Design and Implementation of an Extendable SoC Virtual Platform in SystemC-TLM 2.0

Lei Liang

Master Thesis in System-on-Chip Design
Department of Electronic Systems
School of ICT

Supervisor: Patrik Bohlin Björk
Examiner: Associate Prof. Zhonghai Lu

TRITA-ICT-EX-0039
Abstract

With the increasing design complexity for SoC development, the workload for hardware designer and verification engineer is becoming larger and larger. On the other hand, software and hardware development is unable to be carried out in parallel. This creates a bottleneck in the current design flow. Also, it will be very difficult to deal with the hardware problems which are found during the software development process. To overcome these problems, design at higher level needs to be applied. SystemC is a language which enables the design at the system level and the TLM-2.0 contains different standardized SystemC interface classes, which ensures the portability and interoperability of different IPs.

In this thesis, an extendable SoC virtual platform is implemented in SystemC. It can give exactly the same functions as the design specification required. A standardized SystemC module template is designed which owns all different interfaces of the virtual platform. The template can provide lots of convenience for future module development. One method for wrapping a C/C++ into SystemC is given and a basic framework structure is implemented so that the existing C++/Simics modules can work in the designed SystemC virtual platform. Finally, the comparison on simulation time and workload between RTL modules and SystemC modules is made, which demonstrates that large development time can be saved by using this virtual platform for software development.
Acknowledgements

First and foremost, I would like to extend my sincere gratitude to Ericsson AB for providing me this opportunity to do my master thesis. I am also deeply grateful for my manager Hans Lundén and my supervisor Patrik Bohlin Björk. They have given lots of help on my working for these past six months. I also owe my gratitude to Christian Sauer, Björn Fjellborg, Hans-Peter Loeb, Henrik Svensson, Göran Knutson, Marcus Lövgren, Anders Ulander. They have given me lots of help in SystemC and TLM filed from the start of my thesis. And for all members in Ghost group, I would like to express my sincere appreciation for their help and their recognition to my work.

My heartfelt thanks also go to my parents and all my dear friends for their always supporting in the past years.

Finally, I would like to thank my supervisor and examiner Zhonghai Lu who is working at the Department of Electronic Systems in Royal Institute of Technology.

Lei Liang

2011-10
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CB</td>
<td>Circular Bus</td>
</tr>
<tr>
<td>CBI</td>
<td>Circular Bus Interface</td>
</tr>
<tr>
<td>CM</td>
<td>Common Memory</td>
</tr>
<tr>
<td>CMC</td>
<td>Common Memory Controller</td>
</tr>
<tr>
<td>DMI</td>
<td>Direct Memory Interface</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>IF</td>
<td>Interface</td>
</tr>
<tr>
<td>SSS</td>
<td>Simics/SystemC Switcher</td>
</tr>
<tr>
<td>SV</td>
<td>Supervision</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>TLM</td>
<td>Transaction Level Modeling</td>
</tr>
<tr>
<td>VP</td>
<td>Virtual Platform</td>
</tr>
<tr>
<td>VSP</td>
<td>Virtual System Platform</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1-1 Module development progress today ................................................................. 8
Figure 1-2 Module development progress with SystemC-TLM ........................................... 9
Figure 2-1 Abstract block diagram for the Ericsson SoC ....................................................... 11
Figure 2-2 VP system overall picture .................................................................................... 14
Figure 3-1 Block diagram of CMC implementation in SystemC ........................................... 16
Figure 3-2 Comparison between RTL and TLM map ......................................................... 17
Figure 3-3 Block diagram of memory class ........................................................................... 18
Figure 3-4 Block diagram for Semaphore Unit ................................................................. 21
Figure 3-5 Block diagram for Emdmac unit ......................................................................... 21
Figure 3-6 Arbiter Structure ............................................................................................... 22
Figure 3-7 Future arbiter structure ....................................................................................... 24
Figure 4-1 Original structure of the template ........................................................................ 26
Figure 4-2 Multi-Inheritance Error ....................................................................................... 27
Figure 4-3 New template structure ....................................................................................... 28
Figure 4-4 Block diagram of the existing XIO module ...................................................... 29
Figure 4-5 Block diagram of XIO ......................................................................................... 32
Figure 4-6 Conflict of forward declaration ........................................................................... 32
Figure 4-7 Correct structure of forward declaration ............................................................ 33
Figure 4-8 Forward declaration for wrapper with template class ........................................ 33
Figure 4-9 Functions connection in the wrapper ................................................................. 34
Figure 4-10 Class inside the framework ............................................................................... 35
Figure 4-11 Modules working in both environments ........................................................... 36
Figure 4-12 Function connection in the wrapper ................................................................... 37
Figure 4-13 Function connection in the SSS framework ..................................................... 37
Figure 5-1 Block diagram of the Dummy DSP ..................................................................... 38
Figure 5-2 Block diagram of dummy CPU ........................................................................... 39
Figure 5-3 Block diagram of dummy antenna ...................................................................... 40
Figure 5-4 Test platform for CMC ....................................................................................... 41
Figure 5-5 Test for CM ........................................................................................................ 42
Figure 5-6 Test for semaphore ............................................................................................ 43
Figure 5-7 Test for supervision ............................................................................................ 43
Figure 5-8 Test for arbitration .............................................................................................. 45
Figure 5-9 XIO test platform ............................................................................................... 47
Figure 5-10 Platform for testing CMC CPU runtime .......................................................... 49
Figure 5-11 CPU time for LT CMC ..................................................................................... 49
List of Tables

Table 3-1 Return value format ................................................................................................................. 20
Table 3-2 Files for all CMC .......................................................................................................................... 24
Table 4-1 Basic information for interface ..................................................................................................... 30
Table 5-1 Interfaces for the DSP ...................................................................................................................... 39
Table 5-2 Interfaces for dummy CPU .............................................................................................................. 39
Table 5-3 Initial value for all variable in SV .................................................................................................... 43
Table 5-4 Initial value for masks of SW defined areas .................................................................................... 44
Table 5-5 Set value for masks of SW defined areas ........................................................................................ 44
Table 5-6 Test schedule ................................................................................................................................... 46
Table 5-7 Codes comparison on CMC ............................................................................................................. 48
Table 5-8 Codes comparison on XIO with SSS (Simics & SystemC) ................................................................. 48
Table 5-9 Codes comparison for XIO core ...................................................................................................... 48
Table 5-10 Simulation time comparison ......................................................................................................... 50
Content

1 Introduction.................................................................................................................. 8
1.1 Background Description............................................................................................ 8
1.2 Problem Statement...................................................................................................... 8
1.3 General introduction for SystemC Virtual Platform................................................... 8
1.4 Thesis Contribution..................................................................................................... 9
1.5 Organization of this document................................................................................... 10
2 Foundation Research.....................................................................................................
2.1 General Introduction of the existing SoC ................................................................... 11
2.2 Research for the existing platform............................................................................ 11
2.3 Research on building the new CMC-XIO based VP.................................................. 11
2.3.1 CMC-XIO based virtual platform....................................................................... 11
2.3.2 Dummy modules for testing................................................................................ 12
2.4 Standardized module template and module reusability exploration.......................... 13
2.4.1 Standardized module template............................................................................ 13
2.4.2 Module reusability exploration............................................................................ 13
2.5 Overall picture for the thesis................................................................................... 14
2.6 Module design requirements..................................................................................... 14
2.6.1 Requirements on TLM interfaces...................................................................... 14
2.6.2 Requirements on simulation environment.......................................................... 15
3 The Common Memory Controller Implementation..................................................... 16
3.1 Structure of CMC..................................................................................................... 16
3.2 Supervision Unit ..................................................................................................... 17
3.3 Router ..................................................................................................................... 17
3.3.1 Router in Cadence tool...................................................................................... 17
3.3.2 Memory Map..................................................................................................... 17
3.3.3 Router Generating............................................................................................. 18
3.4 CM & DDR & Memory Initiator.............................................................................. 18
3.4.1 Memory Overview............................................................................................. 18
3.4.2 Memory Class................................................................................................... 18
3.4.3 Memory Initiator.............................................................................................. 19
3.5 Semaphore Unit....................................................................................................... 19
3.5.1 Semaphore function description........................................................................ 19
3.5.2 Semaphore implemented in SystemC................................................................. 20
3.5.3 Structure of Semaphore unit.............................................................................. 21
3.6 Emdmac Unit.......................................................................................................... 21
3.7 Arbiter .................................................................................................................... 22
3.8 CMC Top ................................................................................................................. 24
3.8.1 All file overview................................................................................................ 24
3.8.2 Hierarchy binding............................................................................................. 24
3.9 CMC implementation Summary............................................................................. 25
4 Standardized module template and module reusability exploration............................ 26
4.1 Standardized module template............................................................................... 26
4.1.1 Old template overview....................................................................................... 26
4.1.2 New template structure...................................................................................... 28
4.2 Introduction for existing XIO module...................................................................... 29
4.3 Different methods for the wrapper implementation.................................................. 30
4.3.1 General description for the wrapper................................................................. 30
4.3.2 Mix all the XIO functions in the wrapper.......................................................... 30
4.3.3 Instance the XIO core as an Object.................................................................... 31
4.3.4 Reuse the XIO logical functions and implement it with SystemC....................... 31
4.3.5 Summary of wrapper design methods............................................................... 31
4.4 XIO wrapper implementation................................................................................. 32
4.4.1 Structure of the SystemC XIO .................................................................32
4.4.2 Wrapper implementation .................................................................32
4.4.3 Implementation for XIO interfaces ....................................................34
4.4.4 Overlap the Simics code in XIO core .................................................34
4.5 SSS framework for blocks .................................................................35
4.5.1 Structure of the SSS framework .........................................................35
4.5.2 Interface implementation .................................................................36
5 Testing .....................................................................................................38
  5.1 Dummy Modules ................................................................................38
  5.1.1 Dummy DSP ....................................................................................38
  5.1.2 Dummy CPU ..................................................................................39
  5.1.3 Dummy Antenna ............................................................................40
  5.1.4 Summary for dummy modules .........................................................41
5.2 CM Test ...............................................................................................41
  5.2.1 Test Platform ..................................................................................41
  5.2.2 Blocking Transport and DMI Test ....................................................42
  5.2.3 Semaphore test ...............................................................................42
  5.2.4 Supervision test ...............................................................................43
5.2.5 DMI invalidation Test .......................................................................45
  5.2.6 Arbitration Test ...............................................................................45
5.2.7 A demo test for all functions .............................................................46
5.3 XIO wrapper test ................................................................................46
  5.3.1 XIO test platform ............................................................................46
  5.3.2 XIO test process .............................................................................47
5.3.3 XIO test result ..................................................................................47
5.4 SSS test ...............................................................................................48
5.5 Standard template test ..........................................................................48
6 Comparison between RTL and SystemC implementation ....................48
  6.1 Comparison on complexity .................................................................48
  6.2 Comparison on simulation time ...........................................................49
6 Conclusion and Future Work .................................................................51
  6.1 Conclusion ..........................................................................................51
  6.2 Future Work .......................................................................................51
7 Reference ...............................................................................................52
1 Introduction

