessary parameters (instead of all the parameters), taking advantage of Versat's partial reconfiguration capabilities.

When developing an application for the RV32-Versat, the user can configure the Versat by using these classes directly in the C code that runs in the PicoRV32 processor. For example, in order to configure the ALULite parameters it can simply write `CALULite alulite0 (base, opa, opb, fns);`, where base, opa, opb, fns are the configuration fields, and then write `alulite0.writeConf();` to write the configuration to alulite0. As said before, partial reconfiguration is also supported, saving time in the functional units reconfiguration. If the user wants, for example, to use the same ALULite with the same inputs but performing a different operation, it just needs to write `alulite0.setFNS(fns)`, where fns is the desired operation.

Besides the C++ classes, this file also includes the declaration of the function `defs`, used to initialize the constants that will hold the different configuration addresses for each one of the configured Versat functional units.

In the file `versat_func.cpp` the functions used to configure and use the Versat are declared. This way, the Versat can be easily configured directly from the C code that runs in the PicoRV32 processor by calling this predefined functions. The functions are the following:

- **versatInit:** Function that checks if Versat is operational by writing something to its dummy register and reading it back. If the read value is equal to the written one, then Versat is working properly.

- **versatWriteConf:** Configures the Versat with the specified parameters. For example, if an ALULite is configured using `CALULite alulite0 (CONF_ALULITE[0], sMEMA[0], sMEMB[3], ALULITE_ADD);`, this configuration can be written by calling `ALULite0.writeConf();`.
- **versatClearConf:** Used to clear the current configurations of the Versat by writing ”0” to the `CONF_CLEAR` Versat address. This function must be used before configuring a new Versat kernel.
- **versatRun:** Function that runs the Versat DE. First, it checks if the DE is ready by checking if the register that holds the data engine status is ”0”. If it is, then it writes the value ’1’ to the register that runs the data engine, otherwise it keeps waiting until the DE is ready.
- **databusWrite:** Function used to write a value to a Versat memory directly from the program
- **databusRead:** Loads a value from a Versat memory directly to the program.

```
class CALULite {
   public:

   int base;
   int opa;
   int opb;
   int fns;

   CALULite (){
   }
   CALULite (int base, int opa, int opb, int fns) {
      this->base = base;
      this->opa = opa;
      this->opb = opb;
      this->fns = fns;
   }

   void writeConf() {
      RAM_SET32(VERSAT_TOP_BASE, (base + ALULITE_CONF_SELA), opa);
      RAM_SET32(VERSAT_TOP_BASE, (base + ALULITE_CONF_SELB), opb);
      RAM_SET32(VERSAT_TOP_BASE, (base + ALULITE_CONF_FNS), fns);
   }
   void setOpA(int opa) {
      RAM_SET32(VERSAT_TOP_BASE, (this->base + ALULITE_CONF_SELA), opa);
      this->opa = opa;
   }
   void setOpB(int opb) {
      RAM_SET32(VERSAT_TOP_BASE, (this->base + ALULITE_CONF_SELB), opb);
      this->opb = opb;
   }
   void setFNS(int fns) {
      RAM_SET32(VERSAT_TOP_BASE, (this->base + ALULITE_CONF_FNS), fns);
      this->fns = fns;
   }
};
```

Figure 2.7: Example of a functional unit class declared in `versatUI.hpp`.

The `xdictgen` python script was created to read the Versat Verilog include files, extract the constants, and create a file called `versat_defs.hpp` with all these constants. This file will then be included in the `versatUI.hpp` file and in the C code that runs in the PicoRV32 processor, making the Versat configuration constants available in these files. This way, when configuring the different functional units available, the user does not need to remember the value of the different constants, instead she/he just needs to know their name.

2.3.2 Basic Programming

In order to better explain how the RV32-Versat is programmed, this section shows an example program that writes values into two Versat memories, adds them and then saves the results in another memory. The example program is shown in Figure 2.8.

```
void main() {
   // setup uart
   uart_reset();

   //for 100MHz
   uart_setdiv(868);
   uart_wait();

   // init Versat
   uart_puts("Init Versat\n");
   defs();
   versatInit();
   versatClearConf(); //clean previous configurations

   //Write data to MEM0A and MEM1A
   databuswrite(ENG_MEM[0], 0, 1);
   databuswrite(ENG_MEM[0], 1, 2);
   databuswrite(ENG_MEM[0], 2, 3);
   databuswrite(ENG_MEM[1], 0, 4);
   databuswrite(ENG_MEM[1], 1, 5);
   databuswrite(ENG_MEM[1], 2, 6);

   //This kernel adds 3 numbers in Mem0 [0:2] with 3 numbers
   //of MEM1 [0:3] and saves the result in Mem2 [0:3]

   //Config MEM0 A to read data
   //mem0A (base, start, iter, incr, delay, per, duty, sel)
   CMem mem0A (CONF_MEMA[0], 0, 3, 1, 0, 1, 1, 0);
   mem0A.writeConf();

   //Config MEM1 A to read data
   CMem mem1A (CONF_MEMA[1], 0, 3, 1, 0, 1, 1, 0);
   mem1A.writeConf();

   //Config ALULITE
   CALULite alulite0 (CONF_ALULITE[0], sMEMA[0], sMEMA[1], ALULITE_ADD);
   alulite0.writeConf();

   //Config MEM2 A to save data
   CMem mem2A (CONF_MEMA[2], 0, 3, 1, 3, 1, 1, sALULITE0);
   mem2A.writeConf();

   //Run
   versatRun();

   uart_puts("Done\n");
}
```

Figure 2.8: Example program for RV32-Versat.

The program starts in the `main` function, as usual in C or C++ programs, and the first thing it does is call the UART functions that in this case will be used to print to the computer terminal. The function `uart_reset` initializes the UART, the function `uart_setdiv` sets the baud rate and the function `uart_wait`, as the name indicates, waits until the UART is ready.

After this the program proceeds to initialize the Versat, calling the functions `defs`, `versatInit` and `versatClearConf` that, as explained previously in Section 2.3.1, initialize the Versat and its data bus. When this is finished the memories of the Versat are loaded with constants needed by the program (values 1, 2 and 3 for addresses 0, 1 and 2 of memory 0 and values 4, 5 and 6 for addresses 0, 1 and 2 of memory 1, respectively). In this case, this was done directly in the C code, using the function `databusWrite`. However, for large sets of constants this is not practical and the constants should instead be written into the memories via the testbench of the RV32-Versat.

When the memory writing is finished the Versat's data engine is configured to perform the desired operation, in this case adding the values previously stored in the memories and saving them in memory 2. To do this, the classes defined in the C++ driver (Section 2.3.1) are used. This way, the memories are configured to perform 3 iterations, starting at address 0 and incrementing by one address at the end of each iteration (see Section 2.1.1 for more details on the configuration parameters of the memories).

The ALULite is configured to receive memory 0 and memory 1 as inputs, and to have memory 2 as an output. The operation to be performed by the ALULite is also configured, in this case it is the `ALULITE_ADD` operation. As explained in Section 2.1.1, this operation simply outputs the result of the addition of the two inputs.

The last functional unit to be configured is memory 2 (note that this order was chosen just to simplify the explanation, the configuration order does not matter). Memory 2 is configured with the same parameters as memories 0 and 1: 3 iterations, starting at address 0 and incrementing by one address at the end of each iteration.

After the configuration is finished function `versatRun` is called in order to run the data engine with the current configuration. When it finishes, the `uart_puts` function is called to print in the pc terminal the message "Done".

2.4 Application Example

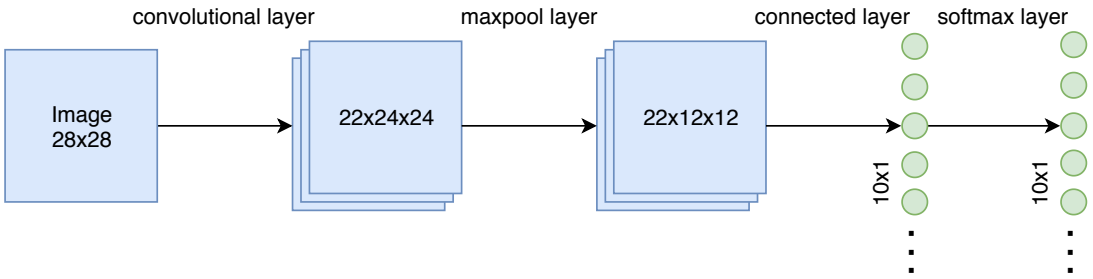

Figure 2.9: CNN architecture implemented in RV32-Versat.

