A.Y: 2008/2009

National Engineering School of Sousse

Sousse University, Tunisia

Master Thesis

Specialty: "Intelligent and Communicant Systems"
Option: micro-electronic embedded systems

presented and defended publicly

by

Nabila Abdessaied

Title:

Design of a Java Simulator for Fast Prototyping of System-on-chip

Supervised by: Matthieu Moy (VERIMAG)
Giovanni Funchal (VERIMAG)
Younes Lahbib (ENISO)

VERIMAG Research Lab
Centre Equation
2, avenue de Vignate
38610 Gieres, France

بسم الله الرحمن الرحيم

"و ما توفيقي إلا بالله عليه توكّلت و إليه أنيب"
"اللّهمّ علّمنا ما ينفعنا وانفعنا ممّا علّمتنا"

إهداء

إلى من أوجب الله شكرهم فقال (أن أشكر لي ولوالديك).

إلى من لهم عليّ حقّ التّعظيم.

إلى والديّ العزيزان.

إلى أخواتي و إلى أخي .

إلى جدّي سالم.

إلى أساتذتي الأفاضل، إلى كلّ من علّمني حرفا.

إلى كلّ العائلة و الأقارب.

إلى كل الأحبّاء والأصدقاء.

نبيلة

Acknowledgments

I would like to express my deep and sincere gratitude to my supervisors, *Matthieu Moy*, assistant Professor in ENSIMAG engineering school and VERIMAG laboratory, and *Giovanni Funchal*, Ph.D candidate in Computer Science at VERIMAG laboratory, their wide knowledge and their logical way of thinking have been of great value for me. Their understanding, encouraging and personal guidance have provided a good basis for the present thesis.

I am deeply grateful to my supervisor, *Younes Lahbib* for his detailed and constructive comments, and for his important support.

I would like to express my gratitude to VERIMAG laboratory, who gave me the opportunity to do this thesis. I am grateful to the assistant Christine Saunier who helped me to complete the administrative papers. Thanks also to the system operators (Nicolas, Jean-Noél) who do a great job for the lab.

During this work I have met many colleagues at VERIMAG, for whom I have great regard and I will keep beautiful memories: Salwa, Ramzi, Unmesh, Raj, Hung, Laurie, Yanhong, Ronjgie and all the other members.

I owe my most sincere gratitude to the ENISO director Miss Najoua Ben Amara and the responsible of the master, Mr Hachmi Dali for their help and their assistance. I warmly thank All the ENISO institute members and Staff who helped me to carry out this work.

I also wish to thank all the friends at the crous residence Hector Berlioz, with whom I shared great moments during my intern ship. Specially when we met together in the kitchen to prepare our dinner: Lamin, Andy, Ibtissem, Anas, Djosepé, Nawel, Sandra and Dima. It was a good opportunity for changing mood, sharing ideas and having fun.

I owe my loving thanks to my family. Without their encouragement and understanding it would have been impossible for me to finish this work. My special gratitude to my brother, my sisters and their families for their loving support. My loving thanks to my oncle Bouderbela, his wife Fatouma and their children who always supported and encouraged me.

My last thanks go to all the people who have helped me, to all those who gave me the possibility to complete my master thesis. Thank you all.

Contents

List of Figures

Chapter 1

Introduction

Contents

1.1 Background

Nowadays, embedded systems are everywhere in our lives: cellular phones, digital cameras, MP3 music players, digital video recorders and tuners, electronic systems in our cars, etc. Such devices require satisfying strong constraints: they have to perform intensive computer processing (for instance to decode digital video) while consuming very little energy, and at a reasonable production cost. Tremendous processing power must be fitted into a very small surface. Traditional computer micro-processors are not well suited for such applications, because they would consume a lot for the required processing and they are expensive. The idea of a System-on-Chip (SoC) is to regroup all the hardware components necessary to the systems operation on a single chip. This includes energy efficient micro processors, local memories, radio-frequency analog parts and hardware blocks that are dedicated to speed-up specific, computing-intensive tasks. Therefore, a System-on-Chip actually consists in hardware but also in embedded software executing on its micro-processors.

Because of the complexity of these systems and time to market constraints, it is no longer tractable to design the software and the hardware separately. The design process must take into account the interaction between these heterogeneous parts. In particular, various models of the system are needed along the design flow, for different usages and with different levels of details. For instance, Register Transfer Level (RTL) models are a common entry point for producing the actual hardware. These models are developed using hardware description languages such as VHDL or Verilog. They are destined to be transformed automatically into models with more details (using so called synthesis and place-and-route software), which ultimately leads to the production of the hardware chip. Separately, developing the software requires its execution on the chip. Because of time-to-market constraints, it is not acceptable that software developers have to wait for the physical chip to be available to start their work. It could be possible to use the RTL models as a support for executing the embedded software, but the slowness of simulation prevents any practical use for complex chips. The fundamental reason why the RTL simulations do not scale well is that the models contain too much detail.

Transaction Level Modeling (TLM) [CG03] is a relatively new type of model of Systems-on-Chip, which has been developed initially to speed-up the execution of the embedded software. To write these models, SystemC [Ini06, Gro02] has become the de facto standard language used in the industry. Transaction Level Models are more abstract, so they are not only faster to simulate, but also require less effort to build. They are thus available far before the RTL in the design flow and allow the embedded software development to start earlier. However, TL models do not replace the RTL as a new entry point for designing the hardware, because there is no way, at least for now, to automatically transform them into RTL.

1.2 Working context

The work presented in this document was carried out during my Master Project, started on 12/01/2009 and ends on 04/07/2009, done into the Synchronous Language and Reactive Systems (Synchrone) team of the research laboratory Verimag [Lab09] in Grenoble, France. Verimag is a leading research center in embedded systems. Over the last fifteen years, Verimag has actively contributed to the development of the state-of-the-art, in particular for synchronous languages, verification, testing and modeling. Research at Verimag provides theoretical and technical means for developing embedded systems, contributing to scientific advancement and industrial progress.

Verimag's strategy is to maintain a good balance between fundamental, experimental and applied research. This is particularly visible in long term cooperation with academic and industrial partners. For several years, one of the topics addressed by the laboratory concerned the SoCs validation at the transactional level. This was realized in collaboration with STMicroelectronics.

In particular, a work [Moy05] on Formal Verification of Transaction Level Models written in SystemC was carried out by Matthieu Moy and defended in 2005. Claude Helmstetter has then studied the scheduling of SystemC/ TL Models in a thesis [Hel07] defended in 2007.

To get preliminary performance analysis to take some decisions about the RTL design Jerome Cornet, in a thesis [Cor08] defended in 2008, participates of an ongoing research effort to construct "PV+T" models from existing PV models. The idea is to introduce a new "T" model which is statically linked to a "PV" model in other to obtain a "PV+T" one. The "T" model, add the number and width of the buses, whose main consequence is to fix the granularity of the transactions in the "T" model. Then, other architectural artifacts, such as pipelines and time, must also be modeled. At last, the execution of both "PV" and "T" models is tuned to achieve "PV+T" behavior.

Then Giovanni Funchal had compared these intermediate models ("PV+T" models) with the RTL models in his master thesis [Fun07]. In the other hand Youssef Bouzouzou [Bou07] in collaboration with ORANGE BUSINESS SERVICES has worked in Semantics-Preserving Parallelization of the SystemC Scheduler for Reduced Simulation Time he has focused on the parallelization of SystemC execution engine to exploit at best the multiprocessors architectures while keeping the semantics of cooperative execution of language.

1.3 Motivation

TLM models are most of the time represented using SystemC language. This due to the fact that SystemC is faithful, largely, in representing the hardware chip to a virtual one. Our motivation, in this work, is to prove that we could write the transactional models, while remaining faithful, even using an execution model other than SystemC. We will work to build a faithful model of execution in capturing the real chips platform in these sides:

1. ***Parallelism:***
   When we focus a physical chip closely, we remark that it's composed of components and blocks arranged in parallel, they work and communicate at the same time. To imitate this specification, JTLM uses the Java Thread platform which allow the parallel description and the parallel execution [OW04].

2. ***Synchronization:***
   Parallel programming is useful when one represents a parallel system but in the other hand we are obliged to synchronize, which is an error-prone task, between concurrent components because they share the same resources. Although it could represent parallel system i. e. parallel description, SystemC has a cooperative scheduler [AHT$^+$] that means sequential execution, that resolves a big part of the synchronization problem.

   For JTLM, it would be an opportunity to see another point of view of components functioning since their behaviors are following a preemptive policy in their execution [LB00], i. e. in parallel execution. But in the same time this way of parallel programming is difficult to be correctly synchronized and difficult to debug, this due to the non reproducible bug, i. e. occurring randomly and for reasons unknown. The main source of such bugs, in contemporary languages like Java, dues to multi-task programming. Despite the Locking tools, it is often difficult to totally avoid unexpected situations and concurrent access to shared resources in memory. As a remedy, Java provides tools for synchronization that help to avoid concurrent access and memory consistency errors.

3. ***Time modeling:***
   To model the time that the physical chip may take to accomplish its work, the JTLM uses in the model execution a variable that represents this time; it is called the simulation time. Whereas the time that may take the simulation of the virtual platform is called the wall clock time. So to summarize the wall clock time is used to designate the time of a platform simulation while the simulation time is used to designate the time that may take the real hardware.

1.4 Objectives

Our objective is to design a simulator for the rapid prototyping of systems-on-chip in order to view the behavior of TLM platforms without recourse to SystemC language. It will help to identify what belongs to TLM from those of SystemC. Our simulator should respect the faithfulness to the hardware and, why not, bring innovative ideas.

1.5 Approach

In the first part of the training, we identify the constraints imposed on the execution of transactional model by a simulator. Then, we design a transactional model simulator by defining a Java language library; we will rely on the Java threads library which provides a set of functions to manage the execution queues. The thread primitives allow to manage their life cycle and provide mechanisms for synchronization and management of critical sections.

1.6 Expected results

As a result of this work, we prove the possibility of modeling transactional platform using JTLM and we give a case study that shows what could this execution model offers for hardware modeling. Moreover we will give a brief tutorial which will provide instructions for the use of JTLM.

1.7 Faithfulness

To be faithful to the physical chip, the simulation using a Transactional level platform should have the same result as the simulation into an RTL platform. But TL models give more behaviors comparing to RTL when executing software into it. Embedded software which works well in TLM platform must have the same result in RTL, i.e. it musts run successfully into an RTL platform. However, the reverse is not always true; it's not guarantee all the time that the software can run well in the virtual TLM platforms, this due to how much the parallelism of the implementation is applied in the execution of the platform.

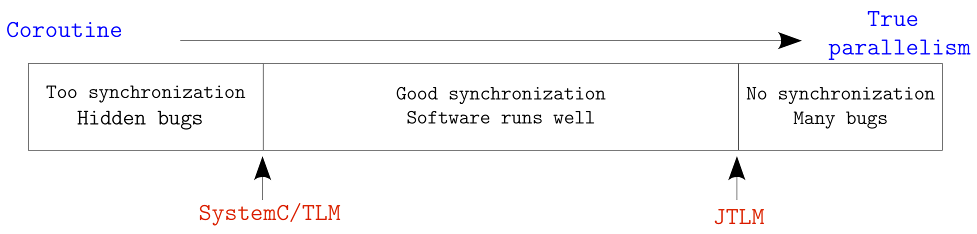


Figure 1.1: The different types of TL models execution

SystemC language provides parallel description and sequential implementation, it facilitates the task of synchronization between platform components but in the same time the big granularity of transactions hides many bugs that programmers could not see them so the embedded software could not work properly. To solve this problem, programmers are used to add instructions to split these transactions to give other processes a chance to run.

In the other hand, if the TLM models are implemented with a language that has parallelism in execution such as Java; the embedded software could never run into theses models because of the absence of the synchronization. For this reason we had add into the JTLM, since it's written with Java, synchronization tools to reduce the bugs number and to let the embedded software works correctly into the TL platform realized with JTLM.

To conclude, as the figure shows 1.1, to guarantee that a software runs properly into a TL platform, this last should not have nor a true parallel executions neither sequential execution but sharing at the same time the two proprieties of execution.

1.8 Related works

There are already modeling approaches that deal with simulators. This section highlights a few of these approaches.

1.8.1 SystemC

SystemC [Ini06] is a C++ based modeling platform supporting design abstractions at the register-transfer, behavioral, and system levels. Consisting of a class library and a simulation kernel, the language is standardized and supported by the Open SystemC Initiative (OSCI), a consortium of a wide range of system houses, semiconductor companies, intellectual property (IP) providers, embedded software developers, and design automation tool vendors.

SystemC separates communication from computation by using port-interface calls. However, it lacks all other separations of concerns, such as behavior-performance and function architecture.

As a result, it is less efficient in modeling reusable system level designs. SystemC also has standard libraries for Transaction Level Modeling and Verification.

1.8.2 Metropolis

Metropolis [BWH+03], based on a meta-model with formal semantics that developers can use to capture designs, Metropolis provides an environment for complex electronic-system design that supports simulation, formal analysis, and synthesis.

Metropolis is a modeling and simulation environment based on the platform-based design paradigm. The key idea is to separate function, architecture, and model of computation into separate models. Although Metropolis allows co-simulation of heterogeneous PEs as well as different models of computation, a refinement or verification flow between different abstraction levels has not emerged. The Metropolis project is supported by the Gigascale Silicon Research Center.

1.8.3 SpecC

SpecC [BWH+03] is a system-level design language (SLDL) and a system-level design methodology. SpecC has been proposed as the standard system-level design language based on C programming language which covers the design levels from specification to behaviors. It can describe both software and hardware seamlessly and a useful tool for rapid prototyping as well.

SpecC methodology and language have been designed and implemented to integrate the specification and the design phases in the SOC design process. Originally developed at University of California, Irvine, with sponsorship from several companies, SpecC language is a system specification description language based on C. It allows the same semantics and syntax to be used to represent specifications for a system concept, hardware, software, and, most importantly, intermediate specification and information during hardware/software co-design stages.

1.8.4 Milan

MILAN [BPL01], a model based extensible framework that facilitates rapid, multi-granular performance evaluation of a large class of embedded systems, by seamlessly integrating different widely used simulators in to a unified environment. The MILAN modeling paradigms facilitate seamless integration of a variety of simulators at multiple levels of granularity, into the framework. A single graphical user interface allows designers to specify different aspects of embedded system hardware and software, and performance requirements.

MILAN [LDNA03] provides an integrated environment where existing development and analysis tools, primarily simulators, can work seamlessly together.

MILAN is a collaborative project between the University of Southern California (USC) and the Vanderbilt University (VU).

1.8.5 Glonemo

GLONEMO [SMMM06]: Global and Accurate Formal Models for the Analysis of Ad-Hoc Sensor Networks approach for the formal modeling and analysis of ad-hoc sensor networks, at various levels of abstraction. It is global because it takes into account all the following aspects: a precise modeling of the hardware that implements a single node; the protocol layers; the application code; an abstract model of the physical environment as viewed by the sensors. The global model enables validation by simulations. Glonemo is a collaborative project between the laboratory Verimag and France Telecom company.

1.8.6 Ptolemy

The Ptolemy project [THG+92] studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interaction between components. A major problem area being addressed is the use of heterogeneous mixtures of models of computation. A software system called Ptolemy II is being constructed in Java. Ptolemy II differs from other commonly used graphical block-diagram languages in that they typically support only one model of computation. In addition, Ptolemy II is a more open architecture in that its infrastructure is open source, and the interfaces to the core mechanisms in the software are published and documented. The Ptolemy project has been under development in Java since 1997.

1.9 Overview of Thesis

This document contains 6 chapters; the first chapter is this introduction.

- Chapter 2, *"General context"* presents the context of work. It was partly written on the basis of Thesis of Matthieu Moy and Jérôme Cornet. We present a brief summary of the history of the systems on chip design flow. Then we present the transactional approach (TLM) for modeling systems on a chip, and then we give a technical presentation of the SystemC library and the Java language.

- Chapter 3, *"Problem description"* presents the problems that belongs to SystemC. We give, in detail, our approach of the design of a execution model for the rapid prototyping of SoCs. Then we highlight the problems of parallel programming and we present the synchronization tools as a solution to that issue.

- Chapter 4, *"Contributions"* present the JTLM library; what it consists of, what it offers to model TLM platform. Then we give the tests performed in order to evaluate this model of execution.

- Chapter 5, *"Case study"* describes in detail a TLM platform implemented in JTLM to improve the capability of the JTLM library to model TLM platform, also to show that TLM could be written in JTLM as it can be written in SystemC.

Finally, chapter 6 *"Conclusion"*, concludes the thesis and provides pointers to future work.

Chapter 2

General context

Contents

2.1 The need for higher abstractions

Transfer register level remains the common way which leads, using synthesis and placement and route software, to the manufacturing of the physical chip. Meanwhile engineers can't begin writing the software parts of the system till the chip would be produced to execute their work on it.