1.1 Background Description

With the rapid development of SoC technology, the design complexity and the integration density are becoming higher and higher. For Ericsson SoC development history, there were only 20,000 gates in 1980’s. But at the year 2000, the number had increased to 20,000,000. The problem appeared with this fast development is that the workload for both design engineer and verification engineer is becoming larger and larger.

At the same time, people in software development need to wait the SoC hardware to be released. Then they can start to design and debug the software on the SoC platform. Figure 1-1 is the development process for SoC today. The software development can only be started after the VHDL code can be virtual typed. And the final product of SoC can only be released when the software development is finished. Normally the release time of the final production would be 1-2 years after the SoC Chip was released. If there are hardware problems found during software development, it would be very time consuming to fix those bugs. Obviously, this process cannot be satisfied nowadays.

<table>
<thead>
<tr>
<th>Functional Model (C++)</th>
<th>Architecture Exploration</th>
<th>RTL Model (VHDL/…)</th>
<th>SoC Chip</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Application SW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Low Level SW</td>
</tr>
</tbody>
</table>

Figure 1-1 Module development progress today

1.2 Problem Statement

In the SoC development, the software development can only be started after all the VHDL codes can be virtual typed or even the SoC chip is released.

1.3 General introduction for SystemC Virtual Platform

SystemC is a language for Electronic System-Level Modeling. It enables the design engineer and verification engineer to work at a higher level compared to the RTL level.

There are two different internal levels existing for a design in SystemC with TLM2.0. One is Loosely Timed (LT) level and the other is Approximately Timed (AT) level. The modules built at LT level would be mainly used for software development or as raw materials for AT implementations. The modules built at AT level would be used for architecture explorations.
In this thesis, the virtual platform technology is adopted in order to solve the problem in the development process today. This virtual platform is constructed by modules implemented at LT level. The main reason for using TLM2.0 standard for all the interfaces is that it provides a high efficiency for implementation and also the reusability for IPs. The whole product development process can be changed to what is shown in Figure 1-2.

<table>
<thead>
<tr>
<th>Functional Model (C++)</th>
<th>LT Model (SystemC)</th>
<th>Architecture Exploration</th>
<th>RTL Model (VHDL/…)</th>
<th>SoC Chip</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Application SW</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low Level SW</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1-2 Module development progress with SystemC-TLM

Compared with Figure 1-1, when the functional model is ready, all the involved modules will be designed using SystemC with TLM LT standard. Then the virtual platform can be constructed by those LT models, which own exactly the same functions as what are defined in the specification at the functional level. After the virtual platform is ready, software development and the left hardware development can be carried in parallel. The virtual platform would be used in software development and the LT modules can be improved to AT modules in hardware developing process. With this improvement, the software development can be started much earlier compared to what is shown in Figure 1-1. Thus the time to market for SoC can be decreased a lot. Moreover, if problems for hardware function are found in software development, it would take less time and effort to solve.

For the functional models, they are built mainly in C++. They are used for testing different algorithms. And these models would not cover all the functions which the SoC have. During the LT SystemC model implementation, these models can be re-used in some cases. Especially for modules like HW-Accelerators, their main function is to process data with different algorithms. Lots of codes from functional models can be re-used during LT model developments.

1.4 Thesis Contribution

In this thesis, the CMC, which is the central component in one Ericsson SoC, has been modeled under SystemC and TLM-2.0 standard. This CMC module contains all main functions documented in the specification and it is constituted by different sub-modules so that existing functions can be easily maintained and more functions can be easily added.

The XIO module can now work well with SystemC virtual platform. Before the thesis was carried, the XIO can only work in Simics virtual platform. With the XIO, the data from the outside world of the platform can then be filled into the system. During the XIO wrapper’s implementation process, a basic structure of the SSS framework is implemented so that one XIO core can now be working in both Simics and SystemC platform. The fully completed SSS framework used in Ericsson is based on that structure developed in this thesis. And the testbench for XIO is also used for testing the fully completed SSS framework.

A new standardized template for modules with different interfaces is developed and tested. The comparison between the new and old template is documented.
1.5 Organization of this document

The reminder of this document is separated into four parts:

Firstly, chapter 2, foundation research gives the demand and request analysis for this thesis.

Secondly, chapters 3 and 4 give the main implementations for different modules and the research on wrapper design methods and the module template.

Thirdly, chapter 5 shows the different dummy modules for testing and tests which are carried out for those implementations described in the second part.

Fourthly, the last chapter states the conclusions and the future work.
2 Foundation Research

2.1 General Introduction of the existing SoC

In this thesis, all the components to be modeled are based on one Ericsson’s SoC platform. The abstract structure for the platform is shown in Figure 2-1. It can be separated into five blocks which are CMC, CPU, DSPs, Interfaces and HW Accelerators. In order to give a basic virtual platform for this SoC board, these five parts with the internal interfaces for binding components like the CBI and CMI shall be modeled and tested.

![Abstract block diagram for the Ericsson SoC](image)

The implementations made in this thesis are mainly based on function descriptions in the given specification documents for the Ericsson’s SoC platform. The demonstration for the consistency between the SystemC implementation and RTL implementation is not made in this thesis.

2.2 Research for the existing platform

In Ericsson, there does exist a virtual platform which contains some basic components before the start of this thesis. But those modules, which are used to construct the VP, are very basic that they even lack of some main functions. Take the CMC for an example. It is the central part of the SoC platform, which is connected to all other modules. However, it only supports the basic write and read operations which are far away from functions expected. Also, there are existing C modules like different HW accelerators. But they can only work in Simics Virtual Platform.

Based on those existing problems, this thesis is to develop a new platform. The method for wrapping an existing C module into SystemC module and standardized module template shall also be considered during the new platform implementation.

2.3 Research on building the new CMC-XIO based VP

2.3.1 CMC-XIO based virtual platform

From Figure 2-1, CMC is the central component for the whole SoC platform. It owns following features that other modules do not have:
- The CMC is connected to all other modules
- Data sharing carried out mainly through the CMC
- The CMC contains all different types of interfaces

As the CMC is connected to all other modules, it shall be implemented correctly for the VP so that further developed module can be plugged in by binding to the CMC. And it is also an indispensable component for testing data communication. Moreover, since it contains all different types of the interfaces, the standardized template for modules can be defined with the understanding on this module.

For the XIO, it is the only interface for the SoC platform to communicate with outside world. All the data, which is to be processed, is filled into the SoC through the XIO and the processed data would finally be sent to Antennas through the XIO as well. So with this XIO and CMC based virtual platform, all the requested data by surrounding modules will be provided. Then modules to be plugged-in in future can be tested easily.

