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

Pedro Magalhães
Instituto Superior Técnico
Oeiras, Portugal
Email: magalha3s@gmail.com

***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.**

I. Introduction

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 in 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 sould 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].

The final goal of this thesis is to present a simulation environment of a processor architecture which is optimized for the implementation of a particular set of bioinformatic algorithms. One of such algorithms is denoted as Smith-Waterman (SW) and it is widely used to determine the optimal sequence 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 x 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

Considering any two strings S_1 and S_2 of an alphabet ε 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 [3].

The recursive relation to calculate the local alignment score $G(i, j)$ is given by Eq. 1 and 2, where $Sbc(S_1(i), S_2)(j))$ denotes the substitution score value obtained by aligning character $S_1(i)$ against character $S_2(j)$ and α represents the gap penalty.

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

$$G(i, 0) = G(0, j) = 0 \tag{2}$$

A. 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.

II. System Description Languages

System description languages [4] are fundamental tools for project design, whether it is an electronic or an informatic project. They are generally used to allow cutting the project into 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 particualr 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.

A. 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. The engineers that built this language 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, constratly to Ada and most programming languages where only ascending indexing is available.

B. 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. It is also used in the verification of analog and mixed-signal circuits. Verilog [5][6][7] 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 (=) and a non-blocking assignment (<=). 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 similiarities with it. This way, Verilog is case-sensitive, has a basic preprocessor, has equivalent control flow keywords like if/else, while, case, etc, and the same operator precedence.

C. SystemC

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 deliberatly 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 wether 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 tipically 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 [8]. Abstrat ports (also called as channels) brought TLM (Transaction Level Modelling) into SystemC, that is, modelling of systems above the RTL level of abstraction.

III. Related Work

To better demonstrate the adequacy of SystemC language and its simulation environmet 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 a I2C bus controller, followed by a CAN bus design and finally a parallel processor interconnection system. The final subsection of this chapter gathers a set of factors that led the designers to choose SystemC as the simulation tool of the projects.

A. I^2C Bus Controller

The Inter-Integrated Circuit (I^2C) bus [9] is a multi-master serial single-ended computer bus. 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.

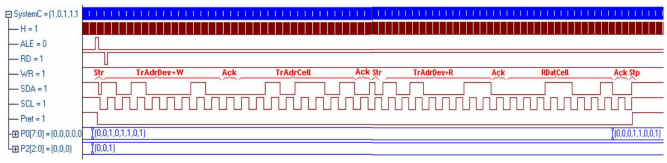

Fig. 1. Simulation of an I2C controller using SystemC.

Figure 1 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 in [10], the designers were able to sucessfully 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 [10].

B. CAN Bus Architecture Model

In this section, it will be described the modeling and simulation of a CAN bus architecture [11], 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.

The CAN controller is composed of four modules. They are the Message Buffer module (MB), the Interface Management Logic module (IML), the Bit Stream 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 [11].

After its implementation and several test scenarios, the designers were able to identify various design flaws and bugs of the CAN bus model [12].

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.
- 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.

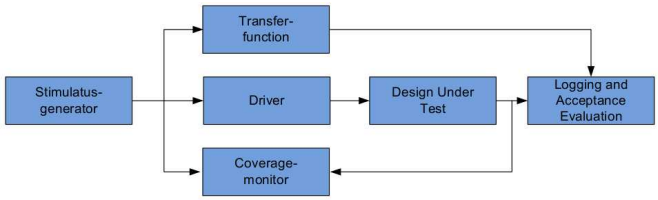

Fig. 2. CAN bus Testbench model.

C. 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 [13]. 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. Beeing 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 pratical. 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 1 in the first project case.

IV. Simulation framework architecture

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 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. Figure 3 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.

A. 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.

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.

Monitor Module - the monitor is in charge of tracing the status of the simulated device. The modules included in this operation are the master and slave modules. Every time a module changes its status, the event is sent to the monitor and recorded in a log file. 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.

