POLITECNICO DI TORINO

Department of Control and Computer Engineering

Master Degree in Mechatronics Engineering

Study and development of software based self-test programs for Automotive ECU

Supervisors

Prof. Luca STERPONE
Dr. Boyang DU

Candidate
Younes Mahboub
Student ID: 213838

March 2018

Last night, in private, I asked the wise old man

To reveal to me the secret of the world.

Softly he whispered, Hush! In my ear:

It's something you learn, not words you can hear.

"Rumi"

Summary

Nowadays, ever-improving technologies related to the semiconductor manufacturing and significant complexity in the design of electronic devices implemented in the automotive systems, lead to a rise in the number of possible defect mechanisms and affecting the level of quality and reliability. As a result, System-On-a-Chip (SOC) testing devices have become the main challenge in the Electronic Control Unit (ECU) of automotive systems. A wide range of demands such as test speed, fault coverage, diagnostic options, cost-effective techniques and test length requires an underlying solution for using an online application of functional and structural tests in key components. Self-testing is an effective mechanism that consists of two main approaches, Software-Based Self-Tests (SBST) and Built-In-Self-Tests (BIST) into E/E (Electric and Electronic) architectures, to ensure the correct operation of ECUs.

Built-In Self-Test (BIST) is an embedded test hardware with limited at-speed testing. Also, BIST resides on the system in nonfunctional mode.

Software-Based Self-Test (SBST) is an embedded test software which uses self-tested memory and processor to test System-On-a-Chip component models. SBST handles testing as an application. Due to the growing of the open-source microprocessors and applying SBST methodology into an open source microprocessor core, SBST is proposed for cost reduction and common use in SOC test environment.

Widespread use of SystemC/TLM simulation and the cycle accurate open source system simulator gem5 in Embedded Industry and test researching lead to facilitate an optimizing virtual platform for testing.

Gem5: Gem5 is a modular platform that built from a combination of the former full system simulator M5 and memory system simulator GEMS for system level design, architectural and performance exploration. This combination makes gem5 as Self-contained simulation framework which is built on a discrete-event simulation kernel, although you are welcome to bind things together. Unlike other simulators, gem5 overcame the limitation of the emergence of multicore systems and deeper.

Cache hierarchies have presented architects with several new dimensions of exploration and present a flexible simulation framework. In addition, gem5 relies on high-quality open-source codes that allow collaborating with colleagues in both industry and academia. Gem5 has presented practical transport interfaces for coupling CPU to Memory and binding different environments. Relying on flexibility and availability features of gem5, it can have a key role in self-test technologies. Through the variety of ISA possibilities which supported by gem5, the ARM should be the best selection for this thesis propose. Gem5 as a simulator for testing base functionality just focuses on requirements, design specifications and the increase of accuracy with code simplicity. ARM selects a set of compile-time build options that control simulator functionalities such as the ISA, CPU models, memory system and peripheral devices to use.

SystemC/TLM: SystemC for system-level design is a set of the class library in C++ which is interoperable for System-On-a-Chip (SOC) modeling platform and allows the exchange of IP models in system-level design and functional verification. SystemC also offers the event-driven simulation interface for design and concurrent development of hardware and software. These facilities make the access to digital simulation models easier for system designers.

SystemC supports elaborating high-level synthesis (Electronic system level) and simulates concurrent processes, furthermore enables a huge set of possibilities for architectural exploration and performance modeling. SystemC concentrates on the functionality of the system instead of its structure. This means that it is possible to focus on the actual behavior of the system more than its implementation details. But detailed implementation can be changed during any alternative of system architecture. In this regard SystemC presents TLM (Transaction-Level Modeling) as an abstraction level framework that based on virtual prototype make communication among modules and digital systems. Communication mechanisms in TLM are implemented modules and ports which are using digital signals and blocks through the channels and interface classes. The Functional model which is provided by TLM increases simulation speed against lots of pin wiggling in RTL. Therefore, high Interoperability between modes in TLM affects directly possibilities of system design exploration.

The goal of this thesis is to investigate the optimal virtual platform for a transaction-based interface which is employed to connect the two environments together through two main simulation models:

1. Computation-centric simulation
2. Communication-centric simulation

This thesis focuses on Communication-centric simulation since both gem5 and SystemC/TLM are functional-based and data-based simulation models and not code-based. SystemC/TLM enables to run gem5 as a thread inside the kernel and keeping the events and timelines synchronized between the two environments. This functionality provides interoperating with a wide range of System-On-a-Chip (SOC) component models for the gem5. MemObjects are connected through master and slave ports. A master module has at least one master port, a slave module has at least one slave port and an interconnect module has at least one of each protocol stack based on Requests and Packets. Likewise, SystemC/TLM can simulate two environments together. The TLM-2.0 presents two main blocks, initiator, and target. An initiator module initiates new transactions, and a target module is a module that responds to transactions initiated by other modules. A transaction is a data structure transferred between initiators and targets using function calls by a generic payload. The same module can act as both an initiator and a target, this would be typically the case for a model of an arbiter, a router, or a bus.

Therefore for coupling two environments, there are two possibilities:

1. Gem5/TLM co-simulation
2. TLM/TLM simulation

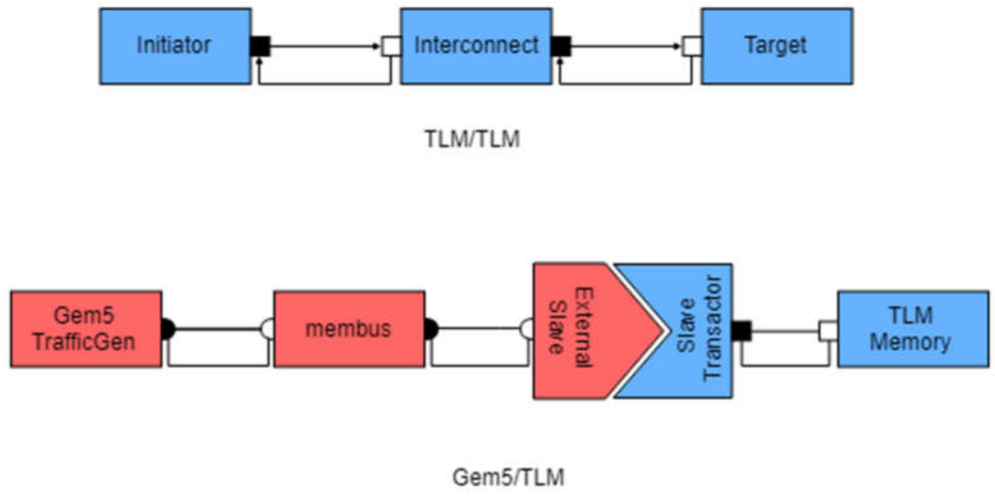


Figure 1: TLM/TLM vs Gem5/TLM coupling mechanism

Upon reviewing sample tests using these models under test to evaluate the diverse characteristics of each model. The results demonstrate which model is the best suited for different scenarios.

Acknowledgments

First and foremost, I would like to thank influence persons, who support and guide me to be possible this work.

Prof.Luca Sterpone, without your guidance and support as ACCLAIM supervisor of my dissertation committee, I would not have felt as prepared and confident during my research, THANK YOU for your support and encouragement throughout this strenuous process!

Dr.Boyang Du, without your additional guidance and support, I would not have focused on deep study. Your questions and discussions during my dissertation created additional layers of my knowledge of self-efficacy and my study of its sources. THANK YOU for illuminating the areas that needed more investigation!

CAD group Buddies, without your support and motivation throughout my research, I would not have felt as confident to press forward. Each of you has exemplified courage, strength, and friendship through your own personal obstacles and setbacks. THANK YOU for always being there to help!

Mom and Dad – Without your support, encouragement, and love, I would not have been able to attain this goal. Knowing that I could always count on you handle my issue to provide such a peace of mind that allowed me to focus on the work at hand. THANK YOU for learning and inspiring me to be patient and believe in God will allow you to accomplish your goals!

My Wife– Without your understanding and support through my frustrations, lack of quality time, and stressful deadlines, I would not have been able to complete my dissertation. You allowed me to spend countless hours working on “this little paper” last year. I would like to dedicate this work to my wonderful my wife Shiva. I LOVE YOU!

God – THANK YOU for the wisdom, and grant me the patience, the perseverance, and the ability!

List of Figure

List of Table

Contents

Chapter 1

Introduction

1.1. Testing

Nowadays, ever-improving technologies related to the semiconductor manufacturing and increasing significant complexity in the design of electronic devices implemented in the automotive embedded systems, lead to a rise in the number of possible defect mechanisms and affecting the level of quality and enhancing validation and reliability. As a result, System-On-a-Chip (SOC) testing devices have become the main challenge in the Electronic Control Unit (ECU) of automotive systems. A wide range of demands such as test speed, fault coverage, diagnostic options, cost-effective techniques and debugging and test length requires an underlying solution for using an online application of functional and structural tests in key components. To create Self-testing is an effective technique that consists of two main approaches, Software-Based Self-Tests (SBST) and Built-In-Self-Tests (BIST) in E/E (Electric and Electronic) architectures, to ensure the correct operation of ECUs.

Therefore, it needs to define the concept of these techniques.

1.1.1. Self-test:

In order to detect permanent fault and malfunction in microprocessor-based embedded systems performance, System-on-chip microprocessor is periodically forced to execute the Self –test code uploading to non-volatile memory during the device testing.

1.1.2. BIST

Built-in self-test is a more general test approach with additional pin or hardware and software into embedded systems to deliver higher fault coverage which enables them to test itself. In another word, testing of itself operation by means of own circuits leads to reduce an external automated test equipment (ATE). This technique provides the capability to execute tests outside the embedded testing environment, thereby it makes easier customer support and needs a way to interface with the outside environment to be effective.

1.1.3. SBST

Software-based self-test methodology is an embedded test software that develops a testing code and runs as an application by memory and processor during regular system operation to test System-On-a-Chip component models. In order to detect system faults quickly after they become visible. Additionally, SBST is the non-intrusive testing approach that provides at-speed testing capability in actual operating frequency without any hardware or performance overheads and monitoring the SOCs without interfering. While SBST can be used in the field, either off-line or periodically on-line with avoiding over-testing.

1.1.4. How does SBST work?

Test code and test data deploy into memory (i.e., either the on-chip cache or the system memory). A low-cost external tester can execute test code and data loading via a memory load interface. The processor executes test programs at speed. The application runs as a test by exploiting the proper instructions to excite faults. Finally, the test code stores. Besides, the test responses back into the processor data memory to propagate the faults. When the test process is supported only on the on-chip cache, the self-test program must be developed so that no cache misses occur during SBST execution. Hence, this is sometimes called cache-resident testing. Eventually, Test responses are uploaded into the tester memory for external evaluation. This mechanism can be implemented on the systemC/ TLM virtual platform.

Due to the growing of the open-source microprocessors and applying SBST methodology into an open source microprocessor core, SBST is the best choice for the industry that limited by the difficulty to write the efficient and effecting test program and to device suitable methodology for cost reduction and common use in SOC test environment.

1.2. Simulation

Dealing Effectively with Fast Growth of complex hardware architectures especially System-on-chip, the hardware/software simulation and co-simulation based on virtual platforms has become a popular approach for the embedded system. Accurate virtual platforms allow to decrease cost and Time-to-Market (TTM), for prototyping space exploration and remove the constraint of transposition of developing hardware and software.

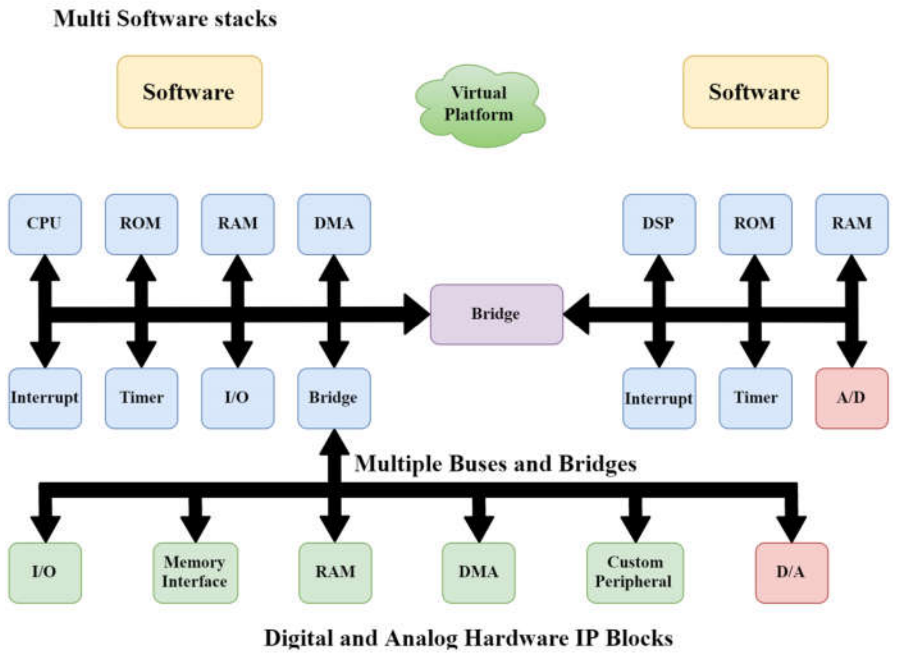


Figure 2: Typically Use-case of Virtual Platform

They are software-based systems that can fully mirror the functionality of a target System-on-Chip or physically hardware behavior rapidly. Also, they are able to emulate high percentage of the behavior of the real system. Virtual platforms have the capability to be reused in future projects and hardware engineers able to fast design space exploration, that means they can try out different things like how big is system cache or what is the number of cores should use or exchange easily.

Some of the important virtual prototypes are functional software models of physical hardware which based on systemC, Furthermore, they provide controllability over the entire system and offer powerful debugging and analysis techniques.

Generally, systemC is a standard from IEEE and different modeling hardware and software components. It extends C++ to an event-driven simulation kernel with various levels of accuracy. SystemC is much faster cycle accurate simulations compared to of RTL, however, the normal cycle accurate systemC is don't fast enough to boot an operating system in a reasonable time. This reason besides the event needs to communication between abstract levels caused to build new communication mechanisms from the actual hardware named Transaction Level Modeling (TLM). Cycle-accurate model has a clock on all the components and pin accurate model simulate on each pin the events that are happening such as waveforms with analyzing waves and pins while TLM simulates all these events communication and gets abstracted with the help of function calls. By encapsulating data and transaction, TLM makes the simulation depending on the level of accuracy which can simulate up to 10,000 times faster than in VHDL.

TLM is widely used in industry as Virtual platform tools and Core Models such as Synopsys Platform Architect, Cadence - Virtual System Platform, Mentor Graphics - Vista Virtual prototyping and ASTC - VLAB Works. Virtual Platform Core Models such as ARM (Fast-models) have only just loosely timed blocking kind of TLM modeling.

There are several use cases for modeling platforms with TLM in different area such as software application developing and software performance analysis in Software developers and in other hand architecture analysis and hardware verification for hardware engineers.

Generally all TLM systems have the CPU and Memory, also the bus in between as interconnect transfers data from CPU to target memory and target I/O

basically for communication Initiator (CPU) needs a so-called generic payload reference and a pointer to a payload object that has information containing Command Address data, Byte Enables, Response Status and other features with function calls, object transferred is pointer to the target (Memory) then It can change (Read & Write) and to transmit a manipulated object back. This is very similar to Gem5 behavior.

By knowing the detail of coding style is able to make interoperability and with getting a model from another company it needs just plug it together with the fit model and it will work properly.

Gem5 is an event-driven simulation kernel with some objects models that compile as a library and in Gem5. By using this logic, the systemC project can couple to Gem5. SystemC introduce a module called Gem5simcontrol for coupling two environments which support codes and pools of each side. The object in systemC implements the Gem5 event queue. It replaces the Gem5 event queue with the hook to the systemC event queue.

Simulated system schedules Gem5 event and on the other side getting systemC event will be scheduled to be executed driven body gem5 Object. So hooking Gem5 to the systemC kernel can make way to communication two environments. Also with employing TLM, is solved the problem of communicating to other systemC modules and Simulated system capable of plugging in any other systemC model and hook it to Gem5 modules properly. As consequence Gem5 already supported the coupling to systemC kernel.

This thesis will compare two virtual communication mechanism to find feasible configurations in order to provide optimal design space exploration for verification and visualization applications.

Chapter 2

Background

2.1. Computer simulation

Computer simulation reproduces a mathematical model of the behavior of the system. The following figure illustrates the classification of computer simulation, the abstraction level leads to specific simulation tools.