This methodology is no longer realistic since it takes too much time and it does not minimize the time to market. As an alternative to this problem, software developers use the RTL models as a support to check their work. But the simulation into these models is too slowly because they contain many details, specially the complex chips. In order to increase the simulation speed, a technique called co-simulation was adopted, which consists of replacing complex components by other much simpler written in C like replacing the memory by an array and the processor by an instruction set simulator (ISS). Hardware devices called hardware emulators used for RTL simulation which they are efficient in terms of speed but they require the RTL model to be available They are very costly and they provide limited debugging capabilities.

Another solution to the problem is to raise the abstraction level by creating models with fewer details so they could be available before the RTL and would be used as simulation platform for embedded software development. This new level is called the Transaction Level Model (TLM).

Figure 2.1 shows an example of compared simulation times for encoding and decoding a picture in a MPEG4 codec.

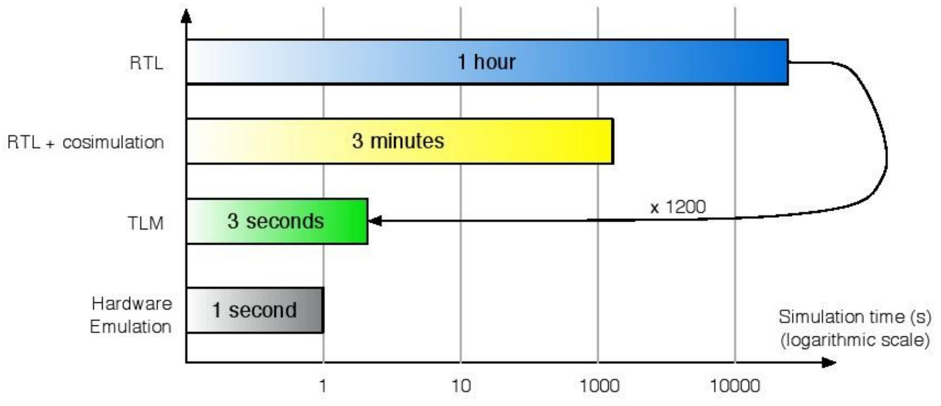


Figure 2.1: Simulation time for the encoding and decoding of one image in a MPEG4 codec

2.2 TLM

2.2.1 Common concepts

A full description of the TLM approach can be found in the book [Ghe05]: Transaction-Level Modeling with SystemC. TLM: Concepts and Application for Embedded Systems.

TLM models represent architecture. The architecture is defined by a set of components, connected by channels, see figure 2.2. Each component contains one or more concurrent processes, ports, and other components. A component could be a ***master*** (initiator) which is a component that starts a transaction or a ***slave*** (target) which is a component that receives and serves transactional requests. The ports represent the inputs and the outputs of components. Components communicate with each other directly or through out a communication channel.

In transactional models, we distinguish two types of communication: transactions and interrupts.

- An *interrupt* is a unidirectional data exchange between components. It proceeds directly to the target module. It is a simple point-to-point communication that do not require target address or to be routed by the bus.
- A *transaction* is an atomic data exchange between an initiator and a target. It is an exchange of a data or an event between two components through a bus model that routes the transaction from an initiator to a target.

The information exchanged via a transaction depends on the bus protocol. However, some of them are generally common to all protocols:

1. The type of transaction determines the direction of the data exchange, it is generally read or write,
2. The address is an integer determining the target component and the register or internal component memory address,
3. The data that is sent or received.

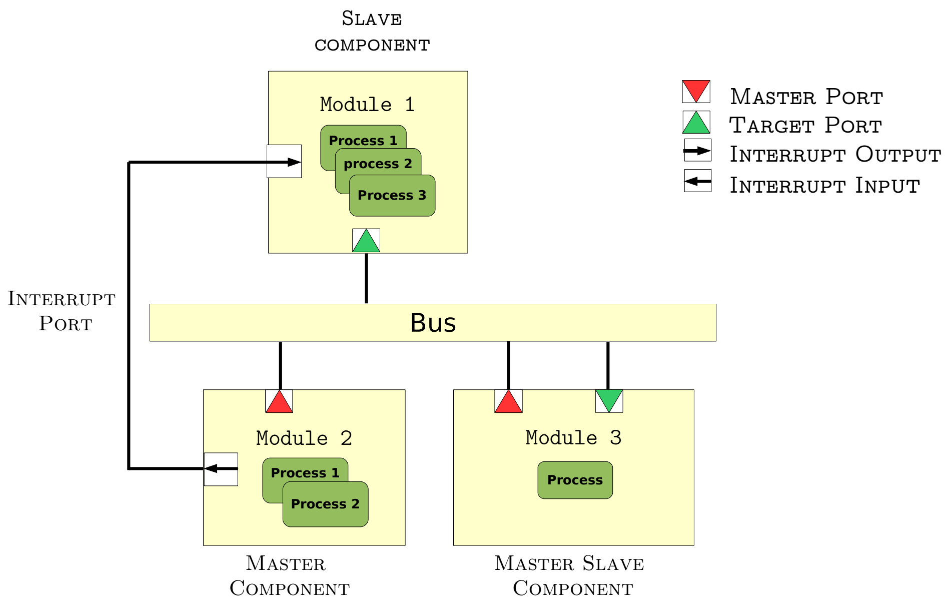


Figure 2.2: An example of a TLM platform

The most basic functionality shared by all buses is to route the transactions to their destination depending on their address. The destination is determined by the global memory address map which associates a memory range to each target port.

2.2.2 Various TLM applications

Depending on their uses, the TLM models are divided into two parts: Programmer View (PV) models called functional models, and Programmer View + Timing (PVT) called Timing models.

- **Programmer View (PV) model:**
  Used as a platform for developing and executing the embedded software because it is ready early and offers a high speed simulation, this amount to the fact that these models abstract so many details. It just contains only the details that allow the software to function as in the final chip. PV model is also more efficient in finding and fixing bugs than in the physical chips because it lets the developers insert debug messages so they can pick out the origin of the bug instead of seeing no output coming from the real chip.

  Functional models are used also to validate the behavior of an RTL block since they are seen as the reference model for the hardware.

- **Programmer View + Timing (PVT) model:**
  This model contains more details than PV models: timing information and new communication granularity that allow to TLM to perform another application which consists of the evaluation of the chip performance, to help architects determine the best compromise between the various factors involved(performance, area, costs,..).

2.3 TLM with SystemC

To implement and execute TL models, a computer language is required. Verilog and VHDL couldn't be the appropriate choice for this task because they were defined basically to describe low levels of abstraction such as RTL. Also we could not depend on the programming languages like C/C++ to satisfy such requirement that's dues to the fact that theses languages don't support hardware description. SystemC [Ini06] was the best alternative because it can satisfy all the needs: hardware and software description, rapid simulation platform and parallel execution semantics.

SystemC is a C/C++ library that has been standardized by the IEEE in 2005 [Ini06]. It affords classes to describe the TL model architecture, which consists of a set of components connected to each other (sc_module, sc_port...). It's behavior is implemented using processes (sc_thread...). Components behave in parallel and synchronized by the mechanism of events (sc_event...). After instantiating the modules and connecting their ports, communications between components are done by function calls through communication channels or directly to the target component. The first type of communication is called transaction and the other one is called interrupt used to model a point-to-point communication.

The *granularity* is the size of the data transported by the transaction. Depending on the real bus width and its capability to bursts transactions, a transfer for example of a picture in memory will actually be transferred line by line, or worse: pixel by pixel. To increase the simulation speed, a SystemC/TLM bus makes the transfer of the whole picture at once, making use of what we will call block transactions. It would be easier and more efficient to transfer big granularity transactions to avoid memory errors and make the simulation speed more important.

The SystemC simulator schedules the SystemC processes. At the first time all the process are eligible, only one process selected by the scheduler go in running phase and the others are waiting for time to elapse or a SystemC event. A process suspends itself by executing a wait instruction in order to let the other process execute their jobs; this is called cooperative scheduling policy.

The figure 2.3 gives an example of a transactional model.

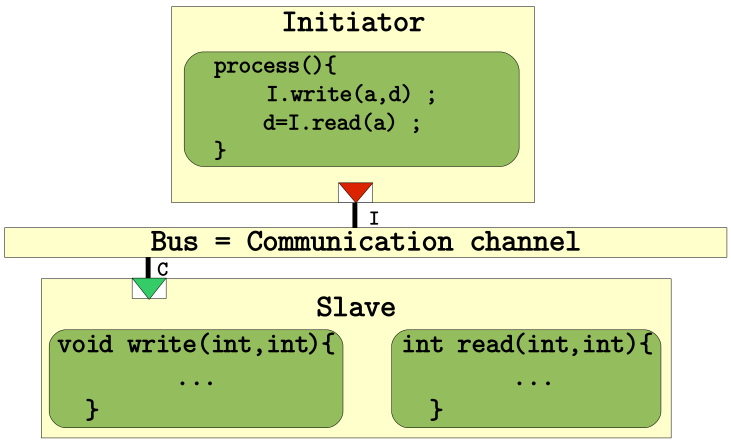


Figure 2.3: Example of a transactional model

2.4 Java

Java [AGH05] is a programming language originally developed by James Gosling at Sun Micro systems. The essential components in the Java platform are the Java language compiler, the libraries, and the runtime environment, called also virtual Java machine [LY99], in which Java intermediate byte code "executes" according to the rules laid out in the virtual machine specification.

The implementation of Java compilers, virtual machines, and class libraries were developed by Sun from 1995. In May 2007, Sun made most of their Java technologies as free software under the GNU (General Public License).

We have chosen Java as a programming language to carry out the building of the JTLM library for these reasons:

1. It allows parallel programming [Lea99] through its class thread.
2. Java threads are generally preemptive, the virtual machine is allowed to step in and hand control from one thread to another at any time. This preemptive scheduling would help us to view in different way the behaviors of TLM models.
3. Java offers a reliable mechanism for the synchronization between JTLM components.

2.4.1 What is a thread?

To avoid ambiguity, it is important to clarify that Thread is not a process. Indeed, the process live in virtual isolation while Threads are lightweight processes that live together in a single process. Unlike process, the threads share the same memory. A Thread is a portion of code to run in parallel with others.
We will use Java threading in order to implement the behaviors of JTLM components.

Chapter 3

Problem description

Contents

3.1 Introduction

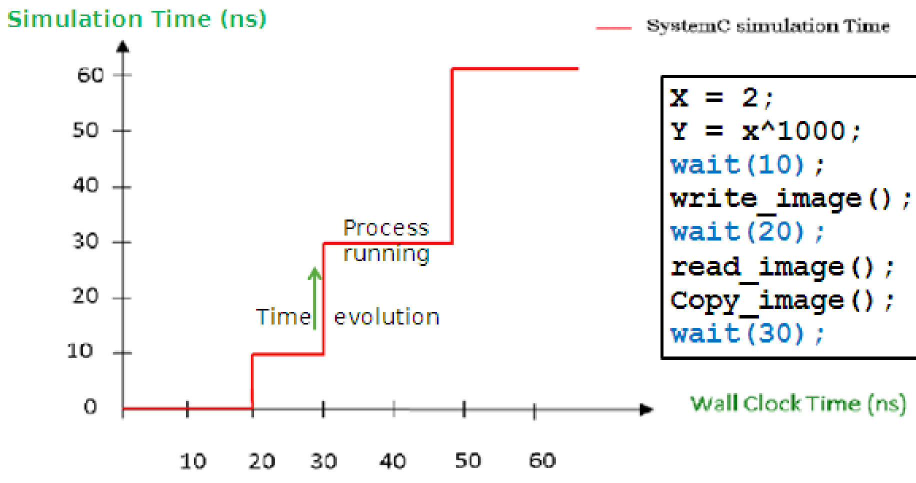
In this chapter, we will present our approach trying to avoid the bad side of SystemC while remaining faithful to the real hardware characteristics.

3.2 SystemC influences

Among the potentially bad habits in SystemC that we try to avoid them in JTLM, we invoke:

- SystemC was firstly designed to write RTL and cycle accurate models. After the adoption of TLM level in the design flow of SoCs, SystemC was chosen as a standard language to write TLM models, some of SystemC primitives were imported to implement the SystemC/TLM library. This fact of copying/pasting has led to having difference between what a SystemC/TLM model represents and what a real TLM model represents. SystemC designs are a bit far from the transaction level.

- SystemC does not model the duration that a task could take. It only invokes the waits between them so, when a SystemC transaction is running, the time remains constant and it elapses when there no eligible process. As the figure 3.1shows, the simulation time does not progress when a transaction is running.



```
X = 2;
Y = x^1000;
wait(10);
write_image();
wait(20);
read_image();
Copy_image();
wait(30);
```

Figure 3.1: Simulation Time of a SystemC/TLM model

- Because SystemC scheduler is cooperative, once a process is given control; it continues to run until it explicitly yields control or it blocks by invoking a waiting statement. For this reason if a process is polling another module it leads to a livelock since it does not give the hand to another module to make it goes out from this kind of looping by modifying the polling value.
- The big granularity of a SystemC transaction is quite good in the term that it helps to avoid memory errors but the problem of this approach is that although it works well in the virtual platform, it does neither guarantee that it works well in the real hardware nor help to find hardware bugs.

3.3 Our approach

In the first part of the JTLM construction, after defining them, we have to implement the basics elements called the core library. It would contain the required element for modeling transactional models such as components, ports, communication channel and interrupts.
All components behaviors are conceptually working in parallel, they are used to model components functionality. The Java programming language provides multi-threaded programming and offer constructs, such as synchronization, for creating functional programs. So we will exploit Java thread class for this issue.

In the second part, we have to design generic components, so called utilities that could be used in each platform such as the router or the memory.

After that we have to do several examples in order to validate the two packages JTLM core language and JTLM utilities but the important thing of this part is the verification of some notion and viewing the behavior of a TLM model implemented in JTLM this stage would help us to distinguish what comes from SystemC from those of TLM. Because programmers get used to to represent TLM models with SystemC, and since SystemC has few properties like having a cooperative scheduler, programmers tend to say that TLM also is cooperative. We can confirm, after using JTLM, which has a preemptive scheduler, that TLM can be cooperative as it can be preemptive. That depends on the language of implementation of a TLM model not on the TLM him self.

Later, to be more faithful, we will add the notion of time on our execution model so it could inform us about the time that may take the real chip. We call it the simulation time in the virtual platform. At this step we will develop examples to resolve some problems that exist already in SystemC/TLM and to know if they are attached to SystemC core language or not.

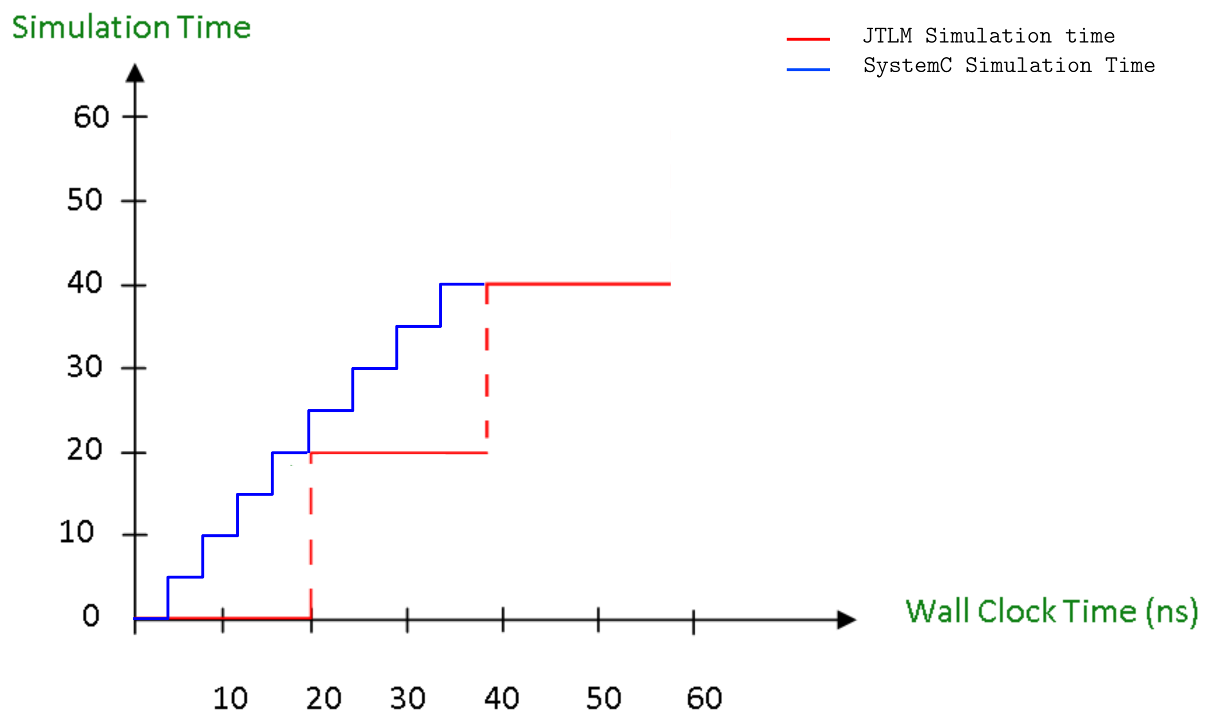


Figure 3.2: JTLM simulation time comparing with the SystemC time

Finally, to innovate, and since SystemC does not manage the tasks duration, JTLM will deal with this limitation, it will give the possibility to make timed tasks and it's benefits in reducing the time of a platform simulation. As the figure 3.2 shows a comparison in simulation time between JTLM and SystemC.