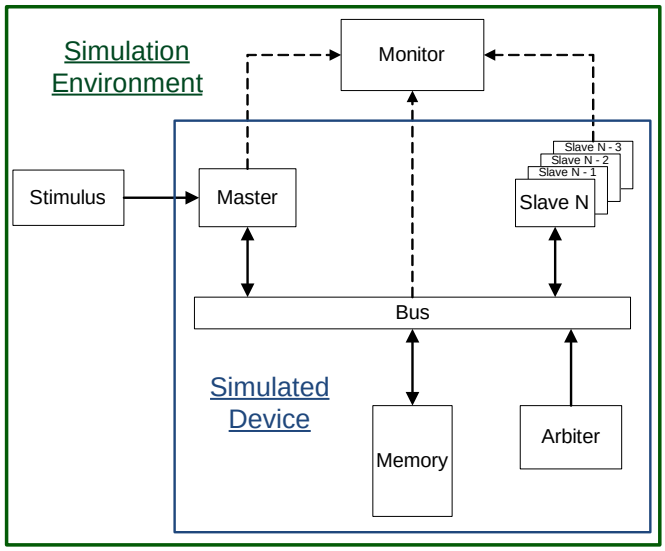

Fig. 3. Environment System Model

B. Simulated Device

The simulated device is composed of one master processor, one shared memory device, one bus that interconnects all the modules, one arbitrer and a variable number of slave processors that are predefined before build time.

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 beeing 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.

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.

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.

Memory - 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 beeing processed with the Smith-Waterman algorithm.

Every module (master or slave) 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 shows the address space made available to each module.

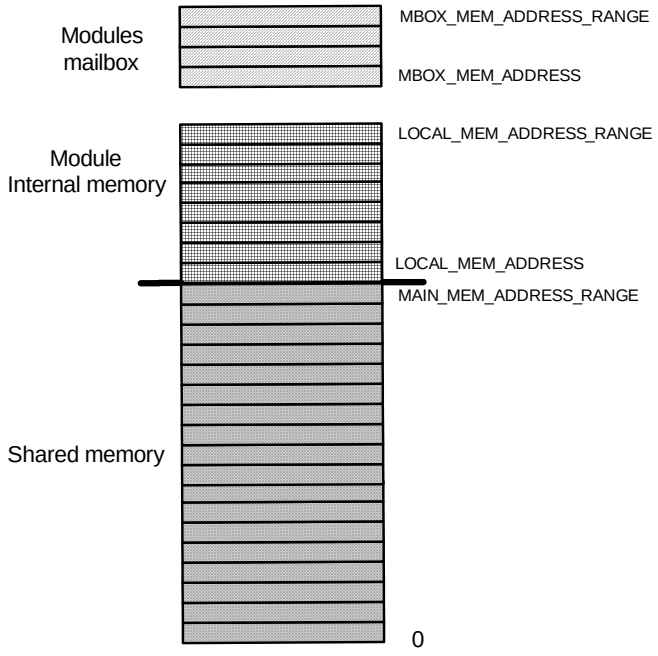

Fig. 4. Module addressing space.

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.

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 arbitrer 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 beeing requested by the master and slaves, but only one module at a time can access the bus.

C. Mailboxes

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.

V. Framework Implementation

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.

A. Main Modules Implementation

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. The main structure of the modules is implemented with the "ModuleStruct" class. 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) and finally the capacity of the module's mailbox where the command messages are stored.

Methods - 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 resposibility 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. 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. 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 varibles - 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.

B. Stimulus

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).

The ”test_port” is used to send to the master the table of addresses (”data”) 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.

C. 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 arbitrer use the monitor’s interface, so that the entire activity of the system is tracked down.

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”. Trough this channel, the simulated device’s modules are able to tell the monitor its status changes at the specific time that they occur.

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. This way, the programmer may easily perform statistical calculations during and or at the end of the simulation.

D. Master

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

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. This slave management is accomplished through three dedicated methods ”next_destiny”, ”isjobFree” and ”finishjob”.

As it was referred before, before the simulation starts, the stimulus module sends to the master the table with the addresses in stored memory of the test data through the

”test_if” test channel. The table is a two dimension integer array with pairs of test data identifiers and the corresponding address, where data is stored in the memory. In order to accomplish this transaction, the stimulus module sends the pointer of the table as a parameter through the test channel interface method.