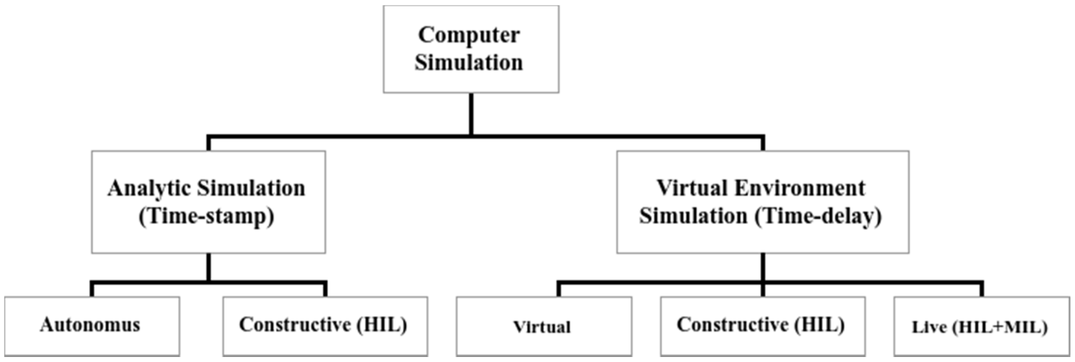


Figure 3: General computer simulation classification

To deal with the increasing complexity of System-on-Chip (SoC), the hardware/software co-simulation based on virtual platforms has become a popular approach for the Electronic System Level (ESL) design flow.
Three approaches to modeling the processor of a virtual platform are addressed:

1. hardware description language (HDL),
2. instruction set simulator (ISS),
3. Formal.

In general, the simulation speed of the HDL approach is far **slower** than that of the ISS and formal approach.
The formal approach, which uses ‘‘compiled simulation’’ to simulate software statically, is always faster than the ISS approach, which uses ‘‘interpretive simulation’’ to simulate software dynamically.

Over the past years, interpretive simulators generally were used in system design but because of poor performance of these type of simulator, system developer started research work based of increasing performance ability and fast compiled instruction-set simulation. Eventually, Instruction set simulators have become a main tool for architecture development and exploration in SOCs. Therefore, the simulation system increase the efficiency and capabilities of system design exploration. While due to constraint compiled technique and being specificity instructions for each program, it was not possible to use in commercial products [4]. This is a lack of industry satisfaction caused the programmer to think about the new way in simulation till achieving to the Virtual Platform which it will be explained in the following.

In fact, the instruction-set simulator is a special kind of functional-level mode in a simulation model which mimics the microprocessors behavior by using internal variables with code in a high-level programming language like C instead of using assembly language. [5]

Instruction-set methodology is used for at least one of the following reasons:
In order to simulate machine code from a hardware or full-system of a computer for compatibility. Also, it uses for monitoring and testing the execution of machine code same as hardware device for debugging propose. In addition, instruction-sets is able to improve the speed of system performance compared to the interpretive simulator. In another side, the instruction-set increasing accuracy of simulation compared to application-specific approaches. While it just mimics hardware behavior functionally.

Common embedded systems as application-centric include a wide range of machines and products like dependable and real-time hardware small electronics components in an integrated circuit. In order to meet different requirements on functionality, performance for a specified product, it is able to whether to redesign or revise frequently hardware and software codes. This ability make to potential benefits for shortening time-to-market and makes alongside reduction cost and consumption energy of the production process. According to Embedded Market Forecasters reported it cause save up to 50% of the scheduled time of projects. [7] But the other side of this effort makes a considerable challenge to the efficiency of design exploration in embedded systems. In another word, this situation allows to fast propagate of system complexity. Generally, all systems are separated to software and hardware part based on system specifications. Using instruction-set allows to software and hardware developed in parallel and simultaneously and the final application is able to test, verify and implement in hardware prototyped or finalized.

An Instruction Set Simulator ISS is a layer of software-based implanting between operating systems and upper layers like applications or lower layer like real hardware, which enables interface the upper layer ISA program so-called source ISA with lower layer software/hardware of ISA so-called target ISA by binary compiling for an Instruction Set Architecture (ISA). Basically, each instruction form Source of ISA translates into instruction set of target ISA which represent source ISA behavior.[6]
Following figure illustrate the main differences between the ISS and virtual machine (VM) based virtual platforms. The first row depicts the ISS-based virtual platform in building a co-simulation environment using a lot of peripheral and interconnect model in order to run an operating system linked to another part of systems. However, it is negligible the hardware models and the system functionalities do not fit completely into an unmodified operating system. If all layers use single

language like SystemC. It is able to retain accessibility between the lower and upper layers. While the second row represents a VM-based virtual platform which all fundamental hardware component to run the operating system are included in the virtual platform, this model simplifies process rather than the ISS model. It requires only extract the information of processor and target ISA in a virtual machine to wrap and accommodate behavior as ISS. Good examples are from VM such as QEMU and Gem5. [15]

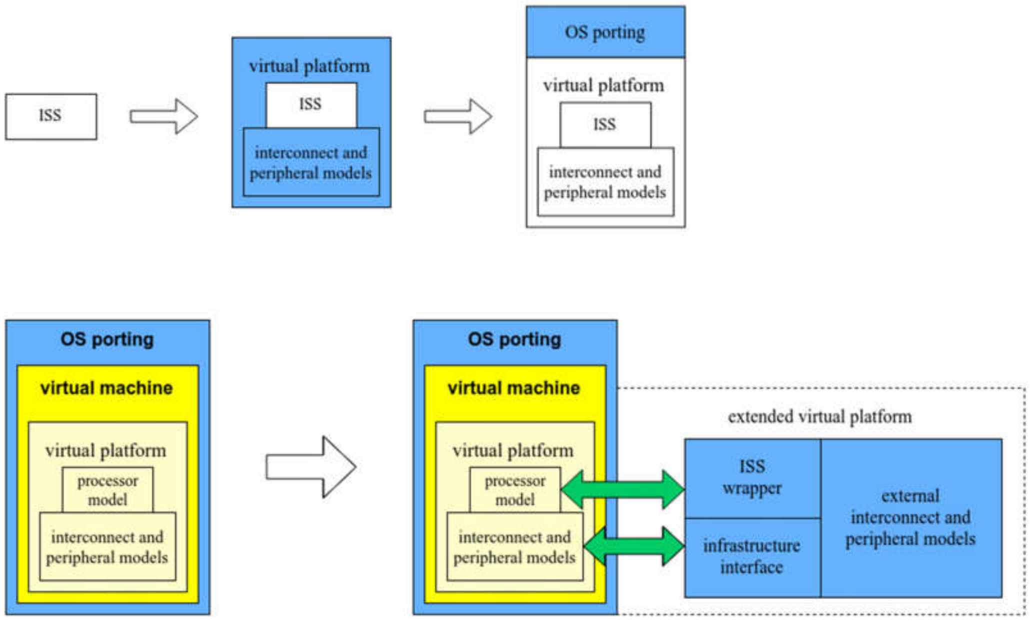

ISS
virtual platform
ISS
interconnect and peripheral models
OS porting
virtual platform
ISS
interconnect and peripheral models
OS porting
virtual machine
virtual platform
processor model
interconnect and peripheral models
OS porting
virtual machine
virtual platform
processor model
interconnect and peripheral models
extended virtual platform
ISS wrapper
infrastructure interface
external interconnect and peripheral models

Figure 4: Instruction Set Simulator and Virtual Machine Comparison

2.1.1. System-Level Design

System-level design is electronic design in higher abstraction level which includes identifying the specific system, verifying functionalities in order to find optimal system architecture. Nowadays complex systems are integrated into a System-On-Chip and contain specific software. SoC in system level design technology cause develops more complex and capability of both hardware and software.

2.1.2. Event-driven memory system

An event-driven memory system as part of event-driven programming which suitable method in embedded system and simulation too with containing all components like caches, crossbars, snoop filters, and DRAM controller model, in order to capture the power of states and system impact of memories and controller. The components need more flexibility, in order to compatibility issues, address ranges and model complex cache hierarchies with heterogeneous memories.

2.1.3. Cycle-accurate simulation

A cycle-accurate simulator simulates a micro-architecture based on cycle-by-cycle simulation. That means the cycle-accurate model has on all the components a clock. In another word, cycle-accurate simulation is trying to simulate timing exact on per-cycle accesses perfectly. So each individual component is simulated at the exact same time, with considering synchronizing at single-clock resolution, which has a costly CPU. All in all the accuracy of this methodologies are close to real hardware not means 100%. The following figure shows the level of accuracy and abstraction of the different model compared to cycle accurate simulation model

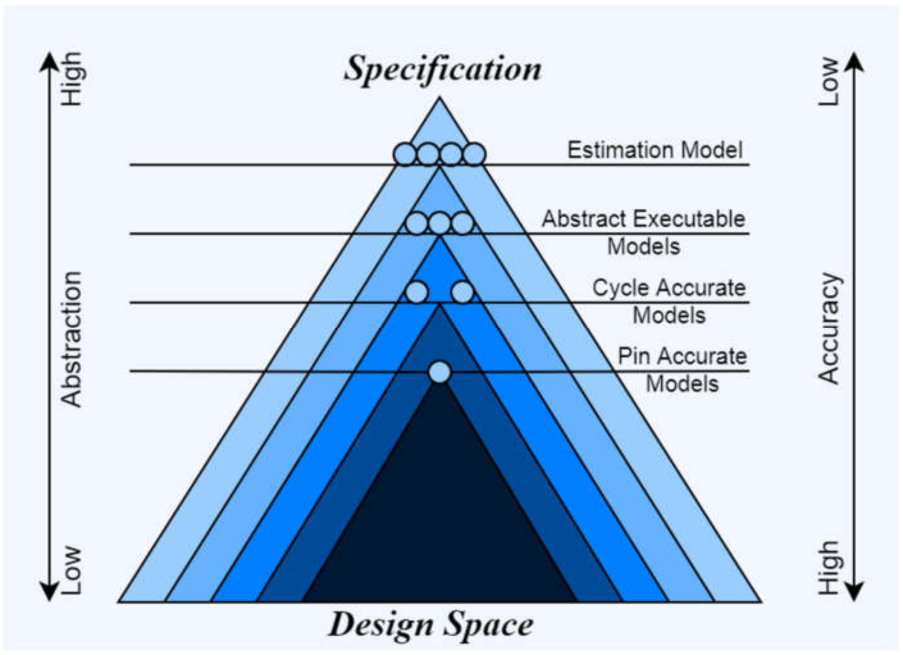


Figure 5: Pyramid of abstraction levels that comprise a system design from the specification to a possible optimal solution

2.1.4. Service-Oriented Architecture (SOA)

Service-Oriented Architecture (SOA), is a software design architectural pattern where a set of components can be invoked and provide a collection of services to each other using a communications protocol. Services can have self-contained functionalities and highly modular which can be swapped depending on the needs of the application, by using different implementation languages and platforms make easy the addition and development of features. Virtual platforms can be intuitively implemented using an SOA, where the functions of different virtual components, such as memories, pipeline, and networks on chip (NoCs) are provided by different services.

2.1.5. Simulator Evaluation Criteria

According to computer simulation literature resource, there are two types of classification of computer architecture simulation.

1. **Level of abstraction**

   Most resources classify computer architecture simulation by level of abstraction, where high-level abstraction models, which result in faster simulation at the cost of a loss of accuracy. Furthermore, these models facilitate analysis iterations around various architectural options as well as software execution, which gives the flexibility to explore more features than in a low-level abstraction model. Using the different level of abstraction by simulator exactly depend on detail demanded by the target system. Also, there are two sub-classification such as gate level and cycle level

2. **Model of simulation**

   Simulation model classifying is used for demanding of the target application. Where they need either the functional simulator or Full-system simulator or cycle-level simulator.

2.1.5.1. Characteristics of a simulator

Characteristics of a simulator presented in [8]

1. **Speed**

   The speed of simulator is measured by the number of instructions executed per run-time second for evaluation simulation model. While it takes to solve an instruction might seem at first peek to be a basic and very useful measure.

2. **Accuracy**

   The accuracy of the simulator is measured by both types of observational error, random and systematic errors which occur during the simulation process. Accuracy depends on the level of abstraction used in the simulator, low-level abstraction provide a more accurate model. Also, accuracy can be analyzed in different modes such as at per-cycle, per-instruction or per-function granularity, each having independent implications.

3. **Flexibility**

   The flexibility of simulation define the capability of the simulator to support different configuration and it can be modified to evaluate different target functionalities, also it provides the development and exploration of design efficiently.

4. **Completeness**

   Completeness defines in developing peripheral of the simulator and potential interaction with other components besides of full- system simulation and communication with another environment for developing and extending of the simulation model.

5. **Usability**

Usability feature relies on more aspect such as ease of setup, able to specify for certain tasks, contributing to a specific purpose or fit to using in tools and devices. While in this thesis I focus on effort and ease of setup.

2.2. Gem5

Gem5 is a modular platform that fabricated from a combination of the former full system simulator M5 and memory system simulator GEMS for system level design, architectural and performance exploration. As a side note, M5 provides a highly configurable event-driven simulation structure, multiple ISAs, and various CPU models. GEMS complements these features with a flexible and detailed and memory system, including support for complex and multiple cache coherence protocols and interconnect models. This combination makes Gem5 as Self-contained simulation framework which is built on a discrete-event simulation kernel, although you're welcome to glue things together.

Unlike other simulators, Gem5 dominate the limitation of the necessity of multi-core systems and complex cache hierarchies have presented architects with several aspects of exploration and present flexible simulation framework. In addition, Gem5 relying on high-quality open-source codes allows them to collaborate with their colleagues in both industry and academia.

2.2.1. Main Goal

Being a community tool focused on architectural modeling is the main goal of the Gem5 simulator. While that has three aspects in the following:

1. The flexibility of modeling to appeal to a wide range of users
2. Extensive availability and utility to the community
3. The expert developer interaction to collaborate

2.2.1.1. Flexibility

Flexibility is the main role in the success of simulation infrastructure. But achieving to high-level of flexible simulator needs to balance simulation speed and accuracy to a specific design. Fine-grain clock gate or coarse-grained model need a different variety of accurate CPU model, System modes, memory system and peripheral devices. The Gem5 simulator enables simulation system with a variety of capabilities and flexibility in components and peripheral hardware. These features allow covering a board range of speed/accuracy exchanges in multiple configurations.

2.2.1.2. Availability

Being compatible with a variety of users with different requirements and goals would be another challenge of each simulator and Gem5 simulator as an open source software is the perfect answer for each requirement.

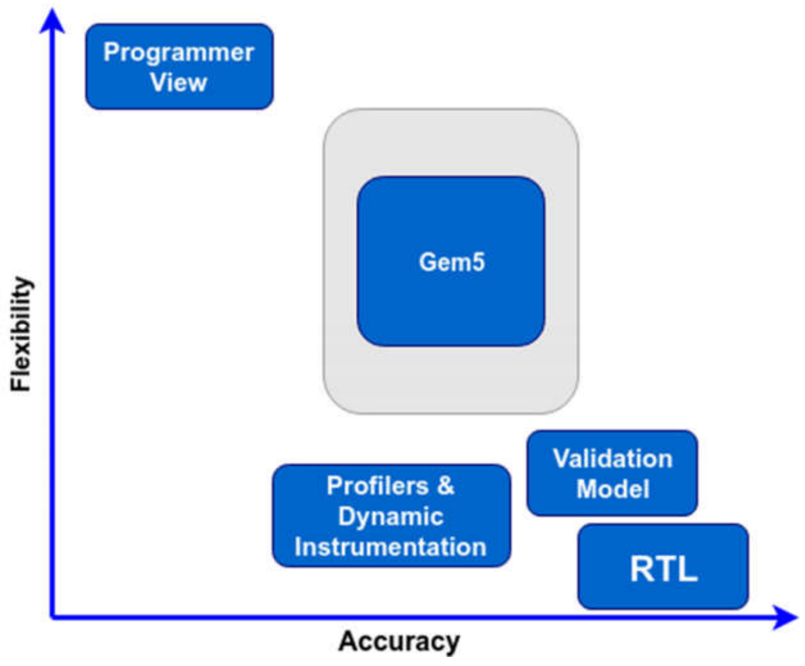


Figure 6: Gem5 in Accuracy & Flexibility

2.2.1.3. High level of collaboration

Currently promoting Gem5 include, mailing lists, wiki, web-based patch reviews, and a publicly accessible source repository is enabled to compatible with collaborative technologies effectively.

2.2.2. Licensing

The Gem5 simulator is released under a Berkeley-style open source license.

2.2.3. Platforms

The Gem5 simulator works on most operating systems. Though, all guest platforms aren't supported on all host platforms.

2.2.4. Tools

Gem5 are backed up by these tools and utilities (Git, GCC/G++ 4.8+ (or clang 2.9+), Python 2.7+, SCons 0.98.1+, SWIG 1.3.40+, protobuf 2.1+)

2.2.5. Compiling

Gem5 has 5 types of compiling target and making binaries for different architectural and flexibility purposes.