The figure 3.3 represent the difference between the way of SystemC and JTLM to model the duration of a process.

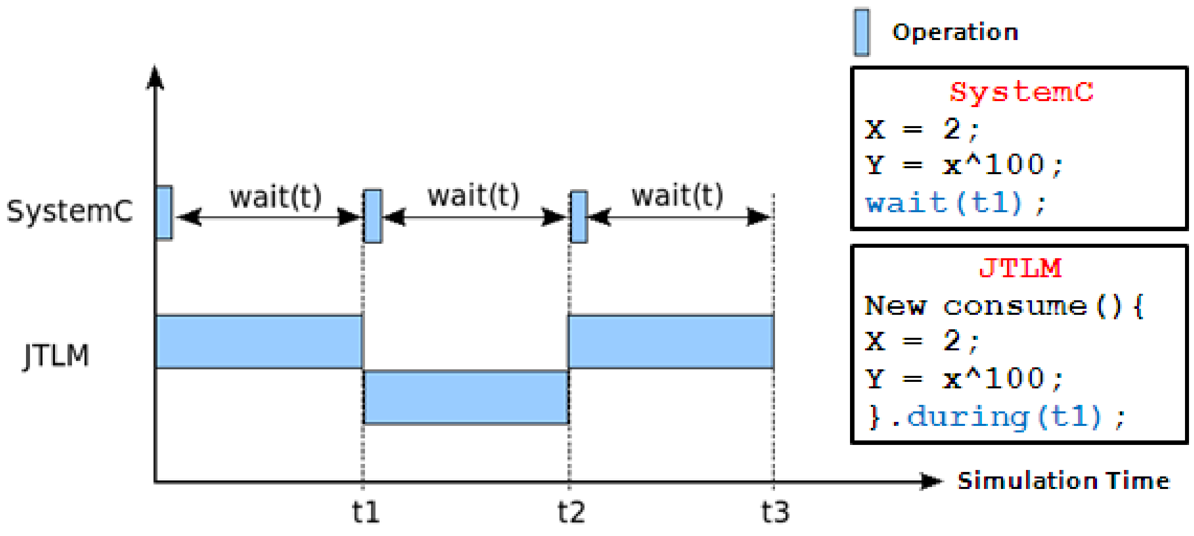


Figure 3.3: The time managment in SystemC and JTLM

In SystemC, the scheduler gives the hand to run to only one component. Suppose for example we have two components executing their work as described in the figure 3.4, the scheduler will give the hand to the first component to execute its first instruction while the other component enter in wait of 5 ns. When the first component finishes its first instruction, the scheduler increase the simulation time to 5 ns, then it gives the hand to the second component to execute its first instruction till it finishes, then the scheduler increase the simulation time to 10 ns and let the first component do its second instruction. As we see, this was the global idea for SystemC to model parallelism between components. In fact SystemC do not allow for two components to execute their work at the same time, for this reason to model the parallelism, the one use the wait between components instructions to enable the explicit parallelism between components and to model the execution time.

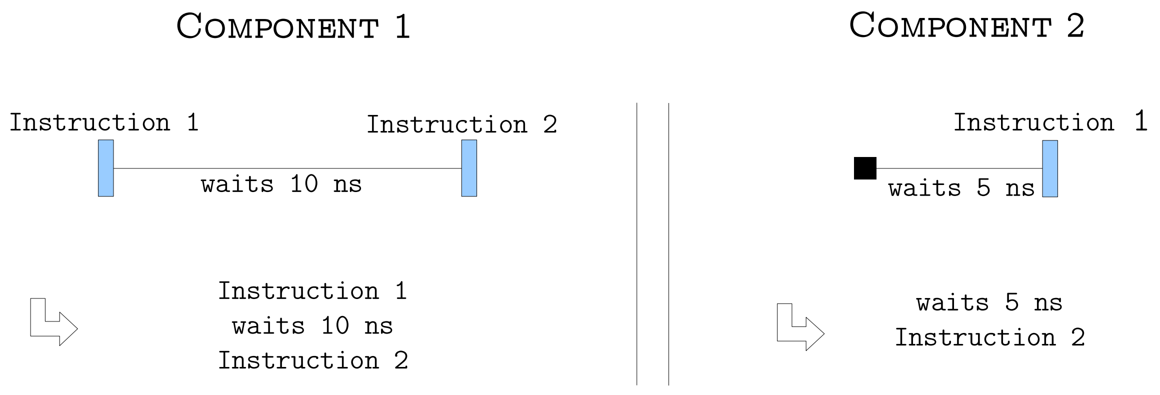
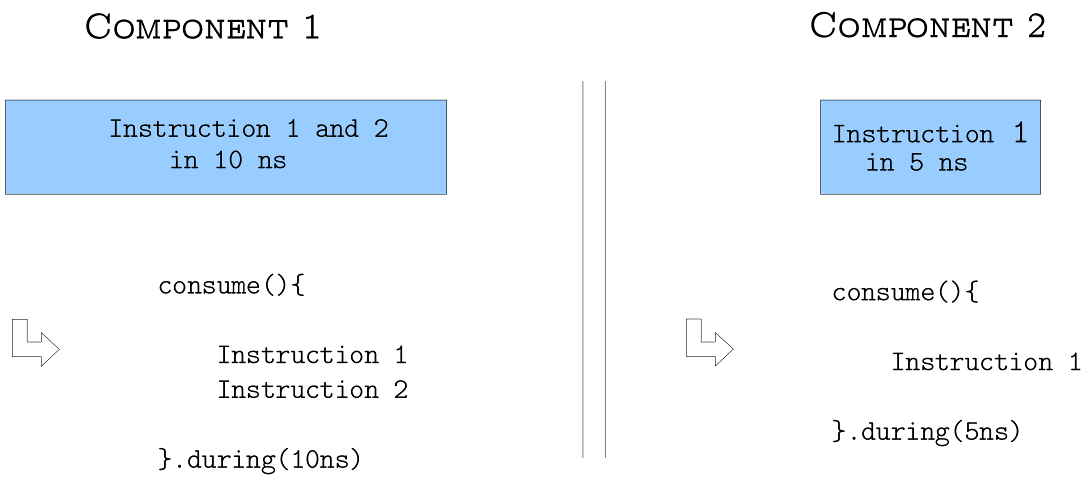


Figure 3.4: The time managment in SystemC

In JTLM, timed tasks is a way to model the simulation time, it allows the parallelism of components execution, For example, we consider two component that are executing their instructions using the duration as the figure 3.5 shows, the instructions of the two component are running in parallel. Even it is modeled to take 10 ns to achieve its job; the first component may finish before the second. In this case, it musts wait for the end of execution of the second component to continue the rest of its work.



```
consume(){

    Instruction 1
    Instruction 2

}.during(10ns)
```

```
consume(){

    Instruction 1

}.during(5ns)
```

Figure 3.5: Time management in JTLM

Since the build of the real chip takes a lot of time to be ready and the RTL platforms are slowly in simulation, the TL level was born firstly to satisfy the need of developers to an auxiliary platform to test the embedded software and also to provide a faster time of simulation comparing to RTL platforms. Even

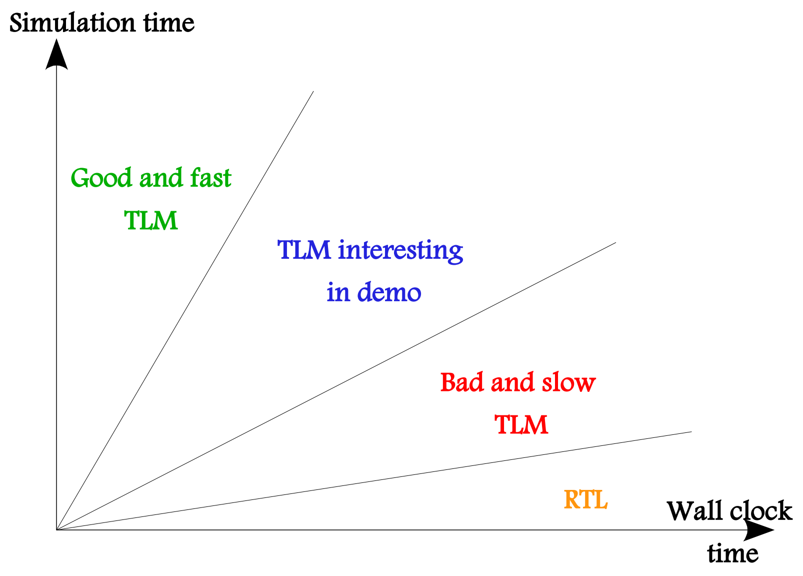


Figure 3.6: Transaction models types

its various advantages, Transactional models could no longer be good enough in term of the time that could take a platform simulation. As we can see in the figure 3.6, the one can find three types of TL models: The first are similar to the RTL simulation time, they are called bad TL models, the seconds provide a reasonable wall clock time when simulating a platform, they are good to make demo. The last ones are those which provide a fast simulation, they are called as good TL models. Researchers and engineers are working to reach this TL modeling type so that developers can test the embedded software in a short time so they could finish in the same time as or before the physical chip would be ready.

As a consequence of the use of timed tasks, many behaviors are eligible at the same time. This result will increase the speed of the simulation so the JTLM will represent a kind of the third type of TL models as explained above.

3.4 Parallelism

Parallelism between JTLM Behaviors let them communicate primarily by sharing access to fields and common resources. This mechanism is extremely efficient because it reduces the time of execution but in the other hand, it makes many kinds of errors. It can lead to the classic problems of parallelism such as deadlock, threads Interference and memory consistency errors and other problems related to the JTLM time manager like the livelock.

3.4.1 Deadlock

A "deadlock" condition can occur when there is a circular chain of tasks that lock on each other. Deadlock refers to a specific condition when two or more processes are each waiting for each other to release a resource, or more than two processes are waiting for resources in a circular chain. For example, thread C waiting on thread B, which is waiting on Behavior A, which is waiting on Behavior C is a deadlock condition. In such a case, no tasks will continue because none has received a signal from the one it is waiting for.

3.4.2 Livelock

A livelock is an infinite branch in the execution tree, in which, from a certain point there is no progress, i.e. no new state is visited. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked; they are simply too busy responding to each other to resume work. For example, consider two threads are sharing a variable x initially set to 0.

Suppose Thread A modifies the value of x before thread B begin its work, in this case it will lead to

Thread A	**Thread B**
x = 1	while(x!=0) { +//do nothing }
	print(x)

a livelock situation. No progress in the program, but thread B remains busy with the infinite periodic reading of the value of x.

3.4.3 Threads Interference

It happens when two operations, running in different threads, but acting on the same data, interleave. This means that the two operations consist of multiple steps, and the sequences of steps overlap.
For example, suppose that two threads are sharing a variable c as described in the following table: Suppose

Thread A	**Thread B**
Retrieve the current value of c	Retrieve the current value of c
Increment the retrieved value by 1	Decrement the retrieved value by 1
Store the incremented value back in c	Store the decremented value back in c

Thread A invokes `increment` at about the same time Thread B invokes `decrement`. If the initial value of c is 0, their interleaved actions might follow this sequence:

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

Thread A's result is lost, overwritten by Thread B. This particular interleaving is only one possibility. Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all.

3.4.4 Memory Consistency Errors

It occurs when different threads have inconsistent views of what should be the same data. For example two thread are sharing a variable x which is set to 0.

Thread A	**Thread B**
x = 1	print(x)

If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1". But if the two statements are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to x will be before the execution of the tread B. The key to avoiding memory consistency errors is understanding the happens-before relationship [MPA05]. This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement.

3.5 Synchronization

The tool needed to prevent multi-threading errors is the synchronization. Java has constructs for synchronization [Mic08] that, when used properly, can ensure that only one thread is accessing a particular object at any given point in the computation. The synchronization construct in Java is the lock.

3.5.1 The keyword synchronized

The Java keyword synchronized is used to manipulate locks. This mechanism provides the possibility to ensure atomic transaction so we can avoid memory consistency errors and threads interference.
To synchronize the behaviors communications, we have used block synchronization and method synchronization.

Block synchronization allows you to lock any object, anywhere in your code. The syntax for block synchronization is:

```
Object x = new Object();
void foo() {

       synchronized(x) { // acquire lock on x on entry

            // I hold the lock on object x in block

       } // release lock on x on exit
}
```

Method synchronization is exactly equivalent to block synchronization of the entire method. The syntax for method synchronization is to add the keyword synchronized at the beginning of a method declaration, such as:

```
synchronized void foo() {

    //Locked method:  critical section

}
```

This method is equivalent to the block synchronization of the entire method bellow:

```
void foo() {

         synchronized (this) {

             //Locked block of code:  critical section

         }
  }
```

3.5.2 Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread can acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables reentrant synchronization. Reentered synchronization is a solution to the deadlock situation.

3.5.3 The wait and notify

The Java language includes three important methods that effectively allow one thread to signal to another. This facility helps to resolve memory consistency errors and livelock errors. The following table gives a solution to the livelock problem highlighted above.

Thread A	**Thread B**
wait()	while(!x) { }
x = 1	notify()

- The **wait()** method of any Java object suspends the thread which call it. The thread is said to be "waiting on" the given object.
- The **notify()** or **notifyAll()** method of the same Java object, when called by another thread, will "wake up" the threads waiting on that object.

If the wait() and notify() methods are not used properly, they will lead to a deadlock, for example when the notification of a thread is done before it starts waiting for this notification.

Chapter 4

Contributions

Contents

4.1 Introduction to JTLM

JTLM is an extension of the Java language, designed to represent transactional models. It includes a Java class library containing hardware models and all building blocks to model a hardware system.

This library, as the figure 4.1 shows, is composed of **the core language package** which contains all the necessary elements to capture a physical hardware to a virtual one, **the utility package** regrouping generic components and **the test package** used to validate both of the two other packages and to verify other functionalities.

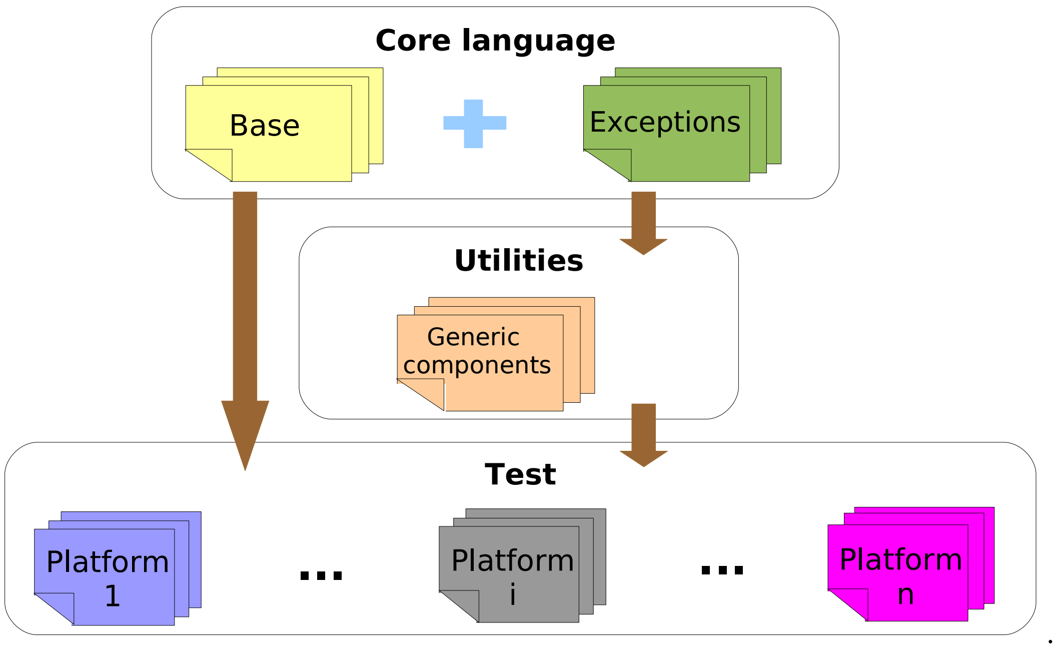


Figure 4.1: The JTLM library

JTLM used as a simulation platform to test the embedded software. JTLM is not created, in this step, as a professional language concurrent to SystemC but to prove that it is possible to implement TLM models using another language rather than SystemC.

4.2 Core language

Before we jump into the details of JTLM language, let's looks at how JTLM models hardware in brief.