E. Slave

The slave’s structure is entirely built on ”ModuleStruct” class structure. The only 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.

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.

F. Memory

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

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 hability to completely specify the address where data is written to or read from. The class that represents a memory unit (”memory_unit”) is used to implement the shared memory of the system, as well as the internal memories residing within the master and slave modules.

G. Bus

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. It 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. The ”ram_port” port is used to perform write and read operations, according to the requests made by the master and slaves. 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”.

The bus module has three independent queues. Each one is used to store the requests according to their priority (high, normal and low). When the modules make requests to access the bus and are not granted to access it, these are inserted into the queue that corresponds to its priority. As long as high priority requests exist, the normal and low priority requests will not be attended. The same logic is present when there are only normal and low priority requests. The queues are implemented with the ”sc_fifo” data type of SystemC.

The ”new_request” method is the interface used by the modules to place a new request in the bus. Another 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.

Finally, the ”handle_request” method is called whenever a request is ready. This method performs the requested operation and signals the monitor about it.

H. Arbitrer

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.

The method ”is_owner” is used by the bus to ask the arbitrer if a specific module is owning the bus at the current instant. 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 immediatly attended or inserted into the queue line.

Every time a different module gains accesss 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.

At the time the main action of the bus is performed, a routine check is run in the request queues. This routine check is executed by the ”arbitrate” method provided by the arbitrer. This routine checks for free slots inside the sorted priority queue and adds one or more requests that are awaiting in the unsorted queue.

VI. Programmer Interface and API

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.

A. Programmer Interface - 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.

1) Building and sending a message: The process of sending data to the shared memory or a module is depicted in figure 5 and is done via four steps:

```
void send_msg_example(){

    REQUEST *temp_req = new REQUEST;
    char* command = new char[8];
    strcpy(command, "SW_ALGO");

    temp_req->size = new_message(temp_req->data, 3,
                0, command,
                1, 12345,
                1, 8935);


    //sending the message to internal/shared memory
    temp_req->address = 3166;

    /*
    or sending the message to another module
    temp_req->address = getModuleMboxAddress(next_recipient);
    */

    write_op(temp_req);     //send the message to the internal or shared memory
                            // or mailbox from another module
    clearREQ(temp_req);
}
```

Fig. 5. Creating and sending a message example.

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 from 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. 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.
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. An important aspect, is that the messages strucutre 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.

2) 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.

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.
4) Once all the parameters are set, the ”read_op” method carries on with the reading procedure.

```
void read_msg_example()

  int value1, value2;
  char *command = new char[2];

  REQUEST *req = new REQUEST;

  //read data from a specific address (internal or shared memory)
  req->address = 3166;

  /*
  or read from my mailbox// No need to specify req->size
  req->address = getModuleMboxAddress(myid);
  */

  req->size = 20;

  read_op(req);         //read message from the shared or internal memory or mailbox

  getarg(req->data, command, value1,    //get command into command variable
                0, 20, false);

  getarg(req->data, command, value1,    //get 12345 into value1
                1, 20, true);

  getarg(req->data, command, value2,    //get 8935 into value2
                2, 20, true);

  int result = sw_algorithm(value1, value2);

  clearREQ(req);
}
```

Fig. 6. Reading a message example.

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.

VII. Simulator Setup

Figure 7 shows the important parameter values that are present within the simulator’s engine. The shown parameters have the greater interest and impact in the simulation results.

