System-Level Simulation Framework for Heterogeneous Multi-Core Processing Structures

Pedro Magalhães

Dissertation submitted to obtain the Master Degree in Information Systems and Computer Engineering

Jury
Chairman: Doutor João Paulo Marques da Silva
Supervisor: Doutor Nuno Filipe Valentim Roma
Member: Doutor Ricardo Jorge Fernandes Chaves

June 2012
Acknowledgments

This thesis was performed in the scope of the research project “HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis”, with reference PTDC/EEA-ELC/113999/2009.

I want to thank professor Nuno Roma for all the support and guidance provided during the entire development of this thesis. Fortunately for me, you were always available for helping me and provide ideas to surpass the challenges. Thank you a Lot Nuno Roma. You are undoubtedly the best thesis leader I have ever known of.

Due to the thesis context, it was not possible to exchange ideas with friends or family, but thank you all guys from Grupo Mais Fixe for the psychological support and meetings that cheer me up during this long walk. Emanuel Silva, it was a great pleasure to go through this journey along with you, so, here’s a special thanks and a warm hug for you. We did it!

Dear Great Family, a long and hard season has finally come to an end for me, as you know. There were several moments when I was not present, due to this important and huge step in my life. Plus, during this long path, I took another giant leap which left us even further away. Either way, you have been of great importance to me. You have supported me all this time, and also, you were right there in front of me when I most needed you while I was seriously talking with two or three people and complete nonsense for most of you. You are the Best! Andreia, we all know you were there too. Mother and Father, Adriana Magalhães and Vítor Magalhães, thank you for who you are.

At last, my dear girlfriend Daniela Pedras, we have spent the first months of our life together which were totally not easy for both of us. We were so close and yet so far. I know that these last months were hard on you as you missed my support for almost everything, so now, it’s time for me to support you. Remember, this is just our first milestone! Let’s just live happy! Love you forever...
Abstract

This thesis proposes an event-driven simulation tool to support the design, development and rapid prototyping of an heterogeneous multi-core processing structure. The main focus of this parallel structure is to efficiently execute a set of widely used bioinformatic algorithms for DNA sequences alignment and processing. Biologists and researchers use those alignment procedures as their main tool to extract useful information from the huge DNA sequences that are stored in large databases. This document starts with a brief review of some of the most widely adopted hardware and system description languages. Then, it compares and discuss these languages in what concerns their suitability for the implementation of the required simulation tool. Finally, it will be presented a detailed description of the simulation framework in what concerns its main components, how they work and the way they have been implemented using SystemC.

Keywords

DNA alignment, Parallel processing, Heterogeneous architectures, System level simulation.
Resumo

Este trabalho apresenta uma ferramenta de simulação guiada por eventos, que será utilizada no desenho, desenvolvimento e criação dum protótipo dumha estrutura de processamento heterogéneo constituída por vários núcleos. Esta estrutura de processamento será utilizada para executar um conjunto de algoritmos do ramo bioinformático mais especificamente alinhamento e processamento de sequências de ADN. Biólogos e investigadores utilizam estes procedimentos de alinhamento de forma a extrair informação de sequências extensas que necessitam de ser guardadas em grandes bases de dados devido ao seu tamanho. Este documento tem início com uma breve revisão sobre as linguagens de descrição de sistemas e hardware mais utilizadas. Em seguida, é feita uma comparação e discussão sobre estas linguagens no que toca à sua adequação relativamente à implementação da ferramenta de simulação em vista. Por fim, será apresentada a descrição detalhada da ferramenta de simulação construída tendo em conta os seus componentes, a forma como funcionam e como foram implementados utilizando SystemC.

Palavras Chave

Alinhamento ADN, processamento paralelo, arquitecturas heterogéneas, simulação nível sistema.
Contents

1 Introduction ................................................. 1
  1.1 Motivation ............................................. 2
  1.2 Application Context .................................. 2
  1.3 Simulation of concurrent processes ................. 3
    1.3.1 Homogeneous systems ......................... 4
    1.3.2 Heterogeneous systems .................... 4
  1.4 Objectives ........................................... 5
  1.5 Structure of the thesis ............................ 5

2 System Description Languages ......................... 7
  2.1 Hardware Description Languages ................. 8
    A – Structured and behavioral descriptions ...... 8
    B – Netlist ......................................... 8
    C – Modularity and Hierarchy .................. 9
  2.1.1 VHDL ............................................. 9
  2.1.2 Verilog ......................................... 11
  2.2 System Description Languages ................. 13
    2.2.1 SystemC ....................................... 13

3 Related Work ............................................. 17
  3.1 I²C Bus Controller ................................ 18
  3.2 CAN Bus Architecture Model .................... 20
  3.3 SystemC Modelling of a Parallel Processor Broadcast Interconnection System .... 22
    3.3.1 Node Module ................................ 23
    3.3.2 Transmit Module ................................ 24
    3.3.3 Receiver Module ................................ 24
  3.4 Discussion ........................................... 24

4 Simulation framework architecture .................. 26
  4.1 Simulation Framework ................................ 28
    4.1.1 Stimulus Module .............................. 28
    4.1.2 Monitor Module ................................ 28
  4.2 Simulated Device .................................... 31
    4.2.1 Modules ....................................... 32
      4.2.1.A Master .................................... 32
      4.2.1.B Slave ...................................... 33
# List of Figures

1.1 Example of an alignment score matrix ........................................ 4
2.1 Modularity and hierarchy .................................................. 9
2.2 Description of a 4 bit counter in VHDL ................................ 11
2.3 Description of a 4 bit counter in Verilog ............................. 12
2.4 Event and notify example .................................................. 15
2.5 4 bit counter example in SystemC ....................................... 15
3.1 I2C bus controller ........................................................ 18
3.2 I2C Controller Architecture ............................................. 19
3.3 Simulation of an I2C controller using SystemC ...................... 20
3.4 Can bus controller architecture ........................................ 20
3.5 CAN bus Testbench model ................................................. 22
3.6 System Architecture ...................................................... 22
3.7 Transmit Module .......................................................... 24
3.8 Receive Module ............................................................ 25
4.1 Environment System Model ................................................. 27
4.2 Simulation Framework ..................................................... 28
4.3 Monitor writing to log file ................................................. 29
4.4 Ploticus text file syntax ................................................... 30
4.5 Monitor log file format .................................................... 30
4.6 Example of a concurrent execution of 8 alignment processes using four slave processors ............................................. 31
4.7 Simulated Device architecture ........................................... 31
4.8 The first 3 steps .............................................................. 33
4.9 Task issue in the simulated device the last steps .................... 34
4.10 Write and/or read operations in the shared memory ............... 35
4.11 Module addressing space ............................................... 36
4.12 A command message path .............................................. 37
4.13 A data message path ...................................................... 37
4.14 Request queues for each priority ...................................... 38
4.15 Mailboxes scheme ........................................................ 39
5.1 Producer-consumer example implemented in SystemC .......... 42
5.2 Fifo’s source code to implement the buffer .......................... 43
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Buffer module using a sc-event type event to signal the consumer module</td>
<td>43</td>
</tr>
<tr>
<td>5.4</td>
<td>ModuleStruct source code</td>
<td>45</td>
</tr>
<tr>
<td>5.5</td>
<td>Mailbox interface definition</td>
<td>46</td>
</tr>
<tr>
<td>5.6</td>
<td>Mailbox interface implementation</td>
<td>46</td>
</tr>
<tr>
<td>5.7</td>
<td>Stimulus source code</td>
<td>47</td>
</tr>
<tr>
<td>5.8</td>
<td>Monitor and corresponding interface source code</td>
<td>48</td>
</tr>
<tr>
<td>5.9</td>
<td>Data structure to store the log events details</td>
<td>49</td>
</tr>
<tr>
<td>5.10</td>
<td>Master source code</td>
<td>49</td>
</tr>
<tr>
<td>5.11</td>
<td>Table of each slave’s status</td>
<td>50</td>
</tr>
<tr>
<td>5.12</td>
<td>Master interface that is used to receive the table containing the addresses and lengths of the test data</td>
<td>50</td>
</tr>
<tr>
<td>5.13</td>
<td>Slave source code</td>
<td>51</td>
</tr>
<tr>
<td>5.14</td>
<td>Memory unit source code</td>
<td>52</td>
</tr>
<tr>
<td>5.15</td>
<td>Shared memory source code</td>
<td>53</td>
</tr>
<tr>
<td>5.16</td>
<td>Bus source code structure</td>
<td>54</td>
</tr>
<tr>
<td>5.17</td>
<td>Arbitrer structure and interface source code</td>
<td>56</td>
</tr>
<tr>
<td>5.18</td>
<td>Interface methods that support the communication between the bus and the arbiter</td>
<td>57</td>
</tr>
<tr>
<td>5.19</td>
<td>Syntax of a command message</td>
<td>57</td>
</tr>
<tr>
<td>5.20</td>
<td>Example of a test data message</td>
<td>58</td>
</tr>
<tr>
<td>5.21</td>
<td>Example of a command message</td>
<td>58</td>
</tr>
<tr>
<td>5.22</td>
<td>Datagram structure</td>
<td>59</td>
</tr>
<tr>
<td>5.23</td>
<td>Request state diagram</td>
<td>60</td>
</tr>
<tr>
<td>6.1</td>
<td>Creating and sending a message example</td>
<td>63</td>
</tr>
<tr>
<td>6.2</td>
<td>Syntax of “new message” method</td>
<td>64</td>
</tr>
<tr>
<td>6.3</td>
<td>Content of “req-&gt;data” after using the “new message” method</td>
<td>64</td>
</tr>
<tr>
<td>6.4</td>
<td>Reading a message example</td>
<td>65</td>
</tr>
<tr>
<td>6.5</td>
<td>Syntax and examples from “getarg” method</td>
<td>66</td>
</tr>
<tr>
<td>6.6</td>
<td>Generating a new task identification and assigning it to a slave</td>
<td>67</td>
</tr>
<tr>
<td>6.7</td>
<td>Tools used for slave management</td>
<td>67</td>
</tr>
<tr>
<td>6.8</td>
<td>Loading data into the simulator</td>
<td>69</td>
</tr>
<tr>
<td>6.9</td>
<td>Master issuing procedure</td>
<td>70</td>
</tr>
<tr>
<td>6.10</td>
<td>Master receiving procedure</td>
<td>71</td>
</tr>
<tr>
<td>6.11</td>
<td>Slave processing procedure part 1</td>
<td>72</td>
</tr>
<tr>
<td>6.12</td>
<td>Slave processing procedure part 2</td>
<td>73</td>
</tr>
<tr>
<td>6.13</td>
<td>Parameters file 1</td>
<td>74</td>
</tr>
<tr>
<td>6.14</td>
<td>Parameters file 2</td>
<td>75</td>
</tr>
<tr>
<td>6.15</td>
<td>Memory Unit wait statements example</td>
<td>76</td>
</tr>
<tr>
<td>6.16</td>
<td>Slave wait statement example</td>
<td>76</td>
</tr>
<tr>
<td>7.1</td>
<td>System snapshot using one slave module for K=0.2</td>
<td>79</td>
</tr>
<tr>
<td>7.2</td>
<td>System snapshot using two slave modules for K=0.2</td>
<td>79</td>
</tr>
<tr>
<td>7.3</td>
<td>System snapshot using four slave modules for K=0.2</td>
<td>80</td>
</tr>
<tr>
<td>7.4</td>
<td>Total execution time and speedup as functions of the number of slave processors</td>
<td>80</td>
</tr>
<tr>
<td>7.5</td>
<td>Total execution time as a function of the number of slave processors</td>
<td>81</td>
</tr>
</tbody>
</table>
7.6 Speedup as a function of the number of slave processors ............... 82
List of Tables

3.1 Simulated Packet Definition ............................................................ 23
4.1 Priority level assignment ................................................................. 38
5.1 Alphabet used for communication .................................................... 55
5.2 Request status table ........................................................................ 60
Introduction

Contents

1.1 Motivation .................................................. 2
1.2 Application Context ........................................ 2
1.3 Simulation of concurrent processes .................... 3
1.4 Objectives .................................................. 5
1.5 Structure of the thesis ..................................... 5
1. Introduction

1.1 Motivation

With the latest developments in computer architectures, modern computational systems are often composed by multiple processors, memories and dedicated structures integrated either in embedded systems or even in Systems on Chip (SoC) devices. However, as the complexity of these systems increases and the design time is shortened by market demands, it is extremely important to simulate these systems before their design achieves the manufacturing process. Furthermore, there are also several difficulties in the pre-design stage when it is needed to specify the system’s characteristics before it is actually implemented. Natural language is ambiguous and open to interpretation. As a consequence, the system specification may be incomplete and inconsistent and there is no way to verify the correctness of such specification [1]. Frequently, these systems target mass production and therefore should be cheap, power-efficient, offer high performance, and give support to multiple applications and standards, which requires high programmability. This wide spectrum of design requirements leads to complex heterogeneous SoC architectures [2].

All these facts have pushed the usage of System and Hardware Description Languages a lot further, in order to make them applicable to system design and behavior simulation. The designers immediately benefited from some of the advantages of using programming languages. Firstly, they provide the control and data abstraction layers necessary to develop compact and efficient system descriptions. Secondly, they allow to simulate systems containing both hardware and software components. Finally, designers are often familiar with the used programming languages and there are a large number of development tools associated with them [1].

1.2 Application Context

The final goal of this thesis is the development and implementation of a simulation environment of a parallel multi-core processor architecture which is optimized for the execution of a particular set of bioinformatic algorithms. One of such algorithms is denoted as Smith-Waterman (SW)[3] and it is widely used to determine the optimal alignment between two DNA sequences.

Nowadays, biologists are already able to determine and process the nucleotide sequence of the Deoxyribonucleic Acid (DNA). These sequences often represent a huge quantity of information (e.g. the size of the human DNA can be as large as $3 \times 10^9$ base pairs) and thus requiring large databases. The information contained in the DNA sequences is mainly extracted by homology, which means that a certain sequence may share an ancestor from any other sequence, therefore requiring a large number of comparisons between sequences. However, the exact search of a given sequence in the whole sequences database is often unfeasible, due to the frequent mutations to which DNA is affected (nucleotide insertion and deletion). This is where the Smith-Waterman algorithm [4] comes in. It is capable of finding the optimal position where as many matching nucleotides are found in the same positions. This is how biologists can determine which sequences match more closely and how they align, in order to identify the zones that are common between two DNA sequences [3].

The alignments can be classified either as local or global. In global alignments, the complete sequences are aligned from one end to the other, whereas in local alignments only the subsequences that present the highest similarity are considered in the alignment. The local alignment
1.3 Simulation of concurrent processes

is generally preferred and the one focused on this project, so a brief description is now presented.

Considering any two strings $S_1$ and $S_2$ of an alphabet $\varepsilon$ with sizes $n$ and $m$, respectively, the local alignment of strings $S_1$ and $S_2$ reveals which pair of substrings of $S_1$ and $S_2$ optimally align, such that no other pairs of substrings have a higher alignment score. Let $G(i, j)$ represent the best alignment score between a suffix of strings $S_1[1..i]$ and a suffix of string $S_2[1..j]$. The S-W algorithm allows the computation of $G(n, m)$, by recursively calculating $G(i, j)$, which will reveal the highest alignment score between the substrings of strings $S_1$ and $S_2$.

The recursive relation to calculate the local alignment score $G(i, j)$ is given by Eq. 1.1 and Eq. 1.2, where $Sbc(S_1(i), S_2(j))$ (in Eq. 1.1) denotes the substitution score value obtained by aligning character $S_1(i)$ against character $S_2(j)$ and $\alpha$ represents the gap penalty cost (the cost of aligning a character to a space, also known as gap insertion). An example of an alignment score matrix is shown in the figure 1.1.

The alignment scores are usually positive for characters that match, thus denoting some similarity level between them. Mismatching characters may have either positive or negative scores, depending on the type of the alignment that is being performed. When the entire score matrix $G$ is completely filled, the substrings of $S_1$ and $S_2$ with the best alignment score are identified by locating the cell with the highest score in $G$. Then, all cells that lead to this highest score cell are sequentially determined by performing a traceback phase. This last phase concludes when a cell with a zero score is reached, identifying the aligned substrings as well as the corresponding alignment. The path taken at each cell is chosen based on which of the three neighboring cells (left, top-left and top) was used to calculate the current cell value using the recurrence given by Eq. 1.1 and 1.2.