A JTLM system consists of one or more components which are linked through a communication channel or by interrupt ports. Each component typically contains:

1. *Behaviors.*
2. *Ports.*
3. *Interrupt ports.*
4. *Methods.*

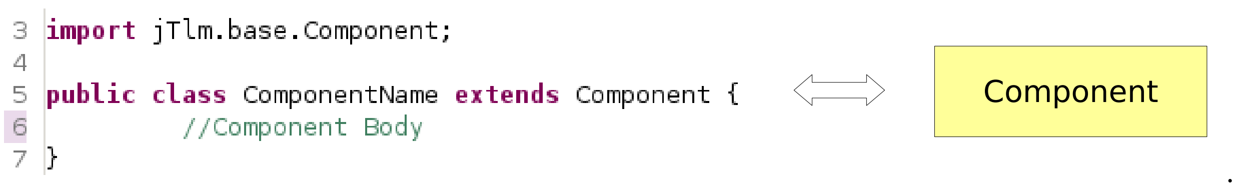
5. *Internal data.*

And Constructor.
In JTLM, components, ports, channels and behaviors are classes from which objects are created.

4.2.1 Components

Components are the basic building block within JTLM to partition a design. Components allow designers to hide internal data representation and algorithms from other components. They are similar to module in SystemC and Verilog and Entity in VHDL. By definition, components communicate with other components through channels and via ports. Typically a component will contain numerous concurrent behaviors used to implement their required behavior. Components are similar to module in SystemC.

In JTLM, any component has to be derived from the existing class **Component**. Below, the figure 4.2 is syntax of a component.



```
import jTlm.base.Component;

public class ComponentName extends Component {
        //Component Body
}
```

Figure 4.2: Component declaration

The source code is available at `jtlm/base/Component`.

4.2.2 Transaction Ports

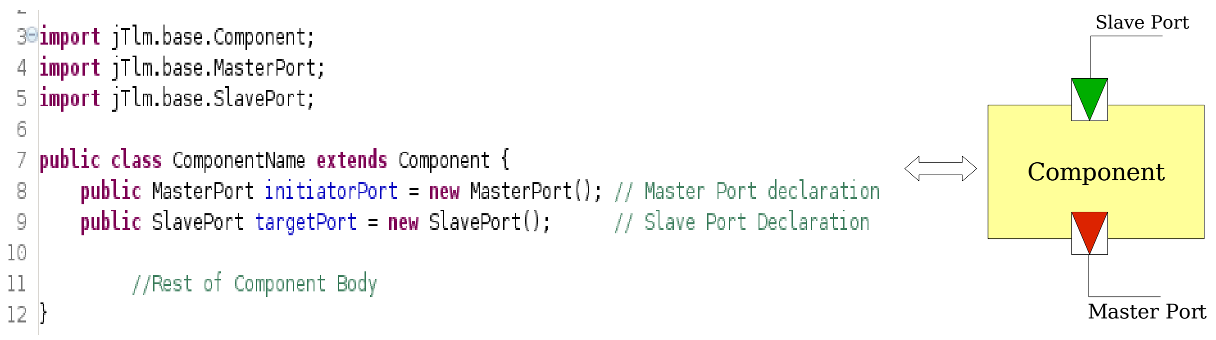
Components could not pass or receive data directly to or from another one; they use ports to communicate to each other through the communication channel which will decode addresses and route transactions. Ports are used by components as a gateway to and from the bus. In a simplistic way, one can consider a port like the pin of a hardware component.

JTLM has two types of ports: Slave and Master ports:

- ***Slave port:*** This port is found in a slave or master-slave component, it is used to receive or serve transactions.
  The source code is available at `jtlm/base/SlavePort`.

- ***Master port:*** This port is found in a master or master-slave component, it is used to send transaction to another component.
  The source code is available at `jtlm/base/MasterPort`.

The figure 4.3 shows the way to declare a port in JTLM.
The ports have two functions that will be used to communicate through them:



```
import jTlm.base.Component;
import jTlm.base.MasterPort;
import jTlm.base.SlavePort;

public class ComponentName extends Component {
    public MasterPort initiatorPort = new MasterPort(); // Master Port declaration
    public SlavePort targetPort = new SlavePort();      // Slave Port Declaration

        //Rest of Component Body
}
```

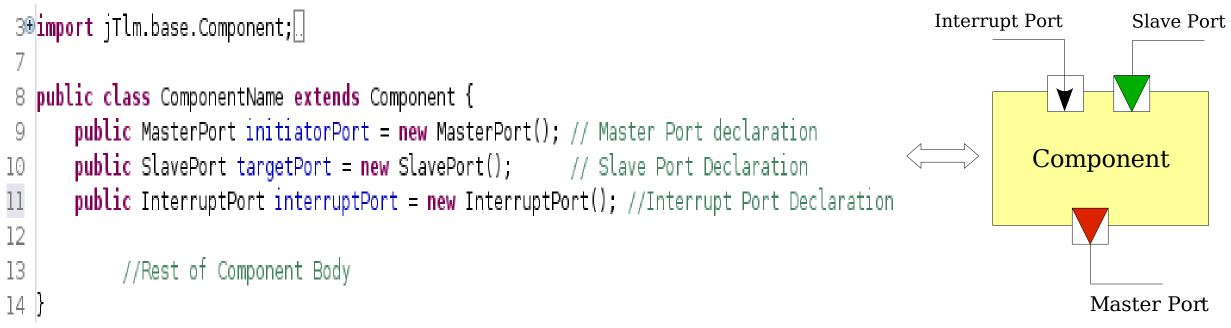
Figure 4.3: Master and Slave Port declaration

- `port_name.write(value, address)// For writing value to port.`
- `port_name.read(address) // For reading value from port.`

4.2.3 Interrupt port

Interrupts are asynchronous signals transmitted by wires indicating the need for attention. To model this idea, JTLM uses an interrupt Port. When a component sends an interrupt through its interrupt Port, it leads to the call of another component function in order to make it do a specific work like for example waking it up from sleeping or processing another job.

Unlike transaction ports, interrupt ports proceed directly to the target component without going through bus. Since interrupts do not pass by the communication channel, neither address decoding nor routing is needed to send interrupts from a component to another.
Bellow the figure 4.4 gives the syntax of an interrupt port declaration.
To wait for an interrupt, a behavior or a method should execute the instruction bellow:



```
import jTlm.base.Component;

public class ComponentName extends Component {
    public MasterPort initiatorPort = new MasterPort(); // Master Port declaration
    public SlavePort targetPort = new SlavePort();      // Slave Port Declaration
    public InterruptPort interruptPort = new InterruptPort(); //Interrupt Port Declaration

        //Rest of Component Body
}
```

Figure 4.4: Interrupt Port declaration

```
InterruptPort.waitInterrupt()
```

To send interrupt:

```
InteruptPort.sendInterrupt()
```

The source code is available at `jtlm/base/InterruptPort`.

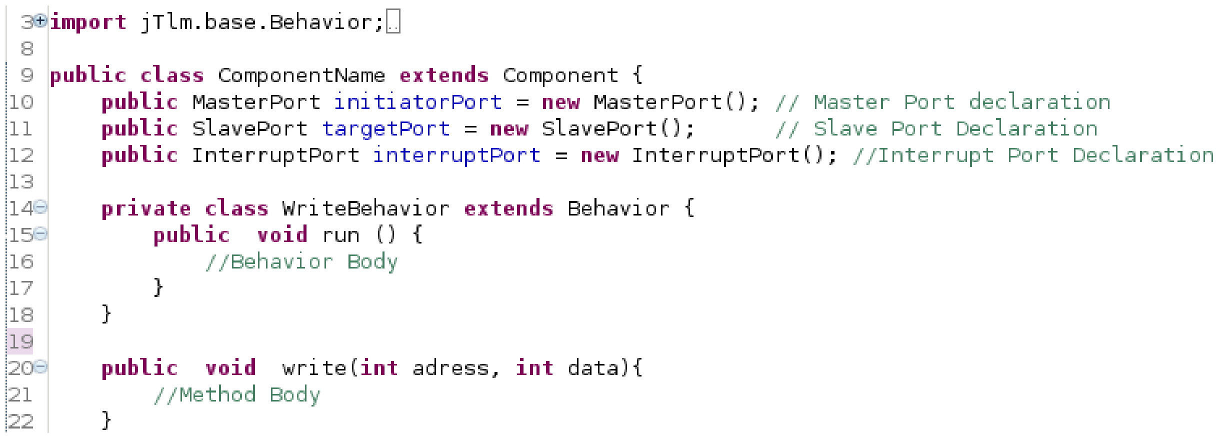
4.2.4 Behaviors and Methods

In TLM the functionality part of the component is known by processes. There are two types of processes in JTLM: Behaviors and Methods:

Behavior:

Behaviors are second kind of process, which when instantiated keeps executing or waiting for some events to occur. Conditions that enable these Behaviors can be sendInterrupt, wakeUpFromEvent or the time has elapsed. The equivalent of the behaviors in SystemC are called sc_thread

To define a component behavior, it should inherit from the class Behavior so it could be considered as a behavior. The figure 4.5 shows the declaration syntax of a method and a behavior in JTLM



```java
import jTlm.base.Behavior;

public class ComponentName extends Component {
    public MasterPort initiatorPort = new MasterPort(); // Master Port declaration
    public SlavePort targetPort = new SlavePort();      // Slave Port Declaration
    public InterruptPort interruptPort = new InterruptPort(); //Interrupt Port Declaration

    private class WriteBehavior extends Behavior {
        public  void run () {
            //Behavior Body
        }
    }

    public  void  write(int adress, int data){
        //Method Body
    }
```

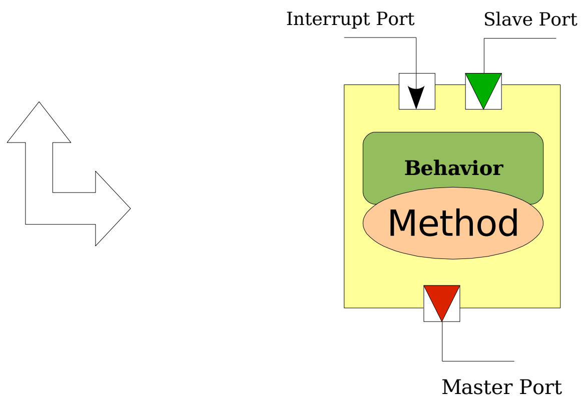


Figure 4.5: Method and Behavior declaration

Behaviors can be suspended and reactivated. The behavior can contain wait() functions, which could be a wait for event, time or interrupt, that suspend its execution. An event, which could be event time

elapse or interrupt capture, will reactivate the behavior process from the statement the behavior was last suspended. The behavior will continue to execute until the next wait().
The source code is available at `jtlm/base/Behavior`.

Methods:

Methods behaves like a function, when called it gets started and executes and returns execution back to calling mechanism. A method is called whenever an interrupt is received, when a slave receives a transaction or when a behavior calls it. JTLM methods are analogous to SystemC sc_method.

4.2.5 Constructor

The component constructor creates and initializes an instance of a component. The constructor creates the internal data structures that are used for the component and initializes these data structures to known values. The constructor is used to instantiate behaviors defined in that component and to link the slave port if it exists to the component.

4.2.6 Debugging and tracing support

JTLM provides two mechanisms to aid the programmer in diagnosing and correcting the platform errors. One is the **Debug** class and the other is the **Trace** class. These two classes contain functions used to show internal data of components, and to check outputs results. They have the same purpose but in a different context.

The major difference between debug and trace message is that debug message is used to display outputs results and tracing messages are used for tracking the internal functioning of the platform.

JTLM Debugging support

The JTLM debugger is used to test outputs of a platform. It is used to find out bugs. JTLM offer text based debug. To use debug, a programmer choose the proper place and insert his debugging message as bellow:

```
Debug.debugMessage("debugging message");
```

The source code is available at `jtlm/base/Debug`.

JTLM tracing support

The JTLM Trace class allows the programmer to identify and understand the problem in a platform. It provides informative messages from the running JTLM platform, such as viewing variable values and recording informations about program execution that can help diagnose problems or finding bugs.

To use tracing, the programmer should determine where he wants the tracing output to appear in the code and add the appropriate trace message as bellow:

```
Debug.traceMessage("tracing message");
```

The source code is available at `jtlm/base/Debug`.

4.2.7 Timed tasks in JTLM

In SystemC, the waits included into components processes are used for two issues:

1. Time elapsing: The one adds a wait statement in a component in order to make it wait a period of time.

2. Tasks duration: Because SystemC do not model task duration, the programmer adds a wait statement to represent the time that may take a task execution.

When a component is executing a task in the real chip, it will take a time to do it. Although SystemC does not support this feature, we had included timed tasks to JTLM in order to better implement the programmers intend and to be more faithful to the physical hardware.
As a consequence of using this feature, it allows to better parallelize between processes i.e. many tasks are eligible at the same time unlike SystemC, because of its cooperative scheduler, can give the hand to only one component to run an instruction.

The figure 4.6 shows the parallelism of components execution. The instruction f() of the cpu1 and the instruction of the cpu 2 are running in the same time. The same thing for the instruction k() of the cpu 1 and the instruction h() of the cpu 3.

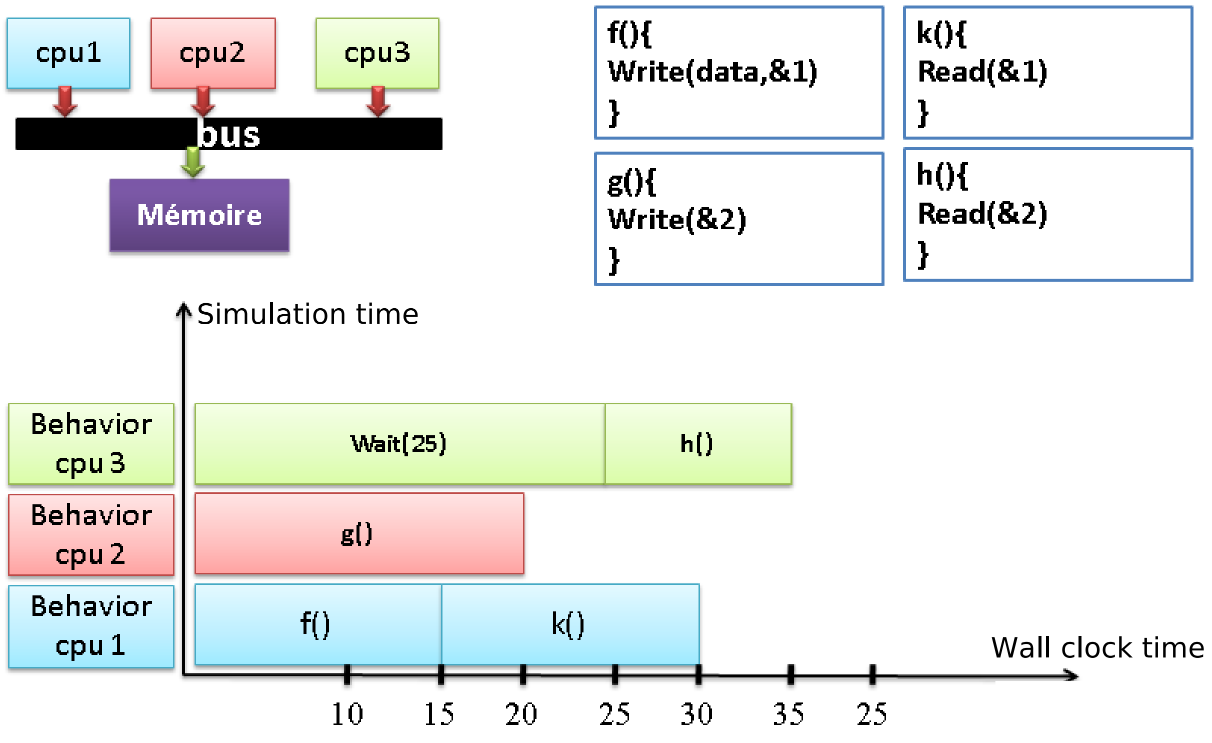
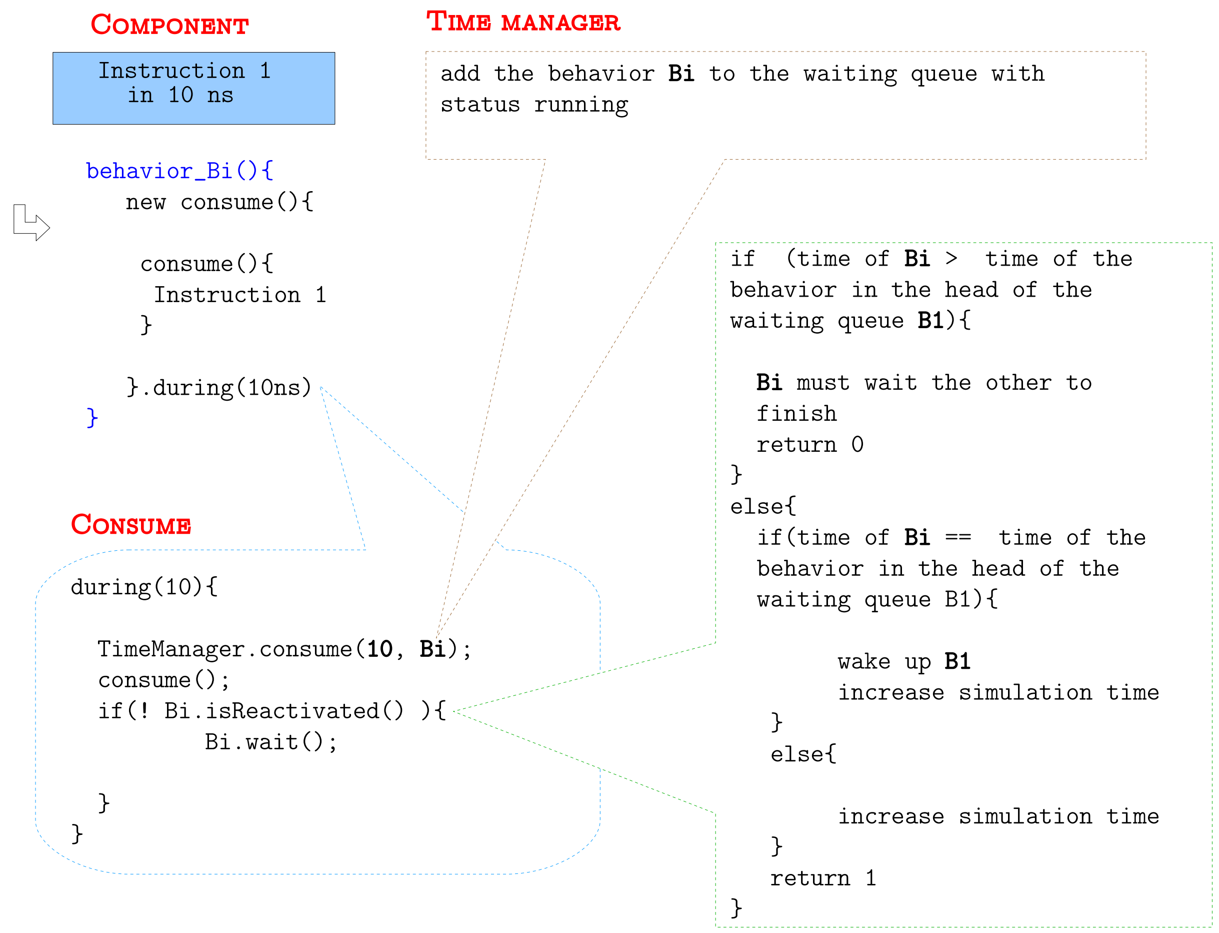