- **Gem5.debug:** It built in debugging and contain symbols, tracing, assert features
- **Gem5.opt:** It built in optimized and contain symbols, tracing, assert features
- **Gem5.fast:** It built in optimized and doesn't contain any symbols, tracing, assert features
- **Gem5.prof:** Same as fast binary target with profiling support
- **Gem5.perf:** Similar to the prof, this target is aimed at CPU and heap profiling using the Google Performance Tools.

2.2.6. System mode:

Gem5 has two fundamental system execution mode

2.2.6.1. System Call Emulation (SE):

This binary mode configures specific applications or set of applications on microprocessor typically it used by calling host OS and simplify address translation model with no schedule. (Running one or more applications by emulating sys-calls).

2.2.6.2. Full System simulation (FS):

Full System simulation FS mode configures whole of a system for booting OSs perfectly with preferred disk images like Linux or other mimics or just in the bare-metal model for implementing devices. It can simulate peripheral components and frame buffer in output and with a class of instructions include interrupt handling, timer control, I/O, exceptions, privileged instructions, fault handlers that can be executed only when the computer is in a special privileged mode that is generally available to an operating or executive system. (Booting an entire operating system).

Processor		**Memory System**		
CPU Model	**System Mode**	**Classic**	**Ruby**	
			Simple	**Garnet**
Atomic Simple	**SE**	**Speed**		
	FS			
Timing Simple	**SE**			
	FS			
In-Order	**SE**			
	FS			
O3	**SE**			
	FS			**Accuracy**

Table 1: Configuration trade-off between speed and accuracy

2.2.7. Multiple ISA support

Also, Gem5 supports multiple of operating system and Instruction set architecture as a flexible architecture simulator for full-system simulation and system-call emulation.

2.2.8. Gem5 capabilities

1. The conjunction full-system model provides instantiate multiple systems with just a single simulation process. This capability of full-system mode enables simulation of whole client-server networks.
2. Flexibility: Multiple CPU models, memory systems, and device models and thus multiple systems can be instantiated within a single simulation process.
3. Accuracy: Gem5 relying on multiple memory systems and CPU model Gem5 can be enough accurate.
4. The speed of simulation: Variety of simulation model makes Gem5 as much as fast compared to other system simulators.

ISA	Maintainer	Level of ISA Support	Full-System OS support	Test coverage	Toolchain availability	Linux kernel availability
ALPHA	None	High	Linux	Medium	Low	Low
ARM	Andreas Sandberg	High	Linux, BSD, Android	High	High	High
MIPS	None	Low	None	Low	Medium	Medium
POWER	None	Low	None	Low	Medium	Medium
RISC-V	Alec Roelke	Medium	None	Low	Medium	Medium
SPARC	Gabe Black	Low	None	Low	Low	Low
X86	Gabe Black	Medium	Linux, BSD,	Medium	High	High

Table 2: Gem5 supported Multiple ISA

2.2.9. Event

GEM5 has an event-driven simulation kernel framework which has diverse abstraction levels, that balance simulation speed, and accuracy. Event-driven can keep the memory overhead down, [9] therefore, using event scheduled mode on Timing accesses memory during process cause to synchronize and manage all requests by Global logical time in “ticks”. In every CPU clock cycles (system-call emulation) or Picoseconds (full system) Objects and CPU schedule their owned events at regular intervals.

2.2.10. Co-simulation with SystemC:

For two world communication, Gem5 can be coupled in a SystemC simulation, running as a thread inside the SystemC event kernel, and maintain the events and timelines synchronized effectively by using the transaction methodology. This feature enables the Gem5 components and SystemC functions to interoperate with a board range of System-on-Chip (SoC) component models, such as interconnects, initiators and target devices. Transaction Level Modelling (TLM) standard interfaces for SystemC provide interoperability and transaction between two environments. Gem5/TLM allows full interoperability between both simulation frameworks, then it facilitates a vast set of capabilities for system level design space exploration.

2.2.11. Gem5's base infrastructure

Here it defines some features related to use-cases.

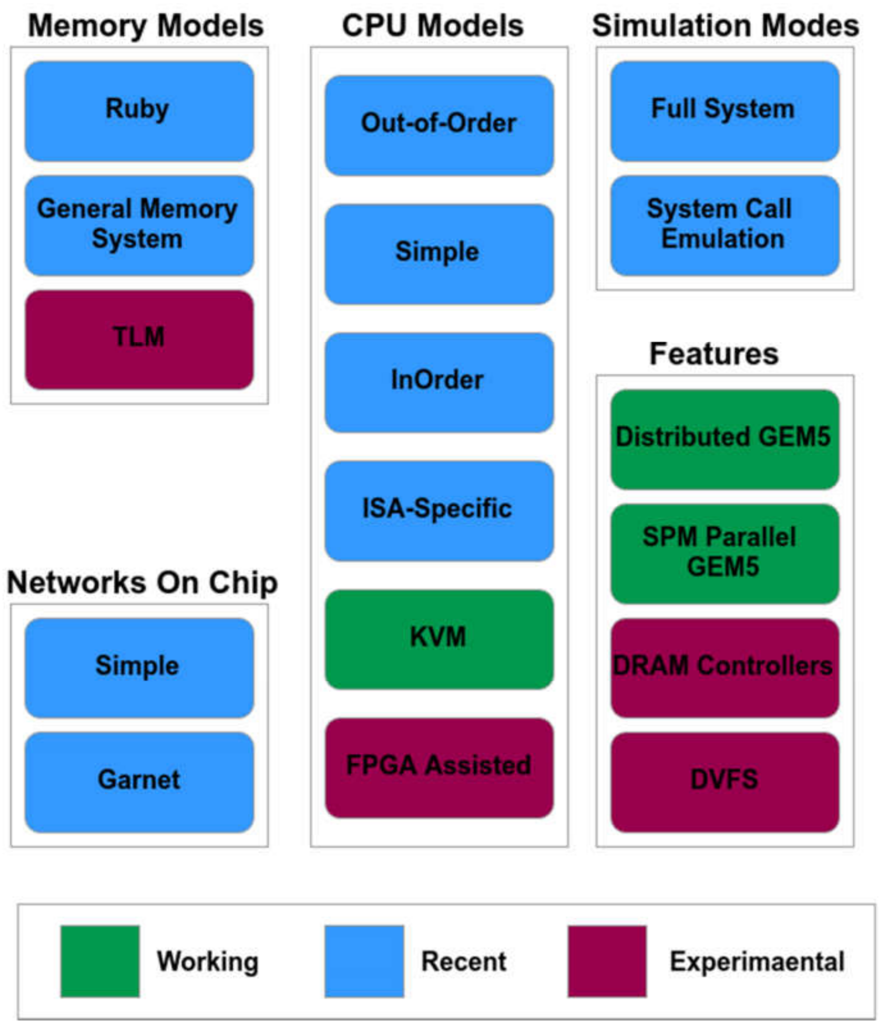


Figure 7: Summary of the main trending gem5 module

2.2.11.1.CPU models

2.2.11.1.1. SimpleCPU

When using a detailed model is not essential. The system can be used SimpleCPU as a fully functional and orderly model is assorted for the simulation system. This can contain warm-up periods, client systems that are operating or control a host system or only making sure that application works as a testing program. (High-level ISA)

2.2.11.1.2. BaseSimpleCPU

The BaseSimpleCPU can't be run by itself. This serves for maintain architect status, Defining functions for checking or fulfill the states like interrupts, fetch request, pre-execute setup or post-execute actions, and just advancing the PC to the next instruction.

2.2.11.1.3. AtomicSimpleCPU

The AtomicSimpleCPU is the version of SimpleCPU which uses atomic memory accesses. It defines the port that is used to hook up to memory and connects the CPU to the cache.

2.2.11.1.4. TimingSimpleCPU

The TimingSimpleCPU is the version of SimpleCPU that uses timing memory accesses. It defines the port that is used to hook up to memory and connects the CPU to the cache. Additionally, it defines the required functions for conducting the response from memory to the accesses sent out.

2.2.11.1.5. O3CPU

It defines a new detailed model which has following pipeline stages such as Fetch, Decode, Rename, issue/Execute/Writeback and Commit.

2.2.11.1.6. Checker

The checker provides dynamic verification of CPUs. Currently, it is not full support for all systems. (high-level ISA).

2.2.11.1.7. OutOrderCPU

Detailed model of an out-of-order CPU (high-level ISA).

2.2.11.1.8. KVM-based CPU

Kernel-based Virtual Machine CPU (Visualization to accelerate simulation)

2.2.11.2. Memory

2.2.11.2.1. Memory system models:

2.2.11.2.1.1. Classic Memory System

The Classic memory enables easy and fasts configurable memory system

2.2.11.2.1.2. Ruby Memory System

Ruby memory system, unlike the previous system, is a flexible and highly configurable memory system which implements a detailed simulation model for the memory subsystem. It designs complex cache hierarchies, interconnection networks, coherence protocol implementations, memory controllers and DMA, various sequencers that initiate memory requests and handle responses.

2.2.11.2.1.3. General Memory System

The General Memory System is a shared infrastructure between the Classic and Ruby models.

2.2.11.2.2. Memory Port system

2.2.11.2.2.1. MemObjects

`MemObject` is a class scripted in C++ and pure virtual functions which all objects that connect to the memory of Gem5 inherit from `MemObject`. Also, it includes a port corresponding to the given name and index. This interface is used to connect the *MemObjects* together.

2.2.11.2.2.2. Ports

Ports are interconnected components which used to interface `MemObject` to each other. They will always are in pairs (Master and Slave) and it refers to the other port object as the peer. The direction defines A master port always connects to a slave port, with the master initiating requests, and the slave providing responses. Every `MemObject` has to have at least one port to be useful.

There are two pairs of functions in the memory port object. Frist is sent and second is receive. The object belongs to Master port call by `send*` functions on the port. As an instance a request packet data in the memory system a CPU call `myPort->sendTimingReq (pkt)` to send a packet data. Another one is `recv` function belongs to the Slave port that which defines the corresponding of each `send*` function that is called on the ports peer. So the implementation of the `sendTimingReq ()` call above get simply `peer->recvTimingReq (pkt)`. Using this mechanism makes only one pure virtual function call penalty while it retains generic ports which can connect any `MemObject` together.

2.2.11.2.2.3. Connections

The ports connection are symmetric that is no difference between `IN.port1 = OUT.port2` and `OUT.port2 = IN.port1` contrary a normal variable or parameter assignment, the connection of the port is part of the configuration of system and script in Python which attributes of `SimObject` such as `Params` function. Two objects can define on their ports should be connected using the assignment operator. Thanks to using "*Vector ports*" the objects have a possibly unlimited number of ports which an assignment to a vector port adds the new peer to a list of connections rather than overwriting the last connection.

2.2.11.2.3. Packets

Gem5 uses a specific transfer mechanism which encapsulates data in Packet and transfers between two `Memobjects`. This is against to a Request whereas a single Request transit from a requester to the ultimate destination and back, possibly being transferred by several different Packets along the way

Accessor methods allow Read access to the many packet fields while it verifies the data in the filed being read is valid.

A packet contains the following features:

- **Address -** The memory location that is used to conduct the packet to own target (If it does not allocate) and to process the packet at the target. Normally it allocates a physical address for the request object. But virtual address may use in the specific situation. It may not be same as the original request address due to address translation.

- **Size –** The allocation address in memory has size and it is able to modify also the size may not be the same as that of the original request, as in the cache miss scenario and different location.

- **Pointer -**A pointer to the data control manipulation of data in the packet. Which sets by `datastatic ()` and `dataDynamic ()`, also the pointer use `allocate ()` function in order to allocate the packet into the address and freed when the packet is destroyed. In addition pointer can retrieve by calling `getPtr ()` or `getConstPtr ()`. Eventually, The `get ()` function execute a guest-to-host endian conversion and the `set ()` function execute a host-to-guest endian conversion.

- **Attributes of packer –** There are a list of Packet Command Attributes related to the packet

- **State of pointer -** A `SenderState` pointer is a virtual structure that is used to maintain state associated with the packet and return the state to the packet's response.
- **Pointer to the request**

2.2.11.2.4. Request

CPU or I/O devices issue the request object by encapsulating the original request. While this request's parameters are continuous during the whole of the transaction. Thus a request object's field is able to write more than once for a given request. There are few of constructor and update value in a request filed which enables to be written at different times. Accessor method provides read access to all request filed when it verified the data in the field being read valid. The fields in the request object are normally used just for statistics or debugging.

Request object fields include:

- Virtual address – If the request was issued directly on the physical address. This field will become invalid. (e.g., by a DMA I/O device).
- Physical address.
- Data size.
- Time the request was created.
- The ID of the CPU/thread that caused this request. If the request was not issued by a CPU this filed will become invalid too (e.g., a device access or a cache write-back).
- The PC that caused this request. Also if the request was not issued by a CPU this filed maybe invalid.

2.2.11.2.5. Packet allocation protocol

In fact, access type determines the protocol for allocation or de-allocation of the Packet objects in `Memobject`. This protocol differs widely with coherence protocol. There are three different memory system modes in the implementation of ports that associated with access types in the following:

2.2.11.2.5.1. Atomic and Functional

The Atomic and Functional transaction function generate packet object in requester by calling `sendAtomic ()` or `sendFunctional ()` in both Master and Slave port, Also return `recvAtomic ()` or `sendFunctional ()` on the peer port. The other side responder using the `Packet::makeResponse ()` method in order to take a request packet and modify then overwrite it in place to be suitable for returning as a response to that request. The requester is able to allocate the Packet object on Stack or statically. The Atomic mode is suitable for fast-forwarding and special situation simulation and warming up the simulator. Also, the functional mode is debugging mode.

2.2.11.2.5.2. Timing

Timing mode generates correct simulation results compared to atomic and functional. Which use two simple message for the transaction, a request, and a response. Unlike previous modes, timing mode allocates Packet object by the sender dynamically. These capabilities allow to manage memory allocation as sender responsible and deallocation as receiver responsible, it may select to reuse the request packet for its response to save the overhead of calling operators to `delete` and then `new` (and gain the convenience of using `makeResponse ()`). The `Packetptr` as a parameter of ports has the main role in the interface of all memory system object. The packet is the request or response to send or receive by using the `sendTimingReq` function for sending a packet which is called on the Master then receive a packet in Slave port by calling `recvTimingReq` and return value of `true` or `false` for allowing using retry mechanism or not. In the case `recvTimingResp` function in master side returns `true`, that is the return value of `sendTimingResp` in the slave. This transaction for the request is complete else slave does not accept the request because of issue or being busy and returns `false` if so as mentioned slave port use retry mechanism by calling `sendReqRetry` and `sendReqRetry` on the master port. It is clear Master port is on CPU side and Slave port is on Memory side or cache.

2.2.11.3. Access Types

The Ports support three types of access (transport interfaces):

2.2.11.3.1. Functional

This interface method is used for loading binaries, debugging, introspection, etc. It completes the transaction in a single function and requests complete before function returns. This method is similar to the debug method in TLM 2.0

2.2.11.3.2. Atomic

It complete transaction in the single function call and Requests complete before the function returns approximate latency without contention or queuing delay. It used for loosely-timed simulation (fast forwarding) coding style. This method is similar to blocking transport method in TLM 2.0

2.2.11.3.3. Timing

This access method provides a model for realistic timings of real memory systems. They use an asynchronous protocol where responses are not instantaneous. This method is Timing accesses in Gem5 are similar to non-blocking transactions in TLM in the sense that they split a single memory access into request/response phases and use forward/backward paths for communication. However, Gem5 and TLM employ a different mechanism for enforcing back pressure. While TLM defines exclusion rules, Gem5 uses the retry mechanism.

2.2.11.4. System Configuration

Gem5 simulation script has two main phases: a configuration phase, and a simulation phase

- Configuration phase: instantiation simulation system is determined by construction `SimObject` and interconnection between its component and parameter which a hierarchy of Python simulation objects;
- Simulation phase, the actual part of system simulated derived from C++ classes and objects whether is `SimObject`, `Membject` or other devices and components.

Both of these phases are controllability via command-line

In order to instantiate the simulation, Gem5 provide the simulator objects so-called `SimObject` such as CPUs, caches, memory, peripheral devices, and buses, etc. Which all `SimObject` has built-in C++ scripts and then export to Python. Then, the Python configuration script interacts with Gem5 simulation core, while the interaction process provides by the Gem5 module is started in Python then translate into C++ function calls. Then in order to create configuration script in python from `SimObjects` needs to set system parameters and determine the interactions between `SimObjects` with choosing any valid Python code in the configuration scripts. This script is able to set up and execute the simulation. Eventually, in order to start the simulation, the instantiation process passes through all of the `SimObjects`. [10]

Gem5 is a complete simulation framework that provides specific models. For a variety of system components and means for easy configuration. [1]

To simulate fast avoiding complexity and just configure selective and necessary components of Gem5 for accuracy, It executes system-call mode While it can simulate full system easy by means of available configuration script also It is necessary to have extensive knowledge about the hardware system. ARM ISA as one of the most popular ISAs used in Gem5.