To fully test the system and the simulation environment an application using a previously trained Convolutional Neural Network (CNN) for digit classification in different images was developed for the RV32-Versat. This application was based on a simple CNN developed by the professor Horácio Neto for the Hardware/Software Co-Design course, and it consists of the following 4 layers, as shown in Figure 2.9:

- **Forward convolutional layer:** This layer tests each feature map in each different position of the image. This is done through a convolution, by multiplying each one of the feature's pixels by the respective image's pixel, and then adding these multiplications. At the end, this is divided by the total number of pixels of the feature map.
- **Maxpool layer:** Here, each one of the matrices generated by the previous layer (24×24) is scanned, and the maximum value of each 2×2 region is found. In the end, 22 matrices with dimension 12×12 are obtained.
- **Fully connected layer:** The previously generated matrices go through a convolution process, being transformed into votes. These votes are expressed as weights, determined by the network's previous training. In the end a 10 element vector is generated (one position for each number from 0-9), with each value representing the amount of votes for that number.
- **Softmax layer:** This final layer use the previously obtained 10 elements and selects the position (and consequently the number) with the most votes. That number is the final guess made by the network. Also, the vector is normalized and a probabilistic distribution is made, being obtained the probability that the number guessed is correct.

To adapt this application for RV32-Versat multiple changes were made. The first change was to convert the program to fixed point (16Q16), since Versat only works with fixed point numbers. Then, the forward convolutional and the fully connected layers were rewritten using the Versat functions in order to accelerate the execution of these layers. The other layers were kept almost unchanged, being executed in the PicoRV32 processor, since their execution time was minimal when compared with the layers run in Versat.

After being implemented and fully tested, this application was used to benchmark the different simulators after the simulation environment was developed. The results are shown in Section 6.1.

From the four layers used in the implementation (forward convolutional, maxpool, fully connected and softmax) only two were implemented using the Versat CGRA (forward convolutional and fully connected), with the two other layers being implemented in software, since their execution time was minimal when compared with the other two layers. As a result, the following functional units were used in Versat:

- 4 dual-port memories with 65.5 KB each (262 KB in total)
- 2 Muladds
- 1 ALULite

In the following sections the datapaths created in Versat to accelerate the execution of the forward convolutional and fully connected layers will be shown and explained, along with the FU's configurations.

2.4.1 Forward Convolutional Layer

This layer can be divided into three main parts: prepare the matrix A (`prepare_matrixA`), transpose the matrix B for and convolutionate it with A (`fixed_gemmBT`) and add the bias values and transpose the matrix (`fixed_add_bias`).

In the `prepare_matrixA` function the matrix that contains the image to be processed is transformed into a $25 \times 24 \times 24$ matrix, making it ready to be convolutioned with matrix B transposed. This transformation is made by copying the values of the original matrix, contained in memory 0, to multiple positions of the memory 3, as shown in Figure 2.10.

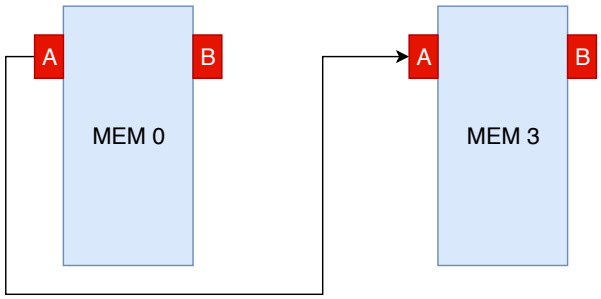

Figure 2.10: Datapath `prepare_matrixA`.

The matrix A is then convolutioned with the transposed matrix B using the Muladd, implementing the datapath shown in Figure 2.11. Both the matrix A and the matrix B are stored in memory 3 (positions 0-14399 and 15022-15571, respectively), and sent to the Muladd. The memory 1 is used to generate a counter between 0 and 25 that will be used to control the number of accumulations done in the Muladd. This way, after accumulating 25 multiplications the Muladd accumulator will be reset, with memory 0 saving the accumulation result in matrix C just before reset.

The matrix C previously obtained is transposed and sent to an ALULite, where the 22 bias values, saved in memory 3 (positions 0-21) will be added. The resultant matrix (Cbias) is saved in memory 3 through port A, as shown in Figure 2.12

2.4.2 Fully Connected Layer

This layer can be divided into two main parts: the convolution of matrix Cpool with matrix B (`fixed_gemm`) and the addition of the bias values to the resultant matrix (`fixed_add_bias`).

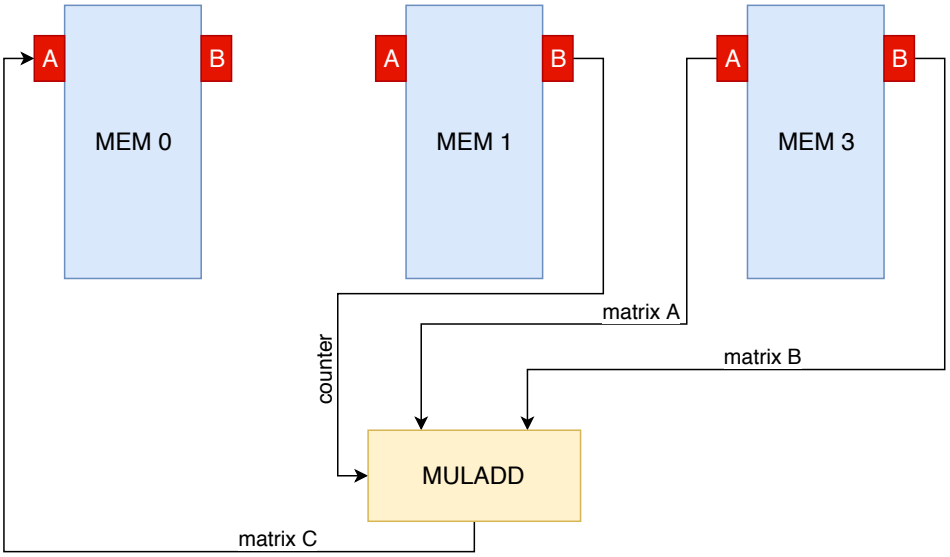

Figure 2.11: Datapath `fixed_gemmBT`.

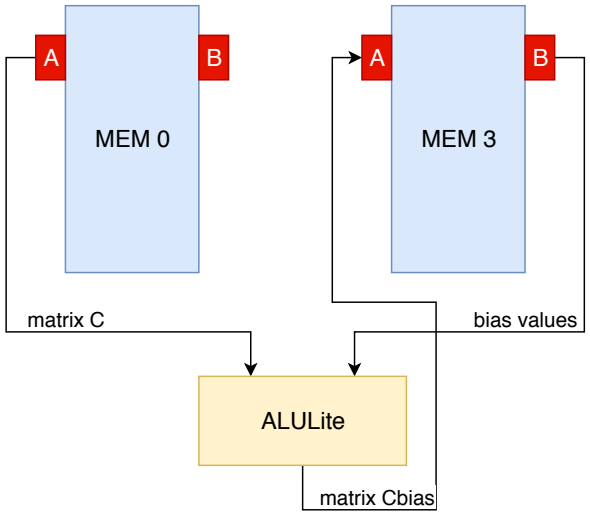

Figure 2.12: Datapath `fixed_add_bias` in forward convolutional layer.

In `fixed_gemm` the convolution of matrix Cpool with the weights is performed using two Muladds (Muladd 0 and Muladd 1), as shown in Figure 2.13. Matrix Cpool is stored in memory 0, while the weights are divided between memory 1 and memory 2 (the 32262 weights can not fit inside a single memory). Memory 0 is used to generate a counter between 0 and 3168 that will be used to control the number of accumulations done in the Muladds. This way, after accumulating 3168 multiplications the Muladd accumulator will be reset, with memory 3 saving the accumulation result (matrix matOUT) just before reset.