Figure 4.6: Example of a platform using timed tasks

We have incorporated a new class in JTLM that manage the time of a task execution. It will allow to a behavior component to define its job with specification of its duration. A behavior, which has finished

its timed task, is obliged to wait for behaviors doing their timed tasks with a smaller time. The figure 4.7 gives an example of using timed tasks.
We call a timed task a consume process. The source code of the time manager is available at `jtlm/base/`



COMPONENT
Instruction 1 in 10 ns
TIME MANAGER
add the behavior Bi to the waiting queue with status running
CONSUME

```
behavior_Bi(){
   new consume(){

    consume(){
     Instruction 1
    }

   }.during(10ns)
}
```

```
during(10){

  TimeManager.consume(10, Bi);
  consume();
  if(! Bi.isReactivated() ){
          Bi.wait();

  }
}
```

```
if  (time of Bi >  time of the
behavior in the head of the
waiting queue B1){

  Bi must wait the other to
  finish
  return 0
}
else{
  if(time of Bi ==  time of the
  behavior in the head of the
  waiting queue B1){

        wake up B1
        increase simulation time
   }
   else{

        increase simulation time
   }
   return 1
}
```

Figure 4.7: The functionning of the timed tasks

`Consume.`

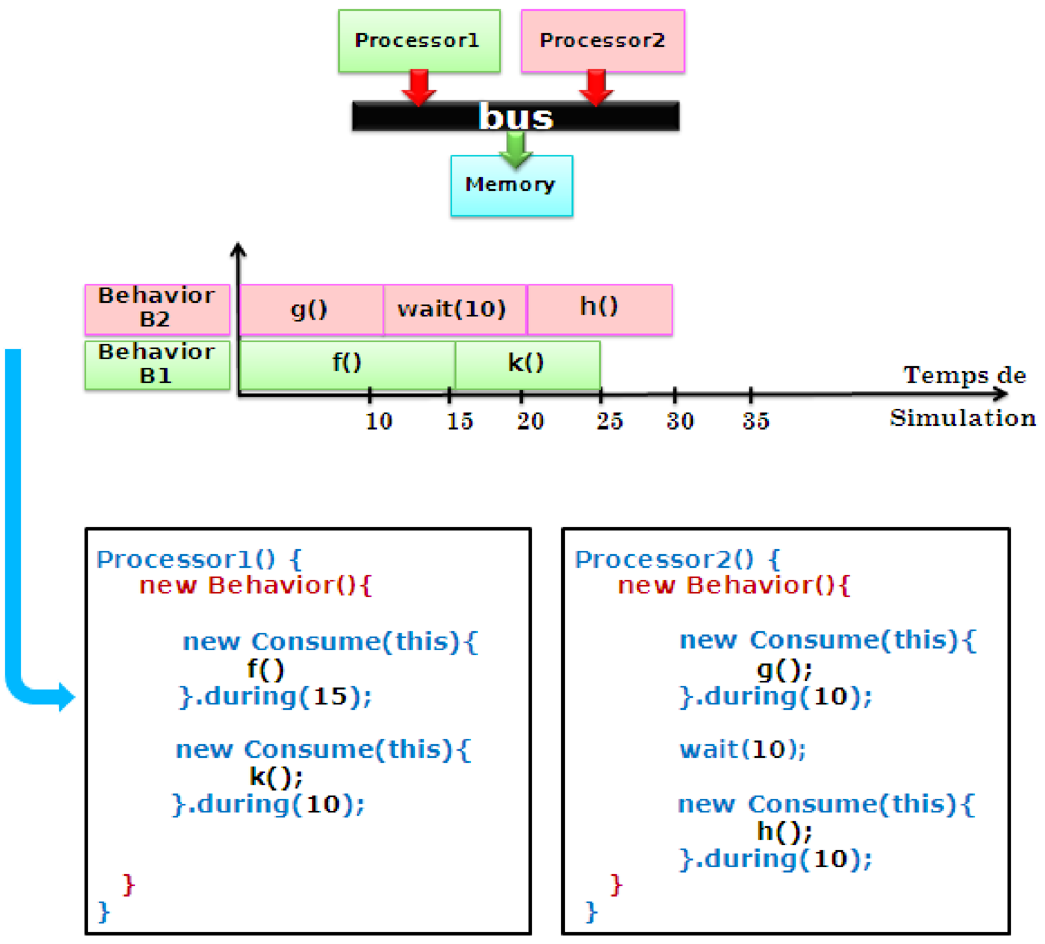
4.2.8 Time Manager

In a first stage, JTLM component could just wait for event or interrupt without neither waiting in time nor giving the time of the system execution. In That stage JTLM could only perform system progress without including the simulation time, see section 1.3.

To include the simulation time, we had implemented a time manager that manages the time between components, gives the simulation time of a platform and provides to a component behavior to wait for time elapsing or to execute a timed task.

In SystemC, the scheduler is the responsible for the progress of the simulation time; it is also responsible for giving the control to the components processes. But in JTLM, The **Java scheduler** is responsible for the preemption between behaviors. It allows the computer system to more reliably guarantee each behavior a regular "slice" of operating time in the processor. And the **JTLM time manager** is responsible for the increase of the simulation time, also it decides which behavior is eligible. The time manager increase the time only when the number of the behaviors executing the waiting process and the consume process is equal to the global number of the platform behaviors.

The time manager waits till all behaviors are waiting or consuming a task. In the other hand, each of the components behaviors that enter in a wait statement (interrupt, event or time) or in executing a timed task will be stored in the time manager queue. When all the behaviors are all stored in that queue, the time manager determines the behavior with the smallest time. If this behavior is consuming a process, the time manager will wait for the end of the execution of this behavior task, and then it will increase the time with the duration of this task. But if this behavior is waiting for time elapsing, the time manager will increase the simulation time with this behavior waiting time and then it will awake it to achieve the rest of its work.



```
Processor1() {
   new Behavior(){

      new Consume(this){
         f()
      }.during(15);

      new Consume(this){
         k();
      }.during(10);


   }
}
```

```
Processor2() {
   new Behavior(){

      new Consume(this){
         g();
      }.during(10);

      wait(10);

      new Consume(this){
         h();
      }.during(10);
   }
}
```

Figure 4.8: Example of a platform

The figure 4.8 gives an example of a platform composed by two processors, a memory and a bus and gives its functioning written with JTLM.
Bellow, as the figure 4.9 shows, we give the time manager functioning in the example mentioned above: Firstly, the two processors have each one its own behavior. The global number of the platform behaviors is two. The processors behaviors begin executing their tasks. The time manager determines the smallest task; in this case it's the task g of the processor-2 behavior. So the time manager will wait for the end of this task, then it will increase the simulation time to 10 ns.

If the first processor would finish its task before the second processor, the time manager will block it till the second processor will finish its task because the duration of its task is smaller than the first.

When the time is increased to 10 ns, the processor-2 continues its work by executing the second task which consists of waiting 10 ns. The time manager will block the behavior of the processor-2 till the simulation time reaches 20 ns.

The time manager, determine the smallest time to wait or to be consumed in executing a task. In our example, the smallest time is 5 ns. So when the first task of the processor-1 is done, the time manager will increase the simulation time to 15 ns.

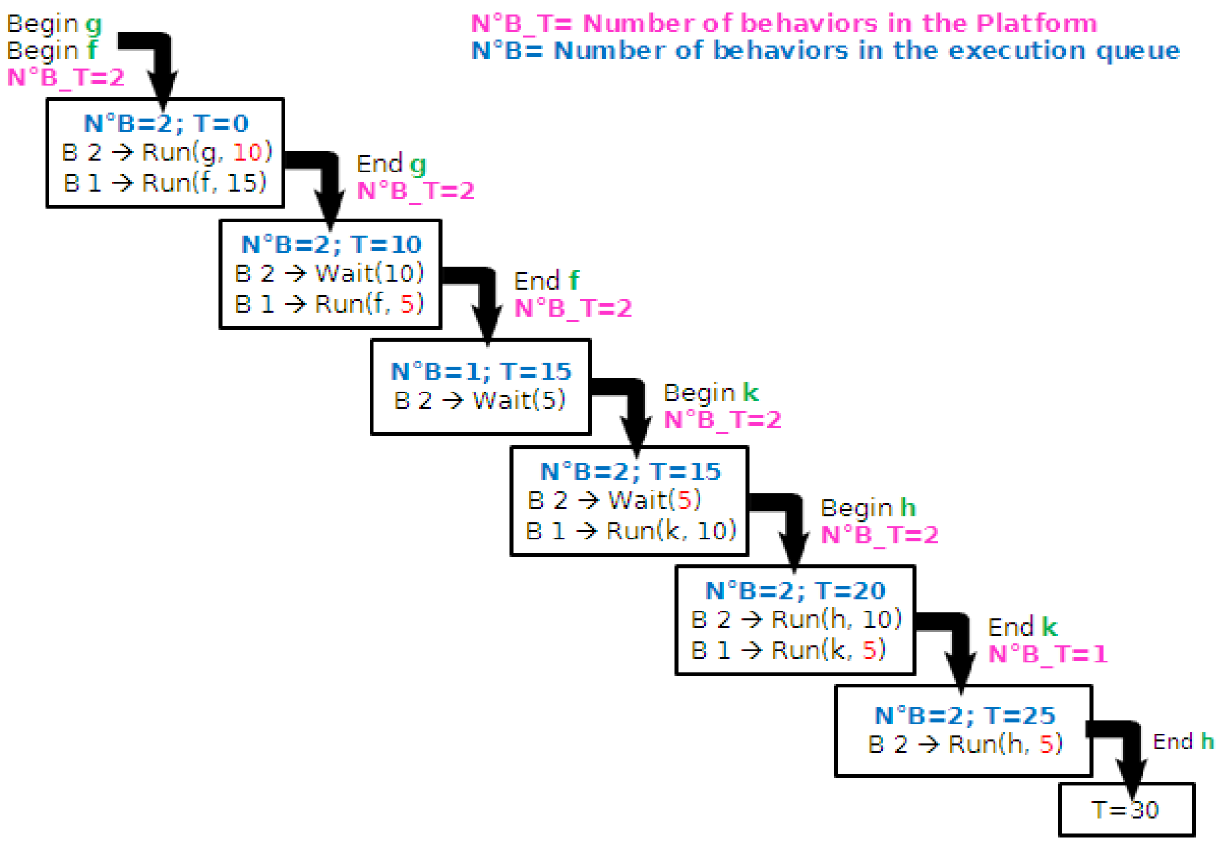


Figure 4.9: The Time manager algorithm

At this step the number of the behaviors which are consuming or waiting is smaller than the global number of behaviors so the time manager does nothing.

The processor-1 begins its second task which will take 10 ns. As usual, the time manager determines the smallest time, in this case, it finds 5 ns as a time left in the waiting process of the processor-2 behaviors, so it will increase the simulation time to 20 and will unlock the processor-2 behavior to continue its tasks.

The processor-2 executes its final task in 10 ns and the time left to finish the second task of the processor-1 is 5ns. So the time manager will wait for the end of the second task of the processor-1. At that time, the time manager will increase the simulation time to 25 ns and will kill the behavior of the processor-1. In this case the number of the global behaviors becomes one.

At the end of the task 3 of the processor-2, the time manager increase the simulation time to 30 ns and kills the processor-2 behavior.

4.2.9 Simulation

The main task of the simulation class is starting all the behaviors of the platform components and instantiating the time manager. After invoking the start function of this class in the main function, the time manager initializes the simulation time and the behaviors begin their work.
The source code is available at `jtlm/base/Simulation`.

4.3 Exception Handling

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. It is an object that contains information about the exception, including its type and the state of the program when the error occurred

Many kinds of errors can cause exceptions such as trying to access an out of bounds array element. When such an error occurs within a method, the method creates an exception object and hands it off to the runtime system.

After a method throws an exception, the runtime system searches until it finds a method that contains an appropriate exception handler. An exception handler is considered appropriate if the type of the exception thrown is the same as the type of exception handled by the handler. The exception handler chosen is said to catch the exception. If the runtime system do not find an appropriate exception handler, the runtime system (and consequently the program) terminates.

JTLM took advantage of the ability of the Java exception handling so that it can itself define and handle its own exceptions. It lets also programmers define their own JTLM exception.

By using exceptions to manage errors, JTLM have the advantages over traditional error management techniques such as grouping and differentiating error types.
JTLM defines three types of exceptions:

1. **AddressOutOfRangeException:** This exception is thrown when a component try to access a component with an invalid address.

2. **NegativeRangeSizeException:** This Exception is thrown when the programmer defines a negative or null address range.

3. **JtlmException:** It is defined for the other JTLM errors.

The source code is available at `jtlm/exception`

4.4 Utilities and generic components

This section presents the generic components which can be parameterized by the developer, all these components are described in more detail in appendix A.

1. **Router:** A router, for more details see this section A.1, or bus is used to route the transactions between components using the target address.
2. **Memory:** A Memory, for more details see this section A.2, used for data storage.
3. **LCDC:** The LCD component, for more details see this section A.3, is used to display videos or data from memory.
4. **ITC (Interrupt Controller):** Some components want to send interrupts but they do not have interrupt outputs, so the ITC, for more details see this section A.4, was created to let these components send interruption by configuring it.
5. **DMA Controller:** The DMA component, for more details see this section A.5, is used to perform data transfers between two memory regions, between a memory and a peripheral, or between two peripherals. It allows data transfers of fixed or variable length without intervention from the CPU.
6. **Timer:** The Timer, for more details see this section A.6, is a slave component which is included in the JTLM development kit as a library component. Timer is used to send interrupt each period of time to another component.