Most of the complication in setting up configuration files come from creating the system object (`SimObject`). For creating new Python objects, which inherit from own ARM class in Gem5 confirmation libraries.

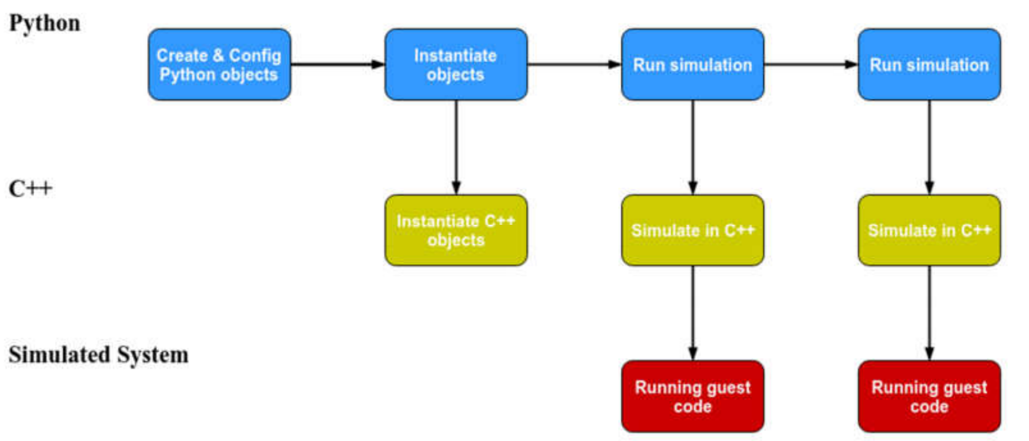


Figure 8: Configuring and Running Gem5

2.2.11.4.1. SimObject

One of the most powerful parts of Gem5's Python interface is the ability to pass parameters from Python to the C++ objects in Gem5 module. Therefore All major Gem5 components are `SimObject` which defines as simulation object that built-in C++ classes it includes the `SimObject`'s state, behavior, and performance-critical simulation model and are accessible from the Python configuration scripts contains the component's parameters and specifications and used for script-based configuration and share a common configuration. In term of the Gem5 module, `SimObjects` include both concrete and abstract models of hardware components such as cores, caches, NoCs, pipelines, and memory controllers. [11]

`SimObjects` can have many parameters which are collected in Python configuration files. The diversity of parameters and allows creating simple or complex simulation system close to the real machine.

2.2.11.4.1.1. Creating a SimObject [12]

In order to create the `SimObject` for system configuration purpose, there is two main instruction which in different simulation phase. First of all, it must derive from the Python class form Python `SimObject.` At this point, it is necessary to define Python parameter, ports, and configuration where parameters in Python are automatically turned into the C++ structure and passed throughout C++ `SimObject`. Then append this Python filed created to `SConscript` or implant it in an existing `SConscript` file in related root. Second it requires to derive C++ class from C++ `SimObject`. In this case, it necessities to describe the simulation behavior and states. Also, it can use a documented library of C++ files in Gem5 sources. Then same as configuration phase, it must add the C++ filename to `SConscript` in the directory of created `SimObject`. Eventually, it needs to make sure you have a create function for the object. The figures 8 and 9 depicts the situation of the Python and C++ parameters in configuration and simulation phases.

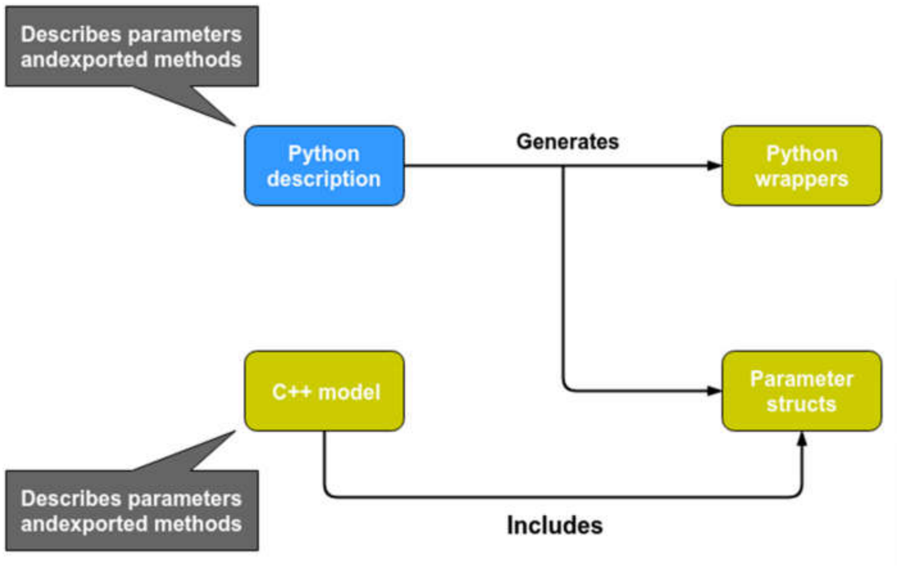

Describes parameters andexported methods
Python description
Generates
Python wrappers
C++ model
Parameter structs
Describes parameters andexported methods
Includes

Figure 9: Implementing the model into the Gem5

2.3. SystemC

Complexity in hardware architectures has forced companies to look for new development tools and approaches, The Open SystemC Initiative (OSCI) is a formed standardize to support and advance SystemC for system-level design as a set of the class library in C++ which is interoperable for System on Chip modeling platform and allows the exchange of IP models in order to system-level design and functional verification. SystemC also offers the event-driven simulation interface for design and concurrent hardware and software development. These facilities make access easy for system designers to digital simulation models and prevent expensive and catastrophic failures. [13]

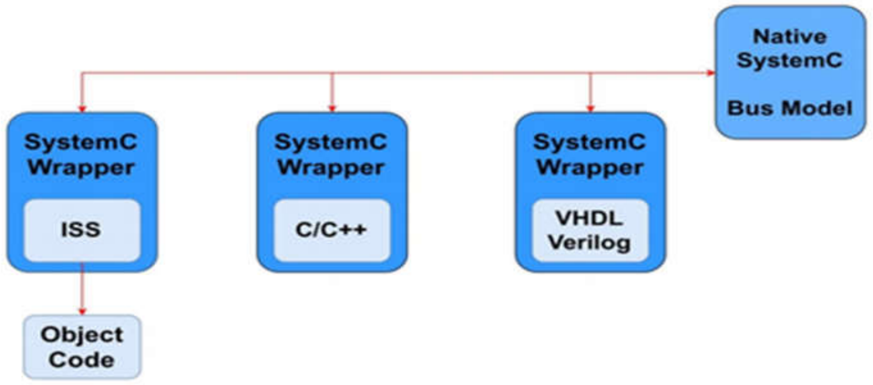


Figure 10: Communication-centric & Mixed Abstraction level in SystemC

SystemC can be used by developers in order to elaborate high-level synthesis (Electronic system level) and simulate concurrent process furthermore enables a huge set of possibilities for Architectural exploration and Performance Modeling. SystemC concentrate on the functionality of the system instead of the structure of system due to avoid to cyclomatic complexity. This means that it is possible to concentrate on the actual behavior of the system more than on its implementation details. But detailed implementation can be changed during any alternative of system architecture.

Virtual platforms can provide software developers with a development platform before the building real machine. Virtual platforms have been integrated into a SystemC Analog/Mixed-Signal-based simulation and verification environment.

2.3.1. Importance of SystemC

Some of the important virtual prototypes are functional software models of physical hardware which based on SystemC, Furthermore, they provide controllability over the entire system and offer powerful debugging and analysis techniques.

Virtual platforms can provide software developers with a development platform before silicon being available. In addition, virtual platforms have the capability to be reused in future projects and hardware engineers able to fast design space exploration, that means they can try out different things like how big is system cache or what is the number of cores should use or exchange easily.

While ago, building FPGA prototyping board and transfer to another country this is more complicated but virtual prototype is just software that hardware engineers can email. Eventually, virtual platforms have been integrated into a SystemC AMS-based simulation and verification environment.

2.3.2. SystemC analogy

SystemC as a language of programming can compare with other hardware description languages like Verilog or VHDL. These type languages only use in Register Transfer Level (RTL) descriptions, hardware verification and using logic gates while systemC exert on system-level modeling particularly virtual platform, software development, functional verification, architectural exploration, and high-level synthesis. Following figure illustrate capabilities of the systemC among the other HDLs

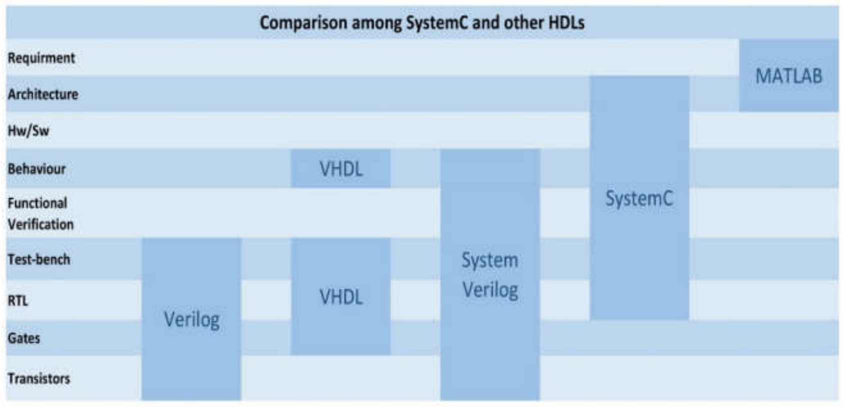


Figure 11: Comparing SystemC & other HDLs

2.3.3. Advantages and disadvantages

SystemC as a language for modeling system has many capabilities such as easy integration in molding product or any software production in systems, more flexible with respect to platforms in the simulation of modules and architecture exploration, classify each module in order to convenient access of each model. Unlike other alternatives, SystemC has written in single programming language C++ that leads to compile faster and convenient understandably by designers. SystemC has some drawbacks in primitive version such as poor support for asynchronous events and necessity of kernel support but they have solved in the present version. SystemC as an alternative modeling platform can solve several challenges in system-level design. All in all leading companies use SystemC to create a wide range of environment for system-level design and verification. Many semiconductor companies in the SystemC community have participated at public events where they described their use of SystemC. Many systems and semiconductor companies currently become a member of systemC community. For example

IBM, Intel, ARM, Cadence, STMicroelectronics, Canon, NEC, Infineon, Motorola, Nokia and etc.….

2.3.4. Features and constructions:

Communication mechanism in systemC defines essential building constructs such as modules, interfaces, ports, exports, and channels to be used in TLM structural to represent hierarchical structures and communication between blocks and models. In this regard, SystemC also presents signals, processors in synthesizable model and events that allows synchronization a between processes.

2.4. TLM 2.0

SystemC for more interoperability between various model needs to effective and detailed functional communication. The TLM-2.0 transaction-level modeling standard from the Open SystemC Initiative was published as abstraction communication instead of abstraction structure among modules and digital systems. Transaction point to encapsulate communication for interaction between components. Modeling in transaction level focused on functionalities help designer to develop virtual prototypes applications and simulate hardware and software in different abstraction levels to optimize performance modeling. Communication mechanism in TLM is implemented modules and ports which are using digital signals and blocks through the channels and interfaces classes. Therefore high Interoperability between models in TLM affects directly on possibilities of system design and Architectural exploration. These attributes enable TLM 2.0 to become a golden model for hardware functional verification. [14]

2.4.1. Interface and blocks in TLM

2.4.1.1. Initiator

The module that initiates new transactions and flows to interconnector or directly to target (CPU)

2.4.1.2. Target

The module responds to transactions initiated by other modules (Memory).

2.4.1.3. Sockets

A socket is an internal endpoint for sending or receiving data (generic payload) at a single node inside or outside system simulated.

2.4.1.4. Transaction

A transaction is a data structure (a C++ object) passed between initiators and targets using function calls.

2.4.1.5. Generic payload

The most significant part of TLM is generic payload because it is template argument to gain interoperability between transaction level models. It encapsulates low-level details in a set of attributes such as address, command, status, and other information related to data package in the transaction cycle.

2.4.1.6. Base protocol

For using of generic payload TlM specifies protocol call base protocol which its standard form of highest level of interoperability among the models.

2.4.1.7. Interconnector

Interconnect device is a general form of bridges between two blocks and allows the flow of data packages and address from one to the other.

2.4.1.8. Router

The router is a transaction device that forwards data packets (Generic payloads) between many blocks.

2.4.1.9. Arbiter

An arbiter is a transaction device used in multi-initiator blocks to select which initiator will be allowed to control and send the Generic payload to the target. The most common kind of arbiter is the memory arbiter in a TLM framework.

2.4.1.10. Bus

The bus on the TLM is referred to a communication system that can move data and commands quickly between the initiator, target and other system peripherals

2.4.2. Interoperability

In addition to Speed of simulation, TLM has another significant capability named interoperability which enables TLM to communicate interoperable API for memory mapped bus modeling and between different models. So TLM avoids adapters where possible.

2.4.2.1. Interoperability layers

Interoperability layers consist of:

- Core interfaces TLM to API
- Sockets
- Generic payloads
- Base protocol (collection of rules)

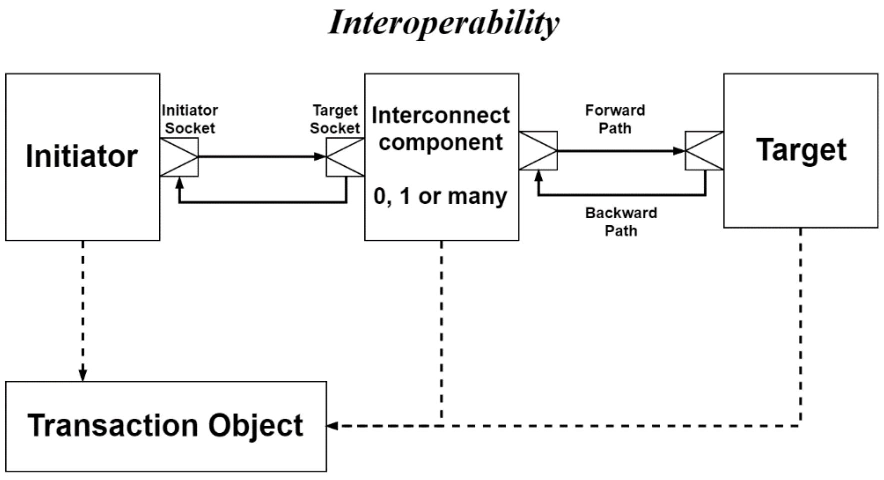


Figure 12: TLM Components Interoperability

2.4.2.2. Interoperability has two different types

2.4.2.2.1. Interconnect

It uses a standard form of protocol call Base-protocol which cause generic payload can ignore extension. Nevertheless Functional incompatibilities still possible.

2.4.2.2.2. Adaptor

It implements new protocol contains Generic payload with an extension which it can't bind sockets to differing protocols. Whereas extension mechanism in generic payload exploited for ease of adaptation reason, generic payload with and base protocol still exploited for acceptance of coding style.

2.4.3. TLM 2.0 vs RTL

RTL is an abstraction level detail of hardware design abstract of the structure of design (logic and timing) which describe register and combinatorial function between those registers any abstract timing by just dealing with state changes or clock cycle boundaries and it needs an event on each individual signal connection those two blocks together. RTL also abstract timing – clock cycle boundaries. While TLM is an abstraction communication replace lots of pin wiggling with a single function, individual events and phases that constitute the communication between blocks in the digital system and instead deal with communication in more coarse-grained terms so a collection of individual events and the pinnacles occurring over some period of time can be abstracted. In particular point of TLM is simulation acceleration functionality is the best captured in a behavior model simply because that's the best match in term of simulation speed you can take and architectural exploration. This facility also enables TLM to boot fast enough software O/S in seconds. The RTL Simulate every event, whereas TLM is 10-100 times faster than RTL. Simulate. As a result of TLM accurate enough to stay in use Post-RTL.