Another reason for choosing the XIO is that there is a XIO working in Simics environment. It can save lots of time comparing to design a new module with all the logical functions from bottom up. And since the XIO is composed of XIO core and Simics Interface classes, it would also be used for doing research on methods for wrapping a core into SystemC module and the structure of the SSS framework. More details about this are given in chapter 2.4.2.

### 2.3.2 Dummy modules for testing

From Figure 2-1, CPU and DSP are also the indispensable modules for the SoC platform. CPU is the central controller for the whole SoC platform. It is impossible to model a real CPU in this thesis but the basic functions like sending configuration mails are necessary for the XIO and CMC module. Thus the dummy CPU module would be implemented to play the hard coded functions.

For the DSPs, since there are SW defined areas in the CMC, dummy DSP shall be implemented so that those functions can be tested. The internal communication through CMI can also be tested with those DSPs. For different platform, the number of DSP is different. To simplify the work in this thesis, not all the DSPs shall be integrated. But the implemented dummy DSP shall be reconfigurable so that two or three DSPs can be instanced to play different operations.

Except for those modules shown in Figure 2-1, the Antenna module needs to be added. The Antenna is the only module that communicates between the enclosed SoC and outside environment through wireless signals. For both XIO function testing and filling data into the CMC, the dummy Antenna shall be implemented. All the data, which needs to be transferred to the VP, can be hard coded in first step in the dummy Antenna and later on the data for Antenna shall be read from a file.
With these dummy modules working with the CMC-XIO based virtual platform, the standardized virtual platform is tested. Since the HW accelerators do not have a new type of the internal interface compared with the existing interfaces, the HW accelerators are not implemented for the VP testing in this thesis. After the SSS framework is fully implemented, the HW accelerator would automatically be able to work in both the SystemC and Simics environments.

2.4 Standardized module template and module reusability exploration

2.4.1 Standardized module template

It can be seen from Figure 2-1 that lots of components are connected by CBI and CMI interfaces. And there are some other interfaces like the interrupt interface and reset interface which are not shown in the picture.

A SystemC module template was created containing all those interfaces so that it would be easier for a designer who is the beginner in SystemC field to design their modules for Ericsson VP. In this thesis, the structure of the template shall be analyzed and the improvement shall be made to provide more conveniences.

2.4.2 Module reusability exploration

In order to increase the reusability of the designed modules, on one hand, not all modules would be directly implemented in SystemC. The reason is that the module implemented in C/C++ would be easily used in different platforms while SystemC modules can hardly be reused for other environments. On the other hand, there are existing modules designed in pure C/C++ and C/C++ mixed with Simics in Ericsson. Lots of time can be saved if the C/C++ and C/C++ mixed with Simics modules can be reused in this SystemC virtual platform. So there are two kinds of wrapper shall be researched based on existing modules.

Firstly, methods for designing a SystemC-TLM wrapper for the existing C/C++ module should be analyzed. With the selected method, the designer can turn the C/C++ modules into SystemC-TLM modules and plug it into the virtual platform. It can be seen from Figure 1-2 that there are C++ modules built at the beginning of the SoC development process. So with the wrapper added, those modules would be easily reused in the SystemC VP. And the XIO module can be used to test this method. Although there are still Simics codes inside the XIO core, all the logical functions are implemented in C. In other words, the codes written in Simics have no influence for the XIO logical functions so that it can be filtered during the implementation.

Second, lots of modules are already modeled and can be run in Simics platform. Especially all the HW-accelerators have the same kinds of interfaces and protocols. The SSS framework is actually a kind of adaptation layer with which all the modules can be switched between SystemC virtual platform and the Simics virtual platform during compilation. And the XIO also is the best choice for this exploration. There are five interfaces for all those modules: W/R interface, reset interface, Circular Bus interface, memory interface, event interface. And the XIO contains the first four of all those interfaces in Simics. So it is the desired module to be used for the research on the basic SSS structure for the adaptation layer. More detail description about the XIO module can be found in 4.2.
2.5 Overall picture for the thesis

The desired result for this thesis described above is shown in the following picture.

---

![Diagram](image)

Figure 2-2 VP system overall picture

2.6 Module design requirements

2.6.1 Requirements on TLM interfaces

Since all the modules to be implemented are at LT level, following functions for the interfaces shall be supported by the developed modules.

- **Blocking transport operation**

  It is the basic interface function used to transport transactions between different modules at LT level. And “b_transport” is used and there is only forward path from initiator to target for blocking transport. There is a timing annotation for “b_transport” but it is set to “ZERO” time since time factor is not considered for the VP.

- **DMI operation**

  DMI would bypass all the supervision carried in the target modules. The simulation time can be faster with the DMI supported. Those modules with no memory block can make forward or filter operation to the DMI request. Modules which cannot receive DMI operation shall return “false” to the DMI request.

- **DMI invalidation**

  It is generated by target to the initiator in correspondent to the granted DMI operations.
- Debug transport

  It is implemented for further verification usage.

**2.6.2 Requirements on simulation environment**

All the modules are designed and compiled with OSIC SystemC 2.2.0 and TLM 2.0 standard. Since all the modules for Ericsson VP is run and debugged in Cadence VSP environment, the “makefile” for every tests and demo shall be added. Both “IRUN” and “Regression Test” shall be implemented inside the “makefile”.
3 The Common Memory Controller Implementation

3.1 Structure of CMC

For SystemC design of the CMC, since the design is at LT level and it is focused mainly on functionalities, timing factors for memory like different access latencies are not considered. The structure of CMC implemented in SystemC is shown in Figure 3-1.

![Figure 3-1 Block diagram of CMC implementation in SystemC](image)

There are all together 1 top module and 6 sub modules for this CMC. Though the DDR module is an off-chip memory, there is no influence on functionality level if it is put inside the CMC. With the DDR put inside the CMC, there will be less binding issues during the VP top constructing time. That's why the DDR is placed inside the CMC in this thesis.

Another major difference is that the memory map in TLM is not the same as the RTL SoC. For TLM 2 interfaces, the address of Write and Read operations are byte aligned while the address in Ericsson RTL SoC is two bytes aligned. Figure 3-2 gives the example for the differences between RTL and SystemC implementations.
3.2 Supervision Unit

The Supervision sub-module would continuously perform supervision on the blocking transport operations carried out through CMI interfaces. The supervised transactions would then be forwarded to the router.

3.3 Router

3.3.1 Router in Cadence tool

Since there are different slaves corresponding to different addresses, the router is needed. For this design, Cadence has provided a tool named fm-integrator, which can generate a fully functioned router according to a given memory map.

3.3.2 Memory Map

For this design, the content in the map has the similar structure as the example given below:

```
common_memory, cms.cmem.tsocket, 0, 0x1FFF, 0x0
sema, cms.sema.sema_in, 0, 0x7FFF, 0x40000
```

For the real CMC, there shall also be a Timing area sub-module which stores the information about time issues. Since the Timing area is not used for the current design, it has not been implemented and it is not considered in memory map for the router as well.

For the Emmdmac unit, since it has not logical function now, it is also not considered in this memory map. The address range in Figure 3-2 is used for this memory map. In this case, there are only two valid slaves for the router.

The region format is listed as follows:

```
Region_Name,target_name,low_addr,high_addr,capacity, system_base_addr
```
The detail explanation for the each variable can be found in reference [1] page 185.

### 3.3.3 Router Generating

The router is generated by a script named RUNME_MMAPGEN located in the `fm-integrator/` directory. The input to the script is the memory map in the file `system_memory.map`. After the script finishes running, the generated parts of the router are contained in the following three files:

- `Router_binding_system_memory.h`
- `Map_system_memory.h`
- `Router_system_memory.h`

With the base file named `simple_router_base.h`, the router can be instanced and connected in the top. The output port numbers can be set by parameter inside the template and there is only one input socket. If more ports are need for the router, a multiplexer can be added between the router and initiators.

### 3.4 CM & DDR & Memory Initiator

#### 3.4.1 Memory Overview

The CM and DDR are the two instantiations of the memory class by different parameters. And the memory initiator is used to provide the initial value for CM and DDR.

#### 3.4.2 Memory Class

![Block diagram of memory class](image)

Figure 3-3 Block diagram of memory class

Figure 3-3 shows the block diagram of the memory class. There are four functions bound to the input port.
The RAM is a pointer defined as “unsigned char” which is correspondent to the byte aligned address. The size of the pointer would be set by parameter in the constructor. And the initialization function can use the instanced initiator object to set the initial value for each address to the pointer.

3.4.3 Memory Initiator

The initiator for original CM is reused in the new CMC for both CM and DDR. It is a pure C++ class which provides initial value to the memory. Three different kinds of value can be set by constructor parameters:

- Random initializer
- Const initializer
- Single file initializer

For some tests, the specified initial value can be put into the CM and DDR using this initiator.

3.5 Semaphore Unit

3.5.1 Semaphore function description

There are two types of semaphores. One could generate interrupts to the corresponding DSPs and the other one cannot. Except for DSPs, the debug module can also access the semaphore unit. No other modules shall try to access those semaphores.