The use of two Muladds in this function allowed parallelization of the algorithm, cutting its execution time by half (when compared with a non-parallel run), and allowing it to be executed in a single run. If only a single Muladd was used, two different datapaths would be needed: one to compute the first half of matrix matOUT (using the weights stored in memory 1), and another to compute the second half (using the weights stored in memory 2).

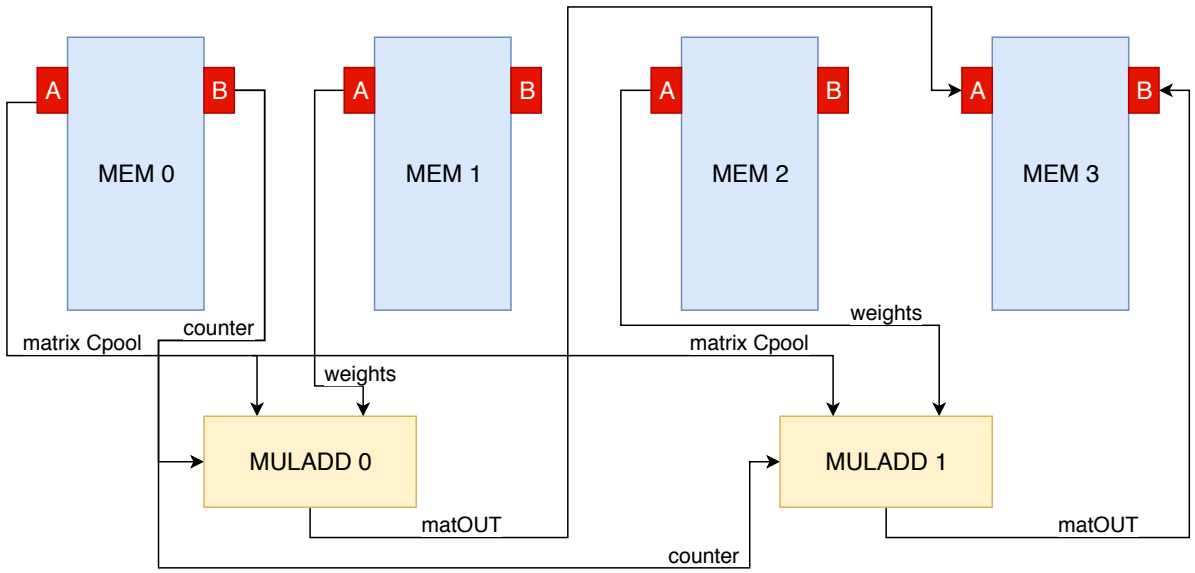

Figure 2.13: Datapath `fixed_gemm`.

The `fixed_add_bias` function is called again, but this time it will add 10 bias values to matrix matOUT without transposing it, with the resultant matrix ConnB being saved in the memory 0, as shown in Figure 2.14.

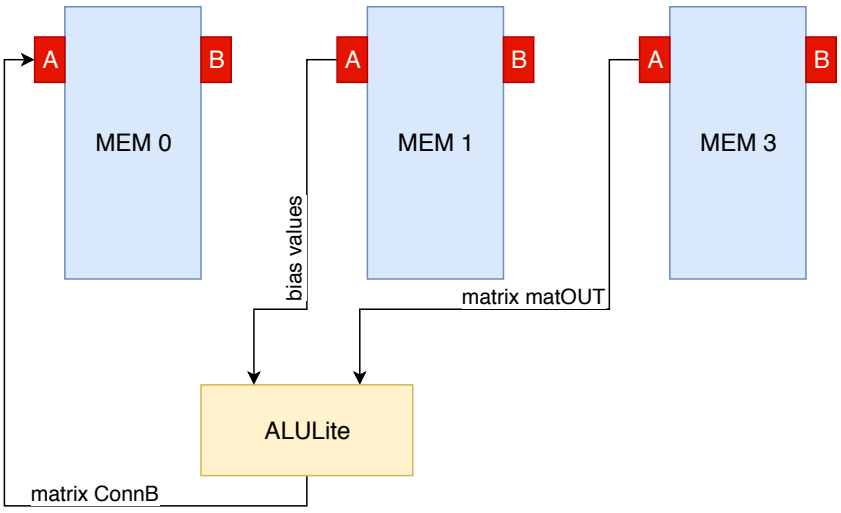

Figure 2.14: Datapath `fixed_add_bias` in fully connected layer.

Chapter 3

Simulating CPU/CGRA Architectures

Digital circuit 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 verify the circuit), they do not work in the same way. In this chapter a study of the state of the art of CPU and CGRA simulation is done in order to understand what are the best solutions to simulate the RV32-Versat architecture, presented in the previous chapter.

Typically, before simulating a circuit, a testbench is created. The testbench is a program, written in a Hardware Description Language (HDL) or in a programming language (like C++ or 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 generates the signals needed to make the circuit work properly. 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 signals match, it means that the circuit is working as intended at least for what it has been exercised for.

The results provided by the testbench might depend on the chosen simulator. This happens because 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.

As it will be seen in this chapter, from all the simulators studied the one that is the most suitable to reach the objectives defined for the new simulation environment of the RV32-Versat architecture is Verilator: it is faster than the more traditional event-driven simulators, it is inexpensive and allows the use of C++ and SystemC to write the testbenches, therefore simplifying the creation of a software and hardware co-simulation environment. Also, changes in the hardware architecture will not require changes in the simulator.

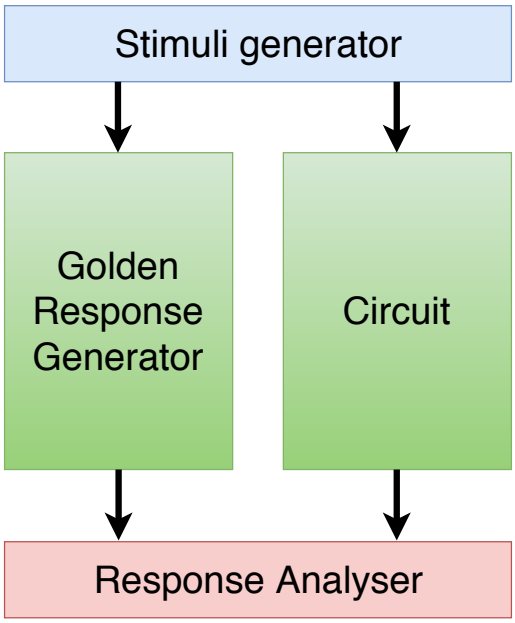

Figure 3.1: Testbench diagram.

3.1 Event-Driven Simulators

The first category of simulators are the event-driven ones [8–10]. Most of the commercially available simulators belong to this category, and they 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 that result from the output changes being added to the queue. This means that the same element might be evaluated multiple times during the same time step due to the feedback from some signals.

It is important to mention that during the simulation process there is 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.

Event-driven simulators are the most common type of simulators, including simulators like the Cadence NCSim [11], the Synopsys VCS [12], the Mentor Graphics ModelSim [13] or the free of charge open source Icarus Verilog [14]. 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. These 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 [15]. Icarus Verilog belongs to this category: it uses a compiler called iverilog to compile the HDL circuit description into the vvp assembly format, accepted by the simulator. After this the vvp file generated is run by the simulator to execute the simulation.

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 [15]. The most used simulators, like Cadence NCSim or Synopsys VCS, 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.

Register-Transfer Level (RTL) simulation is the most used method for circuit verification due to its reasonable accuracy [16]. However, in the last few years there has been a rising trend in the industry to run gate level simulations [17]. 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.