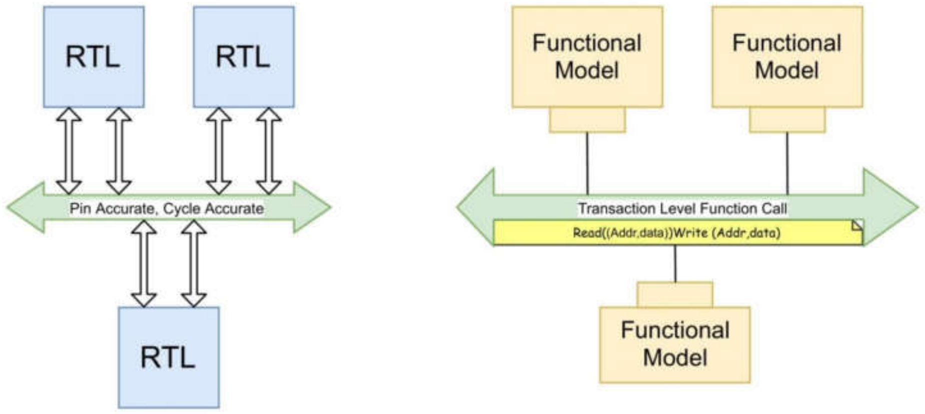

RTL
RTL
Pin Accurate, Cycle Accurate
RTL
Functional Model
Functional Model
Transaction Level Function Call
Read((Addr,data))Write (Addr,data)
Functional Model

Figure 13: RTL vs TLM

2.4.4. TLM Advantages

- Simulation speed.
- Interoperability.
- Capability to take different transaction level.
- Models from different sources and have played together.
- TLM based on architectural modeling for fast executable software development and establish for hardware verification test bench.
- Make easier for system level designer to experiment with a different bus architecture.
- Using of Multiple Processor, Software stacks, Bus Bridge in high level interoperability.

2.4.5. TLM Infrastructure and communication

Communication mechanism in TLM consist of three-layer as following

2.4.5.1. Core mechanism

Foundation layer is a set of mechanisms these are the definitive C++ API for TLM which using a variety of interface such:

2.4.5.1.1. Blocking interface

```
b_transport (TRANS &, sc_time &);
```

This method transaction complete in one call and just in the forward path (unidirectional) also Includes timing annotation and used with loosely- timed coding style (maximize simulation speed).

2.4.5.1.2. Non-blocking interface

```
tlm_sync_enum nb_transport_fw (TRANS&PHASE, sc_time&);

tlm_sync_enum nb_transport_bw (TRANS&PHASE, sc_time&);
```

this method allow multiple calls backward and forward (bidirectional) also Includes timing annotation and phases used with approximately-timed coding style (Accuracy simulation).

2.4.5.1.3. DMI (Direct Memory Interface)

This method is used for transactions direct access to main memory without the interference of CPU and non-intrusive read and write operations. It executes with zero delays, no context, and no side effects. This interface significantly reduces the overhead for simulating transactions.

2.4.5.1.4. Phases

It defines the state synchronization state values between an initiator and a target in non-blocking transport. The base protocol defines 4 phases.

```
(BEGIN_REQ, END_REQ, BEGIN_RESP, END_RESP)

BEGIN_REQ ----------- Beginning of request phase

END_REQ ------------- End of request phase

BEGIN_RESP --------- Beginning of response phase

END_RESP ------------- End of response phase
```

2.4.5.1.5. Generic payload

Previously explained

2.4.5.1.6. Quantum keeper sockets

TLM 2.0 enables a utility class to keep track of their thread global quantum, local quantum and a local time offset.

2.4.5.2. Coding Style

2.4.5.2.1. Loosely Time (LT): *FAST*

Using LT makes simulation fast enough for sufficient timing detail to boot O/S and run Multi-Core systems in Block transport (`b_transport`) method and still need to give each initiator a run and model timer and interrupt. Also, Processes can run ahead of simulation time (temporal decoupling) by way of explanation Allowing SystemC processes to run ahead of simulation time by doing and so minimize the amount of context switching the occur during SystemC simulation that has a significant impact on simulation speed (*Trick1*). Transaction in LT coding style completely just a single function call with minimal timing information just, of course, contribute to speed in addition LT make direct access to an area of memory in the target bypassing the transaction mechanism and therefore further increasing simulation speed. (*Trick* 2)

2.4.5.2.2. Approximately Time (AT): *ACCURATE*

2.4.5.2.2.1. Accurate

Using AT make simulation accurate in order to each process runs at a specific simulation time, according to when activity happens in the target system and has enough timing information for architectural exploration performance modeling. It uses `non-blocking` transport method for the transaction and each transaction has 4 timing points (*extensible*) and allows models from a different source to play to gather. In addition, transactions are annotated with delays which cause the future event to be scheduled. AT coding style processes run in lock-step with simulation time and call wait or notify to consume time. Totally AT coding style accurate enough for performance modeling and sufficient for architectural exploration. AT coding style also known as cycle-approximate or cycle-count accurate.

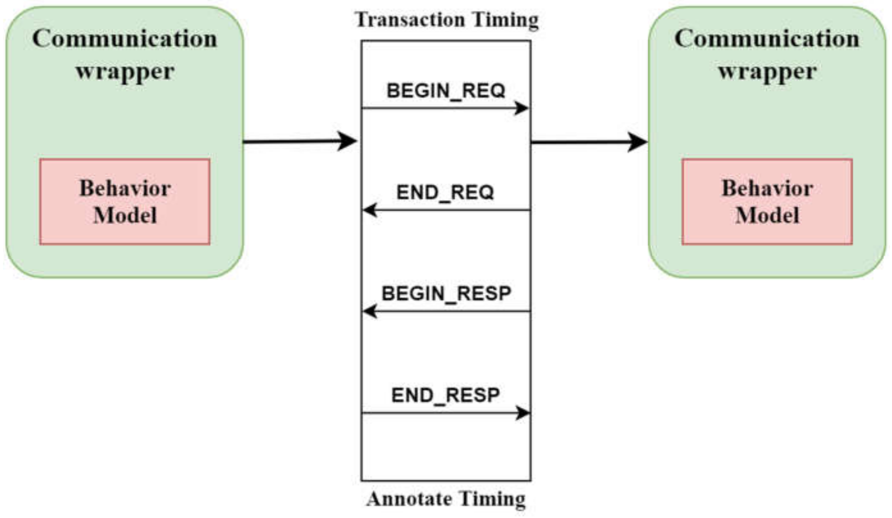


Figure 14: Approximately-Time mechanism

2.4.5.2.2.2. AT in three Steps

1. AT Makes use of TLM core interfaces and sockets.

2. Generic Payload catches a set of attributes that a typical of memory-map buses and it pass through the sockets between the various DM components.

3. The base protocol defines a set of phases making the beginning at the end of the request in a response and also captures a set of rules that are used by initiator and target when making function calls through those standard sockets and passing the standard generic payload backward and forward.

2.4.5.2.3. Concurrent simulation Environment

It is a type of simulation that various simulation in a different part of system simulates concurrently instead of sequentially. In another word varies process are executed in overlapping time without waiting for other computation to complete. This capability is used in new simulators or when using the complex system with mesh and network

2.4.5.3. Services

AT coding style based on its specification serves some capabilities like

- Software Development
- Software Performance
- Architectural Analysis

- Performance Modeling
- Hardware Verifications

2.4.6. Interface method

2.4.6.1. Forward

1. b_transport ( )
2. nb_transport_fw ( )
3. get_direct_mem_ptr ( )
4. transport_dbg ( )

2.4.6.2. Backward

1. nb_transport_bw ( )
2. invalidate_direct_mem_ptr ( )

2.4.7. Initiators, Targets, and Sockets

TLM-2.0 use two type of module for transaction an initiator and target. The transaction process begins with the initiator module and conduct to the target module that responds to the transaction which initiated by other modules. Actually, the transaction is a data structure based on the C++ object and included some attributes transfer between initiator, target and interconnect component by using function calls. Some module can act as initiator and target at the same time. These modules called Interconnect component and formed in such as arbiter, router or bus. An interconnect component does not modify or manipulating the transaction. Another important component which places for each module is a socket. In fact, socket used by the initiator as initiator socket for sending transaction toward the target socket whether target socket owns to target module or buses. Target socket receives incoming transaction. An interconnect component has both a target socket and an initiator socket in term of service to other modules.

2.4.8. Virtual Memory management

Virtual Memory management implements the `tlm_mm_interface` interface with storing in IEEE Std 1666-2011 clauses 14.5 and 14.6. Memory management as a part of CPU can handle of mapping between physical pages in actual memory and their association in the standard addressing schema with sharing memory pool or re-use transaction objects by Memory management utility. It also Avoids to construction and destruction of generic payload (*gp) transaction objects because of the implementation of the extension mechanism. End-of-life: In approximately-timed coding style use transaction objects in several times during the forward and backward cycle. So memory management can control the lifetime of transaction object through calls to acquire and release commands in generic payloads (*gp). Memory management utility performs with Allocating of memory for generic payloads during the back-and-forth transaction and then freeing of memory for generic payloads at the end of the transaction.

2.4.8.1. Memory management in the Interface method

2.4.8.1.1. nb-transport

Memory management (mm) must be set reference count with the non-zero object and it not reallocated while in-use (fixed)

2.4.8.1.2. b-transport:

There is no any Memory management (mm) for `b_transport` whatever sets extension should also clear it.

Chapter 3

Experimentation and analysis

In this section, it will be discussed about the implementation and experiment of two communication mechanism and assess aspects.

3.1. TLM-TLM mechanism

It this section I present a single bus model can make bidirectional connection two environment so-called transactions that enable accurate communication between blocks in the digital systems.

3.1.1. Module classes

Defining the macros `SC_INCLUDE_DYNAMIC_PROCESSES` which use in specific part of TLM in particular simple sockets for creating Initiator and target sockets which spawn dynamic processes.

```
 class Initiator: public sc_core::sc_module

 {.....};

 class target: public sc_core::sc_module

{.....};
```

3.1.2. Top-level module instantiation

1. **Initiator**
2. **Interconnect: Bus, Router, Arbiter**
3. **Target**

```
SC_MODULE(Top)
{
  Initiator *initiator; /// CPU
  Target    *target;
  SC_CTOR(Top)
  {
    initiator = new Initiator("initiator");
    memory    = new Target   ("target");

    initiator->socket.bind( target ->socket );
  }
};
```

Each component must bind by sockets in order to transactions propagating through an interconnect component (bus) placed between the initiators (CPU) and target (memories). The sockets encapsulate data between modules in forward and backward path Initiator socket is bound to target socket. An initiator socket is actually a `sc_port` that has a `sc_export` on the side, whereas a target socket is actually a `sc_export` that has a `sc_port` on the side. An initiator socket is `sc_port` and a target socket is a `sc_export`. The bind operator binds port-to-export in both directions by the single function call.

```
Sc_main function:
int sc_main(int argc, char* argv[])
{
  Top top("top");
  sc_start();
  return 0;
}
```

3.1.3. Initiator and Target modules classes

Namespaces `tlm` and `tlm_utils` introduce TLM 2.0 in C++. `tlm_utils` namespace is created for productivity and convenience and essential for interoperability

```
class Initiator: public sc_core::sc_module
{
    private:
        .....
        ..... .....
    public:
         tlm_utils::simple_initiator_socket<Initiator>
socket;
          SC_CTOR(Initiator) : socket("socket")
          {
                 .....
           }
..... .....

Class Target: public sc_core::sc_module
{
    private:
        .....
        .....
    public:
         tlm_utils::simple_target_socket<Target> socket;
          SC_CTOR(Target) : socket("socket")
          {
                 …
           }
```

These utility classes derived two simple sockets.
The Initiator communicates with the target using `non_blocking` transport method interface. As mentioned above the `nb_transprot` method is two-way communication which is more accurate and cycle-approximate then register callback in the backward path for incoming interface method calls in the initiator and register callback in the forward path for incoming interface method calls in the target.

```
socket.register_nb_transport_bw (this,
                            &Initiator:: nb_transport_bw);
....
socket.register_nb_transport_fw (this,
                            &Target:: nb_transport_fw);
.....
```

defining the `peq_with_cb_and_phase.h` header from `tlm_utils` for `nb_transport` which use in specific part of TLM in particular non-block transport in order to pending an event queue with code blocks and its phase for communicating backward and forward path with waiting in Initiator and target sockets which spawn dynamic processes.

3.1.4. Generic payload

The Generic payload is an underlying part of TLM-2.0 standard due to make interoperability property between transaction level models.

Using a generic payload has two main reasons:

1. General-purpose input/output transaction type for abstract memory-mapped bus modeling. (no detail of bus protocol)
2. The base for modeling a board range of specific protocols at a more detailed level.

There is a standard set of attributes contained in generic payload: command, address, data, pointers, byte enables, streaming width, response status, DMI (direct memory interface)

1. **Command:** the Generic payload support two main commands (Read & Write)
2. **Address:** The address value allocate for commands
3. **Data pointer:** It points to data buffer to the initiator.
4. **Data length:** It gives the length of the data array.
5. **Byte enables:**
6. **Streaming width:** It defines the width of a stream burst
7. **Direct memory interface (DMI):** The DMI represent direct memory interface method.
8. **Response status:**

This attribute must always initiate to the value of `TLM_INCOMPLETE_RESPONSE.`

3.1.5. Phases

The TLM non-blocking transport method includes four phases: `BEGIN_REQ`, `END_REQ`, `BEGIN_RESP,` and `END_RESP`. which present beginning or finishing the request and response Additionally, two functions make communication generic payload in the initiator and target in two different directions:

`nb_transport_fw()` & `nb_transport_bw()`

Which explained above. Both functions receive a phase and a timing annotation as additional arguments.

Three Enumerator base on the base protocol in communication in mentioned functions:

```
TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED
```

3.1.6. Interconnect classes (BUS)

Here is the definition of BUS in the nb_transport method: The Bus has to check out the attribute of each address to determine which socket (initiate or target) should send the transaction out through, target_nr define a number of the socket in the target that it is appropriate for the forwarded transaction. Finally, nb_transport return TLM_COMPLETED then nb_transport_bw replace origin address and return targ_socket.

3.1.7. Utility: Memory Management

1. It uses TLM::tlm_mm_interface for defining Memory Management class contains two main objects allocate & free which allocating memory for "gp" and free memory when the transaction terminates.
2. Simple Memory Management class contain numOfAlloc and numOfFrees function in public for this operation.

```
class MemoryManager: public tlm::tlm_mm_interface
{
  typedef tlm::tlm_generic_payload gp;

public:
  MemoryManager() : numOfAlloc(0), numOfFrees(0){}
  gp* allocate();
  void  free(gp* payload);

private:
  struct space
  {
    gp* payload;
    space* next;
    space* prev;
  };
  space* numOfAlloc;
  space* numOfFrees;
};
MemoryManager::gp* MemoryManager::allocate()
```

```
{    ...    }
void MemoryManager::free(gp* payload)
{    ...    }
```

3.1.8. Top level

Considering that in this example it uses a bus as inter-connector then the Top-level module instantiates multi initiators, a bus, and multi targets. For this purpose Top-level module contains an array for defining initiators and targets. So bus must bind between `init_socket` and `targ_socket` as shown below:

```
SC_MODULE(Top)
{
  int n=4;
  Initiator* init[n];
  Bus*       bus;
  Target*    target[n];
  SC_CTOR(Top)
  {
    bus   = new Bus("bus");
    for (int i = 0; i < n; i++)
    {
      char txt[10];
      sprintf(txt, "init_%d", i);
      init[i] = new Initiator(txt);
      init[i]->socket.bind( bus->targ_socket );
    }
    for (int i = 0; i < n; i++)
    {
      char txt[10];
      sprintf(txt, "target_%d", i);
      target[i] = new Target(txt);
      bus->init_socket.bind( target[i]->socket );
    }
  }
};
```

Currently, the virtual platform already prepared and it can use in the test bench. For testing this platform It can make signals for writing in initiators and reading from targets. While It uses the random function as a simple test, as shown in the following:

```
int sc_main(int argc, char* argv[])
{
  Top top("top");

  sc_start();

  return 0;
}
```

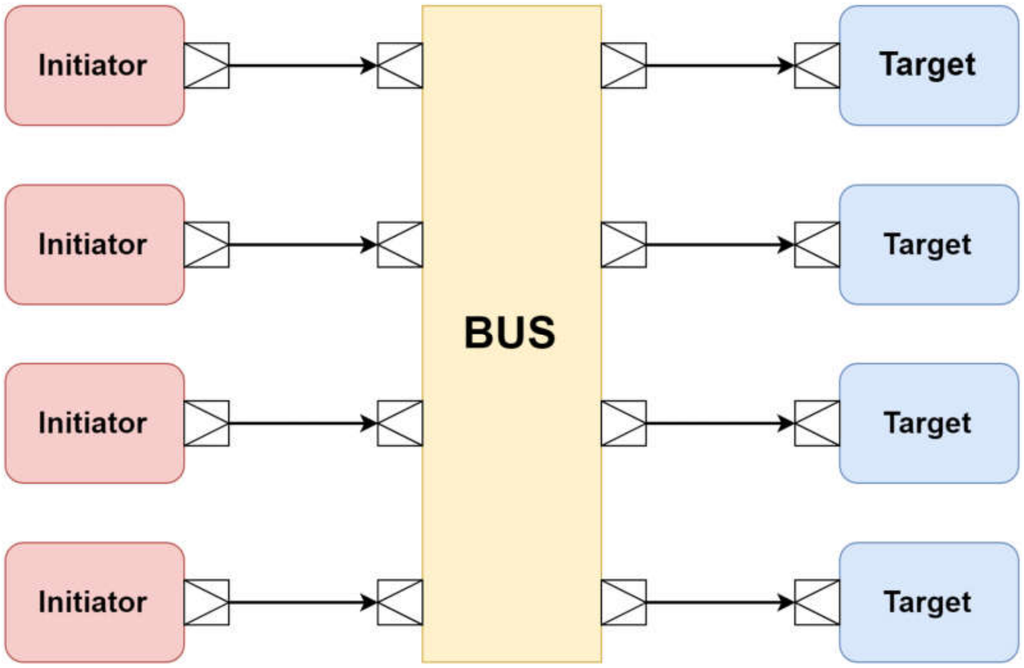


Figure 15: Non-blocking Trans, mem-management through the BUS interconnect

3.2. Gem5-TLM mechanism

In this section will present a couple Gem5 with SystemC and then both Co-simulation environment

3.2.1. Coupling Gem5 with SystemC

Actually, Gem5 already supported the coupling to the SystemC kernel accordingly, Gem5 is an event-driven simulation kernel and some objects models hardware and by compiling Gem5 as a library and using same logic to SystemC project and then It has a SystemC simulation kernel and Gem5 logic and It needs something that couples these two so this is why module was introduced SystemC module `Gem5SimControl`.