A. 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 7 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.

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

```
//ACCESS TIME DELAYS
const unsigned int ACCESS_DELAY_5NS = DELAY_1NS * 5;
const unsigned int ACCESS_DELAY_10NS =  DELAY_1NS * 10;
const unsigned int ACCESS_DELAY_20NS = DELAY_2NS *10;
const unsigned int ACCESS_DELAY_50NS = DELAY_5NS * 10;
const unsigned int ACCESS_DELAY_100NS = DELAY_10NS * 10;

const unsigned int USED_BUS_ACESS_DELAY = ACESS_DELAY_50NS;
const unsigned int USED_MAIN_MEM_ACESS_DELAY = ACESS_DELAY_20NS;
const unsigned int USED_MOD_MEM_ACESS_DELAY = ACESS_DELAY_5NS;
const unsigned int USED_MBOX_MEM_ACESS_DELAY = ACESS_DELAY_5NS;
//------------------------------------------------------------

//MEMORY VALUES
const unsigned int MAIN_MEM_SIZE = MAINMEMSIZE;
const unsigned int MASTER_MEM_SIZE = MASTERMEMSIZE;
const unsigned int SLAVE_MEM_SIZE = SLAVEMEMSIZE;
//------------------------------------------------------------

//MODULES
const unsigned int BUS_TURNS = 10;

//ATTEMPTS TO SEND A MESSAGE
const unsigned int SEND_ATTEMPT = 3;
const unsigned int SLAVE_WORK_TIME = DELAY_1NS * DELAY_2NS;
const unsigned int SLAVES = 4;

const unsigned int MBOX_MEM_ADDRESS = MAIN_MEM_SIZE + MASTER_MEM_SIZE + 30;
const unsigned int MBOX_MEM_ADDRESS_RANGE = MBOX_MEM_ADDRESS + SLAVES - 1;

int getModuleMboxAddress(int ModuleId){return MBOX_MEM_ADDRESS + ModuleId;}
//------------------------------------------------------------

//NUMBER OF TEST DATA TO LOG
const unsigned int LOG_MSGS = 15;
//------------------------------------------------------------
```

Fig. 7. Parameters file.

is defined through a set of constants that are present in another parameters file which is not illustrated in this document. 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 of the shared memory, internal modules memories and the mailbox size

B. Modules behavior

The sections denoted as ”MODULES” and ”ATTEMPTS TO SEND A MESSAGE” in the figure 7 contain values that affect the modules behavior regarding several subjects.

The ”BUS_TURNS” constant specifies the number of consecutive turns, that a module can access the bus without having to wait in the queue after gaining bus access. The next constant is the ”SEND_ATTEMPT”, which defines the number of attempts that a module does in case a certain bus access fails. The simulated period that a module takes executing the implemented algorithm is affected by the constant ”SLAVE_WORK_TIME”. Finally, the number of slaves that are present in the simulation is set in the ”SLAVES” constant.

VIII. Experimental Results

This chapter presents the application of the developed simulation framework to simulate the execution of a parallel implementation of the alignment procedure. The simulator picks a large set of DNA query sequences which are independently aligned to a single (and larger) reference sequence. 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, 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).

A. 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 [14] or dedicated VLSI architectures [15].

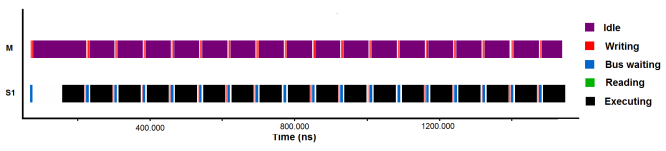

Fig. 8. System snapshot using one slave module for K=0.2.

1) Sequencial execution (one slave): The first graph presented in figure 8 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.

2) Concurrent execution (multiple slaves): The following paragraphs regard figures 9 and 10. 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 9 and 10 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.

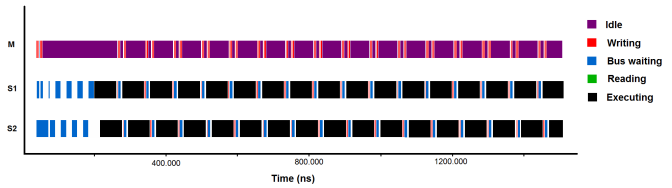

Fig. 9. System snapshot using two slave modules for K=0.2.

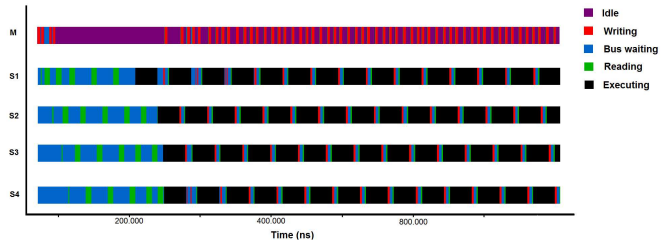

Fig. 10. System snapshot using four slave modules for K=0.2.