4.5 Instantiation

In the main() function the structural elements of the system are created and connected, the figure 4.10 gives an example of instantiating a platform through JTLM

The first part of the main function consists of the instantiation of the component and the channel:

```
Memory memory = new Memory(100);
```

Then it connects the components by attaching channel ports to the appropriate ports:

```
cpu.initiatorPort.bind(bus.getTargetPort(0));
```

After that it defines to each slave or master-slave component its own address range:

```
bus.map(0, 100, memory.targetPort); // binding the memory slave port
to the bus master port and defining 100 as address map for the memory
```

And it assigns the inputs of interrupt ports to the outputs of the suitable interrupt ports:

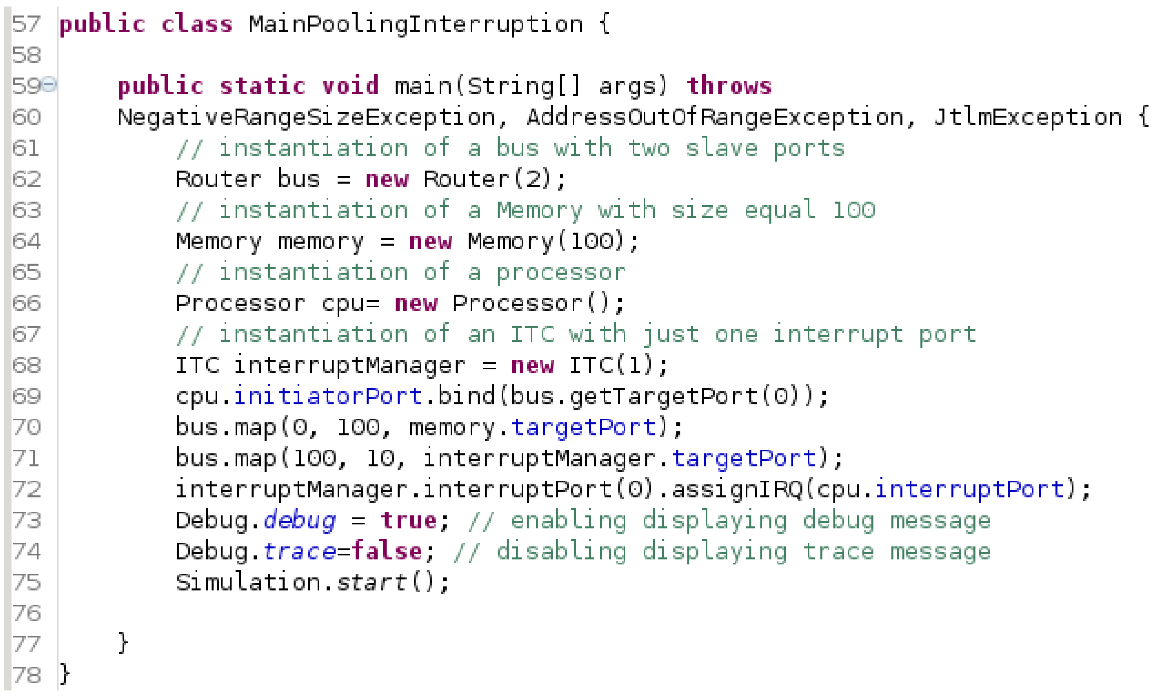
```
interruptPort.assignIRQ(InterruptPort irq)
```

In the final portion of the function, debugging and tracing messages are enabled or disabled:

```
Debug.trace=true;
Debug.debug = true ;
```

The simulation is initiated through the function start() in the end of function main. The time manager will take care of time and manage the components work:

```
Simulation.start();
```



```java
public class MainPoolingInterruption {

    public static void main(String[] args) throws
    NegativeRangeSizeException, AddressOutOfRangeException, JtlmException {
        // instantiation of a bus with two slave ports
        Router bus = new Router(2);
        // instantiation of a Memory with size equal 100
        Memory memory = new Memory(100);
        // instantiation of a processor
        Processor cpu= new Processor();
        // instantiation of an ITC with just one interrupt port
        ITC interruptManager = new ITC(1);
        cpu.initiatorPort.bind(bus.getTargetPort(0));
        bus.map(0, 100, memory.targetPort);
        bus.map(100, 10, interruptManager.targetPort);
        interruptManager.interruptPort(0).assignIRQ(cpu.interruptPort);
        Debug.debug = true; // enabling displaying debug message
        Debug.trace=false; // disabling displaying trace message
        Simulation.start();

    }
}
```

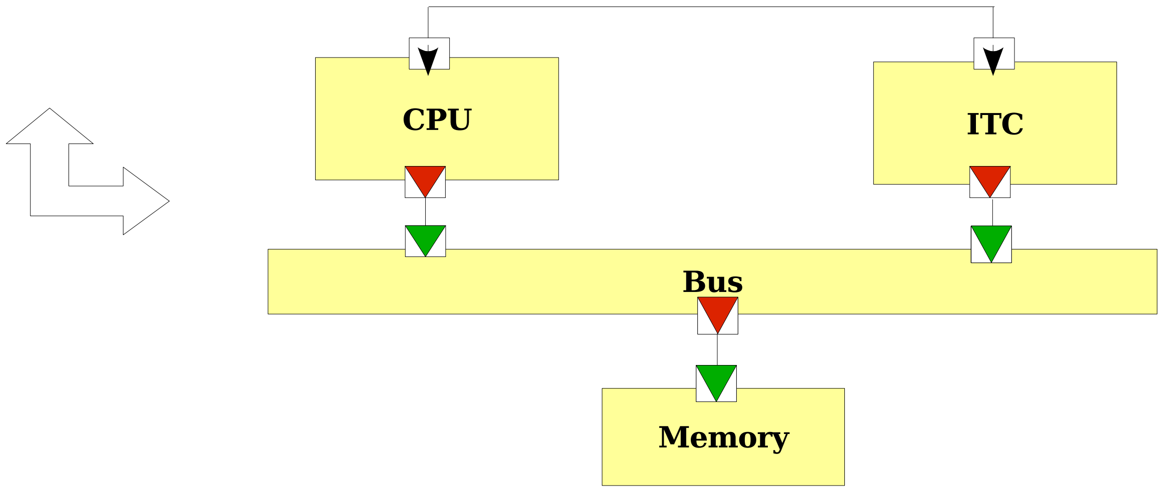


Figure 4.10: Instantiation of a JTLM platform

4.6 Evaluation

We give, in this section, the examples used to evaluate and validate JTLM bases and its utilities. We implemented a platform that contains a generic number of each example's platform to view the behavior of many platforms working together. Also we made a make file, see the appendix B, so we could run each example many times and compare it with a reference which contain true results.
We will, for each example, specify:

1. how it works?

2. what it aims to?

3. help it to conclude some results?

4.6.1 Components test

In order to validate the functionality of the Component Class, we made several examples which helped us to find bugs and correct errors of implementing the Component class.

4.6.2 Generic components test

1. **interrupt Controller ITC test:**
   The interrupt controller was tested in the synchronization test at this section 4.6.3.

2. **DMA test:**
   A full description is available in the source code of the example at: `jtlm/test/dma`.

3. **LCDC test:**
   Two examples are done in order to validate this component:

   - Integration of the time and event: This example consists of a processor, an LCD, a memory and a bus. The LCD begins its work when the processor writes in its start register.
     A full description is available in the source code of the example at:
     `jtlm/test/time-event`.
   - Integration of the time and event in a generic way: This example is a generic number of platforms of the first example.
     A full description is available in the source code of the example at:
     `jtlm/test/time-event-generic`.

4. **Timer test:**
   We tested the functionality of the timer in a platform that contain two processors, memory and a bus.
   A full description is available in the source code of the example at: `jtlm/test/timer`.

4.6.3 Synchronization test

To wait for a device interrupt, a component may poll or wait an interrupt from it. We implemented examples that uses interrupt and polling between components in different ways:

1. **Polling test:**
   Polling a device consists usually of reading its status register periodically until the device's status changes to indicate that it has completed the request. We have implemented an example that shows the polling mechanism, it consists of a platform that has: two processors, Memory and the bus which will route the transaction between them. Each of the two processors uses the polling of the memory to get the right to access it for reading or writing data.
   A full description is available in the source code of the example at:
   `jtlm/test/polling-polling`.

2. **Interrupts test:**
The example written has: two processors, an interrupt manager, memory and the bus. Each of the two processors, to synchronize the memory access, waits for interrupts from each other instead of using polling.

   Since a processor could not have an interrupt input, we have included the interrupt manager to give the two processors the ability to send interrupts through it.
A full description is available in the source code of the example at:
`jtlm/test/interrupt-interrupt`.

3. **Polling-interrupt combination test:**
The platform designed for this example is composed of two processors, an interrupt manager, a memory and a bus. To get the right to access the memory, one processor poll it while the other wait for an interruption.
A full description is available in the source code of the example at:
`jtlm/test/polling-interrupt`.

When a processor is polling a device, it does a busy waiting i.e. it does no tasks except polling periodically another peripheral. This synchronization type is costly both in the real processor consuming and the time of simulation of the virtual platform. In the other hand, in the interrupt waiting mechanism, a processor which waits for an interrupt does nothing but waiting. So, in the real platform, it does not consume energy like using polling. But for this type of synchronization we have to add the interrupt manager in our platform, it does mean to enlarge the chip surface. Two choices: a reduced chip with high consuming energy for the processor or a bigger chip with a better performance.

4.6.4 Timed tasks test

To verify the proper functioning of timed tasks, we have implemented these examples:

1. Timed tasks.
2. Timed tasks with the waiting of time.
3. Timed tasks timed with the waiting of time and events.
4. Timed tasks timed with the waiting of time, events and interrupts. This example is described in the figure. 4.11

All theses examples helped us to validate the **Consume Class** and being careful in management of the duration of tasks to guarantee a good synchronization between components.
A full description of all these examples is available in the source code of the package test at: `jtlm/test/`.

4.6.5 Time manager test

To validate the functionality of the JTLM time manager, we developed various examples:

1. Integration of the time and interrupt.
2. Integration of the time and interrupt in a generic way.

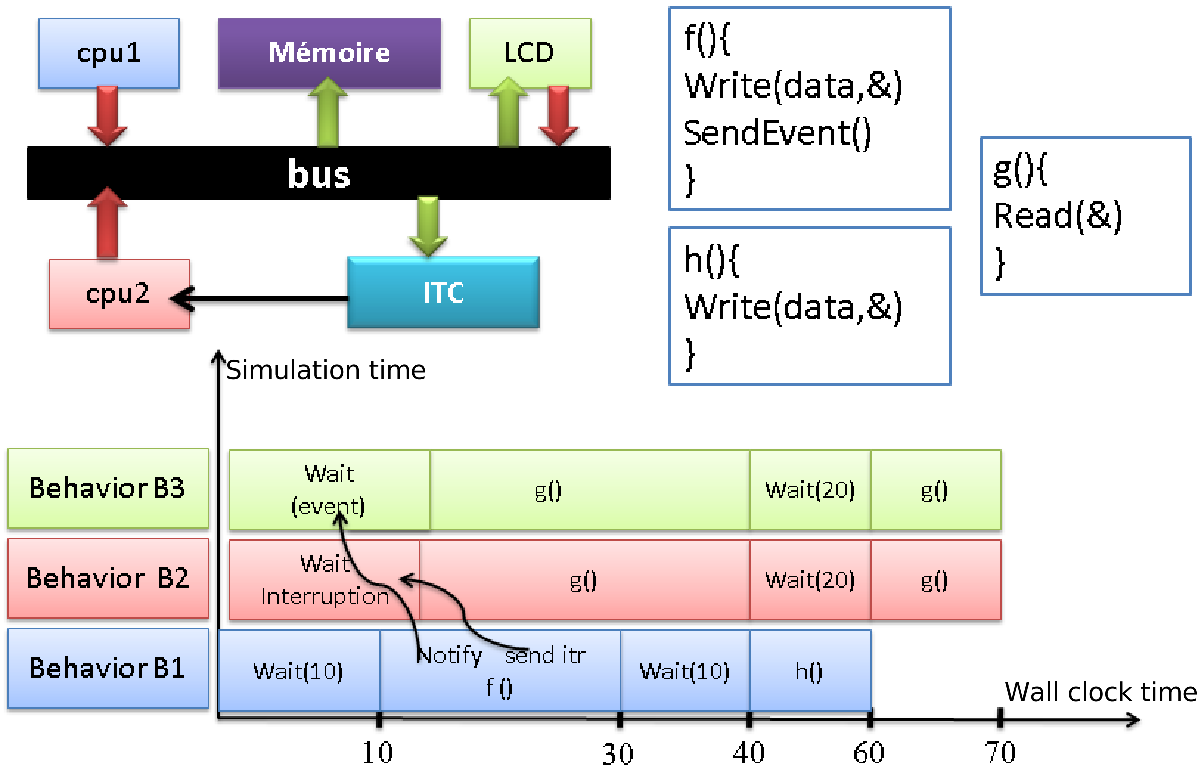


Figure 4.11: Example of a JTLM platform

3. Integration of the time, event and interrupt.
4. Integration of the time and interrupt in a generic way.
5. Integration of the time and polling.
6. Integration of the time and polling rectified.
7. Integration of the time and polling in a generic way.

We quote here the most important examples which have helped us to find bugs, discover other problems that we have not realized before, and provide solutions as remedies to these issues.

- Integration of the time, event and interrupt: In this example we have two processors, an LCD, an interrupt controller, a memory and a router. This platform, uses for synchronization, the wait for events, time elapsing and interrupts.
  This example has helped to check if the simulation time is correct or not and if transactions are respecting the scheduling specified.
  A full description is available in the source code of the example at:
  `jtlm/test/time-interrupt-event`.

- Integration of the time and polling: This example contains two processors, a memory and a bus. To synchronize accessing memory, one processor waits for time elapsing the other polls the memory as bellow:

```
while(initiatorPort.read(POOLING_MEMORY_ADDRESS)==SET_TO_ZERO){
              // do nothing, just poll the memory

}
```

This example is done to verify if the polling problem in SystemC is due to its cooperative scheduler or not. In fact this platform could not work properly in SystemC since processes are working in a cooperative way so for a process which enters to this kind of loop could not give the hand to another process to execute something else so the platform remains in this situation. Because of this reason, researchers used to say that this problem dues to the cooperative scheduler of SystemC. In JTLM, since the time manager could not liberate a behavior till the number of blocked behavior will be equal to the global number of all components behaviors and in the other side, a behavior which is in a loop is not blocked, this platform could not work neither in JTLM when the polling behavior begins before the behavior which will modify the value of the polling. But if the polling is done after the modification of the polling value by another behavior, it will work well without problems of livelock. A full description is available in the source code of the example at `jtlm/test/time-polling`.

- As a solution to the problem described above, in the example integration of the time and polling rectified, we suggest to insert in the polling loop a wait for small time:

```
while(initiatorPort.read(POOLING_MEMORY_ADDRESS)==SET_TO_ZERO){
        waitTime(1);
       // waits for one ns before polling another time
}
```

  So in this case, the behavior which is executing the polling could enter to the waiting queue and let the time manager do his work properly.
  A full description is available in the source code of the example at
  `jtlm/test/time-polling-rectified`.

4.7 Implementation choices

To design our execution model, we rely on many Java classes which satisfy our needs and have the less complexity:

1. We used the **`Java.lang.Thread`** to implement concurrent components behaviors.
2. We used the keyword **`Synchronized`** to guarantee the synchronization in accessing shared resources.
3. We used the Java inheritance to allow the reuse of some JTLM classes.

4. In order to not let the programmer modify functions or parameters definition when inheriting from the JTLM classes, we used the keyword **`final`**.

4.8 JTLM limits

Despite it's capabilities to present TL models, JTLM has lacks in several points. Unlike SystemC, JTLM library, right now, does not give the possibility to represent the hardware data types such as the Bit Type, Logic Type, Unsigned Integer Type and Signal Type because this was not our goal that we are concerned with.

Also it does not support the coarse granularity of transaction as well as SystemC. For example in JTLM a programmer could not make a component writing an image in just one transaction, but in several transactions, so the programmer should be careful and make such kind of transactions in an atomic process to avoid errors. It is not easy to guarantee good synchronization between component when a programmer use the JTLM as a language to write TLM models comparing to the SystemC library, this due to the preemptive scheduler of the Java virtual machine and to the physical parallelism.

The polling problem found in SystemC also exist in JTLM; when programmers include an embedded software, which contain polling, into a JTLM platform this software could not work properly till developers add waiting of time into the polling process.

Finnaly, JTLM does not provide a verification support like constrained and weighted randomization and other verification tasks.

Chapter 5

Case study

Contents

5.1 Introduction

In this chapter we will give a detailed example done in JTLM: its representation, a brief implementation code and the simulation outputs. This case study is shown and described deeply in order to demonstrate with a proof that transactional models could be implemented with another execution model rather than SystemC.

5.2 Case representation

As the figure 5.1 shows, our platform contains these components:

1. A processor CPU 1 owns a master port and an interrupt input port. It performs these tasks:
    - f(): Writing an image into the Memory in 20 ns.
    - g(): Sending interrupt and starts LCD work in 10 ns.
    - Waiting for an interrupt from the DMA.
    - h(): Reading the image written by CPU 2 10 ns. As a result the image should be the same as the one that it has written it in f().
    - l(): Modifying the initial image in 15 ns.
2. An LCD with master and slave port, it realizes these tasks:

- Waiting event to start reading image.
- p(): Reading an image from the memory in 10 ns.