It is important to notice that except for the basic bit-semaphore, there is a 32bit payload binding with each semaphore.

<table>
<thead>
<tr>
<th>bit_semaphore</th>
<th>32bit Payload</th>
</tr>
</thead>
</table>

For Write operation, it unlocks the semaphore and writes the data into the 32-bit payload. For Read operation, it locks the semaphore and reads data back from the payload if the semaphore is free (“unlock” status). Otherwise the Read operation would be blocked until the semaphore is free. The DSPs can release any semaphore, no matter whether the semaphore is in a free status or not. And it is not allowed to access more than one semaphore at a time.

Except for the Write and Read operation, there is one more operation named Peak. The Peak operation is to read the semaphore value without locking it. And it is done by accessing an odd 16-bit address. For example, to peak a semaphore’s value at address X, the address is set to X+1. The data length for peak operation can be set to either 4 or 3 and the format of return value is listed in Table 3-1.
Table 3-1 Return value format

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
<th>Peak Length = 4</th>
<th>Peak Length = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 63 – 48</td>
<td>Payload Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 47 – 32</td>
<td>Payload Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 31 – 16</td>
<td>Status Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 15 – 0</td>
<td>Status Value</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the interrupt semaphores, each one is corresponding to one DSP. They have all the functions the normal semaphores have but with an additional function. Writing to one of these interrupt semaphores would cause an interrupt issued to the corresponding DSP. And there is no protection for the interrupt data.

### 3.5.2 Semaphore implemented in SystemC

In RTL design, the Semaphore payloads are located inside the memory component. Semaphore block receives the W/R operations which access the Semaphore space and send different responses. But for this SystemC design, the Semaphore variable provided by SystemC standard is used. And there is also a 32-bit array defined to store the payload for each semaphore.

```cpp
enum {
    n_int = 0x40,
    n_sema = 0x2000
};

sc_core::sc_semaphore *_sema[n_sema];
uint32_t _payload[n_sema];
```

Since the address is byte aligned in this design, the ID for the semaphore and the payload can be obtained from address shifted by 2 bit.

```cpp
unsigned int id = add >> 2;
```

For Write and Read operation, they can use the provided “lock” and “unlock” to access the semaphore. At the mean time, they need to either write to or read from the payload. But the “sc_semaphore” used here is an integer semaphore. Binary semaphore is not insisted by SystemC. A check is needed before making the Write operation to semaphores. If the semaphore is already in a “unlock” status, only the value shall be written to the payload while the semaphore’s value is still 1 which is the “unlock” status in this design.

```cpp
if(!_sema[id]->get_value()) _sema[id] -> post() ;
_payload[id] = ptr[0];
```

The Peak operation is judged by Read command with the least two significant bit 0x02. And the peak operation will only read the data from the payload without touching the semaphore variable.

```cpp
bool is_peek = add & 0x02;
if (is_peek){
```
```c
uint32_t status = _payload[id] ? (1<<16) : 0;
ptr[0] = status;
ptr[1] = _payload[id];
```

### 3.5.3 Structure of Semaphore unit

![Figure 3-4 Block diagram for Semaphore Unit](image)

When Read, Write or Peak operation is received by blocking transport function, different operations for payload and semaphore data would be made. And it will also call to the Interrupt Generator if the interrupt semaphore is written. The interrupt generator is connected to those DSPs by “sc_out” ports and the interrupt signal is an inversion of the previous value on the port.

The DMI operation for semaphore module is not allowed so it returns a false in the DMI function. And the debug function is connected to the blocking transport since they play the same function. More checking operations can be added for the debug function when they are needed.

### 3.6 Emdmac Unit

![Figure 3-5 Block diagram for Emdmac unit](image)
Now the Emddmac unit is a dummy unit which only plays the binding rule without the real logical functions.

In this unit, the transaction from input port would be directly forwarded to the output port which is bound to the DDR unit. And the empty function has been bounded with the CMI output port so that it can be further developed to send transactions. The CMI out port is connected to the supervision unit. All transactions sent through this unit should also be supervised by the supervision unit.

For the Circular Bus interface, no operation would be made if this module is the destination. And mails with other destination would be forwarded from in port to out port directly.

3.7 Arbiter

It can be seen from Figure 3-1 that there are 3 input sockets for the SV unit but only one output socket. Because this design is at LT level, timing is not considered, which means that the concurrent accesses would not exist. So all the accesses go into the SV would be forwarded to the output socket in sequence.

However, it hopes that modules at LT level can be further reused for the design at AT level. This Arbiter is a try on improving the module from untimed design to a timed design.

A fix delay is added before the transaction can be forwarded to the router so that there would then exist concurrent accesses in the SV unit waiting to be granted by the added Arbiter.

Figure 3-6 shows the structure designed for the arbiter. All the coming transactions would generate a request, which is put into the corresponding FIFO, waiting to be granted by the arbiter. As this is a LT level design, blocking transport is used for all the modules in this platform, actually there would not be any following transactions before the first one finishes. But the FIFO is still implemented preparing for supporting further AT modules with pipelined transactions. And the FIFO is declared as a pointer pointing to another pointer so that both the depth of the FIFO and the number of parallel FIFO can be dynamic defined through the constructor when the arbiter is instanced.
When the request is at the header place of the FIFO, it would try to get the token. It would be blocked if the token is not there. Otherwise the token would be taken and transaction operation can be made. After the operation has finished, the token would be put back. The following two lines are used for the Get and Put operation on the token. The first one shall be put before the transaction is forwarded to the router and the second line shall be put afterwards.

```c
se_arb->scheduler_wait(id);
se_arb->scheduler_post(id);
```

Since there are four banks for semaphore areas, more arbiters can be instanced. Each bank shall have a correspondent arbiter. Then the commands would be changed to following lines.

```c
se_arb[bankno]->scheduler_wait(id);
se_arb[bankno]->scheduler_post(id);
```

The function releasing the token works in a round robin way with “sc_zero” delay. It would first release a token to the first place. If there is a request waiting for the token, then the token is taken. After a ZERO delay, the function would check if there is a token. The token would be put into the second place if there is a token. Otherwise the function would be blocked until the token is put back by the operation which is described in last paragraph. But one problem may arise with this structure. If there is no request then the function would always wait a ZERO delay and make the next operation. In that case, there would be infinite ZERO delay and the module is dead locked. The solution for this problem is that the function would automatically break out from the loop if there is no request for all the FIFOs. And one more connection is made from the header of FIFO to the function releasing the token. Once a new request is at the header of the FIFO, except for trying to get the token, it would also notify the function to start releasing the token.

Because of the limitation of LT blocking transport, every operation shall be finished inside “b_transport”. The handler can not be implemented to handle those transactions which have passed the arbiter. That's the reason why the complex logic in the previous paragraph is implemented to make the function work. When it goes to AT design, transactions can be copied and passed to other functions. Then the following structure can be made to schedule the transactions. The “Forward Trans” function shall only work in a round robin way to put the transaction from input FIFO to output FIFO. After that, every transaction is in a right sequence and it is then the handler's task to make the different operations from the given transactions. And the module shows in figure below can be implemented as an independent module between the SV and the router.
3.8 CMC Top

3.8.1 All file overview

The structure of the whole CMC is show in Figure 3-1. The file names and the ports’ type are listed in the following table.

<table>
<thead>
<tr>
<th>File name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>cms.h</td>
<td>Top</td>
</tr>
<tr>
<td>cms_sv.h</td>
<td>Supervision unit (SV)</td>
</tr>
<tr>
<td>cms_sema.h</td>
<td>Semaphore unit</td>
</tr>
<tr>
<td>cm.h</td>
<td>Memory</td>
</tr>
<tr>
<td>router_system_memory.h</td>
<td>Router</td>
</tr>
</tbody>
</table>

Since all the function blocks are implemented in separate files, all this files can be reused by other SystemC-TLM2.0 platforms. And when all the modules are instanced in a high level top file, the CMC top file may not be needed.

3.8.2 Hierarchy binding

There are two kinds of hierarchy binding ways which are used in this design. They are TLM sockets hierarchy binding and SystemC ports hierarchy binding.

For TLM socket binding, all the units in this design used the “convenience sockets” provided by TLM-2.0. The initiator socket can be directly connected to target socket. But for hierarchy binding, the TLM socket shall be used instead of “convenience socket”. For the ports using “sc_in” or “sc_out”, they can be directly hierarchy bound without any extra request.

tlm_target_socket<32> *cms_in[CMS_IN_PORT_NUMBER];
tlm_target_socket<32> mp_in;
tlm_initiator_socket<32> mp_out;
cms_in[i]->bind(*(sv->sv_in[i]));
3.9 CMC implementation Summary

This chapter has discussed how the different parts of the CMC are designed. From the functional level, the implemented CMC can meet all the requirements in the specification.
4 Standardized module template and module reusability exploration

4.1 Standardized module template

4.1.1 Old template overview

Figure 4-1 shows what the old structure of the template looks like. The Circular Bus is an interface module containing the “sc_module”, which is the base class for modules in SystemC standard. And the mod_template has inherited from the Circular Bus so that not only Circular Bus interface can be used but also more ports and functions are added into mod_template.