SystemC introduce a module called Gem5simcontrol for coupling two environments which support codes and pools of each side. The object in SystemC implement the Gem5 event queue and basically, it replaces the Gem5 event queue with the hook to the SystemC event queue.

By scheduling an event on Gem5, It will get a SystemC event which will be scheduled to be executed driven body Gem5 Object. Although it could hook Gem5 to the SystemC kernel, It is still not able to communicate with other SystemC modules, because there is no way to communicate between two environments. So it needs to work to TLM for the transaction, in order to make the system capable of plugging in any other SystemC model and hook it to Gem5 modules. The CPU is simulated in the Gem5 world and there are transactors that are connected to the memory which is simulated in the SystemC world. The following figure depicts an overview of the communication of memory objects in Gem5. The similarities of the communication mechanisms in Gem5 and TLM are evident.

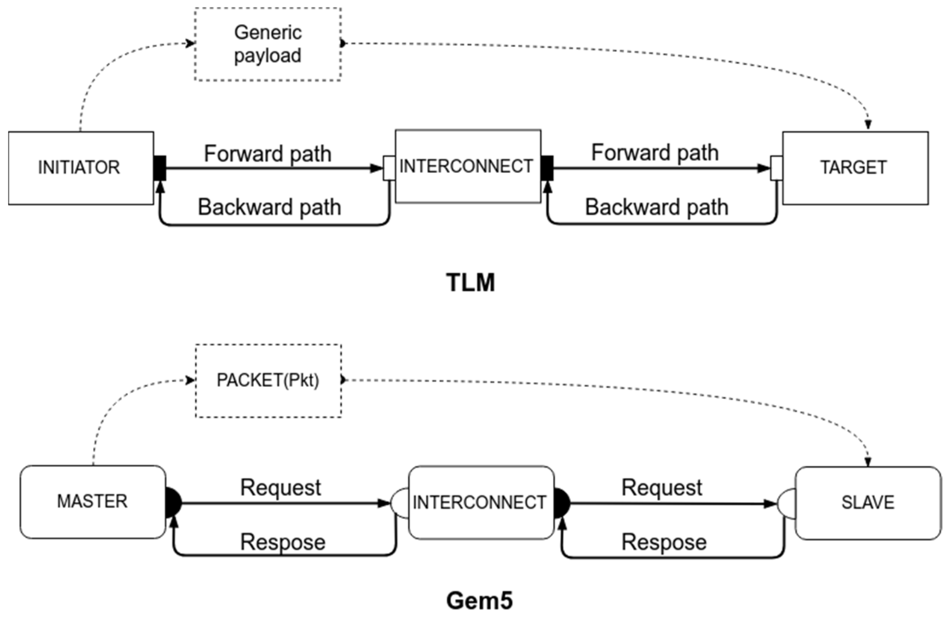


Figure 16: The Similarity between Gem5 and TLM

To send a request to the memory of SystemC world, not to the memory of Gem5 containing the address and size, It must follow these procedures:

1. Forward the packet data to the `externalslave,` which should translate these Gem5's requests in the packet data to the SystemC Generic payload.
2. Create a new Generic payload object and copy all the metadata from the Gem5 packet to TLM Generic payload which the metadata at both sides is very similar to each other, fortunately, it needs to just copy it.
3. Keep track of the original packet that is generated. Then it hooks reference to the packet as an extension to the Generic payload.
4. TLM procedure forwards the payload object to the memory and it will answer the request if it reaches the request within the data and creates a reply, then eventually the reference returns to the `slavetransactor.`
5. Restore our original Gem5 packet by removing the extension from the Generic payload and finally update the packet with the metadata from the Generic payload and store it.

3.2.2. Memobject

Before initiating the coupling develop an explanation, it is fundamental to know about memory system of Gem5 and its object so-called "`memobject`" and similarity between the mechanism of the memory system of Gem5 and transaction mechanism of TLM.

In Gem5, all objects that communicate via the memory system are called memory objects (`MemObject`). These are used to make the design more modular. This class appends the virtual function in order to interface and connect the MemObjects together structurally. Following with functions are in use:

```
getMasterPort (const std::string &name, PortID idx)

getSlavePort (const std::string &name, PortID idx)
```

Memory objects communicate with each other through ports. Ports are always used in pairs consisting of a master port and a slave port. The master port sends requests and receives responses. The slave port receives requests and sends responses. Typically, a master module (e.g. a CPU) has one or more master ports and a salve module (e.g. a Memory) has one or more slave ports. Interconnect components (e.g. bus, cache, bridge) have both port types. A connection between memory objects is established by binding master and slave ports to each other. Each connection binds exactly one master port to exactly one slave port. An interconnect components, such as a cache, bridge or bus, has both `MasterPort` and `SlavePort` instances.

The following figure illustrates that with neglecting to get address function there are two main types of function which makes Interaction between ports and objects. the send and the receive function. As an explanation, packets encapsulate transfer between CPU and memory call `sendTimingReq(pkt)` to send a packet. Corresponding to `send` function in Master port there is a `recv` function which must respond to any request that is sent. Slave port directly call `recvTimingReq (pkt)` in order to return value to port peer. Since the packets contain actual payload data of memory accesses, also Metadata consists of address, size, command, and status. Memory objects interface, by exchanging references to packets in a series of function calls. There is one pure virtual function as a penalty in this mechanism but is able to connect any memory objects together with keeping generic ports. The logic of the communication between memory and CPU in Gem5 simulator is very similar to the logic used in TLM. It goes without saying Gem5 as simulator can take place one of the sides in TLM Structure.

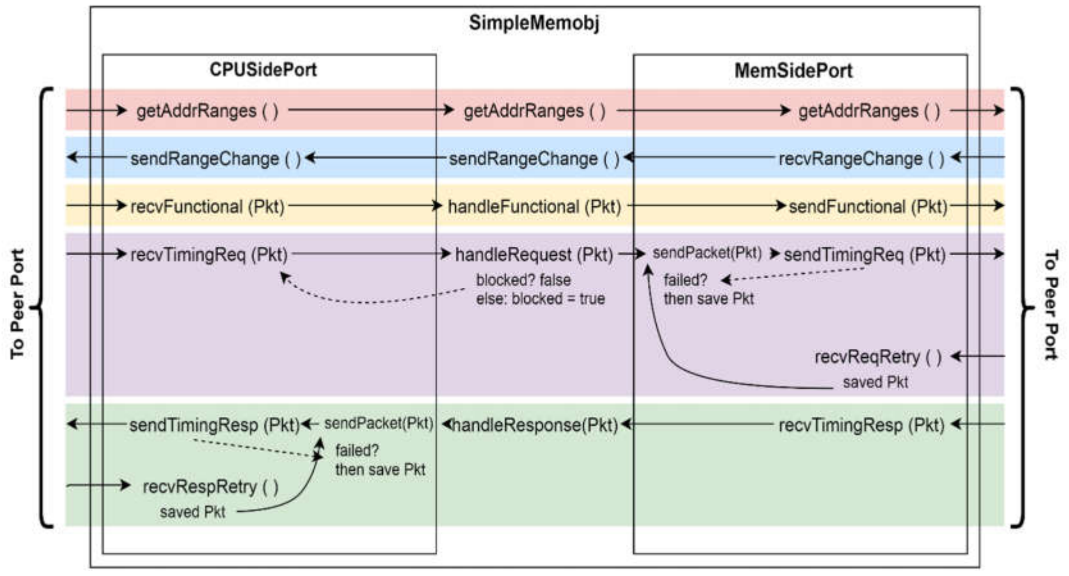


Figure 17: Interaction between SimpleMemobj and its ports

3.2.3. Practical Usage: General Flow

3.2.3.1. Building gem5

This section covers the how to set up and build Gem5

1. In order to preparing system for setting up Gem5 on Ubuntu 16.04. It needs some Requirements tools which listed in the following command used in Linux terminal.

```
>> sudo apt install build-essential git m4

>> scons zlib1g zlib1g-dev libprotobuf-dev \

      protobuf-compiler  libprotoc-dev \

      libgoogle-perftools-dev  python-dev
```

2. In order to obtain the Gem5 source code from GitHub should clone the repository then build Gem5 in the ARM by "`Scons`" which uses the `SConstruct` file (`Gem5/SConstruct`) to set up a number of variables and then uses the SConscript file in every sub-directory to find and compile all of the Gem5 sources.

3. Compile gem5 normally:

```
>> scons build/arm/Gem5.opt -j4 ///or Gem5.debug
```

I used "j4" flag for here to execute the build on 4 of my machine.

4. Compile gem5 as a library:

```
>> scons --with-cxx-config --without-python \

      --without-tcmalloc build/ARM/libgem5_opt.so
```

5. Change directory to `util/tlm`

6. Current directory contains the Gem5 modules that use in coupling, like `Gem5SimControl` and `Gem5SlaveTransactor` and/or `Gem5MasterTransactor` which in your SystemC project and connect them to your SystemC models. Make sure to go through an individual port name to the constructor of each transactor.

7. Compile your project by starting Gem5 with `tlm_slave` Python configuration. Not that It is necessary to create `config.ini` which reveal the Gem5 configuration for running the coupling project.

```
>> ../../build/ARM/gem5.opt \conf/tlm_slave.py
```

8. Run gem5 with a desire python configuration script as a system-call mode or full-system mode with `--tlm-memory=<port-name>` to make `m5out/config.ini`. Make sure to set the `tlm_data` attribute of the External Slaves to the port name of the corresponding SystemC transactor.

9. Run your SystemC project and pass the `m5out/config.ini` file to your `Gem5SimControl` object.

```
build/examples/slave_port/gem5.sc      \m5out/config.ini
```

There is four option flag for setting memory offset, debug mode and verbose output also a number of ticks in the following:

• -o <offset>	-- set memory offset
• -d <flag>	-- set a gem5 debug flag (-<flag> clears a flag)
• -v	-- verbose output
• -e <ticks>	-- end of simulation after a given number of ticks

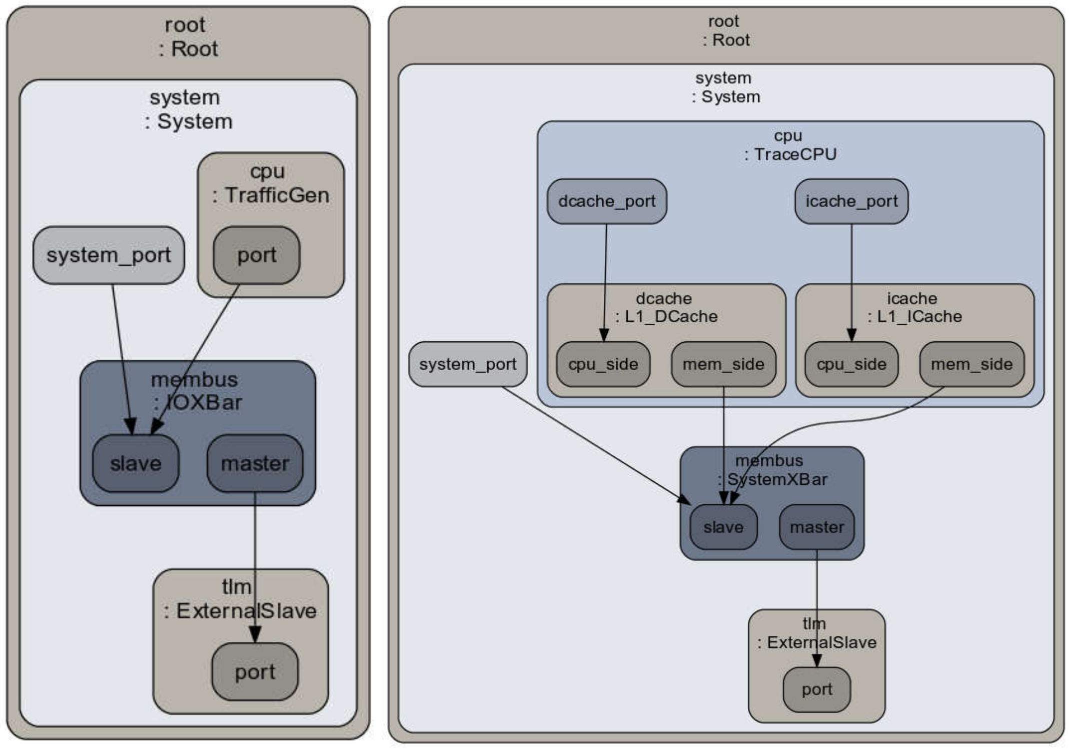


Figure 18: Gem5 to TLM (Slave Transactor) w/o cache

3.2.4. Simulation project

Ports of Gem5 support three different access types: timing, atomic, and functional. Atomic and functional accesses are synchronous. While timing access is use an asynchronous protocol where responses are not instantaneous.

In Atomic access, the master module sends a request by calling `sendAtomic ()` on one of the free master ports and the response is provided immediately when `recvAtomic ()` returns then receive an atomic request packet from the master port. The atomic model Gem5 focus more on fast transactions and doesn't provide so much timing detail and accurate simulation model.

Also at the same hand functional access type has similar attributes like atomic calls `sendFunctional ()` and `recvFunctional ()` returns then received a functional request packet from the master port. While Functional accesses are mostly used for initialization, debugging, and to load binaries to memories. Atomic accesses directly correspond to blocking transactions in TLM. Similarly, functional accesses correspond to debug transactions.

In timing access, there are two main mechanisms, simple and retry, the main difference between them is the sequence of sending and receiving the data packet. Timing access is high-accurate and detailed forwarding and the concept is very similar to the non-blocking transport interface TLM.

All in all the master module sends a request of timing mode by calling `sendTimingReq ()` function via the master ports. The corresponding slave port may accept or reject the packet. The port return value of `sendTimingReq ()` demonstrates the accept/reject status of a request by using `true` or `false`. If a packet is rejected with returning `false`, then retry mechanism will be used, the master module must not send any further packets using this port because of the busy port. When the slave port is ready to receive a request, it calls `sendRetryReq ()` to notify the master port. The master port can then resend the request by repeating the call to `sendTimingReq ()`. However, the request may be rejected again. Once the slave module accepts the request, it may forward the request to another module if there is Interconnect component or process the request if there is a memory. On the response path, a similar protocol is used. The slave module calls `sendTimingResp ()` and the master port may accept or reject the response. After a rejection, the master port calls `sendRetryResp ()` to indicate that it is ready to receive a response.

gem5 Access Type	TLM Transport Interface
Atomic	Blocking
Timing	Non-Blocking
Functional	Debug
---	DMI

Table 3: Gem5 access types and their corresponding transport

3.2.4.1. Coupling types

There are two types of coupling TLM to Gem5.

1. **Slave Transactor**

   The slave transactor translates memory accesses in Gem5 to TLM transactions. For each access, the transactor first converts the corresponding Gem5 packet to a TLM generic payload object. The transactor acquires a generic payload object and initializes it according to the information provided by the packet.