3. An interrupt Controller which will manage interrupts sent by the two processors. It has only one interrupt output port.
4. A Timer used to send interrupts after a period to the CPU 2. This component has a slave and an interrupt output port.

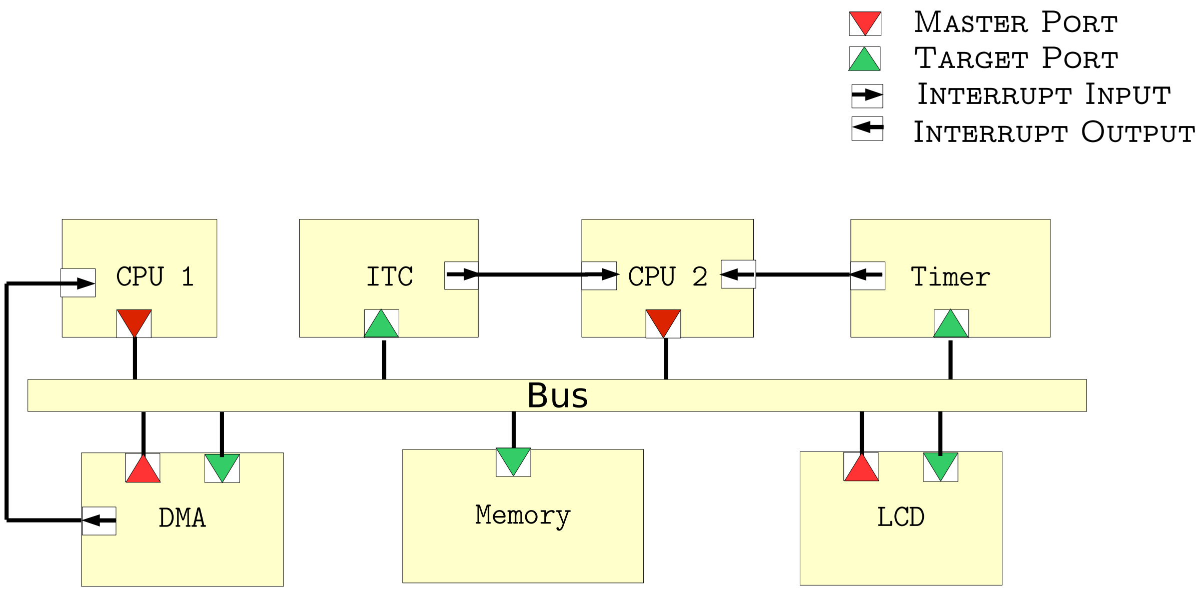


Figure 5.1: JTLM platform

5. A second processor CPU 2 has a master port and an interrupt input port. It executes these tasks:
   - Waiting interrupt.
   - Waiting 20 ns.
   - k(): configuring the DMA to copy an image,
     sending an interrupt to CPU 1,
     configuring the Timer so it will wake it up after 70 ns.
     This task takes 30 ns.
   - Waiting an interrupt from the Timer.
   - m(): Stopping the timer and read the modified image in 15 ns.

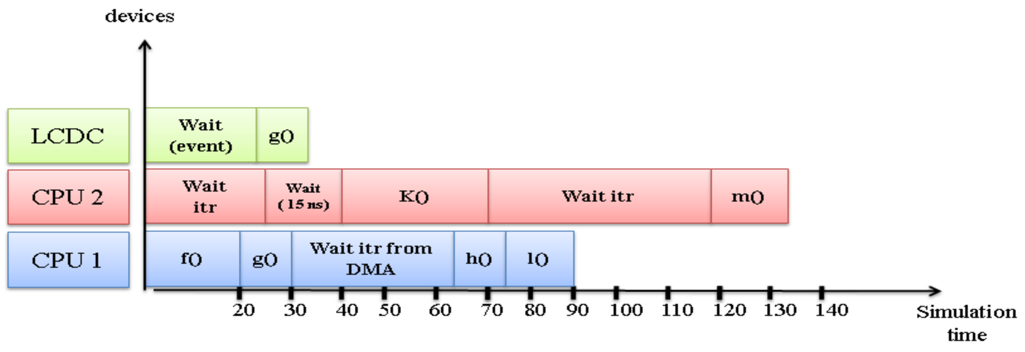


Figure 5.2: Diagram of the chronology of tasks execution

6. A DMA used to read and copy the images without using the processors.
7. A Memory for the data storage, it is accessible through its slave port.

And finally a Router which will route transactions between the components.
A full description is available in the source code at `jtlm/test/caseStudy`.
The figure 5.2 describes the chronology of tasks execution.

5.3 JTLM Implementation

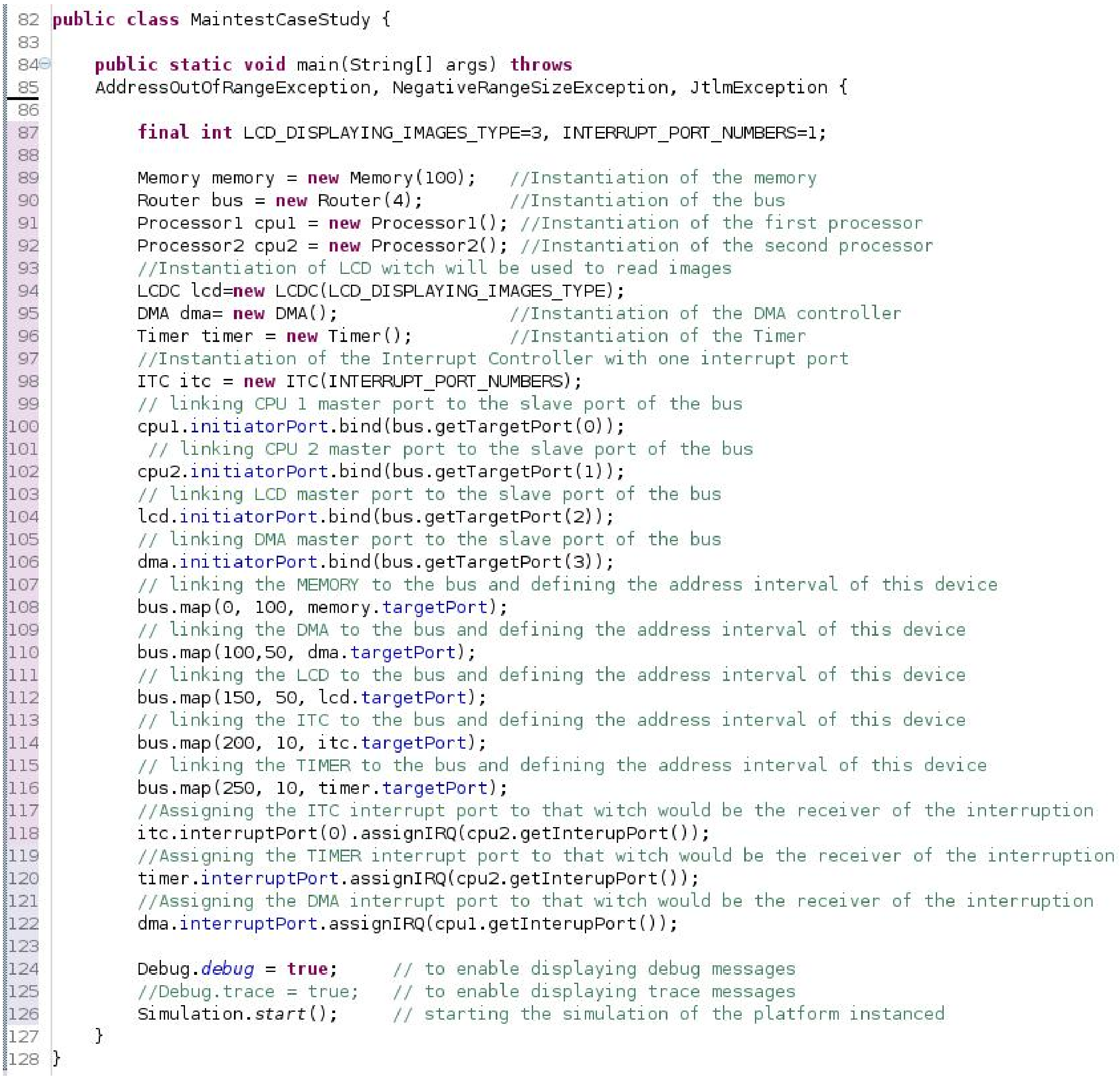
After creating all the components, we instantiate each of them and link each of their ports to the appropriate port in the main class which will start the simulation. The figure 5.3 gives a full description of the main class content.

5.4 Simulation outputs

As the functioning of the platform components described in the section 5.2, we expect to have the following results:

1. CPU 1 writes an image into the memory.
2. LCD reads the image from the memory.
3. CPU 2 reads the image from the memory and copies it.
4. CPU 1 reads the copied image.
5. CPU 1 modifies the initial image.
6. CPU 2 reads the modified image.

Bellow, the simulation output of the platform:



```
public class MaintestCaseStudy {

    public static void main(String[] args) throws
    AddressOutOfRangeException, NegativeRangeSizeException, JtlmException {

        final int LCD_DISPLAYING_IMAGES_TYPE=3, INTERRUPT_PORT_NUMBERS=1;

        Memory memory = new Memory(100);   //Instantiation of the memory
        Router bus = new Router(4);        //Instantiation of the bus
        Processor1 cpu1 = new Processor1(); //Instantiation of the first processor
        Processor2 cpu2 = new Processor2(); //Instantiation of the second processor
        //Instantiation of LCD witch will be used to read images
        LCDC lcd=new LCDC(LCD_DISPLAYING_IMAGES_TYPE);
        DMA dma= new DMA();                //Instantiation of the DMA controller
        Timer timer = new Timer();         //Instantiation of the Timer
        //Instantiation of the Interrupt Controller with one interrupt port
        ITC itc = new ITC(INTERRUPT_PORT_NUMBERS);
        // linking CPU 1 master port to the slave port of the bus
        cpu1.initiatorPort.bind(bus.getTargetPort(0));
         // linking CPU 2 master port to the slave port of the bus
        cpu2.initiatorPort.bind(bus.getTargetPort(1));
        // linking LCD master port to the slave port of the bus
        lcd.initiatorPort.bind(bus.getTargetPort(2));
        // linking DMA master port to the slave port of the bus
        dma.initiatorPort.bind(bus.getTargetPort(3));
        // linking the MEMORY to the bus and defining the address interval of this device
        bus.map(0, 100, memory.targetPort);
        // linking the DMA to the bus and defining the address interval of this device
        bus.map(100,50, dma.targetPort);
        // linking the LCD to the bus and defining the address interval of this device
        bus.map(150, 50, lcd.targetPort);
        // linking the ITC to the bus and defining the address interval of this device
        bus.map(200, 10, itc.targetPort);
        // linking the TIMER to the bus and defining the address interval of this device
        bus.map(250, 10, timer.targetPort);
        //Assigning the ITC interrupt port to that witch would be the receiver of the interruption
        itc.interruptPort(0).assignIRQ(cpu2.getInterupPort());
        //Assigning the TIMER interrupt port to that witch would be the receiver of the interruption
        timer.interruptPort.assignIRQ(cpu2.getInterupPort());
        //Assigning the DMA interrupt port to that witch would be the receiver of the interruption
        dma.interruptPort.assignIRQ(cpu1.getInterupPort());

        Debug.debug = true;     // to enable displaying debug messages
        //Debug.trace = true;   // to enable displaying trace messages
        Simulation.start();     // starting the simulation of the platform instanced
    }
}
```

Figure 5.3: The content of the main class of the Platform

```
    Thread-0 CPU 1 writes into memory
Thread-0 Transaction OK: wrote 1 at address 0
Thread-0 Transaction OK: wrote 2 at address 1
.
.
.
```

```
Thread-0 Transaction OK: wrote 9 at address 8
Thread-0 Transaction OK: wrote 10 at address 9

Thread-2 LCD reads the image from the memory
Thread-2 Transaction OK: read 1 at address 0
.
.
.
Thread-2 Transaction OK: read 9 at address 8
Thread-2 Transaction OK: read 10 at address 9

Thread-1 CPU 2 reads the image from the memory and copies it

Thread-0 CPU 1 reads the copied image
Thread-0 Transaction OK: read 1 at address 10
.
.
.
Thread-0 Transaction OK: read 9 at address 18
Thread-0 Transaction OK: read 10 at address 19

Thread-0 CPU 1 modify the initial image
Thread-0 Transaction OK: wrote 2 at address 0
Thread-0 Transaction OK: wrote 4 at address 1
.
.
.
Thread-0 Transaction OK: wrote 18 at address 8
Thread-0 Transaction OK: wrote 20 at address 9

Thread-4 The Timer:  I had sent the interrupt

Thread-1 CPU 2 reads the modified image
Thread-1 Transaction OK: read 2 at address 0
Thread-1 Transaction OK: read 4 at address 1
.
.
.
Thread-1 Transaction OK: read 18 at address 8
Thread-1 Transaction OK: read 20 at address 9
```

Chapter 6

Conclusion

Contents

6.1 Results and Discussion

In this work, we present an alternative to the SystemC library used for the rapid prototyping of the systems on a chip in order to validate and verify the embedded software before the real hardware would be ready.

Throughout the period of this thesis, we tried and we succeed to prove the possibility to write TL models with a different execution model instead of SystemC. This library was built using the Java programming language because of its capability to deal with multi tasking in preemptive way which allowed us to see another mode of execution of transactional models different from SystemC.

JTLM helped us to make the difference between the SystemC properties and the TLM properties. Because they are used to write TL models with SystemC, and since SystemC has a cooperative scheduler, programmers attributed the cooperation characteristic to TLM. Now, after the design of JTLM, we can say that transactional models could be written with a cooperative scheduler or with a preemptive scheduler depending to the model of execution used to represent it.

After the construction of the JTLM library, we implemented several tests in different ways to validate and test the good functioning of this model. We used a Makefile to repeat the test many times. Then we were interested to solve some SystemC problems such as the polling problem and adding the duration to tasks. We have not arrived to find a solution using JTLM to resolve the polling problem without adding the wait of time into the polling loop; it seems to us it is a more fundamental question. For the second issue, unlike SystemC, JTLM has included the timed tasks.

6.2 Prospects

As prospects to this work, JTLM open the possibility to be extended to satisfy better the programmers' needs such as to include large number of data types to support modeling of Hardware and adding an APIs for transaction-based verification.

JTLM does not support transaction with a big granularity; a future work could be done so JTLM can provide this feature.

It dues to the JTLM time manager algorithm that the problem of the polling exist, as the figure 6.1 shows, the polling has an infinite branch while the status of the program progress remains in its place, To solve this problem, we suggest to remove this infinite branch from the execution tree till the value of the polling loop changes.

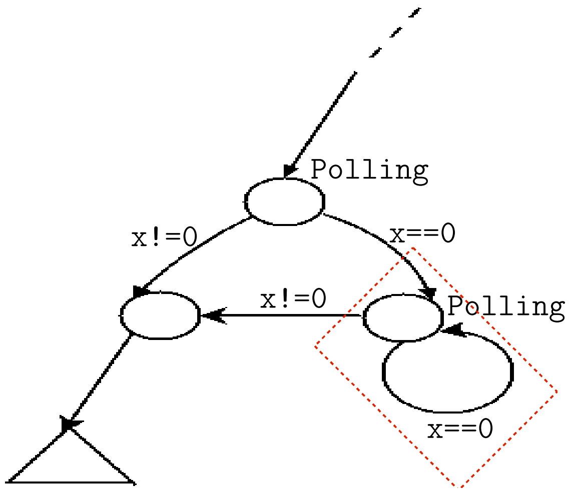


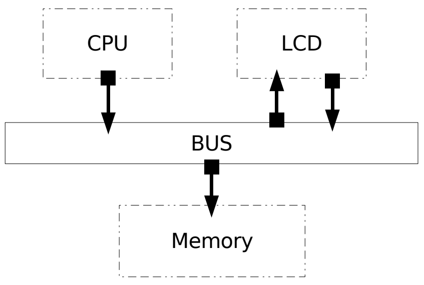
Figure 6.1: The execution tree of a polling process

Appendix A

Utilities description

A.1 Bus

General description



The Bus is included in the JTLM development kit as a library component. The bus is used to decoding addresses routing transaction between components. Our Bus component can be parameterized and is though to be quite generic.
The source code is available at `jtlm/util/bus`.

Functional description

The Bus component is used as a communication channel between
components.

The Bus has a generic number of master and slave ports, normally if there are only one bus in a platform, the number of master ports of the bus should be the same as the global number of the slave ports of the platform components, and the number of its slave ports should equal to the global number of the master ports of all the components existing in the platform.