When people begin to develop a new IP module, the IP can derive from the mod_template. Then the virtual functions which are defined inside the mod_template can be rewritten so that those existing ports can be operated for specified functions. More ports or logical functions can also be added for the IP according to the requirement.

However, there exist some problems in this structure. A new structure is needed so that all the problems listed bellow can be solved.

Firstly, redundancy units exist inside the mod_template. The new IP may not need all the ports and functions in the mod_template, such as Circular Bus interfaces or interrupt faces. But the designer still need to handle those useless ports when they do binding operations or other tasks.
Secondly, the generic functions are not implemented. Take CMI write and read operation for an example, if those Dummy DSPs and CPU are created with the mod_template, those write and read operation shall be coded in both created modules. But those codes are exactly the same actually.

Last and most importantly, now we only have a Circular Bus which owns the sc_module. If there comes an x-bus which also owns the sc_module, then the error would occur since it is not allowed for one SystemC module to have two sc_module. Even the C++ virtual inheritance is not insisted by SystemC for the sc_module. Figure 4-2 shows how the error occurs.

![Figure 4-2 Multi-Inheritance Error](image_url)
4.1.2 New template structure

Figure 4-3 gives the new structure of the template. Instead of putting the sc_module inside the Circular Bus, no sc_module exists inside the interface classes now. Instead, the sc_module is put in the same level with other function blocks. One problem raised by this structure is that functions are not able to be set as threads without sc_module. But this problem can be solved in the next design level.

Class cmi_in {
    simple_initiator_socket<cmi_in> cmi_in;
    ...
}

Class cmi_out {
    simple_initiator_socket<cmi_out> cmi_out;
    ...
}

Each function block may contain two classes that one is for output operations and one is for input operations. All the ports and generic functions are implemented in those two classes. When it comes to mod_template, the sc_module shall be derived first. For SystemC design, the sc_module shall always be derived first so that the port in later derived classes can be bound to the sc_module.

class cm:
    public sc_module,
    public cmi_In,
    public Reset_In,
    public Int_Out,
    public rbif<1, 1, 4, uint32_t>,
    public cm_mail_fun{......};
After that, the designers can choose which operation is needed by the requirement and then let the mod_template derive from the selected classes. And then the threads need to be set in this level. Since the derived functions cannot be set as the threads directly, new functions need to be implemented to call to the derived functions. The newly added functions can then be set as threads. In this way, the problem mentioned in last paragraph is solved. The following lines give the example on how to set a function in Circular Bus class to a thread. A new function “rbif_thread1 ()” is implemented and it calls to the “mp_tx()” which locates in the Circular Bus class. And the “rbif_thread1” is set as thread. In this way, the “mp_tx” now works also as a thread.

```cpp
void rbif_thread1(){
    rbif<1, 1, 4, uint32_t>::mp_tx();
}
SC_THREAD(rbif_thread1);
```

Actually with this structure, designers may not need the mod_template any more. They can start their design by choosing the useful classes from those functions blocks. In this way, all the drawbacks mentioned in 4.1.1 are eliminated.

But there is one problem left for this structure, the IP designer shall take care those threads which shall be set for the derived interfaces. The improvement has already made in SSS framework after this thesis work. The pointer to “sc_module” can be passed to the interface classes so that the thread can be set.

## 4.2 Introduction for existing XIO module

The existing XIO module is implemented under C++ and Simics specifications. And it can already work within the Simics environment.

Figure 4-4 shows the structure of the existing XIO module. It can be separated into two parts. One is the XIO core. It contains all the logical functions for the behaviour of XIO and all these functions are implemented with pure C++. But for the XIO core, it also contains the Simics code. They are used to provide some configuration information for both Simics virtual platform and the XIO logical functions. For this SystemC implementation, the Simics code is not insisted by the SystemC compiler. And there is also a request that the core shall also not be changed at any lines. The method to overcome this problem is stated in the wrapper implementation.
The second part is the derived classes, which are all coded in DML language following the Simics specification. Four of the classes are used for the 4 interfaces of the XIO and the left two is used for configurations. In the interface files, the Macros “#ifdef Simics” and “#ifdef SystemC” are used so that it can switch between the two specifications during compiling. The Simics implementation is already done, while the SystemC implementation shall be done for this design. Table 4-1 gives the basic information for those 4 interfaces. Since no changes shall be made for the XIO core and the classes are inherited by the XIO core class, the same class name as Simics implementation shall be used for all those derived classes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Bounded Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Interface</td>
<td>Unidirection</td>
<td>Common Memory</td>
</tr>
<tr>
<td>XIO interface</td>
<td>Bidirection</td>
<td>Antenna</td>
</tr>
<tr>
<td>Reset interface</td>
<td>Unidirection</td>
<td>CPU</td>
</tr>
<tr>
<td>Circular Bus interface</td>
<td>Bidirection</td>
<td>Circular Bus</td>
</tr>
</tbody>
</table>

Table 4-1 Basic information for interface

4.3 Different methods for the wrapper implementation

4.3.1 General description for the wrapper

There are many different ways to wrap an existing C++ core. Different methods will be analyzed in the later part. And for this thesis, since the XIO core also contains the Simics code, some more issues are to be solved compared with the pure C++ core.

Because the XIO can be separated into two parts, the design for the wrapper can mainly be separated in three steps:

- Overlap the Simics codes in the XIO core
- Implement the Interface part for XIO
- Implement the wrapper which contains all the XIO and TLM sockets.

There are two important factors that needed to be considered for the C++ structure of the wrapper. One is how to pass the received data from SystemC-TLM2 port to the interface of the core. The other is how to receive callbacks from the core. The data can then be sent out through SystemC-TLM2 port.

Following methods have been investigated in this thesis work. General solutions on how the two factors mentioned above are provided for each way. And the way adopted in the XIO wrapper design will be documented in detail in chapter 4.4.

4.3.2 Mix all the XIO functions in the wrapper

The wrapper’s class can inherit from the top class of the core. In this case, every function in XIO interfaces and XIO core will be mixed together with those functions in the wrapper, which are used for the SystemC-TLM2 communication with outside components.
In this design, since all the functions are flat and visible to each other, functions which are implemented in the interface files and bounded with the core can be called directly by functions, which are implemented in the wrapper. While it is the vice verse for the function call from XIO core to outside modules.

But the drawback is that the boundary is not clear. It is more like a software design rather than a hardware design. And since everything is flat, the variables and functions are in danger of hidden disturbance.

4.3.3 Instance the XIO core as an Object

The XIO can be instanced as an Object in the wrapper’s class.

With this method, the functions of the Object can be called in order to pass the data from the wrapper to the object. But the method to receive the callbacks from the object is much complex than the other two way. Detail implementation about this can be found in chapter 4.4.

The drawback for this method is that it has the higher complexity. However, the consistency of the XIO core between the two different working environments can be guaranteed. That’s the main reason for the adoption of this method in this thesis.

4.3.4 Reuse the XIO logical functions and implement it with SystemC

The XIO logic can be extracted from the given code and wrapped with SystemC ports. SystemC ports can be directly added into those functions that are bounded with the XIO logic. And then the bidirectional commutations can be easily done. But with this method, the consistency is destroyed. Or much effort shall be made to demonstrate the consistency of the two implementations.

4.3.5 Summary of wrapper design methods

Based on the pros and cons for the above three wrapper design methods, the second one is adopted in the XIO wrapper’s design.
4.4 XIO wrapper implementation

4.4.1 Structure of the SystemC XIO

The XIO core with derived interfaces is instantiated as an object in the XIO wrapper. While the left blocks in the wrapper contain the TLM sockets and functions which are in charge of passing the operations bidirectional.

If there are template classes in those core files, it will not be able to separate the functions implementations in the CPP file for those template classes. So which structure shall be used is decided by whether the core owns a template or not.

4.4.2 Wrapper implementation

As mentioned in Section 4.3.3, the difficulty for this structure of wrapper is to solve the call back problems. Since the wrapper needs to communicate with the interfaces of the XIO, the design of the functions in the interface classes has big influence on the wrapper’s design.

Because of the bidirectional communication requirement, the wrapper class has to know the definition of the interface classes and the interface classes shall also know the wrapper. The situation would be like the following pictures that they shall “include” each other.

![Figure 4-5 Block diagram of XIO](image)

![Figure 4-6 Conflict of forward declaration](image)
But this situation would lead to “forward declaration error”. So the following method which shows in Figure 4-6 can be used to make the correct forward declarations. A class shall be separated into “.h” and “.cpp” files.

```
wrapper.h
namespace xio { class xIO; }
class wrapper {
    xio::xIO *xio_core;
}
```

```
wrapper.cpp
......
call to m_parent->fun()
......
```

```
circularbus.h
class xio_wrapper;
class circularbus {
    ......
    private:
        wrap *m_parent;
}
```

```
circularbus.cpp
#include "xio_wrapper.h"
......
call to m_parent->fun()
......
```

**Figure 4-7 Correct structure of forward declaration**

In “.h” file, forward declaration shall be made and the pointer shall be defined. While in “.cpp” file, the space for the pointer can be allocated by “new” in the constructor.