To obtain efficient implementations of this algorithm, several hardware architectures have been presented in the literature, either by adopting homogeneous or heterogeneous structures. Independently of the adopted processing platform, it has been shown that efficient and reliable simulation frameworks are regarded as fundamental tools in order to obtain the highest performance levels of these systems.

$$G(i, j) = \max \begin{cases} G(i - 1, j - 1) + Sbc(S_1(i), S_2(j)), \\ G(i - 1, j) - \alpha, \\ G(i, j - 1) - \alpha, \\ 0 \end{cases}$$

(1.1)

$$G(i, 0) = G(0, j) = 0$$

(1.2)

1.3 Simulation of concurrent processes

The development of this project provides the challenge of simulating concurrent processes to be executed in hardware platforms. This is a form of computing in which programs are designed as collections of interacting computational processes that are executed in parallel. These programs can be executed sequentially in a single processor, by interleaving the execution steps of each process, or executed in parallel, by assigning each process to one of a set of processors. The main challenge here is to ensure that the correct sequencing of operations or communications between the different computational processes is achieved as well as to coordinate the access to resources that are shared among the concurrent processes.
1. Introduction

Figure 1.1: Example of an alignment score matrix.

The system that was developed in the scope of this thesis is composed of several concurrent modules and internal functions, all of them running at the same time. This may be a problem when programming only with regular programming languages, such as C, C++, or Java. This problem arises because these programming languages are sequential. There are several different approaches to overcome this problem, like using processes, threads, concurrency, semaphores, spin-locks, shared memory, message-passing, etc., but the main problem is still how to use all those tools and mechanisms [10] and sometimes this can turn out to be a real hard task.

To facilitate all the mentioned problems, in order to successfully design a complex concurrent hardware architecture and to model the execution time referred by this parallel system, the chosen environment to develop this simulation framework was SystemC. The advantages of this language will be described in chapter 2.

1.3.1 Homogeneous systems

There are innumerous research projects in the area of computing architectures focused on developing and improving the computational power to solve huge mathematical and scientific problems [6]. To face the many challenges that arise when designing such systems aiming the maximum performance levels, many approaches use homogeneous platforms, where all computation cores are equal. The algorithms used in this type of platforms, mandatorily need to be properly analysed and prepared to be run on these parallel systems. This is a fundamental requisite, because of the data dependencies that usually arise from within. As a consequence, another challenge often comes in, which is concerned with the inter-core communication, representing a vital aspect for system functionality and coherence [6]. Nevertheless, this approach often suffers from the existing bottleneck when transferring the data to the many cores in the system [6].

There are several proposals based on this technique that use current General Purpose Processors (GPP) [11][12]. Other approaches adopt hybrid platforms, where the algorithms running in the GPPs are accelerated using Graphical Processing Units (GPU) [13][14].

1.3.2 Heterogeneous systems

Heterogeneous architectures integrate dedicated or specialized processing structures, optimized for specific parts of the target application or algorithm. The main problem in heterogeneous systems is that there are not many tools and frameworks that allow simulating this type of
1.4 Objectives

architecture. On one hand, this shortage of tools is tightly coupled with the intrinsic heterogeneity of such architectures, making it considerably difficult to accurately model the several operations being simultaneously executed in such systems. On the other hand, the fact that such systems are often composed by dedicated and very specific processors often imposes the need to conceive specialized and non-generic tools, which can only be applied to very specific and particular targeted systems.

There are some examples concerning this type of systems, such as the Field-Programmable Gate Arrays (FPGA) \[15\] and Application Specific Integrated Circuits (ASIC) \[16\].

1.4 Objectives

During the development of this work, the following objectives will be considered:

- Design and implement a complete simulation framework to model and simulate the total execution time required by a given parallel processing system. All the time periods that each module needs to do its work will be simulated and added to a centralized counter. At the end of the simulation, it will be possible to compare the time that it is needed in a sequential processing execution against a parallel processing.

- Create a flexible design structure so that future changes and added functionalities become easy to integrate. The architecture of the system, including the modules responsible for testing and verification, shall be prepared to run different types of testing and provide proper ways for debugging. The architecture shall be prepared to integrate several homogeneous or heterogeneous parallel processing cores.

- Development of simulation framework that allows to configure several points in the simulated system, such as the bus width, memory size, memory access time, number of processing cores and the time period for each simulated operation.

1.5 Structure of the thesis

In the following chapter, some of the most used hardware and software description languages will be considered, as well as the one that was chosen to develop the proposed system. At the end of the presentation of each description language there is a small example illustrating how a simple 4-bit counter can be implemented, thus providing a good idea of its programming environment. To complement the introduction and motivation of these description languages, in chapter \[3\] it will be presented a set of design examples that illustrate how system description language denoted as SystemC can be used by designers to specify, develop and simulate the hardware architecture that will be manufactured, re-manufactured or even exposed to new types of testing during the simulation process. In chapter \[4\] the architecture of the implemented simulation framework is described as well as its operation. Then, in chapter \[5\] it will be described how this framework is implemented using SystemC. Before the experimental results chapter, it is presented a simple step by step tutorial on how to use, configure and implement an algorithm in the developed simulator. The final chapter presents the obtained experimental results and compares the performance of the system using one or more processing modules. The conceived simulation
1. Introduction

framework simulates the processor behavior at system-level, along with the time that each operation will spend while running in a concurrent processing structure. These operations are also viewed and discussed on the experimental results chapter.
2 System Description Languages

Contents

2.1 Hardware Description Languages .............................................. 8
2.2 System Description Languages .............................................. 13
2. System Description Languages

System description languages [17] are fundamental tools for project design, whether it is an electronic or an informatic project. These are generally used to allow separating the project into smaller pieces and separate them in a logical way. This allows the designer to concentrate on each module design, providing the needed abstraction levels. Furthermore, system description languages brings a major benefit, that is, everyone is able to read, understand, write or change pieces of the project. Moreover, the particular set of system description languages described in this chapter provide an even greater benefit, which is a compiler/simulator tool that are capable of reading the language and simulate the desired system. They will be shortly presented, in what concerns their pros and cons.

2.1 Hardware Description Languages

In the electronics domain, a Hardware Description Language (HDL) [18][19][20] is any language from a class of computer languages for formal description of digital electronic circuits. It can describe the circuit’s operation, its design and organization, and tests to verify its operation by means of simulation. The main feature of HDL is that it contains the capability to describe the hardware from the functional point of view, independently of its final implementation. In contrast to software programming language, an HDL’s syntax and semantics include explicit notations for expressing time and concurrency, which are the primary attributes of hardware.

A – Structured and behavioral descriptions
HDLs are used to write executable specifications of some piece of hardware. A simulation program, designed to implement the underlying semantics of the language statements, coupled with simulating the progress of time, provides the hardware designer with the ability to model a piece of hardware before it is physically created. The descriptions can be either structural or behavioral. Structural description is a textual replacement for a graphical schematic, by directly using the components of the underlying technology process. The design approach follows a hierarchical model, where the lowest level in the hierarchy is composed of primitives provided by the specific language being used. Behavior or functional approach describes what a module does and it does not care about the underlying implementation. The specific circuit for any given module is only generated in the synthesis step. Finally HDLs whose only characteristic is to express the circuit connectivity are classified as 'netlist' languages.

B – Netlist
The connectivity of an electronic design is described by a 'netlist' which usually conveys connectivity information, instances, nets, and attributes. Netlists either contain or refer to descriptions of the parts or devices used. Each time a part is used in a netlist, this is called an instance. Thus, each instance must have a master or definition. The definitions list the connections that can be made to that kind of device and its basic properties. Nets represent the ‘wires’ that connect things together in the circuit.

Hence, HDLs are generally used to write a model for the expected behavior of a circuit before that circuit is actually built. The model is fed into a simulator, allowing the designer to verify that his solution behaves correctly. Next, they are used to write a detailed description of a circuit that is fed into a logic compiler. Depending on the target implementation technology, this logic compiler will be used to configure a programmable logic device that has the desired function.
2.1 Hardware Description Languages

Although not so common, it is also possible to represent hardware semantics using traditional programming languages such as C or C++. In such cases, the function of such program environments must be augmented with extensive and unwieldy class libraries, since such software programming languages do not include any capability for explicitly expressing time.

![Modularity and hierarchy](image)

**C – Modularity and Hierarchy** Modularity allows the partitioning of big functional blocks into smaller units and to group closely related parts in self-contained subblocks, denoted as modules. This way, a complex system can be divided into manageable subsystems. The guidelines for partitioning can differ from design to design. Most of the time, functional aspects are considered for partitioning. The existence of well defined subsystems allows several designers to work in parallel on the same project, as each designer will view his part as a self-contained complete system.

Hierarchy allows the building of a design out of modules, which themselves may be built out of other modules and/or submodules. One level of a hierarchical description contains one or more modules, and each module can even have different degrees of abstraction.

Modularity and hierarchy help to simplify and organize a design project. Some additional advantages are that different implementation alternatives can be examined for each module by simulation. This way, only the corresponding component instantiation needs to be changed for this in the overall model.

### 2.1.1 VHDL

VHDL (Very High Speed Integrated Circuit Hardware Description Language) is a hardware description language commonly used in electronic circuit design to describe digital and mixed-signal systems, such as field-programmable gate arrays and integrated circuits.

In order to document the behavior of ASICs (Application Specific Integrated Circuits) that supplier companies were including in their equipment, around the 1980s, the U.S Department of De-
fense ordered the development of VHDL as an alternative to huge, complex manuals which were subject to implementation-specific details. At this point in time, the cost of electronic hardware became too high to support it. The main cause of this situation was that the various components used to build up integrated systems were individually verified using a wide range of different and incompatible simulation languages and tools. So, two main requirements were established at the creation of VHDL:

1. The language needed to support a wide range of descriptive capability to work the same way on any simulator and to be independent of technology or design methodology.

2. The structure and syntax had to be understandable by humans.

The engineers at the Department of Defense were very familiar with the syntax of Ada programming language. As a consequence, in order to avoid re-inventing concepts that had already been thoroughly tested on Ada's environment, VHDL is heavily based on Ada's in what concerns its concepts and syntax. As such, the initial version of VHDL included a wide range of data types, including numerical (integer and real), logical (bit and boolean), character and time, as well as arrays of bits (bit vector) and characters (string). VHDL is also able to handle the parallelism inherent to hardware designs. It is strongly typed and is not case sensitive as ADA. In order to directly represent common operations used in hardware design, VHDL also provides several features, such as an extended set of boolean operators (like nand and nor). VHDL also allows the arrays to be indexed in either ascending or descending direction, contrarily to Ada and most programming languages where only ascending indexing is available.

Since the publication of the first standard, VHDL has been submitted to review processes from times to times. The first one was accomplished in September 1993, which made its syntax more consistent, allowed more flexibility in the adopted naming, extended the character type to allow ISO-8859-1 printable characters and added the 'xnor' operator. Another feature introduced in 1993 was shared variables. However, it was not possible to use it in any meaningful way due to various limitations when accessing critical regions such as: limitations on the type of statements, critical regions could not be accessed from within another critical region and shared data could not be access typed \(^{[21]}\). This situation led to the addition of protected types (similar to the concept of class in C++) to VHDL in 2000. Next, in 2002, a minor revision of the 2000 version was made with one significant change which removed some port mapping restrictions. These changes were subsequently followed by an amendment in 2006, when VHPI tool was introduced, which allows the VHDL model to interact with programs written in C before and during the simulation. This tool made testbenches more flexible and allow a wider use of VHDL for system-level descriptions.

Some integrated development environments such as Xilinx, ALtera and Cadence allow the programmer to design the desired hardware circuits, generate the appropriate testbench and simulate it showing the waveforms of its inputs and outputs.

The VHDL code presented in figure 2.2 represents the implementation of a simple 4 bit counter. First, the ports are set up according to their direction (I/O) and data type. During the next step, the counter code is described and all the variables needed to transport the data to the output ports are defined. Every time the port's (Clock and Reset) value is changed, the process will be executed \(^{[22]}\).
2.1 Hardware Description Languages

2.1.2 Verilog

Just like VHDL, Verilog is another well known HDL, although it is most commonly used in the design, verification and implementation of digital logic chips at the register transfer level (RTL) of abstraction or specific hardware implementations. This happens, because it is possible to build hardware implementations through a logical description consisting only of elementary logic primitives ('AND', 'OR', 'NOT', 'flip-flops', etc.) that have a direct mapping to real gates. It is also used in the verification of analog and mixed-signal circuits. Verilog was developed at Automated Integrated Design Systems, in 1984, as a hardware modelling language. The next year it was renamed to Gateway Design Automation and finally purchased by Cadence design Systems, in 1990.

Verilog \cite{23,24,25} differs from software programming languages because it natively includes ways to describe the propagation of time and signal dependencies, which is often called sensitivity. There are two assignment operators: a blocking assignment (\texttt{=}) and a non-blocking assignment (\texttt{<=}). The non-blocking assignment allows designers to describe a state-machine update without even needing to declare and use temporary storage variables. With all these factors, designers are able to quickly write descriptions of large circuits in a relatively compact and concise form.

The first designers of Verilog were accustomed to C programming language. As a consequence, Verilog has various similarities with it. This way, Verilog is case-sensitive, has a basic

```vhdl
entity Counter2_VHDL is
  --Ports---
  port( Clock_enable: in std_logic;
        Clock: in std_logic;
        Reset: in std_logic;
        Output: out std_logic_vector(0 to 3));
end Counter2_VHDL;

architecture Behavioral of Counter2_VHDL is
  --Output Port---
  signal temp: std_logic_vector(0 to 3);
begin
  process(Clock,Reset)
    begin
      if Reset='1' then
        temp <= "0000";
      elsif(Clock'event and Clock='1') then
        if Clock_enable='0' then
          if temp="1111" then
            temp<="0000";
          else
            temp <= temp + 1;
          end if;
        else
          temp <= temp;
        end if;
      end if;
    end process;
    Output <= temp;
  end Behavioral;
end Counter2_VHDL;
```

Figure 2.2: Description of a 4 bit counter in VHDL.

2.1.2 Verilog

Just like VHDL, Verilog is another well known HDL, although it is most commonly used in the design, verification and implementation of digital logic chips at the register transfer level (RTL) of abstraction or specific hardware implementations. This happens, because it is possible to build hardware implementations through a logical description consisting only of elementary logic primitives ('AND', 'OR', 'NOT', 'flip-flops', etc.) that have a direct mapping to real gates. It is also used in the verification of analog and mixed-signal circuits. Verilog was developed at Automated Integrated Design Systems, in 1984, as a hardware modelling language. The next year it was renamed to Gateway Design Automation and finally purchased by Cadence design Systems, in 1990.

Verilog \cite{23,24,25} differs from software programming languages because it natively includes ways to describe the propagation of time and signal dependencies, which is often called sensitivity. There are two assignment operators: a blocking assignment (\texttt{=}) and a non-blocking assignment (\texttt{<=}). The non-blocking assignment allows designers to describe a state-machine update without even needing to declare and use temporary storage variables. With all these factors, designers are able to quickly write descriptions of large circuits in a relatively compact and concise form.

The first designers of Verilog were accustomed to C programming language. As a consequence, Verilog has various similarities with it. This way, Verilog is case-sensitive, has a basic
2. System Description Languages

preprocessor, has equivalent control flow keywords like if/else, while, case, etc, and the same operator precedence.

A Verilog design usually consists of a hierarchy of modules. Modules encapsulate its design hierarchy, functions and communicate with other modules through a set of input, output and bidirectional ports. A module can contain its variables, concurrent and sequential blocks and instances of other modules denoted as sub-hierarchies. Sequential statements are placed inside a begin/end block and executed in sequential order within the block. The modules and their own blocks are all run concurrently. A module can contain a wide variety of data types, including reg, wire, integer, real, time, etc. A reg simulates a register, which can store a value as well as an integer, boolean or real type. A wire represents a vector of bits of arbitrary width. The concept of wire consists of both signal values (4-state: “1,0, float, undefined”) and strengths (strong, weak, etc). This system also allows an abstract modelling of shared signal-lines, where multiple sources drive a common net. When a net has multiple drivers (writing entities), the wire’s readable value is resolved by a function that takes its decision accordingly to the source drivers values and strengths.

```verilog
module first_counter(
    clock, // Clock input of the design
    reset, // active high, synchronous Reset input
    enable, // Active high enable signal for counter
    counter_out // 4 bit vector output of the counter
); // End of port List
  //------------------- Input Ports -------------------------------------
  input clock;
  input repeat;
  input enable;
  //------------------- Output Ports -------------------------------------
  output [3:0] counter_out;
  //------------------- Input Ports Data Type----------------------------
  wire clock;
  wire repeat;
  wire enable;
  //------------------- Output Ports Data Type----------------------------|
  reg [3:0] counter_out;
  //------------------- Code Starts Here-------------------------------
  always @ (posedge clock)
  begin : COUNTER // Block Name
    // At every rising edge of clock we check if reset is active
    // If active, we load the counter output with 4'b0000
    if (reset == 1'b1) begin
      counter_out <= #1 4'b0000;
    // If enable is active, then we increment the counter
    else if (enable == 1'b1) begin
      counter_out <= #1 counter_out + 1;
    end
  endmodule // End of Module counter
```

Figure 2.3: Description of a 4 bit counter in Verilog.