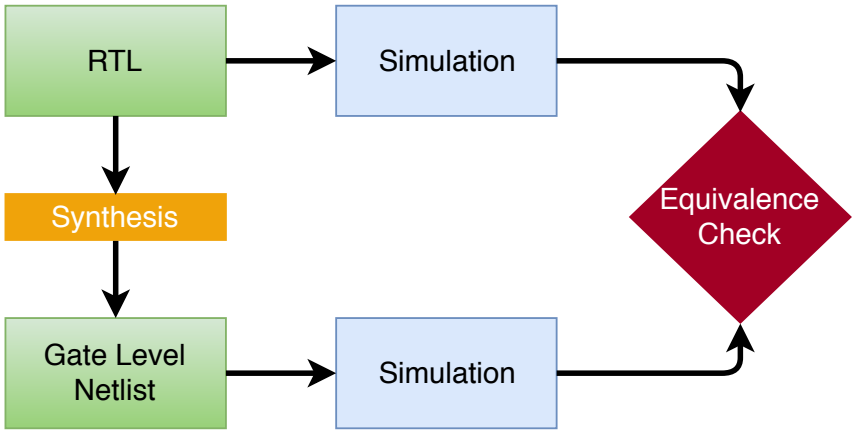

Figure 3.2: Gate level design flow diagram.

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 produced gate netlist behaves as expected, which means that no synthesis problems have been introduced neither by the designer nor by the synthesis tool.

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 [17].

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 event-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) [18]. 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 testbench, 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 [19], that can be used afterwards in a software and hardware co-simulation environment.

3.3 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 presented previously (event-driven and cycle-accurate), 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 behaviour 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 [17]. 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 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.

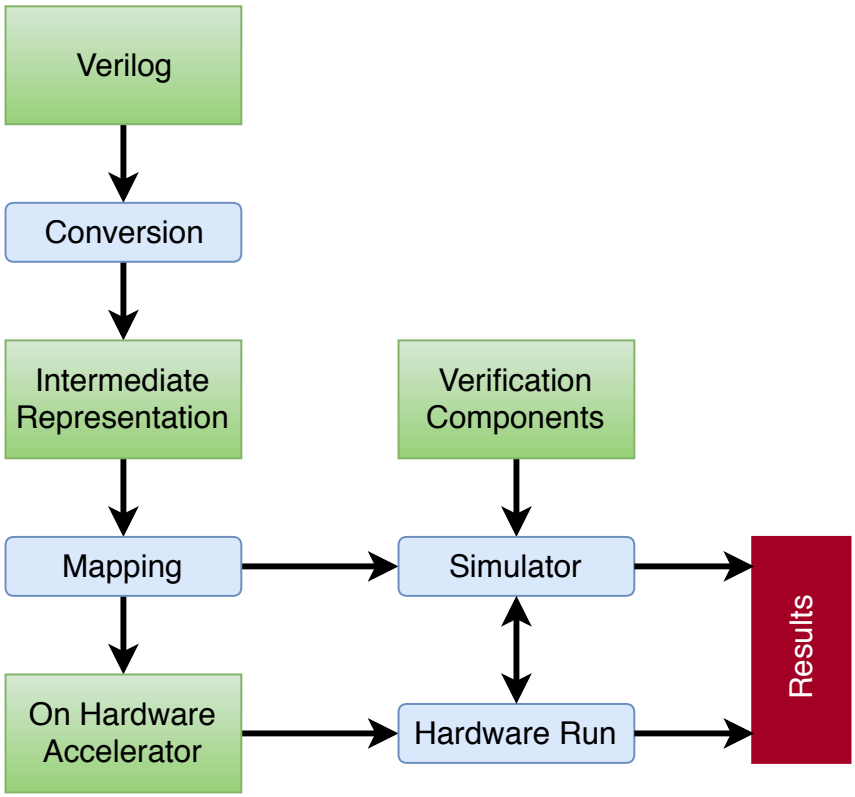

Figure 3.3: FPGA-based simulation flow [17].

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.

On the other hand, Emulator-based simulators rely on ASIC or FPGAs to run their simulations. When compared to an FPGA an ASIC offers limited reconfigurability, but with the advantage of a higher simulation speed.

This type of simulation offers the possibility of testing the software before having it implemented on chip, as the software application can run exactly as it would on the real chip in a real system. It also offers the possibility of testing more complex programs, that would take large amounts of time (sometimes even days) running 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.

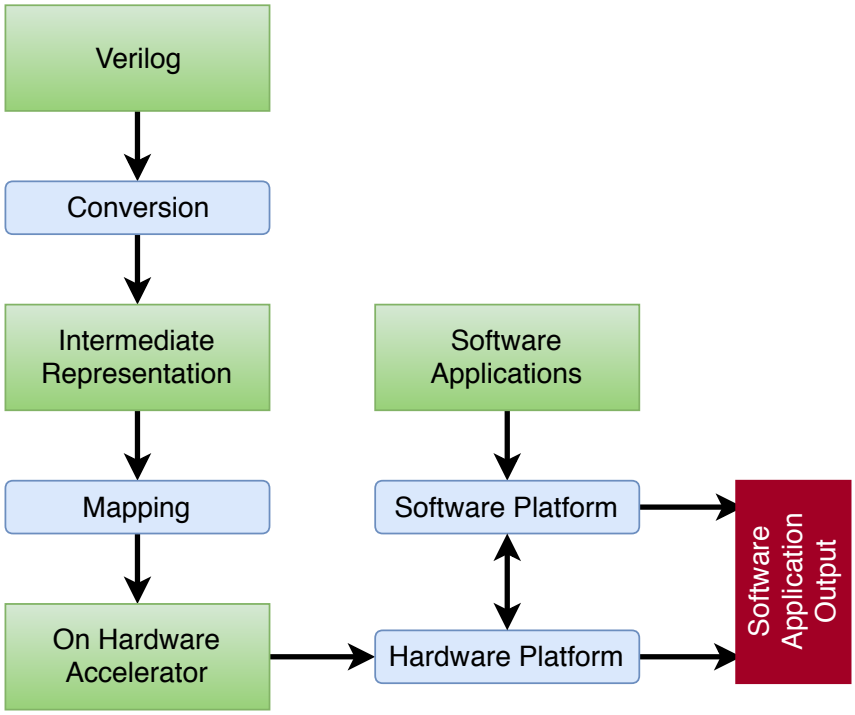

Figure 3.4: Emulator-based simulation flow [17].

3.4 Performance Comparison

In Figure 3.5 a chart comparing the performance of the most popular Verilog software-based simulators available in the market [20] is shown. To run these benchmarks, a slightly modified model of the Motorolla M68K processor is used as the design under test. All the simulators shown were run on a general purpose computer with an 2.2GHz AMD Phenom 9500 processor, 667MHz DDR2 Memory and running the SuSE 11.1 operating system. The benchmark measures the number of clock cycles that a simulator can run in a fixed amount of time, so a higher result means that the simulator has a better performance.

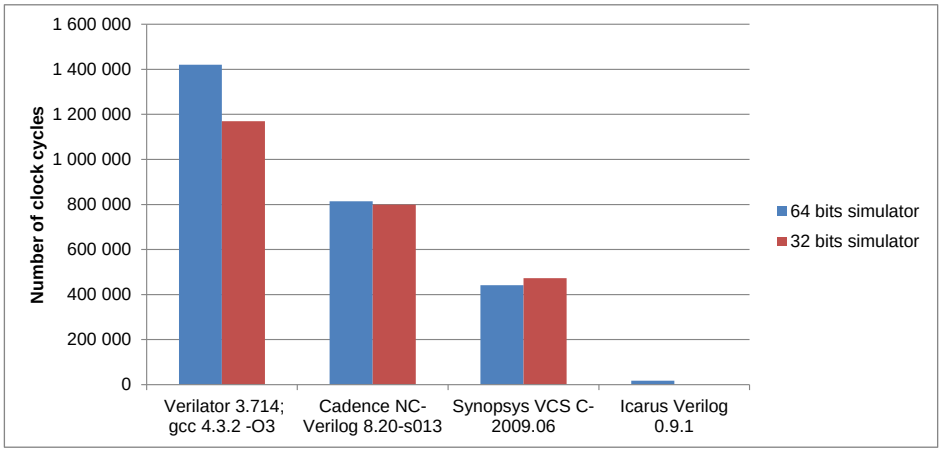

Figure 3.5: Benchmark results for different Verilog simulators [20].

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. This result was expected if we take into account that Verilator is a cycle-accurate simulator, while the other simulators are event-driven. Recall that event-driven simulators have lower performance despite offering more detailed simulations. Among the event-driven simulators, Icarus Verilog is the slowest since it is of the interpreter type while the others are of the compiled-code type. Therefore, Icarus has to deal with the overhead of code translation, as explained before.