But for the existing XIO interface class, the structure shown in Figure 4-7 cannot be used either since there are “template class”, which cannot be compiled with separated “.h” and “.cpp” files, inside the interface files. So a new interface class is added for the wrapper which is show in Figure 4-8. The defined virtual functions are re-implemented in the “.cpp” file of the wrapper. And now for the “.cpp” file of the interface, it only need to know the wrapper interface file so that it can make the callbacks from the core to the function in the wrapper.

```
wrapper_if.h
class wrap_if {
    public:
        virtual void fun (xxx)=0;
    ......
};
```

```
wrapper.h
#include "wrap_if"
class xio_wrapper {
    xio::xIO *xio_core;
}
```

```
wrapper.cpp
#include "ringbus.h"
wrapper :: void fun(XXX){}
......
```

**Figure 4-8 Forward declaration for wrapper with template class**

After the structure of the wrapper is decided, functions for TLM communication are added. As it is the wrapper who contains the “sc_module”, all the TLM sockets or ports shall be defined in this module. There are four interfaces derived by the XIO core. Correspondingly, there are four functions handling different protocols communicating with other modules by TLM sockets. Those functions can be found in Figure 4-5.
Figure 4-9 shows the implementation for CB interface in the wrapper. There is a thread checking the received mails. If there is a Mail for XIO, it would take the mail and reorganize the mail from 8-bit into a 32-bit width array. There is also a port check operation after which the port number and the newly created array would be passed by parameters inside the function call. The called function locates in the interface class of the XIO core. For the opposite direction, after the defined function “mail_i2o” is called, it would set the received data into mail format and then sent it out.

### 4.4.3 Implementation for XIO interfaces

As listed in Table 4-1, two of the interfaces are bidirectional and the other two are unidirectional. SystemC implementations are added into those classes so that the core and the wrapper can be connected.

For passing data from wrapper to the XIO core, there are usually two functions. The first one is called from the wrapper and it calls to the second function. The second function is a virtual function which is declared in this interface class and implemented in the XIO core. With these two functions, the data can reach the XIO core.

For passing data from the XIO core to the wrapper, there is only one function which is called by the XIO core. Inside the function, it calls to the corresponding function inside the wrapper. In order to achieve the calls from the core to functions in the wrapper, there is one function made to store the pointer of the wrapper or wrapper interface in a private variable. This function is called by the constructor of the wrapper. The code is shown below:

```cpp
void cmi_set_parent(xIO_wrap_if *m) {
    m_parent=m;
}
```

```cpp
private:
xIO_wrap_if *m_parent
```

The three interfaces which contain the callback from the XIO core shall be given the pointer in the wrapper's constructor.

```cpp
xio_core->rb_set_parent(this);
xio_core->xio_set_parent(this);
xio_core->cmi_set_parent(this);
```

### 4.4.4 Overlap the Simics code in XIO core

There are three type of Simics codes which need to be overlapped in the SystemC design.
First is the defined Macro in XIO.h file which are used to provide information for Simics platform.

```
SVP_DECL_REG_ATTR(uint32_t, regs, enable);
```

These Macros actually have no use for SystemC platform so that they are defined as empty Macros.

```
#define SVP_DECL_REG_ATTR(...)  
#define SVP_DECL_REG_ATTR_VECTOR(...)  
#define SVP_REGISTER_RESET_IN_INTERFACE(...)  
#define SVP_REGISTER_RINGBUS_INTERFACE(...)  
#define SVP_REGISTER_XIO_INTERFACE(...)  
#define SVP_REGISTER_MEMORY_OUT_INTERFACE(...)  
#define SVP_DEF_REG_ATTR(...)  
```

The second is init_local() function which is also not needed for SystemC design. And it contains the follow code which can not be complied by SystemC compiler.

```
svp::Class<xIO> classDef("xIO");  

classDef  
  << SVP_REGISTER_RESET_IN_INTERFACE(xIO)  
  << SVP_REGISTER_RINGBUS_INTERFACE(xIO, true, 9)  
  << SVP_REGISTER_XIO_INTERFACE(xIO, 4)  
  << SVP_REGISTER_MEMORY_OUT_INTERFACE(xIO);  
```

The Class<xIO> is defined in the file “base.h” under Simics implementation. So an empty class named “Class” is coded in “base.h” and it also contains a friend class shown below:

```
template<class T> class Class  
{  
  public:  
    Class(const std::string& name){}  
    friend Class& operator<<(Class& _this, int i) {  
      return _this;};  
};  
```

The third one is the functions which are coded by DML inside the log file. Since these functions do not influence the logical functions of the XIO, they are defined as empty functions in C++ so that they would be compiled. Further work can be made on this part.

```
void info(int level, const char* fmt, ...) {}  
void error(const char* fmt, ...) {}  
```

With these codes, the XIO core can be compiled. And the XIO modified with those changes works fine in SystemC environment.

### 4.5 SSS framework for blocks

#### 4.5.1 Structure of the SSS framework

![Class inside the framework](image-url)
Figure 4-10 shows the constitution of the adaptation layer. Except for those components mentioned in Figure 4-4 which are derived by XIO core, there are one more interface and several files for configurations. Since the XIO is the only module which is available to be used to test this adaptation, only those used classes are developed for the framework in this thesis.

![Diagram](image)

**Figure 4-11 Modules working in both environments**

With the developed framework, XIO shall be able to work in both SystemC and Simics environment. After the structure of the SSS framework is successful tested, further work can be made to implement the left parts in SystemC so that all the C++ cores following the specification can work in both virtual platforms with this adaptation layer, which is show in Figure 4-11.

### 4.5.2 Interface implementation

With the wrapper of XIO, it can already work in the SystemC environment. The task for this design is to merge all the TLM operations into the interfaces of the XIO core so that the wrapper can be eliminated.

For a System module, it has to derive from “sc_module” first. The XIO core does not contain the “sc_module” so that it has to derive the “sc_module” from those classes in SSS. For the current environment, the CB interface class has already derived from “sc_module” and it is not possible to re-implemented and test a new Circular Bus in this thesis.

```cpp
class xIO :
  public virtual svp::Device,
  public virtual svp::Log,
  public svp::iface::reset::In,
  public svp::iface::RingBus<true, 1 + xio::rbx::tunnels>,
  public svp::iface::xIO<ports>,
  public svp::iface::memory::Out{...
```

But from the given code above, the “Circular Bus” is inherited after “reset::in”. That’s lead to a problem that the socket in “reset” can not bounded with “sc_module”. Because of this, the sequence for inheritance is changed. “Circular Bus” is inherited before “reset” in this thesis.
For further development, the CB can be re-implemented without the “sc_module” which can be put into “Device” class. And the TLM socket in the “reset” can be replaced by “sc_port”.

Except for the problem mentioned above, the left work is only to move all the functions coded in the wrapper into the four interface classes derived by the XIO core. The work for overlapping the Simics code in wrapper’s design can be directly reused in this design for “base.h” and “log.h”

Figure 4-12 Function connection in the wrapper

Figure 4-12 gives the structure of a communication from wrapper to the core. There are three parts which are wrapper, XIO interface and XIO core. The “receive_rb_mail” is the virtual function which is re-implemented in the core. Now with the new implementation, only the interface of SSS and the core exist, which is show in Figure 4-13.

Figure 4-13 Function connection in the SSS framework
5 Testing

5.1 Dummy Modules

5.1.1 Dummy DSP

Figure 5-1 Block diagram of the Dummy DSP

Figure 5-1 shows the structure of the dummy DSP. It contains a socket for CMI operations. Four generic functions are implemented, by which the data and address can be passed through parameters in the function call.

```c
void read(const sc_dt::uint64 &addr, unsigned int &d)
void write(const sc_dt::uint64 &addr, const unsigned int &d)
void dmi_read(const sc_dt::uint64 &addr, unsigned int &d)
void dmi_write(const sc_dt::uint64 &addr, const unsigned int &d)
```

Take read operation for an example, the integer passed would be rewritten with the data at the specified address. Except for CM write and read operation, the pend and post accesses for semaphore area can also be done through this write and read operations. But for semaphore peak operation, since the data length is 8 and it shall return two integers which are the semaphore value and status, an independent function is implemented only for peak operation.

```c
void peek_read(const sc_dt::uint64 &addr, unsigned int &d, int &st)
```

There is also an Interrupt Handler bound with the interrupt port so that it can collect the interrupt signal. In this design, it is an edge trigger interrupt receiver. Either positive or negative edge can be captured by the interrupt handler.

For the Circular Bus interface, there are no operations made for sending or receiving data from CB. If any data is sent to the DSP through CB, it will generate an error message.