Figure 2.3 represents the implementation of a simple 4 bit counter written in Verilog. The first step is to declare the ports that will be used during the simulation. Secondly, the ports are configured by means of direction (input, output or bidirectional) and their data types. Finally, the process in charge of starting, incrementing and resetting the counter is implemented according to
the I/O signals received and sent through the ports. The statement "always @ (posedge clock)" makes the process to be executed every time the clock signal makes a transition from low to high.

2.2 System Description Languages

Classical hardware/software (HW/SW) co-design methods typically start from a single system description that is gradually refined and synthesized into an architecture implementation which consists of programmable and/or dedicated components. The major disadvantage of this approach is that it forces the designer to make early decisions on the HW/SW partitioning of the system, that is to identify parts of the system which will be implemented in hardware and software. The cooperating simulation frameworks that model the classical HW/SW co-design approach generally combine two simulators, one for simulating the programmable components running the software and one for the dedicated hardware[2]. The common practice is to employ instruction-level simulators for the software part, while the hardware part is usually simulated using VHDL or Verilog[26].

Hardware/Software co-design is an integral aspect of the SoC (System on Chip) design. It requires a language that is capable of capturing the requirements of a hardware design, from wire allocations to complex timing requirements, as well as the complexities of current software design. Some examples of such languages are SpecC[27], which is an extension of the ANSI C programming language, and the C++ library extension SystemC[28]. With these languages, the communication between the processing elements is firstly and abstractly defined as channels. This way, the behaviors can be implemented independently from the communication, which is coded on a later step[29]. Another advantage factor using system description languages is the possibility to reuse, exchange and integrate IP(intelectual property) at various levels of abstraction.

As referred before, there is clearly the need of describing digital systems at a higher level before their low-level (Register Transfer Level) implementation begins. The referred languages for high-level simulation have used a more specific nomenclature to modelling digital systems, which is called Transaction-Level modeling (TLM). This approach is followed by SystemC since its 2nd version release, denoted as SystemC 2.0. The goal is to separate the implementation of functional units and communication. Communication mechanisms are presented to modules as interface classes. This way, transaction requests are done through interface functions, which encapsulate low-level details of the information exchange. As a consequence, the emphasis is more on the functionality of the data transfers (what data is transferred to and from what locations) and less on their implementation. This way, it is easier for the system-level designer to test and evaluate different architectures without having to recode the modules. This is why SystemC will be used to implement the architecture of the proposed system.[30]

2.2.1 SystemC

The first version of SystemC was announced in 1999 and was promoted by OSCI, the Open SystemC Initiative. It started out as a very restrictive cycle-based simulator and it was seen as another RTL language. 2002 was the time for a great leap forward, because SystemC became a formal system level modelling language. In 2005, SystemC was ratified as IEEE Std. 1666™- 2005.
This language was built based on standard C++, by extending the language with specific class libraries, and by providing an event-driven simulation kernel in C++. With this environment, the designer is able to simulate concurrent processes using plain C++ syntax. Moreover, SystemC processes can communicate in a simulated real-time environment, by using signals of all data types offered by C++ and some additional ones offered by the SystemC library, as well as those that are user defined. In certain aspects, SystemC deliberately mimics VHDL and Verilog hardware description languages, but is more aptly described as a system-level modeling language.

SystemC supports either hardware, behavioral and register abstraction levels, it is possible to simulate a system whether it has hardware components or software components or even both. Although SystemC has several semantic similarities to VHDL and Verilog, when used as a hardware description language it introduces some syntactical overhead when compared to these. On the other hand, it offers the object oriented environment due to C++. Although it is still strictly a C++ class library, SystemC is sometimes viewed as a language in its own right. In fact, the source code can be compiled with the SystemC library to provide an executable that is capable to be run in every ordinary personal computer. When used for register transfer level simulation, the offered performance, however, is typically less optimal than commercial VHDL/Verilog simulators.

As referred before, SystemC includes the common set of hardware description language features, such as structural hierarchy and connectivity, clock cycle accuracy, 4-state logic (0,1,X,Z) and bus resolution functions. Upon version 2 release (2002), SystemC also includes abstract ports and timed event notifications. Abstract ports (also called as channels) brought TLM (Transaction Level Modelling) into SystemC, that is, modelling of systems above the RTL level of abstraction.

SystemC provides several types of data. Among them, `sc_bit` and `sc_bv` represent a bit and vector of bits correspondly, `sc_time` is able to store time simulation with a time resolution from femtoseconds up to seconds, `sc_logic` can take four values '0' for false, '1' for true, 'Z' for high impedance or 'X' for unknown, `sc_fifo` simulates a fifo channel and `sc_mutex` and `sc_semaphore` are used as a mutex and a semaphore respectively. Furthermore, as a C++ library, SystemC is able to use C++ data types including the floating point type. There is also a set of primitives that define ports in order to establish communication between modules. The ports can be one way only (in or out) or 2 way (bidirectional) and its type of data must be defined in the declaration statement.

In SystemC environment there are various ways for process/thread control. In order to simulate the expected behaviour when a module receives a signal through a port, the port that will receive the signal must be added to the sensitivity list on the constructor code pointing the process/thread that will be run. Sometimes, there is the need to simulate certain events inside the modules, separately from exterior processes. This brings the use of the event declaration, according to the designer needs. As an example, consider the existence of two processes p1 and p2. Process p2 waits for event e1 notification, so it can write some value. Only after p1 notifies event e1, will p2 be able to write. Figure illustrates this example.

Modules behavior and functionality are implemented through three types of processes: methods, threads and/or clocked threads. Methods are the simplest SystemC processes. Their execution is triggered by signal events. When an associated signal is changed, the method is invoked and will run until the end of the routine is reached. Threads are similar to methods. However, their execution may be suspended and re-triggered countless times. Upon a trigger event, the
2.2 System Description Languages

execution resumes from where it left off and run until the next suspend instruction or the module's end is reached. Clocked threads are a specialization of thread processes. Their sensitivity list is limited to one edge of a single clock input that matches the way a clocked hardware device typically functions. A clocked thread can be seen as a watchdog.

```c
SC_MODULE (first_counter) {
    sc_in clk     clock ;   // Clock input of the design
    sc_in<bool>   reset ;   // active high, synchronous Reset input
    sc_in<bool>   enable;   // Active high enable signal for counter
    sc_out<sc_uint<4> > counter_out; // 4 bit vector output of the counter
    sc_uint<4>    count;
    //----------------- Local Variables Here------------------
    // Below function implements actual counter logic
    void incr_count () {
    // At every rising edge of clock we check if reset is active
    // If active, we load the counter output with 0
        if (reset.read() == 1) {
            count = 0;
            counter_out.write(count);
        // If enable is active, then we increment the counter
        } else if (enable.read() == 1) {
            count = count + 1;
            counter_out.write(count);
            <<counter_out.read()<<endl;
        }
    } // End of function incr_count
    SCCTOR(first_counter) {
        SC_METHOD(incr_count); 
        sensitive << reset << clock.pos();
    } // End of Constructor
};
```

Figure 2.5: 4 bit counter example in SystemC.

In figure 2.5 it is presented the same 4-bit counter example, written with SystemC. The first four lines of code contain the declaration of the ports, including each port direction and data type. The next line represents the counter variable, which is an unsigned integer composed by 4 bits. The process responsible for the counter update is defined as a method that will be run every time 'reset' port has its value changed or the 'clock' port receives a rising edge. This process is defined as a method inside the constructor ("SCCTOR") and is sensitive to any changes that occur in the reset port as well as to any rising edge on the "clock" port. In this scope, sensitive means that the
2. System Description Languages

"incr_count" function is run every time an event occurs on both ports.
3

Related Work

Contents

3.1 $I^2C$ Bus Controller ........................................... 18
3.2 CAN Bus Architecture Model .................................. 20
3.3 SystemC Modelling of a Parallel Processor Broadcast Interconnection System 22
3.4 Discussion ............................................................ 24
3. Related Work

To better demonstrate the adequacy of SystemC language and its simulation environment for the purpose of conducting the research driven by this thesis, this section presents some brief descriptions of several other similar projects that were conducted using SystemC. The first example will be an I2C bus controller, followed by a CAN bus design, which is largely used in the automotive industry, and a parallel processor interconnection system. The final section of this chapter gathers a set of factors that led the designers to choose SystemC as the simulation tool of the projects.

3.1 $I^2C$ Bus Controller

The Inter-Integrated Circuit ($I^2C$) bus [32] is a multi-master serial single-ended computer bus, designed at the beginning of the nineties, that is frequently used to attach low-level peripherals to a motherboard or embedded system.

This bus provides communication between a small number of devices on a two-wire bi-directional bus: a serial data (SDA) and a clock line (SCL). Each I2C device integrates a controller composed of two distinct blocks. A digital block manages the protocol, timing and control of specific sequences, while an analog block ensures the access to the I2C bus. Figure 3.1 illustrates the block diagram of a bus controller.

![Figure 3.1: I2C bus controller.](image)

The main features of the I2C bus are summarized as follows:

1. Digital Block Specifications:
   - Interface composed of two wires: SDA and SCL;
   - Transfer rate up to 100kbits/s for standard mode;
   - 7 bits addressing mode;
   - Start/Stop/Acknowledge generation and detection;
   - Arbitration management in multi-master mode, with automatic transfer cancellation when arbitration is lost;
   - Busy bus detection.
2. Data Transfer and Data Frame Protocol:

- A data transfer starts when a falling transition occurs in the SDA line while SCL is high. The data transfer is finished when a rising transition occurs in the SDA line while SCL is high;
- Data is considered valid during the high state of the SCL;
- Each data frame is composed of a start bit, 7 bits for addressing, a Read/Write bit, 8 data bits and 1 stop bit [33].

In the following, it will be presented a SystemC modelling and simulation of an example design of this controller [33]. The main purpose of such description is to demonstrate the potential and feasibility of using SystemC language and its related development tools in early stages of the design. The description of the analog block specifications will not be considered here, because its purpose is to convert from and to digital/analog signals which is not relevant for this thesis. Either way, it is important to say that SystemC-AMS (Analog Mixed Signal) was used in order to simulate the electrical signals of the simulation and hardware implementation of this block. SystemC-AMS is an extension library to SystemC about analog and mixed-signal.

Figure 3.2 presents the SystemC model of the digital block. There are three major modules:

- A micro-controller interface handles all data transfers between the master and the I2C controller. It also interprets the requests from the master to the remaining blocks of the architecture.
- The core of the system is a sequencer, which translates a simple order coming from the master into a detailed sequence respecting the I2C protocol. It is composed of a set of finite state machines.
- A signal generator module handles the behavior of the SCL and SDA lines, according to the sequencer commands.

Figure 3.3 presents the simulation results of a I2C controller using SystemC. It illustrates an example of a read operation in a slave peripheral. At the end of the implementation described
3. Related Work

In [34], the designers were able to successfully simulate and test the I2C bus protocol. More specifically, the simulation results showed that the generated signals are conform with the I2C protocol. After this, the designers added this model to the SOCLIB library (open platform for virtual prototyping of multi-processor system on chip) in order to simulate it as part of a much wider SoC architecture, that includes more microprocessor cores and memories to certify its reusability by other developers [34].

3.2 CAN Bus Architecture Model

In this section, it will be described the modeling and simulation of a CAN bus architecture [35], which is widely used in several domains like industry automation and automotive systems. The implementation of the CAN bus was conducted in SystemC and its main purpose consists in verifying hard real-time constraints, as well as some functional and non-functional properties.

![Can bus controller architecture](image)

The CAN controller is composed of four modules, as depicted in figure [3.4] They are the Message Buffer module (MB), the Interface Management Logic module (IML), the Bit Stream
3.2 CAN Bus Architecture Model

Processing module (BSP) and an Error Management Logic module (EML). The MB is used to store messages (CAN frames) coming from or going to the bus. The IML module is responsible for transforming data and the sender ID received from the host into CAN frames. Next, the BSPU is in charge of the reception (de-serialization) and transmission (serialization) of the CAN messages. It performs bitwise arbitration, error detection, acceptance filtering and CRC checking. Finally, the EML is the module that implements fault confinement rules, according to the CAN specification described in [35].

Figure 3.5 presents the general architecture of a CAN implementation using SystemC, which is composed by six components:

- **Stimulus generator** - used for input stimuli generation for the CAN controller. In this case, the stimulus generator may create specific or completely randomized messages for the Design Under Test (DUT), based on the considered parameterization.

- **Transfer Function** - simulates the input and generates the expected output using a reference model of the CAN controller. This means that the transfer function module receives the CAN frames generated by the stimulus generator as input. Then, these are taken to compute the expected output of the DUT, in order to compare it with the actual output produced by the prototype, in the Logging and Acceptance (LA) evaluation module. Another important task is to compute the transmission durations of the frames (in clock ticks) and store them in the scoreboard of the LA module.

- **Driver** - the task of the driver module is to convert the message objects produced by the stimulus generator into CAN frames, before driving them to the DUT. This is a needed step because the stimulus generator produces standard data types (like char and int) and the CAN model uses sc_logic or logic vectors.

- **Coverage monitor** - it collects information based on the output of the stimulus generator and on the output of the DUT for later scoreboard, according to pre-defined metrics.

- **Logging and Acceptance evaluation** - it is used for data logging and acceptance evaluation based on a scoreboard. It is used for system analysis and for logging of generated and received CAN frames (produced by the generator and received from the DUT). Later, the LA module executes a comparison (and verification) of the output provided by the transfer function module and the output of the DUT.

After its implementation and several test scenarios, the designers were able to identify various design flaws and bugs of the CAN bus model [36].

In order to proper testing the CAN Bus, several test scenarios were made using various CAN Bus instances:

- **Sender/Receiver Scenario** - this scenario verifies the CAN data packet transmission via the CAN bus from a single CAN controller to two CAN controllers.

- **Bus Arbitration Scenario** - in this scenario, three CAN controllers start sending a CAN packet onto the CAN bus model at the same time.

- **Priorities and Arbitration Scenario** - this scenario combines both previous scenarios. This way, the CAN bus arbitration was tested in a multi-sender and receiver CAN network.
3. Related Work

• Heavy Scenario - as a last example, a CAN network was built containing 2032 CAN controllers where each one tries to sends CAN packets through the bus.

![Figure 3.5: CAN bus Testbench model.](image)

3.3 SystemC Modelling of a Parallel Processor Broadcast Interconnection System

Finally, the last project presented here is about the development of a large complex hardware/software system that is the heart of a parallel processor interconnection architecture, and it was accomplished at the University of Alabama, in Huntsville. The developed system allows the investigators to vary the parameters of the system workload, the policy for message passing protocols, and, on the hardware side, the size of elastic buffers and the DMA controller burst size.

![Figure 3.6: System Architecture.](image)
3.3 SystemC Modelling of a Parallel Processor Broadcast Interconnection System

a global bus arbiter, one serial transmit module and N serial receive modules (N represents the number of processors in the system).

Each processing node is known as a network node that will generate data and send it to other nodes (randomly selected) in the net. For the purpose of the simulation, the inter-processor data is transmitted in simple packets, whose format is given by table [3.1]. Three packet types are modeled in the simulation. The first type is a point-to-point data packet, directed from a single node to a single target node (single-cast packet transmission). The second packet type is concurrently transmitted from a source node to all network nodes (multi-cast packet transmission). The last packet type is an acknowledge packet sent in response to a received packet. The following sections describe how the entire system works. A brief description of the SystemC implementation of the elastic buffers that lie within the transmit and receive modules of the system will be also presented.