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.

Also, the benchmark runs were made in single-threaded mode, when 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 over multiple cores in a chip or even across multiple chips, which execute 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 is not 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. Moreover, this would require a great amount of synchronization between the different threads, killing the possible speed-up. Instead, the Verilog top-level code is divided in multiple subsystems, with each one being simulated as a different thread.

Parallelization is the solution found to achieve considerable performance improvements in simulators. This happens because throughout the years the algorithms used in the simulators were optimized to such a point where it became difficult to find further optimizations. So the only viable solution was to start using parallelization techniques.

3.5 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, being one of these areas the simulation of the developed architectures. Unlike a dedicated circuit, in a CGRA the reconfigurable interconnects between the functional units allow building a multitude of different datapaths at run-time. This is an overhead compared to simulating the various datapaths separately without simulating the reconfigurable infrastructure.

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 considerably cut 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 do not take into account the propagation delays inside the CGRA, and also make some simplifications regarding the signal states. This is specially true if the CGRA is in constant development.

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 [21], and another one that proposes a framework for high-level simulation of CGRAs [22]. However, both approaches have a problem: they are specifically tailored for an architecture, so each time there is a significant change in the architecture, the simulators will also need to be changed. With Verilator, for example, that does not happen.

3.5.1 High-Level Cycle-Accurate Simulator

The high-level cycle-accurate approach that is proposed in [21] 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 3.6, 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.

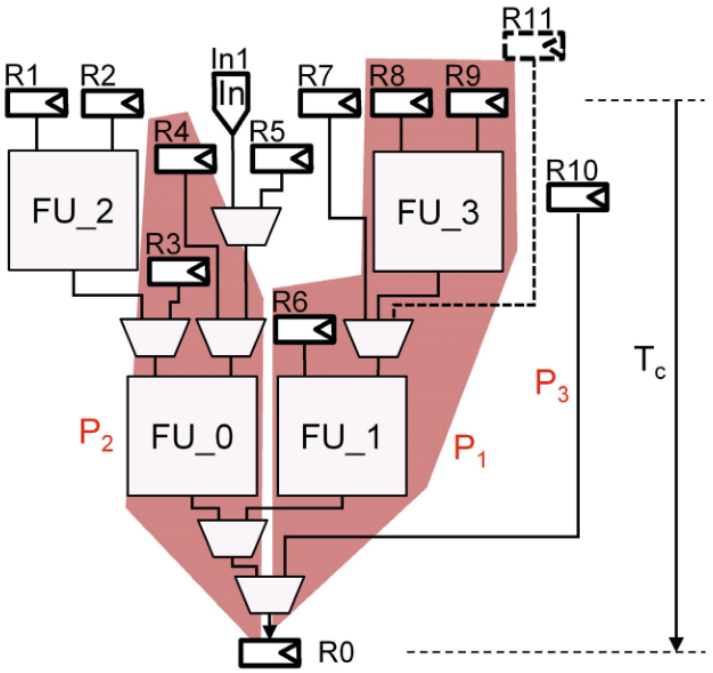

Figure 3.6: Timing-constrained datapath example [21].

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 and ports 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 [23]) and CGRA-2 (similar to [24]), and running six different application kernels (3 on each CGRA). The simulation 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 3.7, normalized with the simulation time for the high-level simulator.

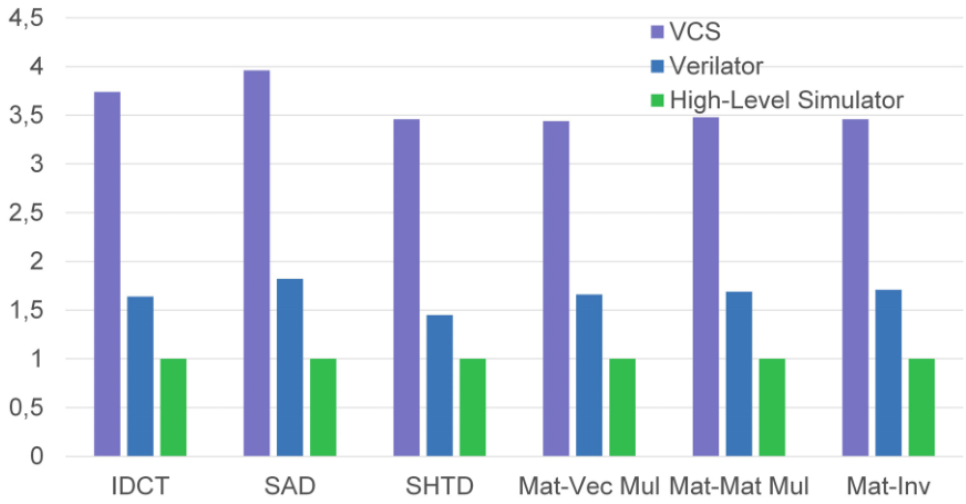

Figure 3.7: Performance comparison between the high-level and other commercially available simulators [21].

From the results in Figure 3.7 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.

3.5.2 CGRA High-Level Simulation Framework

The framework presented in [22] 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 [25], an open-source compiler mainly used for Application Specific Instruction Set Processors (ASIP) 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 Arithmetic Logic Unit (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 FPGA implementations. However, it would be more useful (in the scope of this thesis) to compare how this new framework performs simulating a CGRA when compared with other commercial HDL simulators, using the same architecture.

Chapter 4

Simulating The RV32-Versat Architecture Using Event-Driven Simulators

In this chapter, the simulation environment for the RV32-Versat architecture using commercially available event-driven simulators, more specifically the Cadence NCsim and Synopsys VCS simulators, is detailed. As explained in the previous chapter, event-driven simulators work by taking events sequentially, propagating them through the circuit until the circuit reaches a steady state [8–10]. They produce accurate simulation results with detailed timing information at the cost of the simulation speed.

4.1 Testbench

This type of simulators work with testbenches written in an HDL such as Verilog or VHDL. In Figure 4.1, an example of a simple testbench written in Verilog and used by NCSim and VCS is shown. This testbench has RV32-Versat as the Unit Under Test (UUT) and uses a clock period of 10 ns, meaning that the system is simulated at a frequency of 100 MHz. After 100 clock cycles with the reset enabled (`resetn=0`) the reset is disabled, and RV32-Versat starts running. The data bus external ports are not used, since the memories are initialized at compile time and no input hex file is loaded by the testbench. The simulation finishes when the trap signal is activated (with `resetn=0`), and if the flag VCD is passed when running the simulator a Value Change Dump (VCD) file will be generated. The trap signal is activated when there is an invalid memory access, which is useful for debug and for stopping the simulation, which is caused by the user software that intentionally accesses the memory at a non mapped location.

```
`timescale 1 ns / 1 ps

module system_tb;
   reg clk = 1;
   always #5 clk = ~clk;

   reg resetn = 0;

   initial begin
      `ifdef VCD
         $dumpfile("system.vcd");
         $dumpvars();
      `endif
      repeat (100) @(posedge clk);
      resetn <= 1;
   end

   wire         led;
   wire         ser_tx;
   wire         trap;
   reg          databus_sel;
   reg          databus_rnw;
   reg [14:0]   databus_addr;
   reg [31:0]   databus_data_in;

   system uut (
      .clk            (clk       ),
      .resetn         (resetn    ),
      .led            (led       ),
      .databus_sel    (databus_sel),
      .databus_rnw    (databus_rnw),
      .databus_addr   (databus_addr),
      .databus_data_in (databus_data_in),
      .ser_tx         (ser_tx    ),
      .trap           (trap      )
   );

   initial begin
      databus_sel = 0;
      databus_rnw = 1;
      databus_addr = 0;
      databus_data_in = 0;
   end

   always @(posedge clk) begin
      if (resetn && trap) begin
         $finish;
      end
   end
endmodule
```

Figure 4.1: Example testbench in Verilog for RV32-Versat.