The Run function is the main thread which makes the call to those three different operations. It shall be redefined according to different requests for the DSPs. Except for driving operations, it shall also analyze the result for those operations and store the result into the registers.

The registers are used for collecting the running information and would be used to give the test result for the tested modules. The type defined for the interfaces are listed in Table 5-1.
Table 5-1 Interfaces for the DSP

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmi</td>
<td>simple_initiator_socket</td>
</tr>
<tr>
<td>interrupt</td>
<td>sc_in&lt;bool&gt;</td>
</tr>
<tr>
<td>circular bus</td>
<td>simple_initiator/target_socket</td>
</tr>
</tbody>
</table>

5.1.2 Dummy CPU

![Figure 5-2 Block diagram of dummy CPU](image)

Figure 5-2 show the structure of the Dummy CPU. For the DMI and Blocking operations, they are the same implementations like what are made in the Dummy DSP.

Different from the dummy DSP, the interrupt port and handler are replaced by a reset port and reset signal trigger. The reset signal is a raise edge.

For the Circular Bus operations, since the CPU is the central controller which needs to send out configuration data to the target modules, the function is implemented to send out the configuration data. In the virtual platform developed by this thesis, the XIO and CM are the modules which request to be configured and the configuration information is quite different from each other. So there are two functions that one for each module. The CPU also needs to monitor the received data from CB. In this design, the error messages are received and analyzed by the added function.

For the RUN function, it now can call to those 4 functions according to the design requirement. Both the run function and error message analyze function can write their results into the registers which has the same function as the one in Dummy DSP.

Table 5-2 Interfaces for dummy CPU

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmi</td>
<td>simple_initiator_socket</td>
</tr>
<tr>
<td>reset</td>
<td>sc_in&lt;bool&gt;</td>
</tr>
</tbody>
</table>
5.1.3 Dummy Antenna

Figure 5-3 Block diagram of dummy antenna

Figure 5-3 shows the structure of the Antenna. Since XIO had already worked in Simics virtual platform and there is a similar module which plays the Antenna functions for testing the XIO, this SystemC Antenna follows the same behavior as the one in Simics environment. Like other dummy modules, there are four functions can be handled by the thread Run which needs to be costumed to meet the requirement. And the registers are used to store the runtime information for further analyze.

In Simics, the IQB data is sent in following command:
```
send_xio(    0x05, 0x00, 0x00, 0x00, // header
             0x00, 0x00,            // header
             0x01, 0x00,            // rf
             0x02,                  // hf
             0x0b,                  // columns
             0x00, 0x00,            // padding
             0x00, 0x00, 0x00, 0x00); // size
```

There are two operations for IQB data. One is named IQB RX. It is used for sending data to the XIO module. Four “send_xio” commands shall be sent and the last “send_xio” has one extra parameter which is a data pointer pointed to the data array that shall be sent to XIO.

The other operation is IQB TX. It is used for receiving data from XIO. After five “send_xio” commands are sent, the XIO_IN port can receive the data from requested addresses.

For this SystemC module, the “send_xio” is also implemented so that the commands are sent in the same way as the Simics implementation. All the parameters are put in sequence in an array. The data pointer for the TLM transaction is set to the array and the sent out.

For blocking transport which is used to receive the data from XIO_IN socket, the first byte shall contain the value “0x05”. And the received data starts from the 16th byte of the data pointer.
The reset signal is sent through the reset port. At first, due to the structure of the XIO, the reset port was implemented with TLM socket. So the antenna makes a blocking function call without any address or data set. Later, the reset port is changed to "sc_out". And then the reset signal is an inverter of the previous value on the port.

5.1.4 Summary for dummy modules

These dummy modules are created according to Figure 2-1 and they are used for testing the CMC-XIO based virtual platform. With these dummy modules, not only the logic functions for CMC and XIO can be tested, the communication protocols on different interfaces can also be tested.

5.2 CM Test

5.2.1 Test Platform

Figure 5-4 Test platform for CMC
Figure 5-4 shows the test platform for CMC. Two dummy DSPs and one dummy CPU are connected with the common memory. Compared with the real SoC platform shown in Figure 2-1, there are no interface blocks and hardware accelerators. But since all the components communicate with CMC through CMI and CB interfaces, with this basic platform working fine, other modules later built shall be able to be plugged into this platform. Although there are three dummy modules which can be used in this test platform, the modules to be used can be selected for different tests. For those unbound ports, like the left interrupt ports and all reset ports, fake signals are defined to be connected so that there would not appear “unbound port” error during runtime.

As it is said in chapter 5, the “RUN” thread for different modules shall be implemented according to different requirements. In this CMC test platform, the dummy CPU is mainly a controller which sends configuration mails and receives error mails. And the dummy DSP would make different operations through CMI and receive the interrupt signals. After the simulation is started, it would stop after all the threads in different dummy modules finish. And those dummy modules need to print out the result of the test.

The CMC is composed by 6 sub-modules and each sub-module contains the functions to be tested. For the “Router”, it is created by Cadence tool so that it is assumed to be all right. And since the “Emdmac” contains no logical functions, it is also excluded from the tests. Functions in those left sub-modules shall all be tested.

5.2.2 Blocking Transport and DMI Test

In this test, the supervision is not started. The operations, which go through blocking transport or DMI, can directly reach the CM. And the DMI request is also granted by CM without filtered by supervision unit. Thus the test focuses on the CM sub-module only.

![Figure 5-5 Test for CM](image)

Only one DSP is added in the top for this test case. For the thread “RUN” in the DSP, a set of data is written into the CM with given addresses. Then the data would be read back. Both the blocking transport and DMI would be used for the read and write operations. The comparison made for the written and read data shows that the Blocking Transport and DMI function are working fine.

5.2.3 Semaphore test

There are 4 functions implemented in the Semaphore module that need to be tested. They are pend, post, peek and interrupt operations. Two DSPs are connected with the CM and the supervision is also not started.
For pend and post function test, the DSP0 would continually perform pend operations with hard coded addresses. Since all semaphores are initialized to ‘0’ at the start, the DSP0 would be blocked until the DSP1 releases that semaphore. The first DSP0 would also compare the payload value of the got semaphore with a hard coded value which is the same as the value written by DSP1. The information would be recorded in the registers. In the destructor of the DSP0, the information in the registers would be checked and the result whether the tests had succeeded or failed would be printed out.

For interrupt function test, DSP0 would write to the interrupt semaphore address which is the interrupt semaphore for DSP1. The DSP1 is in charge of receiving the interrupt signal and comparing the payload value of the semaphore with the hard coded value. The information would also be recorded and used by the destructor to print the test result.

For peek function test, the DSP0 would perform some pend and post operations for some specified addresses. Then the DSP1 would make the peek operation for those accessed addresses by DSP0. The value of the payloads for those addresses shall also be compared and the information shall be stored in the registers. Then the destructor would do the same thing as above tests.

5.2.4 Supervision test

One DSP and one CPU are connected to the CM for this test and this supervision test is separated into 2 steps. For easy implementation, the CPU would play both the CPU and DSP functions.

The first step is for CPU to send configuration mails to the supervision unit.

Table 5-3 Initial value for all variable in SV
<table>
<thead>
<tr>
<th>enable_mask[0] – enable_mask[3]</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_irq</td>
<td>0</td>
</tr>
<tr>
<td>M_uint</td>
<td>0</td>
</tr>
<tr>
<td>M_cmcclock</td>
<td>False</td>
</tr>
<tr>
<td>M_cmen</td>
<td>False</td>
</tr>
<tr>
<td>M_setupall</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 5-4 Initial value for masks of SW defined areas

<table>
<thead>
<tr>
<th>Start_addr</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>End_addr</td>
<td>0</td>
</tr>
<tr>
<td>Mask[0] – Mask [3]</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5-3 and Table 5-4 show the variables used for whole supervision unit and their initial value.

In this test, for each area there are three mails sent out which contain the information shown in Table 5-5. The variable “areano” stands for the number of the area. In order to simplify the test, all the masks of different areas are set to the same value. The set value for mask 0-2 means:

- Only DSP0 can access the 32 SW defined areas.
- All the blocks shall not be supervised.

Table 5-5 Set value for masks of SW defined areas

<table>
<thead>
<tr>
<th>Start_addr</th>
<th>areano*0x10</th>
</tr>
</thead>
<tbody>
<tr>
<td>End_addr</td>
<td>areano*0x10+0xF</td>
</tr>
<tr>
<td>Mask[0] – Mask [2]</td>
<td>0x0,0x1, 0xFFFFFFFF</td>
</tr>
</tbody>
</table>

After the 32 SW defined areas setup finished, the destination ID for error message is set and the supervision is then enabled. The other global variables which are not set by mails would be used with their initial values.