<table>
<thead>
<tr>
<th>Packet Location</th>
<th>Packet Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start of Packet Code</td>
</tr>
<tr>
<td>2</td>
<td>Packet ID</td>
</tr>
<tr>
<td>3</td>
<td>Packet Type</td>
</tr>
<tr>
<td>4</td>
<td>Source Address</td>
</tr>
<tr>
<td>5</td>
<td>Target Address</td>
</tr>
<tr>
<td>6</td>
<td>Data Section Word Count</td>
</tr>
<tr>
<td>7</td>
<td>First Packet Data Word</td>
</tr>
<tr>
<td>8</td>
<td>.....</td>
</tr>
<tr>
<td>9</td>
<td>End of Packet Code</td>
</tr>
</tbody>
</table>

3.3.1 Node Module

The node processing element is the source and destination of all data transfers. Three main functions are used to implement its body. The first is intended to process packet arrivals, the second to generate new outgoing packets and the third one to process the outgoing packet transmission process. The processor is regarded as a message generator that produces data to be sent (packets) at a rate determined through a parameters file. When outgoing packets are created, the type of the sent message is selected randomly (single or multi-cast packet transmission) and in case of single-cast packet transmission the target processor is selected randomly too.

The global memory is used to hold incoming and outgoing communication packets. It is implemented as a dual-port random access memory with a processor bus interface and the global bus interface. The aspects concerning the size, data width and access delays are parameterized through the simulation parameters file. The buffer’s global bus interface employs tri-state signals for its data path, since several devices share the same bus.

The global bus arbiter is responsible for controlling the access to the global bus. Every node element within each processor needs to access the global bus in order to exchange information with the global memory module. The global bus arbiter employs hardware handshakes to coordinate the activity on the global bus. Each node that intends to use the bus must request the bus and wait for a bus grant before it may access the bus.
3. Related Work

3.3.2 Transmit Module

The transmit node is the channel used to send data out of the processor (network node) and is composed of a transmit shift register, a FIFO and a DMA unit. The figure 3.7 shows a functional diagram of this module.

The transmit shift register converts parallel data that arrives from the transmit FIFO to a serial data stream that is transmitted through a network datalink.

![Figure 3.7: Transmit Module.]

The transmit DMA unit automates the transfer of outgoing packet data from the global memory buffer to the transmit FIFO. During a packet transmission, the DMA unit must repeatedly request the global bus in case the packet is too large to access in a single burst.

3.3.3 Receiver Module

The receiver module is composed of a receiver shift register, a receiver FIFO, a receiver controller, a local buffer which is a dual-port memory device and a DMA unit. Figure 3.8 shows the functional block diagram.

The receiver shift register accepts incoming serial data from one of the network fiber-optic data links, converts it to a parallel data word and writes it to the receive FIFO. The receive FIFO is identical to the transmit FIFO. It provides an elastic buffer necessary to prevent receiver overruns when the node’s global bus is congested.

All the data that is sent to the local buffer memory is checked for errors (in the receiver controller) and only when valid data is received, a DMA transfer from the buffer memory to the global memory is initiated.

3.4 Discussion

One of the issues in SoC design has been the multiplicity of environments and modelling languages used to describe the software and hardware parts of the system, but also its digital and analog elements. SystemC has provided an answer which is completed by the development of a SystemC-AMS library [38]. With a combination of both languages, it became possible to design an entire mixed-signal IP of the I2C bus interface with a common language.

The test scenarios performed on the CAN Bus were accomplished using the SystemC Verification Library, which allows to randomly generate CAN packets and drive the stimulus generator.
This way, it became possible to lead simulation runs into faulty situations, find corner cases and identify design flaws and bugs. Being able to control the testbenches also allowed to run the test scenarios using different CAN controller configurations. Running simulations through SystemC makes data collecting an easy task. That is why the authors of this work were able to efficiently collect, evaluate the performance of the CAN Bus and controller and draw conclusions on its performance when faced to different case scenarios.

Regarding the final project that was presented, the authors were able to run several simulations using completely different configurations. The system is completely configurable through parameter files that set the various settings such as clock period, global bus clock period, processor clock period, bus width, buffer’s sizes, Fiber-optic data link delay, number of system processors and packet definitions. Run time parameters change too often during a parametric study for simulation rebuilds to be practical. This way, a batch process was defined, which generates a series of run time parameters and calls the simulation executable in a repetitive fashion.

During these projects development, the ability from SystemC to write value dump files (VCD) was used to allow designers to visualize the system performance in a graphical way, as showed by figure 3.3 in the first project case.
Simulation framework architecture

Contents

4.1 Simulation Framework .................................................. 28
4.2 Simulated Device .......................................................... 31
In this chapter, the simulation framework architecture will be presented. The goal of the simulation framework is to provide a simulation environment, where one or more processing units are working together in parallel mode simulating a multi-core processor. More specifically, the target application context is focused on DNA sequence comparison. The process is composed by several and repetitive steps, where one long DNA sequence is compared against one small DNA sequence, so that huge datasets composed by a very great number of DNA sequences are aligned. This comparison is executed through an algorithm known as the Smith-Waterman algorithm [3]. In other words, the created solution is a simulation environment of an heterogeneous multi-core processing structure targetted for bioinformatic applications.

The framework was developed using C++ language, along with the SystemC version 2.2.0 library. This way, it became possible to build (simulate) a project where various processes are running at the same time and communicating with each other. The implementation level is focused on the transaction level process. This means that the resulting simulations of the framework provide feedback on the main events that take place inside the processors, memories, buses, etc. It also provides feedback about the messages and protocols used in the communication between the modules. At the end of the simulation procedure, the programmer is able to track the several stages and status that the modules have been trough. This is done via an event log that is created during the simulation process. With it, the programmer is capable of building a plot or chart using ploticus [39] software package.

During this chapter, the architecture of each component will be described, as well as its purpose, behavior and the way they communicate with each other. Figure 4.1 shows the components that are present in the system. The system is divided in two great areas. The first one is the simulation environment, that contains the simulated device and the modules responsible to support/provide feedback to the programmer during and at the end of the simulation. The second one is the actual simulated device. This area incorporates the processors, memories and buses that are meant to be configured and simulated.

Figure 4.1: Environment System Model
4. Simulation framework architecture

4.1 Simulation Framework

As stated before, the simulation framework is composed by the stimulus and the monitoring modules. These will be used by the programmer in order to provide data to the simulated device and to monitor it. The simulated device can be seen as an independent one, that is inserted into the simulation framework for the specific purpose of testing.

![Simulation Framework](image)

Figure 4.2: Simulation Framework

4.1.1 Stimulus Module

The stimulus module is the starter of the system. When the execution of the simulator is started, the stimulus is the only one awake, except the memory that is always active and only reacts to write and read operations. Its job is focused on providing the system with test data.

The stimulus first step is to read data from the test data files, which in this case have DNA sequences. As data is read, the stimulus sends its content to the memory, along with an unique identifier. For every query/reference that is sent to the memory, a local table saves the address (at the memory) where the data (query/reference) has been stored and its size. After this operation, the table is passed to the master module. The final step of the stimulus is to wake up every processor in the system. Firstly, the processors are awaken because these will do nothing until some data is provided to them. The following module to be awaken is the monitor and finally the master.

4.1.2 Monitor Module

The monitor is in charge of tracing the status of the simulated device. The modules included in this operation are the processor units which are the master and slave modules. These modules are described in the following section. Every time a module changes its status, the event is sent to the monitor and recorded as shown by figure 4.3. The status of the module changes depending on the operation that is going through. The simulated processors can be in one of five states that are known as:

- yield status.
- read status.
• write status.
• executing status.
• bus waiting status.

The yield status means that the module is currently not performing any action. These cases occur when the module has no data to process or to send to any other module in the system. It happens when one processor is waiting for orders or process results coming from other processors. The slaves enter in this state when are awaiting for incoming orders from the master.

The read and write status occur when the modules are reading/writing data from/to the shared memory. Each processor writes data in the shared memory every time it needs to send data to other processor. After the message has been written in the memory, the sender module must signal the receiver module. In order to do so, the sender sends a command through the bus directly to the receiver module, bypassing the shared memory. Finally, a module enters the read status every time it reads data from the shared memory.

A module is in a processing state when data is being processed within it. In a master-slave configuration, the master processor sends data to the slaves through the bus as long as the target slave is not busy. Before this step, the master needs to prepare the message and choose a target slave to send it. After the slave has finished its processing and has sent its results back to the master, the master enters into process state again, in order to process the received results. This happens the same way on the slaves, because every time a slave receives data it will surely be processed according to the algorithm implemented in it.

During the simulation, the master and slaves send various messages through the bus. Every time this occurs, the sender module must receive permission from the bus arbiter in a previous step. As long as the module is waiting for permission, it will be inactive (bus waiting status) due to the spent time waiting to access the bus.

As stated before, the simulation framework (monitor) creates a log file tracing all the modules status and events and corresponding timestamps that can be compiled by the ploticus software package in order to generate a chart. To make this possible, the text file needs to be written according to the ploticus log file syntax, as depicted by figure 4.4. The line represents a status where processor number one is reading data from the memory from time 405 picoseconds to 446 picoseconds, according to the main clock of the simulator. In order to achieve all this data at
once, some decisions regarding data structure and time events had to be made. So, every time
a module changes its status, the information provided to the monitor concerns only the previous
status that has been through. Let’s say that a processor is reading data from memory from time
T1 to T2 and then starts processing the data. At the point in time the change from read status to
processing status takes place (T2), the monitor writes in the log file that the slave has been reading
from time T1 to time T2. During this operation, the monitor saves the timestamp 2 temporarily,
until the processor changes its status again, in order to know the time that the processing status
has begun. Consequently, a time T that represents the end of one’s status, also represents the
beginning of the next status of that same module. At the end of the simulation, the log file will look
like the one presented by figure 4.5.

Figure 4.4: Ploticus text file syntax

# All measures are in pixels
ImageSize = width:1600 height:1200
PlotArea = left:20 right:20 bottom:20 top:20
AlignBars = justify
DateFormat = yyyy
Period = from:0 till:10000
TimeAxis = orientation:horizontal
ScaleMajor = unit:year increment:10000 start:0
# there is no automatic collision detection,
# so shift texts up or down manually to avoid overlap
PlotData=
  width:50 mark:(line,white) align:left fontsize:5
  bar:P0 from:start till:4 color:purple
  bar:P0 from:4 till:58 color:blue
  bar:P0 from:58 till:58 color:blue
  bar:P0 from:58 till:58 color:blue
  bar:P0 from:58 till:59 color:black
  bar:P1 from:start till:118 color:blue
  bar:P1 from:118 till:159 color:green
  ...
  bar:P0 from:93028 till:93029 color:black
  bar:P0 from:93029 till:end color:purple
  bar:P1 from:92629 till:end color:purple

Figure 4.5: Monitor log file format.

Figure 4.6 illustrates a simple example using four slave units, where eight alignment processes
take place in a certain order and point in time. The length of each block represents the time period
spent by the corresponding process. The three colors in each process represent three different
stages: the red color represents the time period to get all the required data in the memory, the
blue color represents the time period spent executing the alignment algorithm and finally, the
green color represents the time period that the process spends to send back the output data.
(the score result from the alignment code). The entire alignment time will comprise whole period, starting at the beginning of ‘P1’ until the end of ‘P8’. On the other hand, if all the processes run in sequential mode, the total time spent by the system would be the sum of all processing times.

Figure 4.6: Example of a concurrent execution of 8 alignment processes using four slave processors.

4.2 Simulated Device

The simulated device is composed of one master processor, one shared memory device, one bus that interconnects all the modules, one arbiter and a variable number of slave processors that are predefined before build time. The architecture is illustrated in figure 4.7.
4. Simulation framework architecture

4.2.1 Modules

The following subsections will present the description of each module within the simulated device, in what concerns its purpose and behavior. The way they work and how they communicate with each other will be also described.

4.2.1.1 Master

The master module is the manager of the simulated device. It sends orders to the slave modules and they answer back with the results of the processed data. This is the main procedure that happens inside the simulated device. This way, all the data that runs in the device is a consequence of the orders commanded by the master module. In this case, the data that is being processed is related to DNA queries and reference sequences. So, the master provides queries and reference sequences to the slaves and these become in charge of executing the DNA alignment algorithm and send back the result.

During the simulation of the device it is assumed that the stimulus has preloaded the shared memory with data or queries/reference sequences and that the master module is preloaded with a table with the corresponding addresses and sizes of the data that has been saved on the shared memory. Hence, for each command to be executed in this multi-core processor, a collection of ordered actions must take place. Before describing these important steps, it is crucial to know that all the messages that travel through the bus are subject to the bus width and consequently need to be divided in several small messages, according to the width of the bus. This process, as well as its details, will be presented in a later section of this chapter. The following topics are described according to figures 4.8 and 4.9:

1. Firstly, the master picks a record from the table that containing the addresses and sizes of the DNA sequences previously stored in the memory. It builds a message with that information and sends it through the bus, directly to one of the available slaves. This message simply transports an order or command along with the address and size of the needed data, available at the shared memory, in order to implement the requested command. (step 1, in figure 4.8)

2. As the message arrives at the slave, it starts reading and interpreting the incoming command. Then, it makes an attempt to access the bus in order to get the data from the memory. As soon as the bus access is granted, the data starts being transferred. (step 2, in figure 4.8)

3. Once the slave has all the needed data, it starts running the alignment task, corresponding to the algorithm using the queries/reference sequences provided for that job. After completion of the algorithm, the result is written into a message. (step 3, in figure 4.8 and step 4, in figure 4.9)

4. The slave sends the message with the results of the alignment to the shared memory. (step 5, in figure 4.9)

5. The same slave sends a command message to the master with the address and command signaling that it has deposited a result from the Smith-Waterman algorithm with the previously provided data. (step 6, in figure 4.9)
6. Finally, the master reads the command that was sent from the slave and makes a read request to the bus to access the memory. Then it reads the corresponding result.

In order to improve the bus usage efficiency, the master module is always aware of the orders that were sent to each slave. For each order that has been sent to a slave, this one is considered as busy until an answer message with the results from the ordered command arrives. Hence, the slaves will receive orders only when they are not busy and as long as there is data to be processed. This process is done via unique identifiers that are assigned to each order. Without this centralized control, the master module would be constantly requesting the bus access as long as there exist tests to be run and consequently spoil the throughput of the system.

4.2.1.B Slave

The slave modules are the main processing cores that integrate the simulated device. They are in charge of most of the computational demanding job carried out by the device. Their job is characterized by a passive attitude, as they only start working only after receiving orders from the master module.

During the performed simulations, the job of each slave consists in executing the DNA matching procedure to compute the best matching score value between two DNA sequences. At first, each slave receives an order from the master processor pointing to the sequences pair is stored in the shared memory. Then, the slave makes requests to the bus in order to read the data from the shared memory. The next step consists in running the implemented software (Smith-Waterman algorithm) upon the provided data by the master module.
4. Simulation framework architecture

The simulated device has a built in shared global memory which is used to transfer data between the master and the slaves. Within the scope of the project, it may be regarded as a big deposit where the DNA queries and reference sequences are stored before being processed with the Smith-Waterman algorithm.

The memory provides two simple operations: read and write. In order to run a write operation, it is only necessary to provide the data to be written and the address where the data will be stored. The same happens in case of a read operation. This means that the implemented software on the master and on the slaves are completely responsible for what happens within the memory. There are no check procedures for the addresses specified by the write and read operations.

When a module intends to read or write data in the memory, it makes requests to the bus in order to gain permission to access the bus and run operations in the memory. But actually, the bus is the one who accesses and executes the operation(s) (write/read) in the memory. It works like an interface for any module who wants to access the shared memory.

Figure 4.10 illustrates write and read operations that usually occur during simulation. Transaction number one represents the moment before simulation starts, when the stimulus module is loading the memory (write operations) with test data (step 1). The next memory transaction takes place when a slave module executes read operations in the memory, to obtain test data (step 2). After processing the test data, the slave writes the processing results in the memory (step 3). Finally, the master module reads from the memory to get the results of the test data (step 4).

This memory has a simple and particular addressing space. It is like a simple vector where...
each cell represents a single byte. In order to point a position within the memory, a simple integer address is needed, as long as its value is within the range of the memory. As an example, let's say that the memory capacity is equal to four megabyte. In this situation, the memory is represented as a vector with 4,194,304 cells and will be addressed from zero to 4,194,303. By simulating the memory this way, it becomes an easy and simple task to the programmer when writing code that will access the memory.