4.2 Verilog VPI

The downside of using an HDL testbench with the event-driven simulators is their limited support for software and hardware co-simulation. One way to overcome this problem is to use the Verilog Procedural Interface (VPI) [26], a C-programming interface for Verilog that consists in a set of access and utility routines that are called from standard C programming language functions. These routines interact with the instantiated simulation objects contained in the Verilog design.

```
#include "vpi_user.h"

void example() {
   vpi_printf("Example function\n");
   return;
}
```

Figure 4.2: Example C code for VPI.

To better explain how the VPI works there is a simple example of a C function in Figure 4.2, called `example.c` that prints text in the terminal, using the `printf` function from the VPI. This function now needs to be associated with a system task. For this, a special data structure needs to be created, of the type `vpi_systf_data`. The code for the creation and registration of the system task can be seen in Figure 4.3. After this stage, the system task must be called in the Verilog testbench of the circuit to simulate. This can be done either in `initial` blocks or in `always` blocks. In Figure 4.4 an example using an `initial` block is shown. The last thing to do is linking the task with the simulator. This is usually made in the terminal when calling the simulator.

```
#include "example.c"

void hello_register()
{
   s_vpi_systf_data tf_data; //data structure

   tf_data.type     = vpiSysTask; //does not return value
   tf_data.tfname   = "$hello"; //name of the task
   tf_data.calltf   = hello_calltf; //pointer for this routine
   tf_data.compiletf = hello_compiletf; //pointer for the compiled instance
                                        //of the task

   vpi_register_systf(&tf_data); //pointer to the structure
}

//Register the new task
void (*vlog_startup_routines[ ]) () = {
   hello_register,
   0
;}
```

Figure 4.3: VPI task creation and registration.

```
module example ();

initial begin
   $hello;
   #10 $finish;
end

endmodule
```

Figure 4.4: Verilog code to invoke the task.

As it can be seen, although it is possible to do software and hardware co-simulation using Verilog testbenches, the process is not direct, making it a disadvantage in simulation environments based on event-driven simulators. If the low performance of this type of simulators is also taken into account (as it can be seen in the benchmarks in Section 6.1), it can be concluded that a simulation environment based on event-driven simulators does not reach the objectives pretended: it is not fast, the simulators licences are expensive and it does not provide a straightforward way of integrating hardware and software co-simulation. Therefore, an alternative is necessary. There is another event-driven simulator (Icarus Verilog) that could solve the cost problems, since it is free, but this simulator does not support the `generate` loops used in the RV32-Versat Verilog code and has a very low performance, as seen in Section 3.4.

Chapter 5

Simulating The RV32-Versat Architecture Using The New Verilator Environment

In this chapter, the new simulation environment for the RV32-Versat architecture using Verilator is detailed. The Verilator environment was designed to overcome the disadvantages of typical simulation environments based on the more traditional event-driven simulators, such as Cadence NCsim and the Synopsys VCS. As detailed in the previous chapters these simulators are relatively slow, require expensive licences and complicated the integration of hardware and software, many times leading to the use of ad hoc solutions.

As explained in Chapter 3, event-driven simulators schedule and process events sequentially, and thus produce more accurate simulation timing information at the cost of the simulation speed.

In contrast, Verilator is a cycle-accurate simulator and evaluates all logic elements of the circuit for every clock cycle using binary (0/1) logic. This produces less accurate results but the simulation is faster. This kind of simulator does not evaluate any kind of timing in the circuit. However, it can be used to quickly identify problems at an early stage or after a change. Additionally, if the architecture is proven in other simulators or in silicon, Verilator can be used to just capture behaviour modifications.

5.1 Working Principle

The simulation environment created for RV32-Versat is based in a set of GNU Makefiles [27]. In order to run the simulation environment the user must go to the root of the RV32-Versat repository and type the command `make simulator_name TEST=test_name`, where the `simulator_name` is the name of the desired simulator (Verilator) and `test_name` is the name of the folder of the test (driving software program) that must be placed inside of the folder `tests/test_name`. The test is compiled, the system memories are initialized with the program code and data and the system is simulated, following the flow shown in Figure 5.1. This flow will be followed for any program that is created and simulated for this architecture.

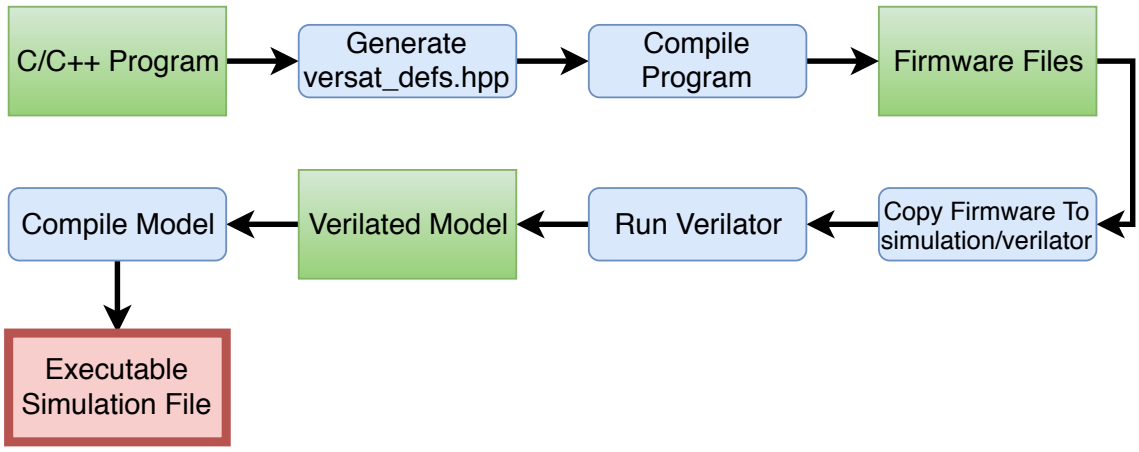

Figure 5.1: Simulation flow of the Verilator environment.

After the `make` command is executed, the Makefile will call another Makefile placed inside the folder `simulation/verilator` that will then call another Makefile placed inside the folder `tests/test_name`. This Makefile generates the firmware hex files containing the RISC-V program and, if necessary, the data to be load into the Versat memories via the testbench, by generating the `versat_defs.hpp` file and compiling the program using the RISC-V toolchain.

The generated firmware files will then be copied into the `simulation/verilator` folder where the respective Makefile was called. The next step is to call Verilator to generate the Verilated model, a C++/System C model obtained from the synthesizable Verilog files. This model is then linked to the testbench and compiled using GNU g++. The resulting executable simulation file will be run and perform the actual simulation.

An additional option was added that allows the simulation to be run in machines that do not have the RISC-V toolchain [28] installed. This option compiles the program via SSH in a remote machine where the toolchain is properly installed. With this option, the user does not need to install the toolchain in the computer where the simulator is installed, since installing the RISC-V toolchain can be a lengthy and complicated process. This option can be activated by adding the flag `TOOLCHAIN=FALSE` in the make command.

5.2 The Verilator Testbench

The Verilator testbench is considerably different from a typical HDL testbench, mainly because it is written in C++, SystemC or a compination of both. The use of these two languages instead of Verilog allows for much greater flexibility by seamlessly emulating the software of a host system that stimulates the simulated system. The testbench shown in Figure 5.2, written in C++, works essentially in the same way as the one shown in the previous chapter (Figure 4.1): it loads the hex files to the Versat memories via the data bus and then runs the program. The simulation finishes when the trap signal in the PicoRV32 processor is activated, together with an indication that the program has finished. Optionally, the simulation can output an hex file with the contents of the Versat memories using the data bus external ports.

```
#include "Vsystem.h"
#include "verilated.h"
#include "verilated_vcd_c.h"

vluint64_t main_time = 0;
double sc_time_stamp () {
return main_time;
}

int main(int argc, char **argv, char **env)
{
Verilated::commandArgs(argc, argv);
Verilated::traceEverOn(true);
Vsystem* top = new Vsystem;
VerilatedVcdC* tfp = new VerilatedVcdC;
top->trace (tfp, 99);
tfp->open ("waves.vcd");

top->clk = 0;
top->databus_sel = 0;
top->databus_rnw = 1;
top->databus_addr = 0;
top->databus_data_in = 1;

int t = 0;

while (!Verilated::gotFinish()) {
if (t > 200)
top->resetn = 1;
top->clk = !top->clk; //Toggle clock
top->eval();        //Evaluate the model
tfp->dump (t);      //Write to VCD file
t += 5;             //Increment clock
if (top->resetn && top->trap == 1)
Verilated::gotFinish(true);
}

tfp->close(); //Close the VCD
top->final(); //Finish the simulation
delete top;
exit(0);
}
```

Figure 5.2: Example testbench in C++ for RV32-Versat.