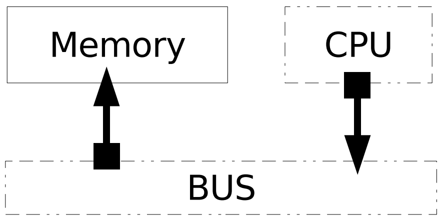
A typical transaction transfer proceeds as follows:

1. Component sends transaction through the bus to the target component.
2. The Bus decodes its address and determines the target components, then it routes the transaction to this component.
3. the target component receives the transaction.

A.2 Memory

General description

A Memory is included in the JTLM development kit as a library component. Memory peripherals are used for data storage. The source code is available at `jtlm/util/Memory`.



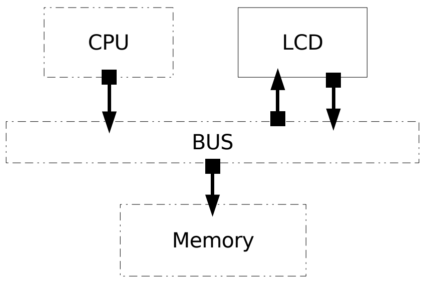
Functional description

Memory is a component that models the behavior of an actual memory.
The Memory component allows data access, storage or update. It has one slave port connected to the bus.

A.3 LCDC

General description

The LCD controller is included in the JTLM development kit as a library component. The LCDC is a master-slave component which connects to the Bus. Our LCD component can be parameterized and is though to be quite generic. The source code is available at `jtlm/util/lcdc`.



Inputs/Outputs

The LCDC provides the following inputs/outputs:

1. A Slave port for configuration.

2. A Master port used to read data from memory.

Functional description

The LCD component is used to display videos or data from memory. A typical LCD transfer proceeds as follows:

1. Software configures the LCD to display data by writing into the START_REG register.

2. Software writes in the start address register. Then LCD begins his work.

3. The LCDC performs read accesses to an external memory device holding the video buffer through its master interface. Video data are supposed to be stored contiguously in memory.

Register map

Offset	Name	Type	Width	value	Description
0	`LCD. ADDR_REG`	rw	32	0x00000000	Start address register
1	`LCD. START_REG`	rw	32	0x00000000	Start register

Registers Reference

Start address register (ADDR_REG)

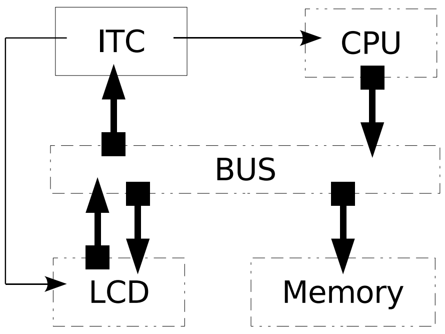
The start address register holds the base address of the video memory buffer as accessed on the AMBA master interface. The value should not be modified while the controller is operating.

Start register (START_REG)

The start register is used to trigger the beginning of controller operation. Writing 0x00000001 into the register starts the LCDC.

A.4 ITC: Interrupt Controller

General description



An ITC is included in the JTLM development kit as a library component. Some components want to send interrupts but they do not have interrupt outputs, so the ITC was created to let these components send interruption by configuring it. Our ITC Component can be parameterized and is though to be quite generic. The source code is available at `jtlm/util/ITC`.

Functional description

The ITC component is used to send interrupts to the peripherals. When peripheral needs to send interrupt and it does not possess an interrupt output, it uses the ITC to send an interrupt to a specific component. The ITC has one slave port for fixing the address of interrupt port.

A typical ITC task proceeds as follows:

1. Peripheral configures the ITC to send interrupt to another peripheral by writing to the addressPort register to specify which interrupt port should send interrupt.
2. The ITC then begins sending interrupt to all the peripherals connected to his interrupt port.
3. All the peripherals connected with the interrupt port receive the interrupt.

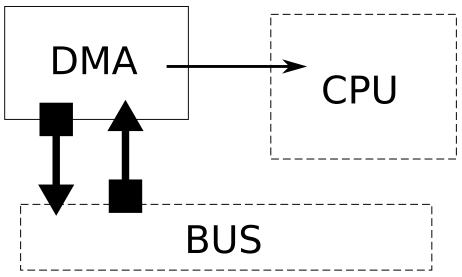
Register map

Offset	Name	r/w	Description
0	`ITC. addressPort`	rw	address of the interrupt port

A.5 DMA Controller

General description

A DMA controller is included in the JTLM development kit as a library component. Direct memory access (DMA) peripherals allow for efficient bulk data transfer between peripherals and memory by removing the CPU from the data path. The source code is available at `jtlm/util/dma`.



Functional description

The DMA component is used to perform data transfers between two memory regions, between a memory and a peripheral, or between two peripherals. It allows data transfers of fixed or variable length without intervention from the CPU. The DMA has one master port and one slave port for controlling the DMA.

A typical DMA transfer proceeds as follows:

1. Software configures the DMA to transfer data by writing to the control registers.
2. Software enables the DMA. It then begins transferring data without additional intervention from the CPU.
3. The DMA master port reads data from the read address, which may be a memory or a peripheral, then writes the data to the destination address (a memory or peripheral). A shallow first-in first-out (FIFO) may buffer data between the read and the write.
4. The DMA transfer ends when a specified number of bytes are transferred. The DMA may issue an interrupt request at the end of the transfer.
5. During or after the transfer, software may determine if a transfer is in progress, or if the transfer ended by examining the DMA's status register.

Register map

Offset	Name	r/w	Description
0	`dma.status`	rw	Status of the transfer
1	`dma.source`	rw	Source start address
2	`dma.dest`	rw	Destination start address
3	`dma.length`	rw	Transfer size

The `status` register indicates particular conditions inside the DMA. The status register can be read at any time by software. Reading zero means the DMA is idle, i. e. transfer finished or no transfer in progress. Any other value means the DMA is busy. Writing zero stops the current transfer if any. Writing any other value starts a transfer (error if transfer in progress).

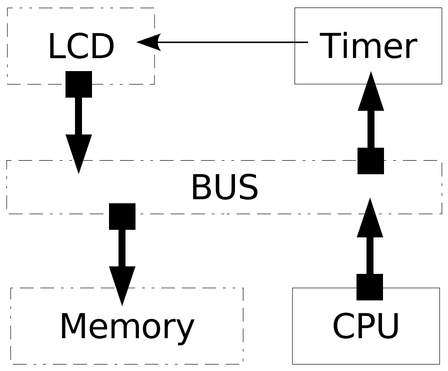
Interrupts

The DMA has a single IRQ output that is sent when the transfer is finished. A typical DMA interrupt handler reads the length register to check if the transfer was completed successfully.

A.6 Timer

General description

The Timer is a slave component which connects to the *Bus*. It is included in the JTLM development kit as a library component. The timer is used to send interrupt each period of time specified in its register PERIOD.
The source code is available at `jtlm/util/Timer`.



Inputs/Outputs

The Timer provides the following inputs/outputs:

- Slave port for configuration.
- Interrupt port.

Functional description

The Timer component is used to send interrupt to a peripheral each period of time.
A typical Timer transfer proceeds as follows:

1. Peripheral configures the Timer to send interrupt by writing a value T into the PERIOD register.
2. If the value T was not null, the Timer starts his work, which consist of raising an interrupt every T milliseconds if the PERIOD register value was not modified (the first would be sent after T ms without verifying the value of the register ACK while the others would be sent if the interrupt was acknowledged).
3. Timer reset the ACK register value to 1 after raising each interrupt.
4. After receiving interrupt, peripheral write a value into the ACK register to acknowledge receipt of the interrupt.
5. If the Peripheral changes the PERIOD register value to T1, the timer will stop his current interrupts raising program and start a new one every T1 milliseconds.
6. If the Peripheral writes into the PERIOD register the value 0 the timer will stop his work.

Register map

- The `PERIOD` register indicates the period of raising interrupts.
  - Writing any value in the PERIOD register will reset the ACK register to zero and will change the raising interrupt period.
  - Writing zero stops sending interrupts.
  - Reading PERIOD register dues the period.

Offset	Name	r/w	Description
0	`Timer.PERIOD`	rw	period of interrupt transmission
1	`Timer.ACK`	rw	Source start address

- The `ACK` register indicates if the peripheral has send an acknowledgment when it receives the interrupt or not.
  - Writing of any value in the registry ACK acknowledge the interrupt.
  - Reading of the ACK register returns 1 if the interrupt has not yet been acknowledged.

Appendix B

MakeFile

```
EXTRA_PROGRAMS =
                MainPollingPlooling
                MainPollingInterruption
                MainInterruptionInterruption
                MaintestDMA
                TestDMAgeneric
                MaintestTimeEvent
                TimeEventGeneric
                MaintestTimeInterrupt
                TimeInterruptionGeneric
                MaintestTimeInterruptEvent
                TimeInterruptionEventGeneric
                MaintestTimePolling
                MaintestTimePollingCorrection
                TimePollingGeneric
                MaintestTimeConsume
                MaintestTimeEventConsume
                MaintestTimeEventInterruptionConsume
                MaintestTimeInterruptTimer
                MaintestTimeInterruptTimer2
                MaintestCaseStudy

iterations = 5

all: compile

compile:
        @ant build

clean:
        @ant cleanall
        rm -f *.aux *.log *.out *.toc *~  *.blg
        rm -r bin

createReferences:compile
        @echo -e Creating references ...'\n'
        @for prog in $(EXTRA_PROGRAMS) ; do \
                ant $$prog > reference$$prog  ;\
        done ;
        @echo Finished.

runall:compile
        @echo -e Running ...'\n'
        @for prog in $(EXTRA_PROGRAMS) ; do \
                ant $$prog ;\
        done ;
        @echo -e '\n' All runs passed.



test:compile
        @echo Running test suite...
        @for prog in $(EXTRA_PROGRAMS) ; do \
                ant $$prog > ref;
                diff -rupN ./reference$$prog ref  ; \
        done ;
        @echo All tests passed.


multipletest :compile
        @echo -e Running test suite... '\n'
        @for prog  in $(EXTRA_PROGRAMS) ; do \
                echo -e   PROGRAM NAME : $$prog '\n'; \
                for ((j=0;j<$(iterations);j+=1)); do \
                        echo   Running test number $$j; \
                        ant $$prog > ref;
                        diff -rupN ./reference$$prog ref  ; \
                done ; \
                echo -e '\n' " ************* "'\n'  ;\
        done ;
        @echo All tests passed

MainInterruptionInterruption:compile
        @ant MainInterruptionInterruption

MainPollingPlooling:compile
        @ant MainPollingPlooling
```

```
MainPollingInterruption:compile
        @ant MainPollingInterruption

MaintestDMA :compile
        @ant MaintestDMA

TestDMAgeneric :compile
        @ant TestDMAgeneric

MaintestTimeEvent :compile
        @ant MaintestTimeEvent

TimeEventGeneric :compile
        @ant TimeEventGeneric

MaintestTimeInterrupt :compile
        @ant MaintestTimeInterrupt

TimeInterruptionGeneric :compile
        @ant TimeInterruptionGeneric

MaintestTimeInterruptEvent :compile
        @ant MaintestTimeInterruptEvent

TimeInterruptionEventGeneric :compile
        @ant TimeInterruptionEventGeneric

MaintestTimePolling :compile
        @ant MaintestTimePolling

MaintestTimePollingCorrection :compile
        @ant MaintestTimePollingCorrection

TimePollingGeneric :compile
        @ant TimePollingGeneric

MaintestTimeConsume :compile
        @ant MaintestTimeConsume

MaintestTimeEventConsume :compile
        @ant MaintestTimeEventConsume

MaintestTimeConsumeEvent :compile
        @ant MaintestTimeEventInterruptionConsume
```

```
MaintestTimeInterruptTimer :compile
        @ant MaintestTimeInterruptTimer

MaintestTimeInterruptTimer2 :compile
        @ant MaintestTimeInterruptTimer2

MaintestCaseStudy :compile
        @ant MaintestCaseStudy
```

Bibliography

[AGH05] K. Arnold, J. Gosling, and D. Holmes. *Java (TM) Programming Language, The*. Addison-Wesley Professional, 2005.

[AHT$^+$] A. Adya, J. Howell, M. Theimer, W.J. Bolosky, and J.R. Douceur. Cooperative task management without manual stack management.

[Bou07] Yussef Bouzouzou. Acceleration des simulations de modeles de systemes sur puce au niveau transactionnel. 2007.

[BPL01] A. Bakshi, V. K. Prasanna, and A. Ledeczi. Milan: A model based integrated simulation framework for design of embedded systems. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 82–93, New York, NY, USA, 2001. ACM.

[BWH$^+$03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM Press.

[Cor08] Jérôme Cornet. *Separation of Functional and Non-Functional Aspects in Transactional Level Models of Systems-on-Chip*. PhD thesis, INPG, Grenoble, France, April 2008.

[Fun07] Giovanni Funchal. Comparison of Embedded System Models: From Signals to Transactions. Master's thesis, Université Joseph Fourier, 2007.

[Ghe05] F. Ghenassia. *Transaction level modeling with systemc: Tlm concepts and applications for embedded systems*. Springer Verlag, 2005.

[Gro02] T. Grotker. *System design with SystemC*. Kluwer Academic Publishers Norwell, MA, USA, 2002.

[Hel07] Claude Helmstetter. *Validation de modèles de systèmes sur puce en présence d'ordonnancements indéterministes et de temps imprécis*. PhD thesis, INPG, Grenoble, France, 2007.

[Ini06] O.S.C. Initiative. IEEE Standard SystemC Language Reference Manual. *IEEE Computer Society*, pages 1666–2005, 2006. `http://www.systemc.org/`.

[Lab09] VERIMAG Research Lab. Verimag web site, 2009.

[LB00] B. Lewis and D.J. Berg. *Multithreaded programming with Java technology*. Prentice Hall PTR, 2000.

[LDNA03] Akos Ledeczi, James Davis, Sandeep Neema, and Aditya Agrawal. Modeling methodology for integrated simulation of embedded systems. *ACM Trans. Model. Comput. Simul.*, 13(1):82–103, 2003.

[Lea99] D. Lea. *Concurrent Programming in Java.: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[LY99] T. Lindholm and F. Yellin. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[Mic08] Sun Microsystems. The Java Tutorials, 1995-2008.

[Moy05] Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, December 2005.

[MPA05] J. Manson, W. Pugh, and S.V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391. ACM New York, NY, USA, 2005.

[OW04] S. Oaks and H. Wong. *Java threads*. O'Reilly Media, Inc., 2004.

[SMMM06] Ludovic Samper, Florence Maraninchi, Laurent Mounier, and Louis Mandel. Glonemo: global and accurate formal models for the analysis of ad-hoc sensor networks. In *InterSense '06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, page 3, New York, NY, USA, 2006. ACM.

[THG$^+$92] Le T, Soonhoi Ha, Li Ght, Joseph Buck, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems, 1992.

Abstract

The work carried out in this document deals with designing a prototype of a transactional simulator avoiding preconceived ideas and stereotypes of SystemC. We identify the constraints imposed on the implementation of a transactional model in a simulator. We define then a library from Java language for writing TL models. The mechanism of Java threads is particularly suited for this study. Thus we demonstrate the ability to model transactional models using a language other than SystemC. Then we observe the behavior of platforms made with JTLM and we try to solve problems already exist in SystemC using the new execution model JTLM.

Keywords: System-on-Chip, TLM, Java, JTLM, SystemC.

Résumé

Le travail effectué dans le présent document porte sur la conception d’un prototype de simulateur transactionnel en évitant les idées préconcues et les clichés de SystemC. On identifie les contraintes imposées sur l’exécution d’un modèle transactionnel par un simulateur. On définit dans une seconde phase une extension du langage Java pour la modélisation transactionnelle. Le mécanisme de threads du langage Java est particulièrement adapté pour cette étude. Ainsi on démontre de la possibilité de modéliser des modèles transactionnel en utilisant un langage autre que SystemC. Après on s’intéresse à voir le comportement des platform réalisées avec JTLM et on essaye de résoudre des problèmes existant en SystemC en utilisant le nouvel modèle d’exécution JTLM.

Mots Clés: Systèmes sur puce, TLM, Java, JTLM, SystemC.

ملخص

العمل المنجز في هذا البحث يتمحور حول تصميم نموذج "ج ت ل م" لمحاكاة عمل رقاقة حقيقية، و ذلك باستبعاد الأفكار المسبقة والصور النمطية للسيستام س. في هذا العمل يتم تحديد الضوابط المفروضة لإنجاز هذا النموذج بلغة مغايرة عن السيستام س. ثم بتم إعداده و تنفيذه باستعمال جافا كلغة للبرمجة. بذلك يتم إثبات إمكانية تصميم نموذج بلغة مختلفة عن السيستام س. بعدها يتم دراسة سلوك الشرائح الافتراضية المنجزة باستعمال ال"ج ت ل م ". في المرحلة الأخيرة يقع محاولة حل بعض المسائل المتعلقة بالسيستام س باستخدام النموذج الجديد

الكلمات المفاتيح: نظام على رقاقة ، جافا ، سيستام س، ت.ل.م، ج ت ل م