At this point, it is important to describe the address space that is made available to the master and the slave modules. Every one is able to access the system's shared memory, its own internal memory and its own mailbox. The shared memory is usually used to exchange data between modules, whereas the mailboxes are used to transfer command messages. Each module may also use its internal memory, to implement its own algorithm. Figure 4.11 shows the address space made available to each module. The shared memory is contained in the range from zero until its size minus one, corresponding to "MAIN_MEM_ADDRESS_RANGE" address. The addresses corresponding to the modules internal local memory are contained right next to the shared memory's range corresponding to a size of "LOCAL_MEM_ADDRESS_RANGE". Lastly, the mailboxes address range is not continuous to the local memory addresses, since the size of each module's internal memory may vary from one to another. This way, the mailboxes address range is placed at the ending limit of the whole addressing map so that it can be fixed and equal to all.
4. Simulation framework architecture

4.2.1.D Bus

The bus is the main entity to make communication possible within the simulated device. It is the bridge for data and command messages traffic between master, slaves and the shared memory. Master and slave modules use it as an interface to access the shared memory and to send messages between each other.

Communication through the bus is done in two different ways. Whenever one module (M1) wants to send data to another module (M2), it does it through the shared memory. After that, the M2 module will read the corresponding data from the shared memory. This is made possible because right after writing data to the shared memory, M1 sends another message (command) directly (through the bus) to M2 with the needed information (address, size, etc.) about the data previously written in the shared memory. In an example scenario, where M1 is the master module and M2 some slave module, the process will repeat, but in reverse order, when the slave sends the result back to the master module. Firstly, the slave saves the result data in the shared memory and then sends a command message to the master. Figures 4.12 and 4.13 illustrate the two types of messages travelling through the bus.

The other way to use the bus corresponds to using it as an interface for straight communication between master and slaves and is limited to command messages only. Command messages are expected to be short (in length), containing only important parameters and commands. That is why they travel directly from one module to another one through the bus, skipping the shared memory. If every module had the possibility to send DNA data directly to other modules, the bus would become a real bottleneck because every request would take a long time to be accomplished, due to the amount of data needed to be transferred between modules.

4.2.1.E Arbitrer

The arbitrer plays an important role in the simulated device. In fact, every message that travels through the bus has to be previously granted. As a consequence, the entire communication
between master, slaves and memory depends on the arbiter module. This important element is responsible to apply the necessary rules on the communication process, so that bus traffic is made efficiently and in an orderly way.

During simulation, bus access is constantly being requested by the master and slaves, but only one module at a time can access the bus. For every request, its sender and type of message is first checked in order to assign a level of priority, so that the arbiter can decide which module to grant bus access. Table 4.1 shows the assigned priority levels, according to the sender module and type of operation. The master module plays an important role managing the data that has to reach the slave modules before being processed. This way, the requests from the master module
are very important, thus are all considered as high priority. On the contrary, in order to simplify, the requests of the slave modules are assigned with low priority.

Priority level 0 represents the top level priority. Priority level 0 and 1 are the most important because the corresponding operations precede a critical point during simulation, which is when a slave module executes the alignment procedure. Priority levels 2 and 3 are the least important because these operations are executed after the alignment procedure.

<table>
<thead>
<tr>
<th>Priority level</th>
<th>Sender Module</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Master</td>
<td>write</td>
</tr>
<tr>
<td>High</td>
<td>Master</td>
<td>read</td>
</tr>
<tr>
<td>Low</td>
<td>Slave</td>
<td>write</td>
</tr>
<tr>
<td>Low</td>
<td>Slave</td>
<td>read</td>
</tr>
</tbody>
</table>

When a module requests a bus access, it will probably have to wait due to the amount of pending requests during the simulation. When this occurs, the request is inserted in a request queue depending on the request priority. There are three request queues. One for high priority requests, one for medium priority requests and another one for low priority requests.

After the bus has been released, the arbiter picks the next request that is inside the highest priority queue. Therefore, as long as there are requests in the high priority queue, the requests of the both medium and low queues will not be attended. The same logic is present when there are requests only inside the medium and low priority queues. There priority queues work in a first-in first-out manner, so, the requests of each queue are attended as they arrive. The figure 4.14 illustrates the three implemented queues.

As stated before, communication is done via two different ways. When one module needs to send a message to other one (M1 to M2), M1 sends data to the shared memory in a first step and then sends the command message directly to M2. These two types of communication are treated in different ways too. If one module requests the bus in order to write or read DNA data to/from the shared memory, the bus will be assigned to that module during a certain amount of consecutive bus accesses, limited to a specific number of accesses. During these period, the requesting module will not have to wait to access the bus whereas all other modules are waiting for their turn.

In case a module requests the bus access in order to send a command message, once the bus is assigned to him it will be granted for one single access to write the command message
for once. This means that a transmission of a command message is assumed to be an atomic procedure. Hence, it will not need to make several access bus requests in order to send an entire command message.

4.2.1.1 Mailboxes

As described before, the processors communicate with each other via two different ways. Sending data from one processor to another one is done via the shared memory and sending instructions is done via a mailbox that each processor has. The messages that are transmitted through the mailboxes represent instructions that one processor is sending to another one. Therefore, these messages tend to be short in length and contain only important parameters that are used by the receptor processor to execute a specific instruction. This way, the instruction messages usually transport address and size parameters, so that the receptor processor can read data from the shared memory in order to execute the related tasks from the implemented algorithm. Every processor in the system contains a mailbox as illustrated in figure 4.15.
## Framework Implementation

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 SystemC briefing on concurrent applications</td>
<td>41</td>
</tr>
<tr>
<td>5.2 Main Modules Implementation</td>
<td>43</td>
</tr>
<tr>
<td>5.3 Communication protocol</td>
<td>57</td>
</tr>
</tbody>
</table>
5.1 SystemC briefing on concurrent applications

The previous chapter described the architecture of the entire system. Each module was described in what concerns its purpose, how they work and react to events from the exterior. At the same time, it was possible to demonstrate how the test data is provided to the system, transmitted inside the system and processed according to an implemented algorithm on the unit cores.

Despite the structure of the current chapter is quite similar to the previous, the main focus of this one is different. Each section describes how the modules of the system have been implemented using SystemC. It includes the description of the main classes, functions and variables that are present in each module. The way how the module’s classes/functions interact with each other will also be described.

Although SystemC can be seen an extension library of C++, it can be regarded as an independent language, though its syntax and semantics still look like C++. Hence, some distinctive concepts and syntaxes (when compared with C++) will be introduced. The brief description of some concepts related to SystemC that is presented in the following sections represent some of the points that allow the simulation of several processes running in parallel mode and at the same time communicate with each other.

5.1 SystemC briefing on concurrent applications

Before moving on to the implementation of each module, it is important to present and describe the main concepts of SystemC that were used to implement this system. This section presents a brief example of a concurrent implementation of the popular "producer-consumer" problem, which will serve to introduce some of the most important concepts that were applied in the implementation of the aimed multi-core simulation framework.

Figure 5.1 depicts a simple implementation of the considered example. There is one producer, one consumer and one buffer with five data slots. The producer creates data from time to time and writes it into the buffer as long as there are free slots in it. Every time a slot gets filled in the buffer, a signal is sent to the consumer which will read the data from the slot as soon as possible.

When a certain system is composed of several modules, the way how to interconnect them is one of the first subjects which is discussed and defined. For such purpose, SystemC provides the programmer with ports that can transport any type of data known from C++ and SystemC. These ports can be used either to simply transport data from one point to another or to implement interfaces.

Implementing a simple connection requires only the definition of two ports (one in each module) and the specification of the type of data that will travel through it. The source code corresponding the buffer and to the consumer, shown in figures 5.2 and 5.1, respectively, contains the definition of a boolean port that travels from the buffer to the consumer. Every time a slot from the buffer gets filled, the current value of the 'new_message' port is changed. This change fires an event within the consumer module, because this port is assigned to the sensitivity list which fires the read method.

An interface between the two modules is just a channel that is defined through a C++ function. Figure 5.1 shows the source code of the both producer and the consumer. Each of them may communicate with the buffer through port "buffer" ("sc_port<fifo_if>buffer"). This connection is defined through an interface which is called 'fifo_if'. The implemented buffer interface provides two different methods to access the buffer: the read and write operations. By using this methods,
5. Framework Implementation

The modules that connect to the buffer through this interface may read or write the data stored in the buffer. The implementation of these operations concern only the buffer, as the producer and the consumer face it like a function in a black box (they provide data input and wait for the output).

In SystemC, functions may behave as usual in C++ and may also be classified as a method or thread. A function declared as a method becomes a process that is fired after certain events have taken place, like ‘read_slot’ from the consumer module, which is sensible to the "new_msg" entry. A method is assigned to the sensitivity list of a module and can be fired upon several events. Events may occur either from ports, triggers, semaphores, mutexes and from a specific SystemC type of event called "event". An example is presented in the last paragraph of this section, after introducing threads and using a SystemC event showing a different approach to this problem.

A function declared as a thread can only be run once during the simulation, contrastly to methods. Threads are not fired due to events but are sensible to them during its execution. Figure 5.1 shows a simple example where the producer implements its main procedure. The "wait" statement inside the "while" cycle is a common way used when designing systems with SystemC. This way, the producer’s main procedure is run every time a specific event occurs. In this case, the producer produces data and writes it on the buffer every two hundred nanoseconds. A different approach could be an active clock entry that would be assigned to the sensitivity list of the "main" thread and the producer would make his job every time that wire's value got changed.

Hence, when the buffer signals the consumer that there are new slots filled with data, a different approach would be: the consumer implements a main thread (just as the producer does), but instead of assigning the thread sensitivity to a timer or port, an event of type "sc_event" would be defined, as shown in figure 5.3. Every time the buffer receives data, the event called "new_data" is triggered and the "wait" statement on the consumer side gets unlocked. However, although it
5.2 Main Modules Implementation

Before starting the description of the implementation of each module, it is crucial to point out that a certain set of modules within the system share a common structure: the master, the slaves.

is an easier approach to the programmer, it may not be as much applicable to reality in hardware systems. The reason why is that, in hardware, in order for a specific component to detect changes or events from other components, some kind of message must be transmitted from one point to another.

5.2 Main Modules Implementation

Before starting the description of the implementation of each module, it is crucial to point out that a certain set of modules within the system share a common structure: the master, the slaves.
and the stimulus modules. All these modules perform common actions such as sending and or receiving data through the bus, memory access or sending messages to other modules. Hence, their main cycle is constantly checking for new messages or commands and react to it. These behaviors are such that the structure of the source code is similar. Figure 5.4 shows the main structure shared by the modules which is called "ModuleStruct". The source code is divided into four parts:

**Ports** - a clock entry is used to keep the main cycle ("main_action") of the module running. Otherwise, the main cycle would run only once and stop for the rest of the simulation. The 'bus_port' is the interface to access the bus so that sending/receiving messages is possible. Finally the monitor port represents the channel used to keep the monitor up with the status of the module.

**Constructor** - the constructor of the module incorporates the main settings that are established during modules construction, which are: size and access delay of its internal memory, a message counter for received messages (used for log purposes), the capacity of the module’s mailbox where the command messages are stored. Finally, the target module which is the identification of the next module that a message will be sent at (‘next_recipient’) is initialized. In order to ease the task of choosing the target module, each one has a unique id, so when a module needs to send a message to another one, all it has to do is to set the address of the message with the address translation function ("getModuleMboxAddress") providing the id corresponding to the target module.

**Methods** - the first four methods represent the main procedures that incorporate a module whether it is a slave, a master or a stimulus. The “main_action” is the procedure that is constantly active within the module, checking for tasks that need to be attended. The “msgRecv_process” method is fired every time the module receives a command message. Whenever the module has data ready to be sent, the “msgSend_process” method is fired and takes responsibility for sending the data. The “init” method is responsible for setting up certain parameters (will be specified in the next subsections) during the beginning of the life period of the module. These methods are declared as "virtual" because although each one has the same purpose, their actual implementation is different in each module.

All the following methods (except "log_status") are used to build, send and receive messages through the bus. The "incoming_msg" method is a thread (defined within the constructor) that is awakened every time the module receives a command message and its job is to increment an integer variable which represents the number of received command messages that have not yet been attended. The next two methods are "send_message" and "send_message_command", whose responsibility is to send a message to the shared memory and to send a command message to other module, respectively. These two methods are also responsible for slicing each message in small pieces, according to the bus width, and sending each one at a time. Whenever a small piece of a message is ready, the "send_operation" method takes charge and sends it. The "check_incoming_message" is run inside the main cycle to check whether there are command messages that have not yet been fulfilled. The "read_message" method is used to read data from local or shared memory, so it is usually run after the module receives a command message and it needs data from the memory to fulfill that command. Finally, the "log_status" method is responsible for reporting to the monitor the status of the module whenever there is a status change.

**Class variables** - As for the protected class variables, there is an internal local memory, represented by "mem", which is characterized by "mem_size" and "mem_delay", regarding to its size and
access delay, respectively. The "my_id" variable specifies the id of the module. "messages" contains the number of command messages that have not yet been attended and "total_msgs" contains the complete number of messages received. The mailbox of each module is implemented through an "sc_fifo" type of data, provided by SystemC. The first and last addresses of the internal memory are provided by "LOCAL_MEM_ADDRESS" and "LOCAL_MEM_ADDRESS_RANGE", respectively.

Mailboxes

Mailboxes are implemented with 'sc_fifo' object type provided by SystemC. It works as a First in First out (FIFO) objet type. Therefore, the instructions that a processor receives are processed in a first in first out manner. The statement "public mbox_if" in the first line of the source code in figure 5.4 means that the module has to implement the methods according to the mailbox interface. The bus module uses this interface to access the mailbox of each one and insert the messages. Figure 5.5 shows the definition of this interface.

Every processor has a mailbox, so the corresponding implementation is located in the 'ModuleStruct.hpp' class too. Figure 5.6 shows the source code responsible for the mailboxes implementation. On the right side, it is the actual implementation of the mailbox interface. When an instruction message travels through the bus to a mailbox, this is the source code that is run in order to insert the message in it. As soon as a message arrives at the mailbox, the event denoted as "new_msg_event" is fired, which triggers an event within the "incoming_msg" function (on the left side, in figure 5.6). This function increments the variable "messages", which is responsible for storing the number of messages that have not yet been attended as described before. After this
set of actions take place, the processor will find out that there is one or more pending instructions in the mailbox through the "messages" counter.

The following subsections describe the implementation of each module and how the main modules use their main structure to fulfill their job during simulation.

### 5.2.1 Stimulus

As described before, stimulus is responsible for providing test data to the simulated device and for initiating its simulation. The stimulus job is short and processed only before the simulation starts.

The stimulus first mission is to read data from one or several input files and to store it in the shared memory. This process is done via three different stages. Firstly, it reads the data from the input file, secondly it sends the data to the shared memory and finally it stores the address (where data has been saved in memory) in table "data", together with an unique identifier. This process is repeated for every test data instance that is read from the input file(s).

Figure 5.7 shows the stimulus source code structure. The first section presents the ports that make the stimulus module unique. The "test_port" is used to send to the master the table of addresses where the test data was saved in memory. The process of starting the simulation of the simulated device is done via 2 steps: the first step is to wake up the slaves and monitor modules, and the second step is to wake up the master module. Waking up the slaves and monitor is done via a vector of ports, each one connected to each processor. Waking up the master module is done via a different approach. The stimulus and master modules share an "sc_event" data type through class "testa.hpp". This data type works like a mutex, with which the stimulus module notifies or unlocks the event/mutex and consequently unlocking the main_action from the master module. All these actions are the body content of the "simulation_start" function, which is in the right upper part of the stimulus source code, depicted in figure 5.7.
5.2 Main Modules Implementation

5.2.1 Stimulus

The parameters of the constructor, as well as the first five methods, are already known from "ModuleStruct" class. In this case, those methods (except "init") are empty because during simulation the stimulus module has no activity and it does not interact with any module, so it does not need to react to received messages neither to sent messages. However, the need to inherit from "ModuleStruct" class is due to the bus interface implementation.

Stimulus task has been completely incorporated into "init" method in an attempt to avoid creating more methods. As figure 5.7 shows, on its right lower part, the stimulus iterates through test data provided from input files and saves it on the shared memory. Thereafter, it sends the table of addresses through the "test_if" interface and runs the wake up call to start the simulation (method "simulation_start" on the right upper part in figure 5.7).

5.2.2 Monitor

Monitor’s module structure is completely based on an interface to the modules of the simulated device. All the modules except the shared memory and the arbiter use the monitor’s interface, so that the entire activity of the system is tracked down. Figure 5.8 shows both the monitor and its interface structure.