The C++ header file created by Verilator from the Verilog model (the Verilated model) must be included in the C++ testbench. After this, the `sc_time_stamp` function is invoked, so that the simulator knows the current time. The simulation setup is done in the `main` function: the runtime arguments are analysed (`commandArgs`), the computation of the traced signals is enabled (`traceEverOn`), the module to be simulated is declared (`Vsystem*top`), along with the Verilated VCD model (`VerilatedVcdC* tfp`) and the number of hierarchy levels to be traced are defined (`top->trace (tfp, 99)`). As with the Verilog testbench example, in this example the data bus external ports are switched off in order to keep the example simple.

The clock of the circuit is toggled inside the while cycle. For each time that this cycle executed, the clock signal will be negated, the simulation time will be checked (to verify if the reset has been disabled), the circuit will be evaluated (`top->eval()`), the signals will be written to the VCD file (`tfp->dump(t)`) and the clock signal time will incremented by 5 ns in each half period, resulting in a frequency of 100 MHz.The testbench will be executed until the (`Verilated::gotFinish`) condition is true. This will happen when the trap signal is activated and the reset disabled (resetn=1), making the program leave the while cycle. This will finish the simulation and exit the testbench.

Since the testbench is written in C++/SystemC, it is easy to do software and hardware co-simulation, given that the host application can be directly embedded in the testbench. This is a major improvement over other simulation environments using Verilog testbenches. This is not the only advantage of this new environment: the simulations using Verilator are considerably faster, as it can be see in the benchmark presented in the Section 6.1, performed for the CNN application presented in Section 2.4. Also, since Verilator is a completely free open-source project, no licence fees are due. This advantages suit perfectly the objectives defined for the new simulation environment. The major disadvantage of this new simulation environment is its lower precision when compared with environments using event-driven simulators. This happens because Verilator does not provide timing information, since it does not take into account propagation times and because Verilator uses a 2-state model, assuming that all the signals in a circuit have a value that can either be 0 or 1. However, this disadvantage is many times irrelevant in the context of the RV32-Versat architecture, since this architecture has already been extensively tested in FPGAs and with other simulators.

Chapter 6

Results

In this chapter, the experimental results for the new simulation environment developed for the RV32-Versat architecture are presented and discussed. In Section 6.1, the performace of the new Verilator-based simulation environment is compared against the two arguably best event-driven commercial simulators: Candece NCsim and Synopsys VCS, using the CNN application developed during this thesis. In Section 6.2, the FPGA implementation results of running the CNN algorithm on the RV32-Versat architecture are presented, in order to validate the results obtained with the new simulation environment.

6.1 Performance Results

In order to test the performance of the new simulation environment against the two mentioned commercial event-driven simulators, the CNN-based digit recognition application developed in this work (Section 2.4) is used as a benchmark and run on the RV32-Versat architecture.

The benchmark consists of two tests: (1) using the simulators debug mode, which enables internal assertions and debugging messages and generation of a VCD waveform file (debug+VCD mode); (2) running the simulators without debug and waveform file generation, and setting the -O3 g++ compiler flag when compiling the Verilator model, which reduces the simulation time at the cost of a slightly higher compile time (normal mode).

The tests were executed in a 64-bit machine, with an Intel i5-4430 processor and 8 GB of memory, running Ubuntu 16.04.3 LTS. The versions of the simulators used were the following: Verilator 4.014 (released in May 2019), Synopsys VCS 2017.03 (released in March 2017) and Cadence NCSim 13.10 (released in 2013). While Verilator was at the most recent stable version available, for the other simulators the versions used corresponded to the available software licences made available by the Europractice service. The Europractice service was launched by the European Commission in 1995 as a successor of the Eurochip program (1989-1995) to enhance European industrial competitiveness in the global market. Over the past 25 years, Europractice has provided the industry and academia with a platform to develop smart integrated systems, ranging from advanced prototype design to volume production. Obviously acquiring new licences is out of the question due to their high-cost, so universities must rely on support services such as Europractice.

All the simulators were run in 64-bit mode using a single thread. Although it would be possible to use multi-threaded simulation in the three simulators, which would be faster as discussed in Section 3.4, this option was not used because the goal was to evaluate the performance of the base simulation engines of the three simulators. Nonetheless, the following problems have been identified with multi-threading simulation. NCSim did not support multi-threading with a System Verilog testbench, so it would be necessary to rewrite the testbench for doing so. Both NCSim and Verilator required the processes to be manually distributed among the threads which could make the comparison unfair if the same distribution was used for both. For VCS apparently the distrbution was automatic but no performance improvement was observed with multi-threading done in this way. Finally, to minimize the effects of random errors in the simulation time, each test was performed 20 times for each simulator, with the final value being the average of the measured times. The experimental results are presented in Table 6.1 and the same results are presented graphically in Figures 6.1 and 6.2.

Table 6.1: Benchmark results.

Simulator	Average execution time	Standard deviation	Average execution time (debug+VCD)	Standard deviation (debug+VCD)
Verilator 4.014; gcc 5.4.0 -O3	3.70 sec	0.05 sec	21.80 sec	0.13 sec
Synopsys VCS 2017.03	5.15 sec	0.07 sec	27.00 sec	0.12 sec
Cadence NCSim 13.10	6.39 sec	0.07 sec	82.97 sec	1.11 sec

As expected, the simulation times are longer in the debug+VCD mode. This happens because the simulators enable more internal assertions, therefore slowing down the simulation performance. Also, the VCD file generation further slows down the simulations, since they have to write a considerable amount of data to the computer disk. This was probably the main reason for the low performance of NCsim, since it produced a VCD file considerably larger than the other simulators.

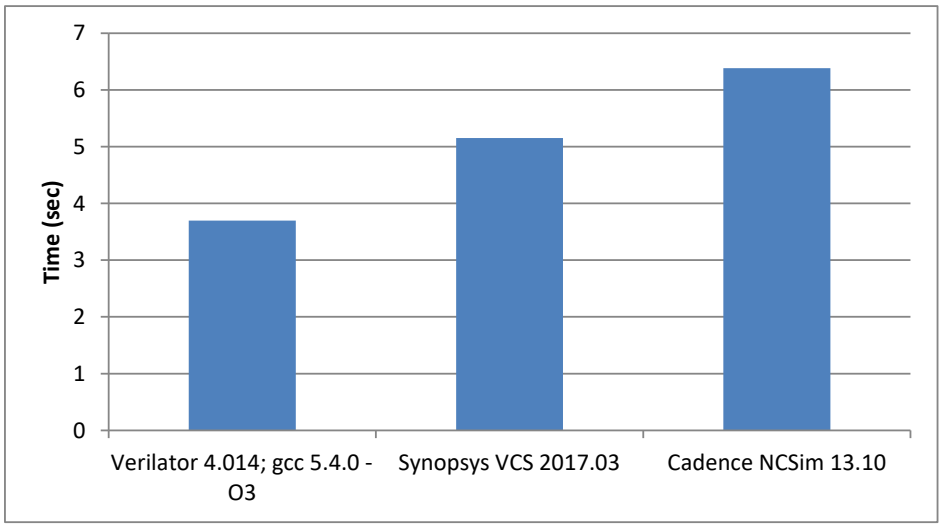

Figure 6.1: RV32-Versat simulation time. Normal mode.