2. **Master Transactor**

   The master transactor translates TLM transactions to Gem5 memory accesses. For each transaction, it first converts the corresponding generic payload to a Gem5 packet. The transactor allocates a new packet and initializes the common fields.

3.2.4.2. Explanation mechanism of Gem5 to TLM (Slave Transactor)

In this thesis I assume slave transactor due to following reasons:

1. Gem5 in the master side has more alternative in architecture and it can increase the usability of co-simulation framework. (Usability)
2. Gem5 as the master port is capable configure in a variety of system components in system-call or full-system mode. (Completeness)
3. CPU built-in Gem5 provide to achieve memory-system (cache-hierarchy, interconnects and main memory) performance exploration in a fast and reasonably accurate also let's connect the translation lookaside buffer and cache ports on the CPU. (Flexibility)

First of all the system simulation need to define some socket for the communication and some storage which has defined a constructor for in a special way following the SystemC. Next, the system needs to define the TLM interface that implemented communication and apart from Gem5 actual memory so-called external memory. For transferring packet of data from Gem5 to TLM needs to encapsulate relevant attributes in Gem5 called "Gem5 packet" to equivalent attributes in the TLM generic so-called "payload object". The `cmd`, `data`, `addr`, and `size` attributes of the Gem5 packet are directly converted to their equivalent in the generic payload in TLM.

Gem5 Packet (Pkt)	TLM Generic Payload (gp)
flags	**---**
cmd	**command**
data	**data_ptr**
addr	**address**
size	**data_length**
---	**byte_enable_ptr**
---	**streaming_width**

Table 4: Comparison of the fields used to encapsulate transfers

Since the `flags` are Gem5 specific, they are only checked but not converted. The `byte_enable_ptr` and `streaming_width` in attributes of the generic payload are simply initialized to their default values, While Gem5 does not support features that are equivalent to the `byte-enable` and streaming features of TLM. In order to remember the original packet, the transactor attaches a reference to the packet than to the generic payload which using its extension mechanism. The generic payload in TLM supports two commands, `read` and `write` (R/W). Therefore, the command attribute is set to `read` or `write` at random as input data in the master side that here is Gem5. Then Gem5 packet of data needs to allocate address but for avoiding construction and destruction of generic payload transaction objects due to the extension mechanism implementation, the module should pool or re-use transaction objects by the implementation of memory manager interface `tlm_mm_interface`. Memory manager can control over a lifetime by counting a number of allocations, the transaction object which is distributed across the hops and is coordinated.

Regardless of transacting atomic and functional accesses, this thesis selects Timing access with multiple calls backward and forward (bidirectional). However, in order to correctly model the timings, It needs to convert the TLM timing annotation (referenced to time object) to the Gem5 annotation (return value denotes the number of ticks). Timing accesses, however, are more difficult to translate. The transactor needs to correctly implement both, the Gem5 timing protocol and the TLM base protocol. Most notably, the transactor needs to enforce back pressure in both directions.

Next, I discuss slave transactor non-blocking transport working mechanism.

As shown in figure 19 on the forward path when the master (Gem5) sends a request by calling `sendTimingReq()` function to transactor. The transactor replies to this request by returning `true` or `false`. If `true` receive by master module so-called `Gem5SlaveTransactor`. It then calls `nb_transport_fw()` passing `BEGIN_REQ` to `SCSlavePort` then initiate a new transaction and to forward the request to the SystemC target(Memory) module. The transactor then needs to wait for the target module to advance to the `END_REQ` phase. If transactor returns `false`. Then transactor follows TLM logic that means `BEGIN_REQ` phase is busy and transactor reject all another request till be free. Then master module resend request by calling `sendRetryReq ()`.soon after the target module forwards the `END_REQ` phase.

On the backward path. Target module (Memory) on TLM side must respond to transactor. So begin to a response by calling `nb_transport_bw ()` by passing the `BEGIN_RESP` phase in the TLM base protocol. Recovering original packet of data from generic payload when target module (Memory) transfer generic payload to transactor by calling `packet2payload ()` function like the following table. Next slave transactor transfers the packet to `makeResponse` and sends the `makeResponse` to the master module by calling `sendTimingResp ()`.

Then it transforms the packet to a response and sends the response to the master module by calling `sendTimingResp()` same as the forward path this time `slave transactor` wait for the reply of the master module (Gem5) `true` or `false`. If it returns `True` the transactor completes the transaction by calling `nb_transport_fw()` and passing `END_RESP`. Otherwise, it will return `false` the master module is busy and transactor has to resend the response to the master module by calling `sendRetryResp()` pending master module return `true` finally The transactor is able completes the TLM transaction. All the phase will be complete if TLM returns `TLM_ACCEPTED` message.

In this section, it needs check transaction objects which transfer to TLM environment. Therefore, `check_transaction` Check value returned from `nb_transport_fw` and `BEGIN_RESP` by reading the state of command (`cmd`), exact address (`adr`) and pointer of data in the master side.

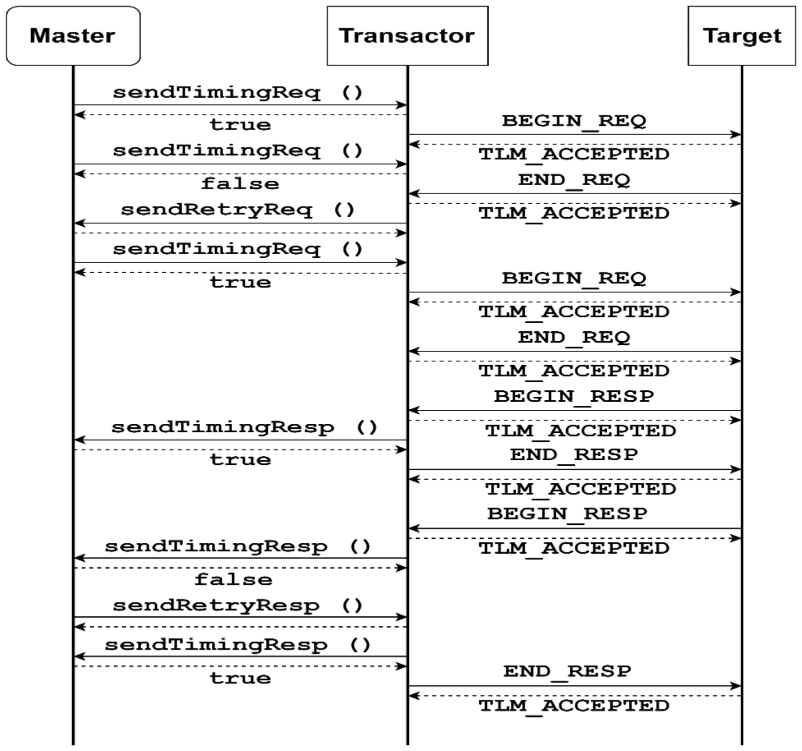


Figure 19: Gem5 to TLM with back pressure

Finally, in order to instantiate for completeness, when using the OSCI simulator, you will also need the following `sc_main` function Therefore the Top level of simulation instantiate systemC modules defined in initiator and memory modules, the initiator and target sockets have to be declared and constructed explicitly, In other words for coupling two environments like a master in Gem5 side and slave in TLM side, as follows:

The `Gem5simcontrol` module It also instantiates a transactor and then the memory Next all module should bind to proper transactors and sockets in order to begin simulation and execute tasks. Consequently, it performs all the connections and then starts the simulation.

There is some information hidden in these `config` notes that it will spread by passing the `config.ini` file through the `simcontrol` module.

But the simulation system requires another side which it should configure and implement in Gem5 domain side.

Due to creating a coupling simulation system. It requires creates a `trafficGenerator` on CPU side then creates a cross by bus connects them. Also, it creates an external slave and connected to the bus and then again it starts the simulation. Similar TLM domain the `Gem5simcontrol` module build the Gem5 file system instantiate all the objects.

In this thesis, the project of a coupling system builds by `scons` through the passing `Sconstruct` file.

Finally, simulation system needs to generate the configuration file for executing normal Gem5 with the Python script. Configuration file provides parameters in python script which are automatically turned into a C++ and run the project and its binaries generated from the SystemC TLM codes. Output as result of simulation shows in *Result section* that the simulation actually runs successfully.

3.2.4.3. Configuration file

A typical simulation script has two phases: a configuration phase, where the target system is specified by constructing and interconnecting a hierarchy of Python assembling objects; and a simulation phase, where the actual simulation takes place in C++ simulation objects.

System simulation has many integrated objects which named `SimObject` that is structured by C++ classes for forming the features and state then is placed into simulation system by a hierarchy of Python object for composing parameters and relationships with other components of the system. In fact, Gem5 enables a collection of Python object classes that correspond to its C++ simulation object classes. In another word Parameters in Python are transferred to a C++ structure and passed to the C++ object. For instantiating system simulation, Gem5 provide configuration files that it uses `SimObject` and their parameters synthesized from Python structure. These Python classes are described in a Python module called "m5.objects". It can be found in `.py` files in `src` in Gem5 sub-directory corresponding to C++ class location.

It needs to create a special configuration file for salve transactor. This is python file that will be executed by the Python interpreter compiler into Gem5.

First, it must import the m5 library and all `SimObjects` and classes that we've compiled.

```
import m5

from m5.objects import *
```

Creating the first `SimObject`: The all System object inherit of the `SimObjects` in the Gem5 simulation system. The `System` object contains most of the simulation information In the form of functionally such as various type of CPU, physical memory ranges, a memory bus, the systemCrossbar, the root clock domain Xbar and memory, the root voltage domain, the kernel (in full-system simulation), etc. `SimObjects` is instantiated like python class.

```
 system = system ()
```

Transaction data between two environments need a number of regular ticks for this reason Gem5 system simulation needs to set the clock on the system. First, it requires to create a clock domain and set `'1GHz'`clock frequency on that domain. Finally, It needs to configure `'1V'` voltage domain for this clock domain.

```
     system.clk_domain = SrcClockDomain ()

     system.clk_domain.clock = '1GHz'

     system.clk_domain.voltage_domain = VoltageDomain
                                                (voltage = '1V')
```

For converting non-blocking TLM to Gem5 equivalent access type It must use timing mode for the memory simulation. Note that in the python configuration scripts, whenever a minimum size is required and allocate that size in common '512MB'.

```
     system.mem_mode = 'timing'

     system.mem_ranges = [AddrRange ('512MB')]
```

In this system, there are two environments and it needs a component to connect their rout as master and slave, here we use Gem5 as master and we have to configure CPU on it and TLM as slave side and also we have to configure all memory and peripheral in slave side. First, we create a CPU. But we need the Traffic Generator as an initiator module that randomly generates new transactions. We modified the Traffic Generator so that it measures the host CPU time required to complete each transaction. On an Intel i7-4790 host CPU, we measured an average of 818 ns=transaction for a total of 1:5 million transactions. So to create the Traffic Generator as CPU and the relative file we can simply just instantiate the object:

```
     system.cpu = TrafficGen (config_file =
    "conf/tgen.cfg")
```

We can create an interconnect module called `membus` as the system-wide memory bus.

```
system.membus = SystemXBar (width = 16)
```

It will connect the I-cache and D-cache ports directly to the `membus`. In this system, It has no caches.

So what we can do now is we can create transactors so we already have external ports and Gem5 and we created external ports that connect to TLM and then we created some transactor modules and SystemC what we can do is we can hook to together and transact from Gem5 to TLM and vice versa by simply translating these interface SystemC interfaces to the Gem5 models and vice versa.

In this case, we should connect CPU `TrafficGem` port directly to `membus` slave port, otherwise, by defining cache it is able to connect caches' `mem_side` to a `slave` port of `membus,` it is possible to will connect the I-cache and D-cache ports directly to the `membus`.

But as mentioned it needs to define connection ports on both sides for the transaction. It is necessary to an object that able to connect `CPU` to `ExternalSlave` object as the last object in the Gem5 side. Also, `membus` can play a significant role in the interconnect module which has two different types of socket. First one is a `slave` port that able to bind to `CPU port` and also `system port` and second is `master` port which directly connected to `ExternalSlave`.

```
system.cpu.port = system.membus.slave

system.system_port = system.membus.slave

system.membus.master = system.tlm.port
```

The last object that must define is `ExternalSlave,` also clearly we know `ExternalSlave` acts as a bridge between the Gem5 world and TLM environment. In fact `ExternalSlave` transfer data to `SlaveTransactor` inside TLM and vice versa.

```
system.tlm = ExternalSlave ()

system.tlm.addr_ranges = [AddrRange ('512MB')]

system.tlm.port_type = "tlm_slave"

system.tlm.port_data = "transactor"
```

Finally, we should instantiate and simulate all configured system and start execution whatever simulated and assembled before we must create the Root object. All of the python `SimObject`s are turned into C++ equivalents by instantiation process and passed to the C++ object.

```
    root = Root(full_system = False, system = system)

    m5.instantiate ()

    m5.simulate ()
```

See figure 18

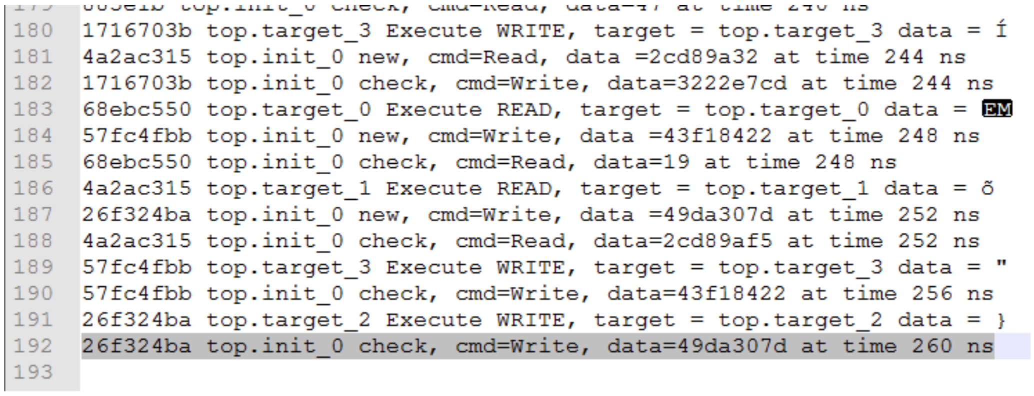
Chapter 4

Results

After running the project in two different mechanisms we can see the results and compare:

4.1. TLM/TLM

The following result achieved from TLM/ TLM experiment which this simulated system as it mentioned it has one initiator send randomly generated data to Bus then data transfer to four targets one after another. See Figure 15

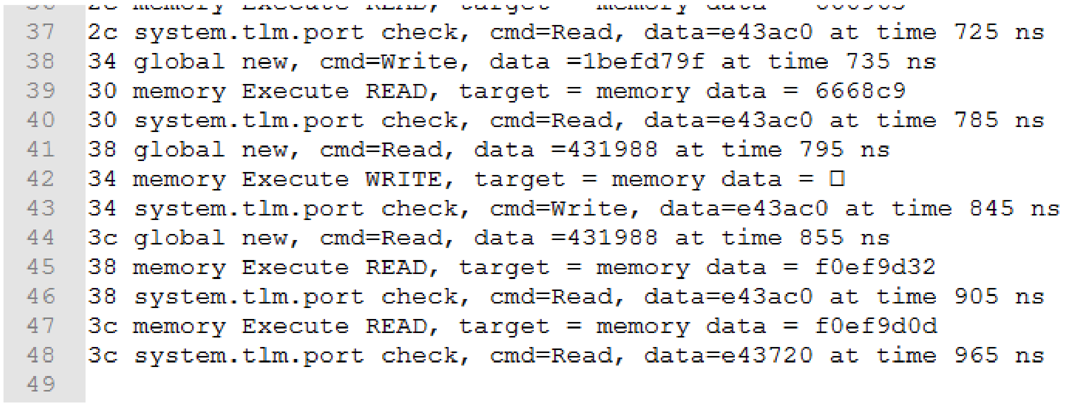


```
1716703b top.target_3 Execute WRITE, target = top.target_3 data = Í
4a2ac315 top.init_0 new, cmd=Read, data =2cd89a32 at time 244 ns
1716703b top.init_0 check, cmd=Write, data=3222e7cd at time 244 ns
68ebc550 top.target_0 Execute READ, target = top.target_0 data = EM
57fc4fbb top.init_0 new, cmd=Write, data =43f18422 at time 248 ns
68ebc550 top.init_0 check, cmd=Read, data=19 at time 248 ns
4a2ac315 top.target_1 Execute READ, target = top.target_1 data = õ
26f324ba top.init_0 new, cmd=Write, data =49da307d at time 252 ns
4a2ac315 top.init_0 check, cmd=Read, data=2cd89af5 at time 252 ns
57fc4fbb top.target_3 Execute WRITE, target = top.target_3 data = "
57fc4fbb top.init_0 check, cmd=Write, data=43f18422 at time 256 ns
26f324ba top.target_2 Execute WRITE, target = top.target_2 data = }
26f324ba top.init_0 check, cmd=Write, data=49da307d at time 260 ns
```

Figure 20: result execution of TLM/TLM mechanism

4.2. Gem5/TLM

The following figure illustrates the result of Gem5/TLM Mechanisms which Gem5 as CPU send random data to External Slave in Gem5 domain then transfer to Slave transactor in TLM domain.



```
2c system.tlm.port check, cmd=Read, data=e43ac0 at time 725 ns
34 global new, cmd=Write, data =1befd79f at time 735 ns
30 memory Execute READ, target = memory data = 6668c9
30 system.tlm.port check, cmd=Read, data=e43ac0 at time 785 ns
38 global new, cmd=Read, data =431988 at time 795 ns
34 memory Execute WRITE, target = memory data = □
34 system.tlm.port check, cmd=Write, data=e43ac0 at time 845 ns
3c global new, cmd=Read, data =431988 at time 855 ns
38 memory Execute READ, target = memory data = f0ef9d32
38 system.tlm.port check, cmd=Read, data=e43ac0 at time 905 ns
3c memory Execute READ, target = memory data = f0ef9d0d
3c system.tlm.port check, cmd=Read, data=e43720 at time 965 ns
```

Figure 21: result execution of Gem5/TLM mechanism

4.3. Comparison

With observing this two mechanism it can realize the TLM/TLM mechanism can transfer same data about ~3.7 time faster than Gem5/TLM mechanism. So it can be concluded Gem5/ TLM mechanism is slower because of it use two more components and translate data attributes in the Gem5 domain to TLM logic. On the other hand, it is worth noting the Gem5 simulator has more capability which provides a more complete and accurate model.