Firstly, the "start" port is used only once at the instant in time when the stimulus wakes up the monitor. This event triggers the "activate" thread, as it is assigned to its sensitivity list. This thread starts the monitor, so that it can receive log operations from the simulated device. Monitor's boot is done right after its construction, due to the delays from accessing and writing the initialization
parameters in the log file. The boot method is called “init” and declared as a “SC_METHOD”. Hence, it is run right after the module construction, hence it is run at the same time the stimulus module is loading the shared memory with test data.

One of the most important methods from the monitor’s interface is the “log”. Through this channel, the simulated device’s modules are able to tell the monitor its status changes at the specific time that they occur. The module that is being tracked down needs only to provide its identification (integer value), current status and the point in time when the event took place.

In addition to registering the status tracking on a log file, the monitor also stores all the log data in an array (“OP_TABLE”) of a specific struct. This table stores the identification, status, start timestamp, end timestamp and period of every status that a module has been through the simulation. Figure 5.9 shows the complete struct. This way, the programmer may easily perform statistical calculations at the end of the simulation.

Finally, and as it was described before, due to an imposed limitation of the chart ploticus application, the log tracking should be suspended after a certain amount of of log samples have been registered. This is because it is not possible to build charts with an unlimited number of log lines as generated by the simulations. Therefore, the monitor’s interface provides a method called “finish”, so that the log tracking suspends the registration of new lines in the log file. The control of this suspension is in the hands of the master module, which decides the stopping instant according to the number of test data already processed and number of slave modules.
5.2 Main Modules Implementation

![Data structure to store the log events details.](image)

5.2.3 Master

The master's implementation is almost completely covered by the inherited structure from "ModuleStruct" class, although its methods are locally implemented.

The logic that is present on the master module is straightforward. As long as there is test data yet to be processed, the master sends it to a slave that is not busy and note it down when the corresponding result arrives. Furthermore, it is responsible to signal the monitor to stop writing in the log file at the right point in time.

```cpp
#include <testa.hpp>

class Master : public ModuleStruct,
    public test_if
{
public:
    sc_in<bool> start_in;

    Master(sc_module_name name_,
        unsigned int mem_size,
        unsigned short int mem_delay,
        unsigned short int myid,
        unsigned short int _modules);

    void init();
    void checkForMsgs(int& results_received);
    void main_action();
    void msgRecy_process();
    void msgSend_process();
    short int next_destiny(int msgid);
    void insert_test(int **v);

private:
    int **data; int **work;
    bool isJobFree(int moduleid);
    void finishJob(int msgid);
};
```

![Master source code structure](image)
5. Framework Implementation

The include statement in the first line of the master's source code (figure 5.10), carries an object of type "sc_event" from SystemC, which is responsible for signaling the master when the simulation starts. As described before in the stimulus implementation section, it is through this event that the stimulus signals the master. The master's main action source code starts precisely with a "wait" statement that is locked on this event, waiting to be signaled.

The main action procedure is constantly checking for events and tasks that need to be performed. In this case, it concerns the arrival of messages, test data that needs to be sent and stopping the monitor from writing log lines. As stated before, "msgRec_Process" and "msgSend_Process" are responsible for receiving and sending messages, respectively. During these operations, an important task is done. Each time a result from test data is received or new test data is sent, one slave is tagged as free or busy, according to the situation. In order to do this, there is a simple table that relates each slave to its current condition. This table is a two dimension integer array which stores the identifier of each slave and its current condition: a minus one integer stands for a free slave and a positive task identifier for a busy one. Figure 5.11 shows an example, where slave number one is free and slaves number two, three and four are busy. This slave management is accomplished through three dedicated methods "next_destiny", "isjobFree" and "finishjob". The first one returns the identifier of a slave that is currently available for performing a task and set it as busy with the provided task identifier parameter. The method "isjobFree" tells whether a specific slave is currently available or busy, and the last one sets the corresponding slave as busy.

![Table of each slave's status.](image)

As it was referred before, before the simulation starts, the stimulus module sends to the master the table with the addresses of every test data that is stored in the shared memory. The table is a two dimension integer array with pairs of test data identifiers and the corresponding address. This table is sent to the master through the "test_if" channel. The stimulus module sends the pointer of the table as a parameter through the test channel interface method as depicted in figure 5.12. Since the simulator is a C++ executable, data is accessible from any class as long as the data is declared as public and the object pointer is present.

```
class test_if
{
    public:
        virtual void insert_test(int **v) = 0;
};
```

![Master interface that is used to receive the table containing the addresses and lengths of the test data.](image)
5.2 Main Modules Implementation

Finally, the writing of the log file is controlled by the master. During the simulation, the master and slaves change their status thousands of times. As a consequence of this phenomenon there are thousands of log lines to be written in the output file. This situation would become a potential problem because the chart builder (ploticus) is not able to build charts with too many records. This way, the writing of the log file needed somehow to be contained. As a consequence, the control of the log file was assigned to the master. The period that the monitor writes to the log file is confined to a certain number of tests that is defined at build time. When the master receives the results of a test, it checks the number of tests already accomplished and if those are equal or greater than the predefined number of monitored tests, the master signals the monitor to stop the writing into the log file.

5.2.4 Slave

The slave's structure is entirely built on "ModuleStruct" class structure. As figure 5.13 shows, the structure is completely similar to its mother class. The only noticeable difference is an integer data type variable, called "work_time", that contains the time that the module should take to execute the data processing algorithm, according to the considered execution model.

```c
class Slave: public ModuleStruct
{
  public:
  sc_in<bool> start_in;
  Slave(sc_module_name name_,
       unsigned int _mem_size,
       unsigned short int _mem_delay,
       unsigned short int _myid,
       unsigned short int _modules,
       unsigned int _work_time);
  SC_HAS_PROCESS(Slave);
  void init();
  void main_action();
  void msgRecv_process();
  void msgSend_process();
  short unsigned int next_destiny();
  int work_time;
};
```

Figure 5.13: Slave source code

As it was described before, the slave modules have a passive attitude during the simulation. The purpose of the slaves is to run the implemented algorithm whenever the test data arrives. This is done via three different methods: "main_action", "msgRecv_process" and "msgSend_process".

The "main_action" process is permanently active and its job is to check for incoming messages. As soon as a message arrives, the method which is responsible for reading and processing the messages is called. This method is called "msgRecv_process". Usually, a command message requires retrieving data from the shared memory, processing it according to the command message...
and sending a result message to the master. Therefore, after retrieving the corresponding data from the shared memory to process the request, it is necessary to call the "msgSend_process". The "msgSend_process" picks the result of the implemented algorithm, writes it into the shared memory and then, a command message is directly sent to the master alerting that the result is stored in the shared memory.

5.2.5 Memory

Memory is a key point of the system, since it is the communication bridge between the master and the slaves.

Figure 5.14 shows the source code structure of a simple memory unit. It is defined by its size, access delay and write and read methods. The memory object is accomplished with a vector of characters. In order to perform read and/or write operations, it is necessary to provide the parameters regarding the address where the data will be written/read from, the length of the corresponding data and the actual pointed to by the supplied address. Hence, the programmer has the hablity to completely specify the address where data is written to or read from. This class is used to implement the shared memory of the system, as well as the internal memories residing within the master and slave modules.

```cpp
class MemoryUnit {
public:
    MemoryUnit(unsigned int size,
        unsigned int access_delay)
        : size(size),
          access_delay(access_delay) {
        mem = new char[size + 1];
    }

    ~MemoryUnit();

    int write(char data[], int address, int data_size);
    bool read(char data[], unsigned int address, unsigned int data_size);

private:
    char *mem;
    unsigned int size;
    unsigned int short access_delay;
};
```

Figure 5.14: Memory unit source code.

Figure 5.15 shows the source code corresponding to the shared memory of the system. It is composed by a memory unit from class "memory_unit", which is the actual memory, and the methods that make part of its interface. The memory interface is the channel used by the bus to access and to perform operations with on the memory like read and write. It is declared on "memory_if" class and its methods are implemented inside the memory.

5.2.6 Bus

In order to make the communication between modules and memory possible and efficient, the bus is the channel that interconnects both the master, the slaves and memory, aided with an arbitrer that manages the requests according to their priorities.
5.2 Main Modules Implementation

The bus module is composed by the methods that are part of its interface, a main action method and built in ports that communicate with all the modules within the simulated device. According to figure [5.16] it also has a clock entry to keep the main action thread running. The “arbitrer_port” port is used to communicate with the arbitrer, in order to sync each request status (this interface will be described in the arbitrer’s section). The “ram_port” port is used to perform write and read operations, according to the requests made by the master and slaves.

As the bus module is aware of the time that the requests are kept waiting for bus permission and for performing memory write and read operations, it is also responsible for notifying the monitor about the time that these operations have took. Hence, it needs an interface to the monitor which is the “monitor_port”.

As command messages travel from one module directly to another through the bus, this operation requires having a specific channel, so there is a port for each slave and master. This port is a specific interface built to insert messages in the mailbox of each processor. Although the interfaces used to write data in the memory and the one used to write in the mailboxes are different (in order to ease the implementation), the programmer’s interface remains untouched. The simulation modelling is not affected in any way and another important fact, is that these type of requests are treated the same way as the requests to access the shared memory.

As stated before, when the modules make requests to access the bus and are not immediately granted to access it, these are first checked about its sender and operation and then inserted into the corresponding priority queue. The implementation of the three request queues is achieved through a SystemC object data type, which is denoted as ‘sc_fifo’. This data type provides the programmer a queue with the corresponding methods that read and write data into. These meth-

---

**Figure 5.15: Shared memory source code.**

```c++
1 class Memory : public sc module
2 {
3     public:
4         Memory(sc_module name name,
5                  unsigned int size,
6                  unsigned short int mem delay
7             )
8             : sc module(name_)
9                  , size(size_)
10                  , mem delay(mem delay)
11     {
12         mem = new MemoryUnit(size, mem delay);
13     }
14
15     void read(REQUEST *req);
16     void write(REQUEST *req);
17
18     private:
19         MemoryUnit *mem;
20
21     unsigned int size;
22     unsigned int short mem delay;
23
24 }
```

---
5. Framework Implementation

ods (together with this data type) perform the needed operations to implement a first-in first-out queue.

Back to the bus structure, the main action is responsible for checking if there is any request ready to be dispatched. A ready request is a request that is currently on the top of its queue and there are no requests with a higher priority. In case there is a ready request, this one is dispatched and the arbiter is signaled about the corresponding module which becomes the owner of the bus.

The "new_request" method is the interface used by the modules to place a new request in the bus. Within this method, the arbiter is questioned about the owner of the bus and if it matches with the current request sender, this one is immediately dispatched, skipping the queue. In case, it corresponds to a module that is not the current owner, the request is inserted in queue according to its priority. If the request is immediately dispatched, the bus signals the monitor that the corresponding module is currently writing or reading (according to the operation) at that moment. If

```cpp
1 class SimpleBus
2 : public sc_module
3  , public simple_bus_if
4 {
5  public:
6  
7  sc_in clk clock;
8  sc_port<simple_bus_arbiter_if> arbiter_port;
9  sc_port<memory_if> ram_port;
10  sc_port<simple_bus_monitor_if> monitor_port;
11  sc_vector<sc_port<mbox_if> > alert_modules;
12
13  SimpleBus(sc_module_name name_,
14  , unsigned int mem_delay_
15  , int modules )
16  : sc_module(name_)
17  , mem_delay(mem_delay_)
18  , modules(modules_)
19  {
20   SC_THREAD(main_action);
21   sensitive <= clock.neg();
22
23   requests = new BufferQueue(BUS BUFFER SIZE, BUS BOTTOM_PERCENTAGE);
24
25   alert_modules.init(modules);
26 }
27
28 void main action();
29 void new request(REQUEST *req);
30 void end_transmission(int id);
31
32 private:
33  
34  unsigned short int modules;
35  unsigned int mem_delay;
36  unsigned short int buffer_size;
37  BufferQueue *requests;
38
39  void handle request(REQUEST *req);
40 
```

Figure 5.16: Bus source code structure
5.2 Main Modules Implementation

not, the monitor is signaled that the corresponding module is waiting to access the bus.

During the period when a processor has the bus access granted, no other requests from other processors will be attended. This period ends only after the maximum number of continuous accesses has been reached or the requestor processor ends the transmission. After this, this same processor will not have permission to access the bus until every current pending requests have been dispatched. This mechanism guarantees that the instruction messages are sent continuously with no interruptions, since these represent sensible information.

The last method that makes part of the bus interface is the "end transmission". It is used by the master and slaves whenever their transmission is over or the maximum number of messages after gaining bus access is reached.

The "handle_request" method is called whenever a request is ready. This method performs the requested operation and signals the monitor about it.

Finally, the default bus width is thirty two bits wide. Such decision was made accordingly to the DNA alignment software running on the slave modules. As described on the first chapter of the thesis, the software running on the simulated device is the Smith-Waterman algorithm, hence, the type of data flowing through the simulated device are DNA sequences, represented as integer numbers and one data separator:

1. The DNA sequences are merely composed of four different characters (A, C, G and T), which means that only two bits are needed to represent them.

2. The integers are used to represent sizes, addresses and other parameters that will be described in the next chapter. So, four bits are needed in order to represent all the ten different digits (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9).

3. One more character is needed to separate data inside the messages and the chosen one is forward slash which needs one bit only.

So, the total number of different used characters are fifteen. In order to represent such amount of values in binary, only four bits are needed. Hence, the bus width is wide enough to transport eight characters of the alphabet at once. Table 5.1 shows all the used characters.

| Symbol | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | A | C | G | T | / |
|--------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

5.2.7 Arbitrer

The arbitrer module is responsible for managing the access to the bus. Every module that performs a request to access the bus needs to be previously authorized by the arbitrer.
To establish a perfect understanding between bus and arbitrer, the arbitrer has a built-in interface that allows the bus to query it about the situation of every module that inserts a request on the bus. It queries the arbitrer about the modules and takes actions according to it. Figure 5.17 presents the arbitrer's source code structure, which is completely based on its interface. Figure 5.18 shows the structure implemented in the arbitrer interface that supports the communication between the bus and the arbitrer.

The first method is the "is_owner", which the bus uses to ask the arbitrer if a specific module is owning the bus at the current time. Every time a new request arrives at the bus, the arbitrer is questioned about the status of the requester module, so that the request is immediately attended or inserted into the queue.

Every time a different module gains access to the bus, this one becomes the owner of the bus. This operation is executed in two steps: firstly, the bus queries ("is there owner") the arbitrer about the current owner and if the answer is that there is not a current owner, then, the requestor module becomes the owner of the bus through the "set_owner" method.

When a module signals the bus that the transmission is ended, the bus signals the arbitrer that the current owner of the bus has finished its transmission through "end transmission" method.

When there is no module owning the bus access, the main action of the bus is performed. This action executes a routine check in the request queues. This routine check is executed by the "arbitrate" method provided by the arbitrer. This routine checks for pending requests firstly on the higher priority queue. In case there are no requests in this one, it tries on the medium priority queue and lastly on the low priority queue in case there are no requests in the medium priority queue.
5.3 Communication protocol

In the previous section it was described the implementation of the system. This is the architecture of the system, the purposes and functions that each piece within the system does and its workflow. The description of the system included the way how the components communicate with each other and what they communicate.

In this last section of the chapter, it will be described in detail the syntax of the messages that were used to exchange data between the modules inside the system in order to run the Smith-Waterman algorithm. The used syntax is completely changeable and flexible, in order to meet the needs of the programmer and of the source code that may be implemented.

Let’s say that the master sends a message to some slave, ordering him to pick some DNA test data from the shared memory. This message has some mandatory parameters, such as an address, size and command. The adopted syntax is shown in figure 5.19. The first parameter is the command, so that the slave knows what to do with the data. The second parameter is the address where the DNA data is stored in the shared memory, the third one is the length of the data that must be read from the memory, starting at the specified address. The last parameter is the address where the slave will write the result from the execution of the algorithm with the test data.

<table>
<thead>
<tr>
<th>Command</th>
<th>Address</th>
<th>Size</th>
<th>Address result</th>
</tr>
</thead>
</table>

Figure 5.19: Syntax of a command message.
Figure 5.20 illustrates an example of a test data message. It contains a DNA reference sequence, the reference sequence length, one DNA query, the DNA query length and, in the last position, one unique message identifier which is used by the master module to control the task assignment of each slave.