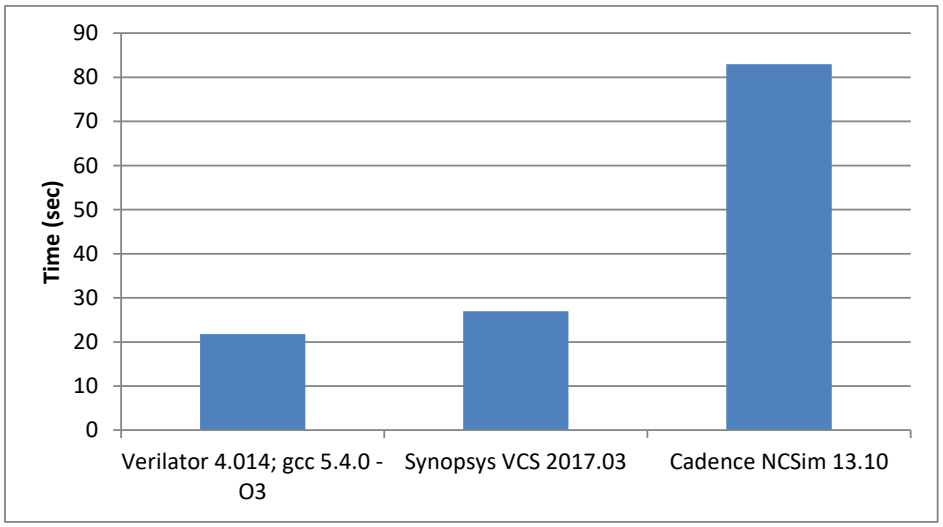

Figure 6.2: RV32-Versat simulation time. Debug+VCD mode.

It can be seen that in both tests Verilator is the fastest simulator. This was expected since Verilator is a cycle-accurate simulator whereas the other two are event-driven simulators. Event-driven simulators are typically slower since the algorithms used are more complex than the ones used in cycle-accurate simulators, as explained in more detail in Section 3.4. The second fastest simulator was Synopsys VCS, which was 1.39 and 1.24 times slower than Verilator, for the normal and debug+VCD modes, respectively. Finally, the slowest simulator was Cadence NCSim, 1.73 and 3.81 times slower than Verilator for the normal and debug+VCD modes, respectively. As said before, event-driven simulators (like Cadence NCSim or Synopsys VCS) provide more detailed results, being specially useful in the early stages of development or to simulate individual modules. However, RV32-Versat is a complex system and each one of its modules has already been vastly tested, so it makes sense to save time (and money) by using Verilator, since there is a low risk of getting wrong simulation results due to the simplifications implemented by Verilator.

Comparing the above results with the ones obtained in [20] and shown in Section 3.4, some differences can be noticed. In spite of Verilator being the fastest simulator in both results, in this work the difference is smaller. Also, while in [20] NCSim is the second fastest simulator, here that does not happen and VCS is faster than NCSim, especially when the debug mode and VCD generation are enabled.

These differences can be caused by multiple factors: the computers used to run the simulations were different, and the same applies to the simulated models: in [20] a Motorolla M68K processor derivative was used whereas here the RV32-Versat system was used. The versions of the simulators used were also different, and that might be a reason for NCSim being so slow compared to VCS. In fact, the NCSim version is from 2013. This of course does not invalidate the results for the present work but it is worth noticing for the sake of rigour.

6.2 FPGA Validation Results

The RV32-Versat architecture was implemented in an FPGA, more precisely a Xilinx XCKU040 device of the Kintex UltraScale product family. The whole system was implemented and tested using the CNN algorithm, previously described in Section 2.4.

The initial values of the Versat memories, which in the simulations were loaded through the testbench, were now used to initialize the FPGA Block Random Access Memories (BRAM) at compile time, preventing the need to send large quantities of data to the device. This situation is not ideal, since each time one wants to use a different set of values, for example, to analyze a different image, the FPGA configuration bit stream needs to be built again, which takes considerable time.

However, in this work the FPGA implementation was only used as a way to validate the simulation results, and further work in this area would be out of the scope of this dissertation. The simulation results were successfully validated and the FPGA implementation results are presented in Table 6.2, where the amount of LUTs, BRAMs and Digital Signal Processor (DSP) used are shown.

Table 6.2: FPGA implementation results for RV32-Versat configured for the CNN digit recognition application.

Resources	LUTs	BRAMs	DSPs
Used	7081	62	8
Total	242400	600	1920
%	2.92	10.33	0.42

Chapter 7

Conclusions

In this thesis the simulation environment developed for the RV32-Versat architecture was presented. This simulation environment presents a considerable improvement over the typical simulation environments using event-driven simulators: it is faster, it is inexpensive, not requiring the acquisition of licences, and allows a direct implementation of software and hardware co-simulation, avoiding the use of the VPI or inefficient ad hoc solutions.

This ambient was developed after a detailed study of the state of the art of CPU and CGRA simulators, that did not only include the event-driven and cycle-accurate simulators, but also included custom simulators specifically developed for CGRAs. This was done to ensure that the simulation environment would correspond to the defined objectives.

To test the new simulation environment an application using a CNN was developed and its results were compared with the ones obtained with the RV32-Versat architecture implemented in an FPGA. This way it was possible to ensure that the results obtained with the new simulation environment were correct.

7.1 Achievements

The first achievement of this thesis was the development of a faster simulation environment. As it was seen in the benchmark presented in Section 6.1, this simulation environment can be up to 3.81 times faster than a simulation environment using the traditional event-driven simulators. This can help to cut the time needed to develop and debug applications for the RV32-Versat.

Another important achievement is the reduction of the amount of money spent in licences. This happens because the new simulation environment uses Verilator, an open-source simulator, in contrast to the event-driven simulators that require expensive licences. This is particularly useful for small companies, allowing them to spend their limited resources in other areas.

The third achievement is the improved support for hardware and software co-simulation in this new simulation environment, dismissing the use of special interfaces (like the Verilog VPI) or ad hoc solutions. With this new environment the testbench can be written in C++ or SystemC, therefore allowing a seamless integration of hardware and software co-simulation.

The last achievement was the development of a simulation environment that is independent from eventual changes in the RV32-Versat architecture. This means that if the architecture of RV32-Versat is changed, the simulation environment will keep working. This would not happen if a simulator developed for this specific architecture was used, instead of Verilator.

7.2 Future Work

The work developed during this thesis can be developed in three different paths. The first is the development of a high-level simulator for this architecture that works at the functional unit level instead of the RTL. It could use a high-level simulator specially developed for the Versat architecture, similarly to [21], extracting the data from the memories and executing the desired high-level operations (no bit-level operations) by reading the configuration bits. This would allow for an even better performance, but it would also have a disadvantage: architecture changes in Versat would also require changes in the simulator architecture. 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.

Another path would be to develop a software and hardware co-simulation environment for this architecture using the new simulation environment that, has explained before, makes this much easier. This would allow to run the software in a computer, adding a C++ class to simulate the RV32-Versat hardware.

The third path is a much more ambitious project: create an Application Programming Interface (API) that would allow to generate and simulate Versat datapaths with an almost complete abstraction from the hardware. This could be done using Versat's C++ Driver, described in Section 2.3.1, as a basis, adding other classes to both generate the datapaths programmed in C++ and to simulate them. With this API the whole system could be evaluated just in software, therefore not needing Verilog simulators. For further tests the Verilog code generated by the API could be simulated using the Verilator co-simulation capabilities or it could be used directly in an FPGA. This solution would allow to have an even faster way to simulate Versat without needing to significantly change the API.

Bibliography

[1] Picoversat - a 16-instruction pico controller to replace complex state machines, 2019. URL https://bitbucket.org/jjts/picoversat/src/master/.

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

[3] 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.

[4] 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.

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

[6] *Versat Specification Report*. inesc-id, March 2016.

[7] Picorv32 - a size-optimized risc-v cpu, 2019. URL https://github.com/cliffordwolf/picorv32.

[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] 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.

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

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

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

[15] 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.

[16] 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.

[17] 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`.

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

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] C. Dawson, S. K. Pattanam, and D. Roberts. The verilog procedural interface for the verilog hardware description language. In *Proceedings. IEEE International Verilog HDL Conference*, pages 17–23, Feb 1996. doi: 10.1109/IVC.1996.496013.

[27] P. D. S. Richard M. Stallman, Roland McGrath. *GNU Make*. Free Software Foundation, May 2016. URL `https://www.gnu.org/software/make/manual/make.pdf`. Version 4.2.

[28] Gnu toolchain for risc-v, including gcc, 2019. URL https://github.com/riscv/riscv-gnu-toolchain.