3) Discussion: Figure 11 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. 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.

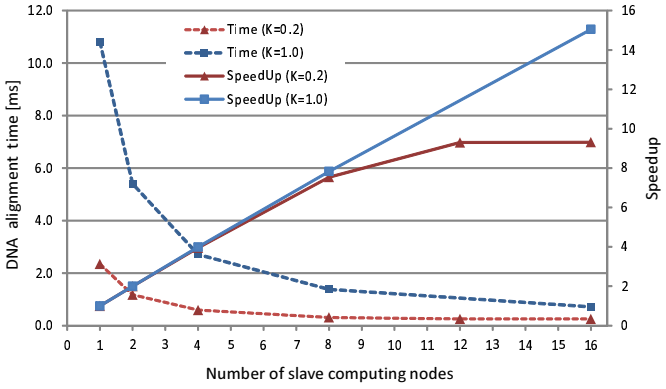

Fig. 11. Total execution time and speedup as functions of the number of slave processors.

IX. Conclusion

The simulation framework is flexible and allows the implementation of algorithms aside from DNA alignment. It allows the programmer to customize it in several different subjects such as: number of cores, memory sizes, memory access delays, simulated procedure times, etc. At the end of simulaiton runs, it is possible to build a chart using the log file produced during simulation in order to visualize the performance of the system according to the programmer implementation metrics. Therefore, the established objectives have been achieved.

The experimental results (in chapter VIII 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.

X. Future works

The figures (8, 9 and 10) that illustrate the performance of the system are very usefull. These figures allow the programmer to understand the behavior of the system and to detect bus contention situations. In order to entirely visualize 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.

ACKNOWLEDGMENT

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.

REFERENCES

[1] J. Gerlach and W. Rosenstiel, “System level design using the SystemC modeling platform,” *University of Tubingen*, 2001.

[2] Çağkan Erbaş, “System-level modelling and design space exploration for multiprocessor embedded system-on-chip architectures,” Ph.D. dissertation, Advanced School for Computing and Imaging, Turkey, 2006.

[3] N. Roma, N. Sebastiao, and P. Flores, “Hardware accelerator architecture for simultaneous short-read dna sequences alignment with enhaced trace-back phase,” *Microprocessors and Microsystems: Embedded Hardware Design (Elsevier)*, 2001.

[4] D. Gajski and L. Cai, *Transaction Level Modeling: an overview*. ACM New York USA, 2003.

[5] R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description Language and Design*. Jonathan Allen, 1998.

[6] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*. Kluwer Academic Publishers Norwell MA USA, 1996.

[7] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers Norwell MA USA, 2000.

[8] K. Bjerge, “Usage of system level modelling with systemc,” Danish Technological Institute, Tech. Rep., 2008.

[9] R. Mitchell, N. Damouny, C. Fenger, and A. Moelands, “An integrated serial bus architecture: Principles and applications,” *IEEE*, 1985.

[10] M. Alassir, J. Denoulet, O. Romain, and P. Garda, “A systemc ams model of an i2c bus controller,” *EURASIP*, 2008.

[11] R. B. GmbbH, *CAN Specification Version 2.0*, 1991.

[12] G. B. Defo, C. Kuznik, and W. Muller, “Verification of a can bus model in systemc with functional coverage,” *IEEE*, 2010.

[13] O. S. Initiative, *SystemC AMS Extensions User’s Guide*, 3 2010.

[14] M. Farrar, “Striped smith-waterman speeds database searches six times over other SIMD implementations,” *Bioinformatics*, vol. 23, no. 2, p. 156, 2007.

[15] N. Roma, N. Sebastiao, and P. Flores, “Integrated hardware architecture for efficient computation of the n-best bio-sequence local alignments in embedded platforms,” *IEEE Transactions on Very Large Scale Integration Systems*, 2011.