![Example of a test data message](image)

Before finishing the communication protocol section, a last example is described. Figure 5.21 shows a command message, sent by a slave to the master, saying that the result is available at the shared memory and it can be picked up. The containing parameters are a command (result from alignment procedure is available at the memory for pick up) in first place, the address in second and the length of the message at the third position.

![Example of a command message](image)

### 5.3.1 Message structure

This last section describes the structure of the messages that travel through the system. The adopted structure can be seen as a datagram and is splitted in two parts. The first part is the user part, where the programmer writes the data or the body of the message itself and the address or module id that must receive the message. The second part contains several attributes that are assigned and used to control and report the status of the message during its trip from one module to another.

The first part of the datagram is composed by two attributes:

- "data" is the first attribute of the datagram. It contains the actual message or data that is being sent from one point to another.
- The second attribute is the address where the message is being sent. The value of the address specifies two different things. First, the target memory: this can be the shared memory, internal local memory of the module or a mailbox of a module. Secondly, it also specifies the memory address where data will be saved. In case the target memory is a mailbox of a module, the address is used like a port to access the mailbox.

The second part of the datagram is composed by seven attributes:
5.3 Communication protocol

![Datagram structure](image)

The first attribute contains the priority of the request that is assigned when the request is sent to the bus. This attribute is crucial, because it influences the time that the request may have to wait until being dispatched.

The attribute "write" specifies the type of operation that the request contains. This can be either a write, read or command operation. This is important so that the bus module is able to rapidly know if the request is performed on the shared memory or on a mailbox. Without it, another translation address operation would have to be made.

The length or size of the message is a crucial attribute for any write or read operation, independently of the target memory.

Keeping track of the status of the message is useful for programming. Every message contains an attribute with the current status of the message. This attribute is used to inquire about the successfulness of the requested operation. It can be used in the future in case the programmer needs to implement a system where the requests are wanted not to be blocking and during the operation it should be possible to know the status of the request. The status of a request can be one of eight different values, as described in table 5.2.

The "sender" attribute carries the identifier from the module that created the message. It is used to calculate the level of priority assigned to the message as described in the previous chapter with the table 4.1.

The "recipient" attribute carries the identifier from the target module that must receive the message. This attribute is calculated during the address translation, so that finding the target module is easier.

The last attribute of the datagram is the event data type, which is used by the bus module to signal the requestor module that the request has been dispatched independently from its success or failure.

Figure 5.23 shows the state diagram of the requests.
5. Framework Implementation

Table 5.2: Request status table

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILD</td>
<td>currently being built</td>
</tr>
<tr>
<td>WAIT</td>
<td>waiting for bus access</td>
</tr>
<tr>
<td>ERROR</td>
<td>the message could not reach the target</td>
</tr>
<tr>
<td>ATTENDING</td>
<td>request being dispatched</td>
</tr>
<tr>
<td>PROCESSING</td>
<td>request is being processed by the target module</td>
</tr>
<tr>
<td>DONE</td>
<td>the message reached the target</td>
</tr>
<tr>
<td>EMPTY</td>
<td>the current request is empty</td>
</tr>
</tbody>
</table>

Figure 5.23: Request state diagram.
6

Programmer Interface API and Setup

Contents

6.1 Programmer Interface and API ........................................ 62
6.2 Simulator Setup ...................................................... 70
During the development of any simulator construction, an important goal should be constantly present. A simulator is built to simulate a specific type of system and should also allow future modifications and adjustments, so that its use does not end up on the first target simulation. Therefore, documentation about the system and how to use it is undoubtedly required.

The chapter is divided into two sections: the first one describes how to use the simulator's tools and presents a step by step guide explaining how to implement algorithms in the simulator. The second and last section describes how to configure the simulator, according to the programmer needs.

6.1 Programmer Interface and API

This section describes how to use the tools provided by the simulator. Firstly, it is described how to make the modules to communicate with each other and how the master assigns tasks to each slave. The second part of this section presents a walkthrough tutorial on how to implement an algorithm in the simulator.

6.1.1 Message building

Message building is one of the most critical points where a programmer must devote some attention in order to implement its algorithms. Therefore, this subsection will describe how to exchange data between modules. The first paragraph explains how to create a message and send it through the bus whether its target is a memory or a module. The second paragraph explains how to read a message.

Building and sending a message

The process of sending data to the shared memory or a module is depicted in figure 6.1 and is done via four steps:

1. The first step is to create or select an object of type “REQUEST” and initialize it if needed.

2. The second step starts the construction of the message. Firstly, the address is set, which points to a specific address of the shared memory, or to a mailbox of a slave or master module. Selecting the address from the shared or internal memory is as simple as assigning the integer address to the "address" attribute from the "REQUEST" structure, as long as the address is within the bounds of the shared/internal memory. In order to assign the address of another module's mailbox, the programmer may use the provided method called "getModuleMboxAddress" and provide the identifier of the target module. This method will return the actual address of that module's mailbox.

3. The third step of the process is the actual construction of the data that will be sent to another point in the system. It can be assisted using a specific tool, which is provided by a method called "new_message", and is implemented in the "SimpleBusTools.hpp" file. As figure 6.2 shows, the method requires two mandatory parameters, namely, the pointer of the structure where the data will be written and the number of attributes that the message incorporates.

Let's consider an example where the message contains a command (string format) and two integer values. According to figure 6.2, the "new_message" method only accepts char* and integer values as arguments. In order to provide the attributes to the method, it is necessary
to specify, for each one, whether it is a char or integer value with one standing for integer and zero for char*. In the case of an integer value, the parameter that follows is the integer value itself. If the case of a char*, the following parameter is the length of the string and then it is provided the pointer to the string. Figure 6.3 shows the result of the message construction. The provided attributes are written in a first come, first served basis and separated by “/” character. An important aspect, is that the messages structure is not limited or defined in any way. Therefore, the method “new_message” is only a provided tool that the programmer may or may not use it to fill the attribute denoted as “data” in the “REQUEST” structure. So, the programmer is completely free to choose the structure of the message.

4. Finally, the “write_op” method carries on with the message send operation.

**Reading a message**

The process of reading data from the memory or its own mailbox is made out of four steps that follow according to figure 6.4.

1. Creating or selecting a data object of type “REQUEST” is the first step.

2. In the same way as sending a message, the second step is to provide the memory address where the message is stored and must be read from. If the message is stored in its own mailbox, it is advisable to use the “getModuleMboxAddress” and provide the id of the module (which is constantly stored on the global variable “myid”, for each module).

3. The next step is to set the size of the message that will be read. In case the read operation is pointed to its own mailbox, this step is skipped.
Once all the parameters are set, the "read_op" method carries on with the reading procedure. Once the message is completely obtained, the next step is to read the content of the message. The "getarg" method is implemented in "SimpleBusTools.hpp" and is used to read the attributes of a message. Figure 6.5 shows the syntax and the three provided examples illustrate how it is invoked. The first parameter is the object which contains the message. Depending on the type (char* or integer) of the attribute that will be read from the message, the value of the read attribute will be assigned to the second (char*) or third (integer) parameter, though both parameters are mandatory. The fourth parameter ("arg_index") is the position of the attribute within the message that is meant to be obtained. Therefore, zero stands for the first parameter (which in the considered example is "SW_ALGO"), one for the second attribute (which is "12345") and two for the third (which is "8935"). The "msg_size" parameter carries the length of the message and finally the last one specifies whether the attribute is of type integer or char*. The length of the message has to be provided by the instruction message that has been received before reading the data from the shared memory.

### 6.1.2 Slave Management

The master module has a built-in tool used for slave management. Since the master is responsible for assigning tasks to each slave, it is also responsible for scheduling those same tasks in an efficient way. This means that it should assign tasks to a slave only when it is free, and do it equally by all. This management is guaranteed through three simple methods (depicted in figure 6.7) and must be correctly used by the programmer.

As described in the implementation chapter, the master manipulates a table used for slave
management. This table maps the slaves to tasks by using slaves and task identifiers. This way, the programmer must ensure that each task that is assigned to a slave must have an unique integer identification. This can be done by sending an identifier in each task that is sent to a slave. Conversely, when the slave answers to the master, the message must contain the same identifier that was previously received.

Once the task identification is guaranteed, the slave management is done in two spots. Figure 6.6 shows the point in the source code where a task issuing message is being built before sending to another module. After generating an identification to the task, choosing the target module is accomplished through the method "next_destiny", which provides the task identificator. The "next_destiny" method returns the identification of a free slave module, which will be later used to obtain the address of that same module's mailbox.

A slave's task is finished when it sends the result or a message telling the master module. This message must contains the identification of that task, which will be subsequently used by the master to know which slave it is. At this point in time, the master module runs the "finishJob" method with the received task identification. The method searches in the table which slave is mapped with that task identification and sets it as free. An example is shown in figure 6.6 (lower part).
6.1.3 Algorithm implementation: step by step guide

This subsection presents a step by step guide on how to implement an algorithm in the developed simulator. It is divided in three major parts. The first part describes how to load and provide data to the system. The next part explains how to manage the loaded test data and spread it through the slave modules. The final part describes how to implement the code in the slaves side and process the received test data. The presented tutorial uses the Smith-Waterman algorithm as an example for implementation.

6.1.3.A Loading data to the simulator

Loading data to the simulator is the first process that the programmer needs to pay attention. Without data to be processed, the simulator becomes useless because the master does not have tasks to assign at each slave. The data loading into the system is done within the "init" function, from the stimulus module. The final goal of this method is achieved through two procedures:

1. Create a two dimensional table, to be managed by the master module where each line contains a pointer (address) corresponding to each test data instance, stored in the shared memory, as well as the corresponding length.

2. Wake up the master and slave modules.

Figure 6.8 shows an example procedure to load data into the simulator and wake it up. This figure is divided into four sections:

1. Create and initialize the necessary data structures. "msg_id" is used for identifying the loaded test data. In order to sequentially write all the test data in the shared memory, the
6.1 Programmer Interface and API

Figure 6.6: Generating a new task identification and assigning it to a slave.

```c
void send_task(){
    ....
    REQUEST *req = new REQUEST;
    message_id++;
    //generate new identification
    // get module mailbox address from free slave
    req->address = getModuleMboxAddress(next_destiny(message_id));
    ....
} 

void receive_task(){
    ....
    REQUEST *req = new REQUEST;
    read_op(req);
    ....
    getarg(req->data, 2, dummy_char, message_id, //get message id
           1, 10, true);
    finishJob(message_id); //set the corresponding slave as free
    ....
}
```

Figure 6.7: Tools used for slave management.

short int next_destiny(int msgid)
bool isjobFree(int moduleid)
void finishJob(int msgid)

memory pointer “mem_ptr” is incremented at the end of each write. Table “data” is initialized according to the quantity of test data that will be provided to the system. The character sequences “reference” and “query” are the variables that contain the actual test data. Finally, the declaration of a “REQUEST” object type is used to send the messages through to the bus.

2. Section two starts the process of sending the test data to the shared memory. Each data test is written into the “REQUEST” object along with the corresponding identifier, and sent to the shared memory.

3. The memory address where the data was stored is saved on the “data” table, along with its
6. Programmer Interface API and Setup

length. The memory pointer is updated, so that the next test data is stored right next to the previous one. Sequences “reference” and “query” are changed, so that the next test data instance is different and consequently the result is different too. Finally, the “REQUEST” object is cleared in order to use to send the following test data instance.

4. The last step is responsible for providing the master with the table containing the references of the stored data. Finally, the master and slave modules are awaken up so that the simulation can be started.

6.1.3.B Master’s algorithm

The implemented algorithm only requires that the master module keeps issuing tasks sending to the slaves until no more data is available. Therefore, inside the master’s main action procedure, the “msgSend_Process” is called as long as there is still data to be processed. Figure 6.9 shows the source code of the master’s send procedure which is divided into three sections:

1. On the first section, the master module searches for a free slave. If none is available, the procedure ends and another attempt to send the message will be run later. The next instruction updates the monitor about the status of the master module.

2. Section two contains the source code that assigns a task to a slave. Since the test data to be processed is already stored on the shared memory, the master module only needs to send a command to the slave, telling where the data is stored. This way, the address of the message is the slave’s mailbox (line 16, in figure 6.9). The command message contains several attributes, such as operation, test data address, test data length and another address, which points to a specific cell within the shared memory, where the slave will write the result of the operation.

3. The last section of this procedure updates the amount of test data that has not yet been processed. It also updates the iterator of the test data table, clears the “temp_req” object and updates the monitor, telling that the master has been doing some work.

After assigning a task to a slave, all that is left besides assigning tasks to other slaves, is to wait till a slave responds with the result from the operation. Therefore, the master has to ensure that it receives the results and runs the corresponding data processing, in case it is needed. This means that the programmer needs to focus on the data arrival, which is the “msgRecv_Process”. This process is automatically triggered as soon as a message arrives to the mailbox. Figure 6.10 shows the source code of this procedure.

This procedure is divided into two sections. In the first section, the master reads the content of the obtained message from the mailbox and reads the address and size from the data stored in the shared memory. In the second section, the master module reads the contents of the message, which has the result from the operation and corresponding task identifier. Depending on the implemented algorithm, the result of the operation can be or not processed in some way. At the end of that procedure, the master updates the status of the slave as free.

Implementing the slave’s algorithm

Once the source code in master module side is done, the remaining implementation lies in the slave side. The slaves are constantly awaiting for tasks (message commands) to arrive. Therefore, the source code is basically inside its “msgRecv_Process” method. Each time a command
message arrives, the "msgRecv_Process" method is triggered the same way as it happens in master side. The source code to process the received command message is divided into three sections, as figures [6.11] and [6.12] show.

Section one reads the contents of the command message which arrived at the mailbox. The contents of that message are the operation that will be performed by the slave, the address where the test data is stored, its length and the address where the slave will store the results of the operation. As soon as this message is processed, the slave reads the test data from the shared

Figure 6.8: Loading data into the simulator.

```c
1  const int SMITH_WATER = 0; //command to run smith waterman algorithm
2  char* generateSequence(); //generate a simple ADN sequence
3  int length(char* str); //get length from provided parameter
4  void Stimulus::init(){
5     //SECTION 1
6     int msg_id = 0;
7     int **data = new int *[TEST_DATA];
8     int mem_ptr = 0;
9
10    char* reference = generateSequence();
11    char* query = generateSequence();
12
13    for(int i = 0; i < TEST_DATA ; i++)
14       data[i] = new int[2];
15
16    REQUEST *temp_req;
17    temp_req = new REQUEST;
18
19   //SECTION 2
20    for(int i = 0; i < TEST_DATA ; i++){
21
22       temp_req->address = mem_ptr; //address where data will be stored (shared memory)
23       temp_req->size = new message(temp_req->data, 4,
24          1, SMITH_WATER,
25          1, length(reference), /*sequence #1 length*/
26          0, length(reference), reference, /*sequence #1*/
27          1, length(query), /*sequence #2 length*/
28          0, length(query), query, /*sequence #2*/
29          1, msg_id++); /*test data ID*/
30
31    send_message(temp_req);
32
33   //SECTION 3
34    data[i][0] = mem_ptr; //shared memory address where the data has been saved
35    data[i][1] = temp_req->size; //message size
36
37    reference = generateSequence(); //change test data values
38    query = generateSequence(); //change test data values
39
40    mem_ptr += temp_req->size; //update shared memory pointer
41    clearREQ(temp_req); //clear REQUEST object
42
43   //SECTION 4
44    test_port->insert_test(data); //pass test data pointers to the master
45    simulation_start(); //wake up the master and slaves
46 }
```
memory according to the address and length parameters received from the command message.

In section two, the test data is read into the slave’s local memory. After that step, the specified algorithm is run, according to the parameters received from the master. As soon as the algorithm is finished, section three carries on with the operation of sending the results to the master. Firstly, the results are stored in the specified result address and then a command message is sent to the master, confirming that the result has already been written in the shared memory and it is available for pick up.

### 6.2 Simulator Setup

Figures [6.13](#) and [6.14](#) show all the parameter values that are present within the simulator’s engine. The first one corresponds to the main parameters file, because it has the parameters that have the greater interest and impact in the simulation results. The second one incorporates in its majority the most basic configuration settings.