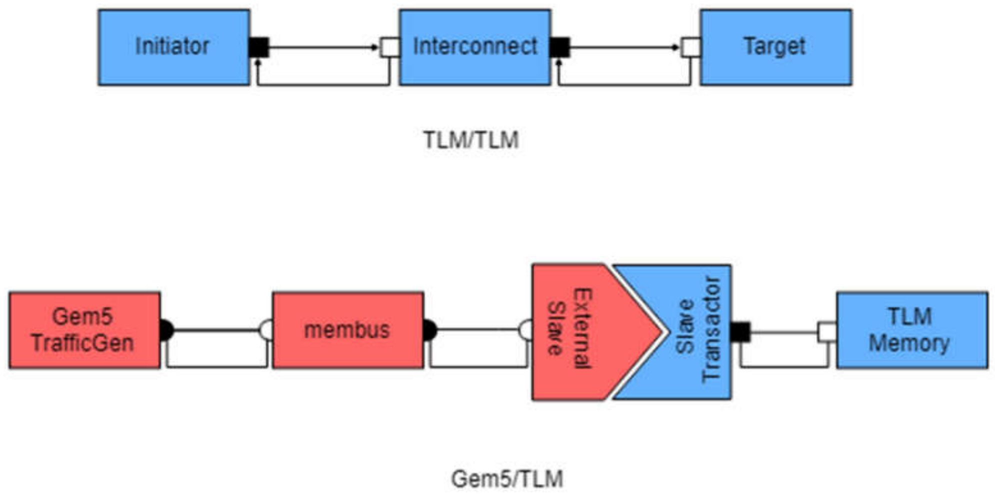


Figure 22: Schematic comparison of TLM/TLM and Gem5/TLM mechanism

Chapter 5

Conclusion

Both TLM-TLM and Gem5-TLM communication mechanisms proved to be suitable and feasible communication between the environment to varying degrees of usability and accuracy depending on the needs and resources of the application being modeled.

Selection the optimal mechanism allows focussing on efficiency in the simulation with regards to five simulator evaluation criteria such as Speed, Accuracy, Flexibility, Completeness, and Usability.

My final subjective assessment of both mechanisms can be summarized as listed below:

Speed: The Gem5-TLM communication mechanism has to employ master and slave transactors along with External master and slave components using TLM logic for communication while TLM-TLM mechanism can bind directly. This reason illustrates TLM-TLM is much faster.

Accuracy: With respect to Cycle-Accurate model both scenarios offer higher accurate performance. But the accuracy depends on the memory traffic. Therefore, Gem5 provide two types of memory model such as Ruby memory model and support for various cache coherence protocols. This feature able to Gem5 simulator possible to explore various possible combinations of parameters.

Flexibility: The Ruby memory system is much more elaborate and allows the accurate and flexible definition of SLICC (Specification Language for Implementing Cache Coherence) and network topologies along with supporting a large number of systems. It provides a domain-specific language for cache coherence protocols, with defining cache memories, DMA controllers enable high flexibility simulation. On the other side, SystemC TLM supports cache coherence protocol too. But it has limitations to use another type of memory system.

Completeness: Gem5 simulator can support full-system simulation with multiple ISA whereas systemC is not fast enough to boot an operating system in a reasonable time. Also, systemC needs to implement IP-Cores.

Usability: With focusing on ease of setup and execution in usability there are many reasons as following that SystemC has played the main role in developing Virtual platforms among vendors.

1. SystemC is an open-source free C++ class library and is very compatible with software and easy to use in applications and virtual platforms.

2. SystemC able to quickly simulate HW and SW systems on different levels of abstraction in order to estimate and optimize the performance and power for different applications.

3. SystemC is used for High-Level Synthesis, usually focused above RTL and models with considering much less detail. In another word, it takes less time to develop code and much faster.

Theses advantage of SystemC has become a reference model for verification and more popular language in system-level modeling, in order to optimize the performance and power for different applications. Furthermore, gem5 needs to systemC TLM logic for easing of setup and execution to reach more usability.

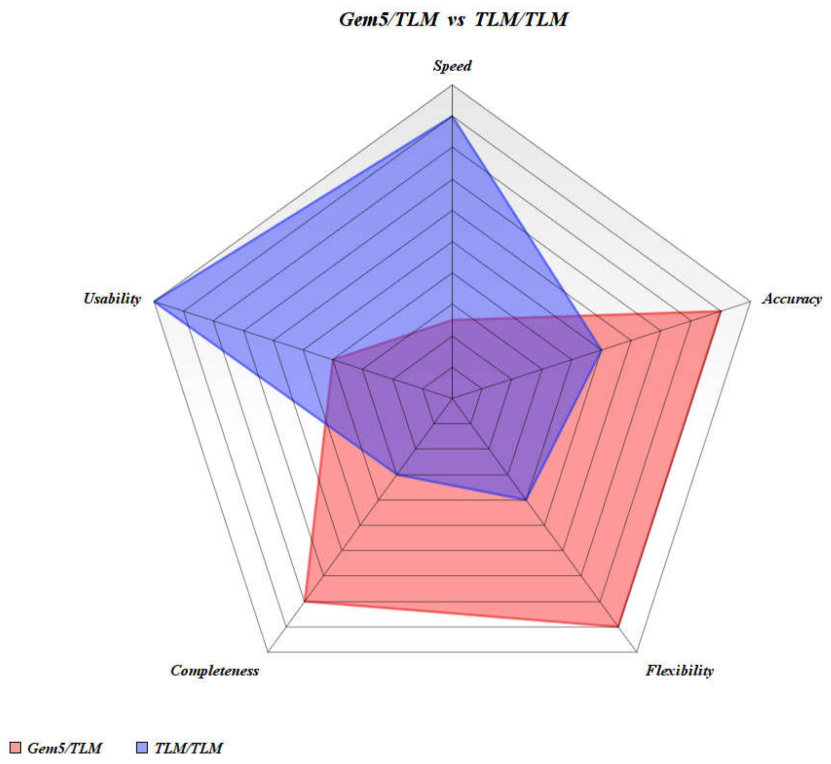


Figure 23 : The Conclusion of the comparison of the two mechanisms

Chapter 6

Future work

By coupling Gem5 to TLM, Network-on-chip (NOC) can achieve several tiles where each processing element present Memory or CPU connect by interface data transfer components. Each processing element can present sub-system simulated Gem5 domain which couple to SystemC domain. NOC model is able to use in the virtual testing environment.

Reference

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness,Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. "*The gem5 simulator*". Newsletter ACM SIGARCH Computer Architecture News. Volume 39 Issue 2, May 2011 Pages 1-7. DIO: 10.1145/2024716.2024718

[2] Anastasiia Butko, Rafael Garibotti, Luciano Ost, Gilles Sassatelli, "*Accuracy evaluation of GEM5 simulator system*", 7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), July 2012, Pages 1-7, DOI: 10.1109/ReCoSoC.2012.6322869

[3] Christian Menard, Jeronimo Castrillon, Matthias Jung, Norbert Wehn, "*System simulation with gem5 and SystemC*", 2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), July 2017, Pages 1-8, DIO: 10.1109/SAMOS.2017.8344612

[4] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, Andreas Hoffmann, "*A universal technique for fast and flexible instruction-set architecture simulation*". DAC '02 Proceedings of the 39th annual Design Automation Conference, Pages 22-27, DIO: 10.1145/513918.513927.

[5] Islam Almasri, Gheith Abandah, Ali Shhadeh, Anas Shahrour, "*Universal ISA simulator with soft processor FPGA implementation",* 2011 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT), December. 2011, DOI: 10.1109/AEECT.2011.6132512

[6] Mingsong Lv, Qingxu Deng, Nan Guan, Yaming Xie, Ge Yu, "*ARMISS: An Instruction Set Simulator for the ARM Architecture*", 2008 International Conference on Embedded Software and Systems, July 2008, DIO: 10.1109/ICESS.2008.73

[7] R. Leupers and O. Temam, Eds., "*Processor and System-on-Chip Simulation*". Boston, MA: Springer US, Book · January 2010, DIO: 10.1007/978-1-4419-6175-4

[8] Hari Angepat, Derek Chiou, Eric S. Chung, James C. Hoe, "*FPGA-Accelerated Simulation of Computer Systems*", Book, August 2014, DIO: 10.2200/S00586ED1V01Y201407CAC029

[9] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, Muneeb Ali, "*Protothreads: simplifying event-driven programming of memory-constrained embedded systems*", SenSys '06 Proceedings of the 4th international conference on Embedded networked sensor systems Pages 29-42, November 2006, DIO: 10.1145/1182807.1182811

[10] "*Gem5 Tutorial*" Available: http://learning.gem5.org/book/index.html

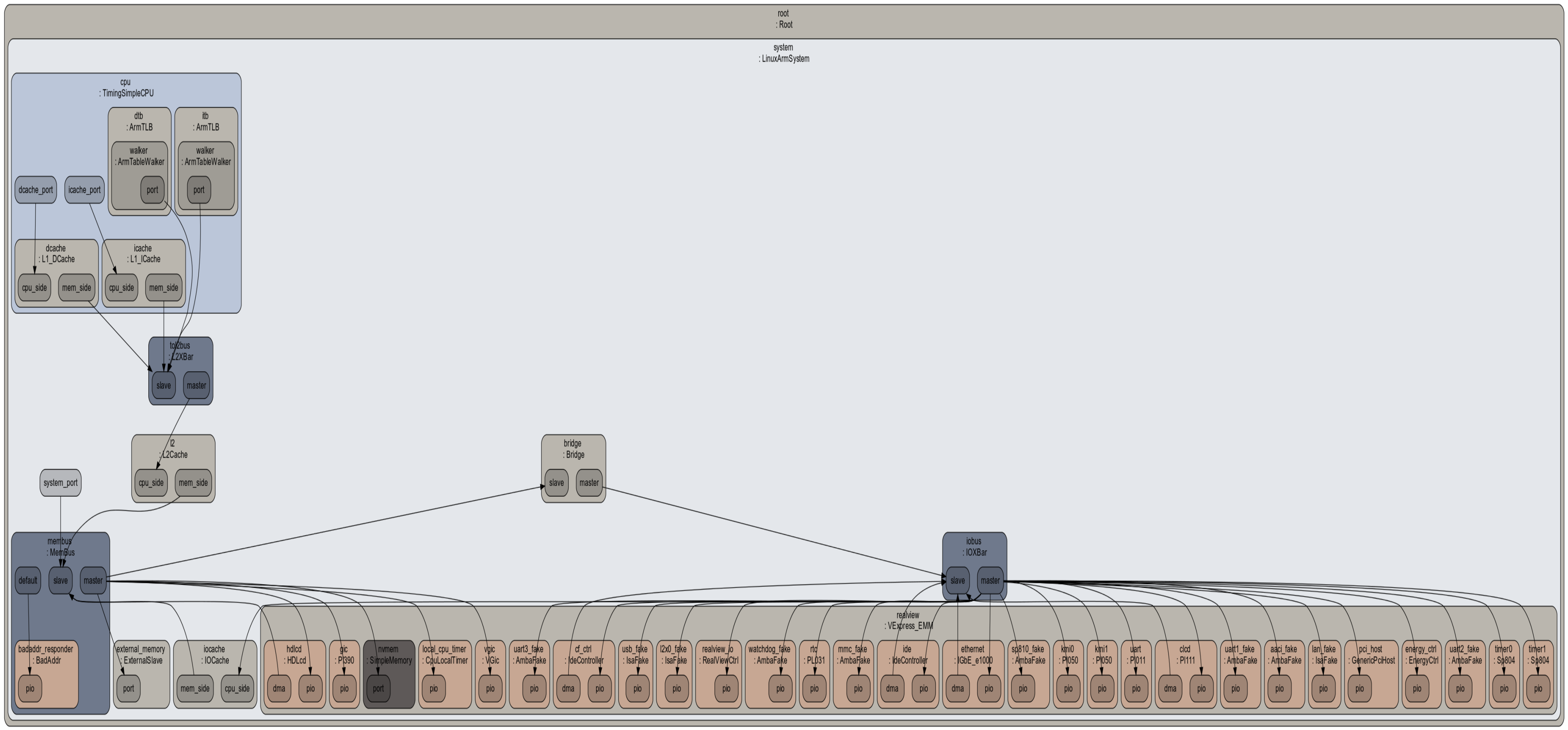
[11] "SimObjects - gem5." Available: http://www.gem5.org/SimObjects

[12] "*Simulating Systems not Benchmarks*" HiPEAC Computing Systems, Week 2012, Ali Saidi, Andreas Hansson, http://gem5.org/dist/tutorials/hipeac2012/gem5_hipeac.pdf

[13] " *SystemC* " Available: https://www.doulos.com/knowhow/systemc/

[14] " *SystemC TLM-2.0* " Available: https://www.doulos.com/knowhow/systemc/tlm2/

[15] Tse-Chen Yeh, Ming-Chao Chiang, "*On the interfacing between QEMU and SystemC for virtual platform construction: Using DMA as a case*", Journal of Systems Architecture: the EUROMICRO Journal archive Volume 58 Issue 3-4, March 2012 Pages 99-111, DOI: 10.1016/j.sysarc.2012.02.002


root
: Root
system
: LinuxArmSystem
cpu
: TimingSimpleCPU
dtb
: ArmTLB
walker
: ArmTableWalker
port
itb
: ArmTLB
walker
: ArmTableWalker
port
dcache_port
icache_port
dcache
: L1_DCache
cpu_side
mem_side
icache
: L1_ICache
cpu_side
mem_side
tol2bus
: L2XBar
slave
master
l2
: L2Cache
cpu_side
mem_side
system_port
bridge
: Bridge
slave
master
membus
: MemBus
default
slave
master
iobus
: IOXBar
slave
master
badaddr_responder
: BadAddr
pio
external_memory
: ExternalSlave
port
iocache
: IOCache
mem_side
cpu_side
realview
: VExpress_EMM
hdlcd
: HDLcd
dma
pio
gic
: Pl390
pio
nvmem
: SimpleMemory
port
local_cpu_timer
: CpuLocalTimer
pio
vgic
: VGic
pio
uart3_fake
: AmbaFake
pio
cf_ctrl
: IdeController
dma
pio
usb_fake
: IsaFake
pio
l2x0_fake
: IsaFake
pio
realview_io
: RealViewCtrl
pio
watchdog_fake
: AmbaFake
pio
rtc
: PL031
pio
mmc_fake
: AmbaFake
pio
ide
: IdeController
dma
pio
ethernet
: IGbE_e1000
dma
pio
sp810_fake
: AmbaFake
pio
kmi0
: Pl050
pio
kmi1
: Pl050
pio
uart
: Pl011
pio
clcd
: Pl111
dma
pio
uart1_fake
: AmbaFake
pio
aaci_fake
: AmbaFake
pio
lan_fake
: IsaFake
pio
pci_host
: GenericPciHost
pio
energy_ctrl
: EnergyCtrl
pio
uart2_fake
: AmbaFake
pio
timer0
: Sp804
pio
timer1
: Sp804
pio