The content from each file are not completely presented in these figures, since some of the parameters are not relevant. The following subsections describe the purpose of each group of parameters, showed in these figures.

```cpp
1 void Master::msgSend_process()
2 {
3     //SECTION 1
4     int next_recipient = next_destination(msg_id);
5     if(next_recipient == -1)
6         return;
7     log_status(IDLE); /* until this point, the master module has not made anything useful for assigning tasks, so it is considered as IDLE*/
8     //SECTION 2
9     REQUEST *temp_req = new REQUEST;
10    temp_req->address = getModuleMboxAddress(next_recipient);
11    temp_req->size = new_message(temp_req->data, 4,
12                                   SMITH WATER, 1, data[0], // data address
13                                   data[1], // data size
14                                   780000 + 10*msg_id); // address to write result
15    msg_id++;
16    write_op(temp_req);
17    //SECTION 3
18    queries--;  // decrement number of test data
19    data_iterator++;  // data test table iterator increment
20    clearREQ(temp_req);
21    log_status(EXECUTING);
22 }
```

Figure 6.9: Master issuing procedure.
6.2 Simulator Setup

6.2.1 Memories

There are several memory units in the simulator, such as the shared memory, the internal memory of each module (slaves and master) and their corresponding mailboxes. These memories may vary on size and access delay.

The first portion of the parameters file illustrated in figure 6.13 corresponds to the setup of the access delays for each memory. The actual amount of time that a read/write operation takes can be edited in the "MemoryUnit.hpp" file, in order to compute the actual access time based on the amount of data that is written/read. Figure 6.15 shows an example of a memory device that takes ten times more time to write the same amount of data when compared to reading.

The "MEMORY VALUES" section sets the capacity (bytes) of the internal memories from the master and slave modules, as well as the shared memory. The capacity of the mailboxes is defined through a set of constants that are present in the parameters file illustrated in figure 6.14 of the parameters file, on "MAILBOX SIZES" section. In this case, the capacity of the mailboxes is set in number of command messages that can be received and not in bytes.

The address space of the system is automatically calculated based on the memory sizes.
This process is shown in the "MEMORY ADDRESS RANGE" section from both parameter files. Parameter file 1 is responsible for calculating the mailboxes address space and parameter file 2 for all the other memories.

### 6.2.2 Bus

The bus module has some important parameters, such as width and queue sizes. Parameter file 2 configures the queue sizes and bus width. Section "BUS BUFFER SIZE"
defines the total amount of slots available with both queues together (see section 4.2.1.E) and its division. Both sorted and unsorted queues have “BUS_BUFFER_SIZE” slots together and the unsorted queue has “BUS_BOTTOM_PERCENTAGE” percentage from “BUS_BUFFER_SIZE”.

The bus width is an important parameter that must be defined. In practice, the simulator is able to send one or a million bytes through the bus. This way, the “WORD_SIZE” parameter at the end of the file must have the intended number of bytes that are allowed to pass through the bus. In order to set this parameter correctly, a simple calculation is performed:

1. Let’s say that the algorithm implemented in the system uses sixteen different characters and no more will pass through the bus.

2. The second step is to calculate the number of bits needed to specify all the needed characters (four, in this case).

3. Finally, let’s say that the programmer needs a 32 bit bus width, so the “WORD_SIZE” is equal to 32 divide by four, which is eight. So, each request that travels through the bus may contain eight characters.

The next parameter is the burst block size. Once a module gains permission to access the bus, that permission is granted for a limited number of times. This parameter is called “BUS_TURNS” which is set in parameters file 1 in the “MODULES” section.

If a certain module tries to send data to the memory or to another module and it fails, it is able to try again for a limited number of times, which is defined by “SEND_ATTEMPT” in parameters file 1.
6.2.3 Monitor

To perform the monitor invocator in order to log the status of the system, the programmer only needs to call the "log" method from the monitor along with the identifier and status of the corresponding module. In the end of the simulation, the status of each module is presented in a chart through a defined color. Parameter file 2 maps all the possible status of a module, together with the colors that are presented in the chart. Section "MODULE STATUS VALUES" incorporates the status through an integer constant, which is used by the monitor to select the corresponding color before writing to the log file.

As described in the monitor sections of the previous chapters, the monitor should limit the number of log lines. In order to make this an easy task for the programmer, this number is set through the "LOG_MSGS" constant, at the bottom of parameter file 1. This value defines the number of test data that needs to be processed and communicated to the master module until it finishes the log write, independently of the total amount of test data that will be processed by the system.

6.2.4 Slave Modules

One of the most important parameters that needs to be set before the simulation starts is the number of slaves that will be present. This is set through a constant integer called "SLAVES".
6.2 Simulator Setup

Figure 6.14: Parameters file 2.

which is defined in parameter file 1.

Another important factor in the simulation is the amount of time that each slave will take to process its algorithm. Since the developed simulator does not assume any particular internal architecture of the slaves processors, this value is defined by the user according to the considered processing model of the target algorithm. A base time is set through the "SLAVE_WORK_TIME" parameter which will affect the wait statements present in the slave modules. In order to provide a reliable simulation, the programmer must set an amount of time proportional to the expected processing time of the algorithm in the target processing platform. This way, the value from "SLAVE_WORK_TIME" will influence the processing time of each slave module in a proportional...
6. Programmer Interface API and Setup

```cpp
1 int MemoryUnit::write(char data[], int address, int data_size) {
2     wait(acess_delay + data_size * 10, SC_NS);
3     //write data
4     //...
5 }
6
7 bool MemoryUnit::read(char data[], unsigned int address, unsigned int data_size) {
8     wait(acess_delay + data_size, SC_NS);
9     //read data
10    //...
11 }
```

Figure 6.15: Memory Unit wait statements example.

```cpp
1 void Slave::msgRecv_process() {
2     result = //run implemented algorithm
3     wait(SLAVE_WORK_TIME * data_length, SC_NS);
4     //...
5 }
```

Figure 6.16: Slave wait statement example.

way. Figure 6.16 shows an example of how to set the time spent by each slave module (influenced by "SLAVE_WORK_TIME").
7

Experimental Results

Contents

7.1 Benchmarking algorithms ................................................. 78
7.2 Implementation of the Smith-Waterman algorithm in a heterogeneous multi-core platform ............................................. 78
7.3 Discussion ........................................................................ 83
7. Experimental Results

7.1 Benchmarking algorithms

This chapter presents the application of the developed simulation framework to simulate the execution of two parallel implementations of the alignment procedure. It is divided in two parts: firstly, the simulator picks a large set of DNA query sequences which are independently aligned to a single (and larger) reference sequence. In the second implementation, the simulator executes the complete algorithm of two long DNA sequences (e.g: a human chromosome) by adopting data-level partitioning techniques, i.e., the sequences are split into small blocks and each one is aligned in one of the slave modules by respecting the data dependencies composed by the algorithm. Each part of this chapter presents the simulated performance of the system, using either sequential and parallel implementations with multiple slave processors. The simulated performance allowed to infer the attained total execution time and speedup. Within this application context, the main aim of the presented experiment was to evaluate the ability of the developed framework to assess the parallelization scalability that is offered by the prototyped multi-core architecture, by evaluating not only the speedup values that may be achieved, but also the possible degradation losses that arise due to the inherent contention in the bus when the several nodes concurrently try to access the shared memory. Such degradation will depend not only on the number of slave nodes that will be incorporated in the multi-core processor, but also on their specific performance (achieved through software optimizations or dedicated architectures) to implement the alignment procedure. For such purpose, (in the first implementation only) it was defined a processing-balance parameter \( K \) in the considered model of the multi-core architecture, in order to model the alignment performance of the slave modules. This parameter represents the relation between the amount of time (clock cycles) that each slave node needs to compute one single cell-update in the scoring matrix and the time that it needs for a single local memory access (one read or write operation).

7.2 Implementation of the Smith-Waterman algorithm in a heterogeneous multi-core platform

7.2.1 Alignment of multiple queries against one reference sequence

The following three subsections regard to the experimental results of the execution of the alignment algorithm that has been considered along this document. In this particular case, the Smith-Waterman instances in each slave module are completely independent from each other and there is no data dependency. The master module only needs to tell each slave which query it should process and collect the corresponding results. The considered data-set comprises 1000 nucleotides long reference sequence extracted from the Homo Sapiens chromosome 1 GRCh37 primary reference assembly, and 500 query sequences, each one with 35 nucleotides, extracted from the Homo Sapiens genome (Run ERR4756 of study ERP000053).

Each read operation in the shared memory takes the length of the data converted to clock cycles as well as each write operation. The spent time by the slave modules executing the alignment algorithm is proportional to the length of the query and reference sequences, where each cell-update in the score matrix is computed under a twentieth of a clock cycle. Therefore, this simulator setup represents a highly optimized implementation which is capable to compute more than one cell-update per clock cycle, achieved either by using SIMD programming models \[40\] or
7.2 Implementation of the Smith-Waterman algorithm in a heterogeneous multi-core platform

dedicated VLSI architectures [7]. The system snapshot

7.2.1.A Sequential execution (one slave)

The first graph presented in figure 7.1 illustrates the operation of the architecture considering only one single slave processor. It shows the slave module accessing the bus and processing the DNA data. In this situation, the bus wait time is very short because it is the only module accessing it. The master processor simply sends commands to the slave module as soon as the previous task is finished. In the beginning, the slave module takes some significant time reading the wider sequence. As this operation is finished, the slave module needs only to read the queries before executing the alignment procedure.

7.2.1.B Concurrent execution (multiple slaves)

The following paragraphs regard figures 7.2 and 7.3. As the number of slave modules increases, the number of tasks that are simultaneously executed increases too. As a result, the bus contention increases and the time that the several modules have to wait to access the bus also increases. Therefore, the amount of time between each read operation to the shared memory becomes larger. Consequently, in order to read a complete set of data from the shared memory, the total spent time is larger too.

Figures 7.2 and 7.3 illustrate the operation of the architecture by considering two and four slave processors, respectively. It is observed that the system performance increases with the addition of more slave modules.

Figure 7.1: System snapshot using one slave module for K=0.2.

Figure 7.2: System snapshot using two slave modules for K=0.2.
7. Experimental Results

Figure 7.3: System snapshot using four slave modules for $K=0.2$.

7.2.1.C Discussion

Figure 7.4 illustrates the alignment time and speedup offered by the prototyped multi-core architecture to process the benchmarked DNA data-set by considering a variable number of slave modules. The alignment performance (speedup) increases linearly with the number of slaves. To illustrate the relation between the processing balance parameter ($K$) and the resulting contention in the bus, this chart represents two distinct scenarios, for $K=1$ and $K=0.2$. The relation between the total execution time between one and two slave modules represents a speedup that is equal to 2,00. The speedup is 3,95 when comparing the simulations that use one and four slaves. The speedup reaches 7,54 with 8 slave modules. Since then, the speedup starts decreasing its growth ratio.

Figure 7.4: Total execution time and speedup as functions of the number of slave processors.
7.2 Implementation of the Smith-Waterman algorithm in a heterogeneous multi-core platform

7.2.2 Alignment of two DNA sequences

This section corresponds to the execution of the second algorithm referred in section 7.1. The Smith-Waterman instances in each slave module execute the alignment procedure between two small parts of the two considered DNA sequences. Hence, the master module responsibility is to split the two DNA sequences in small substrings (each one has the same length) and send each pair of substrings to the slave modules. In this case, the Smith-Waterman algorithm receives two sequences with 1024 characters each and processes the alignment blocks measuring 1024 by 1024 characters. The obtained results are presented in figures 7.5 and 7.6. In this case, it is not possible to present the charts that demonstrate the system performance for each module due to the high level of recorded operations in such short period of time. This relates to the described limitation on the number of log records in the monitor module of the system.

Each read operation in the shared memory takes a tenth of the length of data read in clock cycles as each write operation half of the length of the data length in clock cycles. The slave modules spent half of 120 times 1024 (block size considered in the alignment procedure) clock cycles to execute the Smith-Waterman algorithm. The bus word size is also eight as in the previous situation. Finally, the bus burst size is 20.

7.2.2.A Discussion

The performance gains in this implementation are not as high as the first implementation. In this one, the amount of data that is sent between the modules is significantly higher as a consequence of the size of the processed blocks by the alignment procedure. As referred in the
previous section, the architecture of the modelled system must be adapted to the implemented algorithm. Therefore, according to the amount of data that travels through the system, the time that read and write operations spend must be significantly decreased.

Regarding to the execution time spent by the slave modules, it is reasonably well defined. This simulated time is proportional to the size of the processed blocks that are 1024 by 1024 long. This way, the proportion between read/write and execution operations is once again reasonably set.

Figure 7.5 illustrates the total execution time of the system as a function of the number of slave modules. The figure shows that the performance gain is somehow linear as the number of slave modules is doubled. In this case, it is important to note that the amount of bus accesses is extremely high as a consequence of the amount of transmitted data. Therefore, the bus contention situation is much heavier and propitious to happen. Though, it is expected to obtain a speedup of 4 before using 16 slave modules which is not true (illustrated in figure 7.6) with this modelled architecture.

In order to achieve a higher level of speedup growth, it is clearly important to adapt and review the modelled architecture in this case.
7.3 Discussion

The presented results of the first implemented algorithm show that the simulator is working well and reacts properly according to the configuration set.

The first algorithm implementation presented satisfactory results. From the obtained results, it can be observed that not only is the conceived framework able to accurately model the parallel scalability offered by the prototyped multi-core architecture, but it can also represent the bus contention effect when the access rate to the shared bus increases for lower values of $K$. From the time diagrams that represent each processing module state it clearly illustrates the bus contention effect. Whenever more than one processing module requests access to the bus, the arbiter only grants access to one single module, thus making all the others remain in the bus waiting state.

In the second algorithm implementation, the bus access issue is present too. This situation is clearly noticeable when the simulator has more than four slave modules which is illustrated in the respective speedup and total execution time charts.
Conclusions and Future Works
8.1 Conclusions

This document started with the presentation of a common and actual paradigm, when designing hardware components to be run in parallel. There are lots of restrictions and limitations when designing hardware without simulating it in first place. Some of them are concerned with early decisions about the hardware, the probability of a new protocol that is designed and does not work within the hardware, and finally and very important, the designer can only test the system after the hardware has been manufactured. Hence, some hardware description languages that are used nowadays overcome some of the problems mentioned before. By using such description languages, the designers are able to simulate the system and to run several types of tests, like performance, correctness, behavior, etc. They are also able to simulate and count the runtime periods in the system the way they need to continue their development test. The challenges that are often posed when designing different types of systems (heterogeneous vs homogeneous) and concurrent processes were also briefly described.

Since biologists were able to determine the DNA sequence of a human cell, lots of testing and research has been going through. A DNA sequence is known to be very long and this is why biologists need huge databases to save all the data. As a consequence, the DNA alignment procedures represent an enormous demanding task, because of the many comparisons that are necessary. This is the main reason of this thesis. This thesis presented a simulation framework which is capable of simulating a multi-core processor architecture that is able to run parallel algorithms that use one or more processing cores. In this case, the algorithm is the Smith-Waterman alignment algorithm, which performs the local alignment procedure of DNA sequence.

The accomplished simulation framework was built with SystemC. It is flexible and allows the implementation of many different algorithms, aside from DNA alignment. It allows the programmer to customize it in several different aspects, such as: number of cores, memory size, memory access time, simulated procedure times, etc. At the end of simulation, it is possible to build a chart using the log file produced during the simulation, in order to analyse the performance of the system according to the programmer implementation metrics. Therefore, the main established objectives have been achieved.

The experimental results demonstrated that the framework provides the system designer with a very useful preliminary characterization of the prototyped architecture. In particular, the presented evaluation demonstrates the relation between the number and performance of the computing modules, and the resulting alignment performance gain, as well as the inherent bus contention losses in the shared resources.

8.2 Future works

The experimental results showed that the simulation framework is able to run different algorithms that may or may not contain data dependency between the executed data in the slave processors. In order to improve the simulation framework regarding to its efficiency, some improvements are worth to be executed, such as:

1. The figures 7.1, 7.2 and 7.3 that illustrate the performance of the system during the first implementation are very useful. These figures allow the programmer to understand the behavior of the system and to detect bus contention situations. In order to entirely visualize
8. Conclusions and Future Works

the behavior of each simulation, a different tool must be used or created which is able to completely output the state of each processing module from the beginning until the end of the simulation.

2. The performed simulations of the second implemented algorithm demonstrated that from a certain number of slave modules, some of them started to be almost useless and the speedup soon stabilized. This is due to the fact that, during the complete simulation, there is always more than one slave available to execute tasks, therefore there are always slave modules that are not executing anything. The obtained results from these simulations prove that every algorithm implementation must be properly adapted concerning the modeled architecture characteristics.
Bibliography