The information for all variables for the 32 SW defined areas would be print out in the console like the example shown below:

For area 0 Start address is 0x0; End address is 0x1e; DSP mask1 is 0x1; DSP mask2 is 0x0; Block mask is 0x7fffffff
The second step is for two DSP to generate some operations for different address to see if the supervision can find the invalidation by the given configurations. According to the configuration made in last step, only DSP0 can access those 32 SW defined areas. In this situation, the right behavior for CM is that it would send out the error mails containing the validation information for DSP1. The CPU is in charge of receiving the mails and recording them into the registers. After the entire test finished, the destructor need to print out the test result according to the data in the registers.

5.2.5 DMI invalidation Test

For the DMI invalidation test, one DSP and one CPU are connected to the CM. And the whole test is separated into 5 steps.

The configuration used is the same as the one used in the step 1 of the supervision test. After that, the “enable_mask[0]” is changed from initial value to “0x00000003”, which means no DSP shall be supervised.

For step 3, the DSP0 requests DMI for the CM and the request is granted. Then in step 4, the “enable_mask[0]” is changed to “0x0”, which means that all DSP shall be supervised. Because of this change, the DMI invalidation function is triggered.

And in step 5, the DSP0 shall receive the DMI invalidation function call and record the information. The destructor would print out the test result in the end.

5.2.6 Arbitration Test

This arbitration test is made for testing the performance of the arbiter. There are 3 DSPs connected to the CM in this test.

![Figure 5-8 Test for arbitration](image)

For this design, all the CM values are initialized to 0 and the write operation shall be any number but not 0. The three DSPs would continually access the CM at the same time. DSP0 and DSP1 would only do the write operation while the DSP2 would make the read operation.
The same test would run twice for the CM. For the CM without arbiter, the DSP2 would either get the value write by DSP0 or DSP1 or a CM initial value '0'. But for the CM with the arbiter, it would always get the value written by DSP1.

5.2.7 A demo test for all functions

In order to demo the performance of all the implemented functions, a combined test contains all the tests mentioned above has been carried out. One DSP and one CPU are used. Like the test for supervision, the CPU would also play the function of DSP1.

Table 5-6 Test schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>DSP0/ CPU</th>
<th>Expected Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0ns</td>
<td>Send Configuration mails for supervision</td>
<td>Print out configuration information</td>
</tr>
<tr>
<td>1ns</td>
<td>Blocking Write operation</td>
<td>Blocking Write operation</td>
</tr>
<tr>
<td>1ns+0</td>
<td>Disable supervision for DSP0 and DSP1</td>
<td></td>
</tr>
<tr>
<td>2ns</td>
<td>Blocking Write operation</td>
<td>DSP0 shall not be supervised</td>
</tr>
<tr>
<td>3ns</td>
<td>Receive and count Interrupt</td>
<td>Access interrupt semaphore for DSP0</td>
</tr>
<tr>
<td>4ns</td>
<td>Semaphore pend</td>
<td>DSP0 is blocked</td>
</tr>
<tr>
<td>5ns</td>
<td>Semaphore post</td>
<td>DSP0 is unlocked</td>
</tr>
<tr>
<td>6ns</td>
<td>DMI read operations</td>
<td>DMI granted and read operation succeed</td>
</tr>
<tr>
<td>7ns</td>
<td>Enable supervision for DSP0 and DSP1</td>
<td>DMI invalidation received for DSP0 &amp; DSP1</td>
</tr>
<tr>
<td>8ns</td>
<td>DMI request</td>
<td>DMI denied</td>
</tr>
<tr>
<td>9ns</td>
<td>Post and Peak semaphores</td>
<td>Correct payload got from peak operation</td>
</tr>
</tbody>
</table>

5.3 XIO wrapper test

5.3.1 XIO test platform

For the XIO wrapper test, the modules in Figure 5-9 are used to construct the test platform. The CM in Figure 5-9 is the same CM sub-module in Figure 3-1. The CMC implemented before is not used because only the Write and Read operations are made by XIO. As the CMC is well tested, it shall also work fine with this platform.
5.3.2 XIO test process

Since the XIO core is already fully tested, the test in this design focuses on if all functions for TLM communication and interface functions work in a right way. IQB data communication is selected since all the interfaces are used.

Firstly, the CPU would send IQB RX configuration mails to the XIO. The data for configuration mails are reused from Simics test file but they are reorganized in this SystemC Circular Bus format.

Secondly, the dummy Antenna would send IQB commands with data to XIO. For this IQB data, it would be directly forwarded to specified address in CM.

Thirdly, the CPU would then send IQB TX configuration mails to the XIO.

Fourthly, the dummy Antenna would send IQB command to the XIO, after which the XIO would read data from CM and forward them to the dummy Antenna.

Fifthly, the dummy Antenna would compare the data sent with the received. Since the address for Write and Read operation are the same, the written values are hoped to be read back after IQB TX operation. The correctness is stored in the registers.

5.3.3 XIO test result

In destructor, it would check the register. The value is 18, which means that all the data read back matches the data sent into the XIO. So the test is successful and the “XIO test passed” would be printed out on the console.
5.4 **SSS test**

All the modules and test steps in the previous test are reused for this SSS implementation with XIO core. Both implementations give the same result, thus the SystemC SSS structure is tested and proved to be working fine.

5.5 **Standard template test**

The CMC is made with the new interface structure described in chapter 4.1.2 and the demo test is reused for the CMC implemented in this way. Since for both CMC the test has given the same result, this new template is proved to be all right. And as it is said in last paragraph in chapter 4.1.2, the new SSS framework also made improvements for this module template.

5.6 **Comparison between RTL and SystemC implementation**

5.6.1 **Comparison on complexity**

The following three tables show the comparison on the factor of the amount of files and code lines for CMC and XIO. For Table 5-8, it is worth to be noticed that the files for SSS contain both Simics and Simics code, which would be used for all HW-accelerator.

<table>
<thead>
<tr>
<th>Languages</th>
<th>files</th>
<th>code lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemC</td>
<td>9</td>
<td>1203</td>
</tr>
<tr>
<td>VHDL</td>
<td>184</td>
<td>25159</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Languages</th>
<th>files</th>
<th>code lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemC</td>
<td>23</td>
<td>6953</td>
</tr>
<tr>
<td>VHDL</td>
<td>34</td>
<td>17644</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Languages</th>
<th>files</th>
<th>code lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemC</td>
<td>8</td>
<td>3638</td>
</tr>
<tr>
<td>VHDL</td>
<td>34</td>
<td>17644</td>
</tr>
</tbody>
</table>
It can be seen from those three tables that the complexity for SystemC implementation at LT level is largely reduced compared to the RTL design.

5.6.2 Comparison on simulation time

In reference [2], a test case file with read and write operations for that RTL CMC is generated. The test case file contains 100 thousand commands and an ID mark for each command. As shown in Figure 5-10, those DSPs would read from the file and make the operations according to obtained commands and the corresponding ID. The simulation time for that test case is shown in Table 5-10.

![Figure 5-10 Platform for testing CMC CPU runtime](image)

The same test platform is made for the CMC implemented in this thesis. Those dummy DSPs are instanced. They would read from the same test case file and then make the blocking write or read operations with those commands. The same test is run for 100 times and the CPU runtime for each simulation is recorded as shown in Figure 5-11.

![Figure 5-11 CPU time for LT CMC](image)
The average CPU runtime for Figure 5-11 is recorded in Table 5-10.

<table>
<thead>
<tr>
<th>Languages</th>
<th>Simulation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemC</td>
<td>0.081381 S</td>
</tr>
<tr>
<td>VHDL</td>
<td>16446 S</td>
</tr>
</tbody>
</table>

As we can see from the table above, the CPU runtime for SystemC modules is around 200,000 times faster than that of VHDL modules.

Usually, the SystemC module is around 100 -1000 times faster compared to the VHDL module. In this thesis, the fairness in the simulation is not analyzed. For an example, the supervision is not enabled for this SystemC CMC test. For future exploration, please refer to reference [2].
6 Conclusion and Future Work

6.1 Conclusion

The CM-XIO based virtual platform works fine for all the functions that the CMC and XIO shall have. Since different operations through CMI, CBI, Interrupt and Reset are tested, more modules can be plugged in according to the TLM interface standard.

Different methods for wrapping C++ modules to SystemC are compared. The best method is tested and detail information is documented. It would be much easier to plug those existing C/C++ modules into this virtual platform by following the given method.

The SSS structure is tested. With the given SSS in this thesis, after the “event” interface is added, all the HW accelerators which are working in Simics can directly be switched between the SystemC and Simics platforms.

From the comparisons on the complexity and simulation time, modules built in SystemC are proved to have less workload and can give a much faster simulation performance.

6.2 Future Work

The SSS framework shall be improved to give a better support to all different interfaces so that all the HW accelerators can be switched between Simics and SystemC VP.

The data to be transferred for the dummy antenna in this thesis are hard coded. The “xio_file” which is written in Simics shall be implemented in SystemC so that the data can be dynamically filled into the CMC from given files.

The dummy CPU and DSP shall be replaced by modules which behave the same as the real CPU and DSP at the functional level.

After all the modules are ready, operating system shall be installed on the VP and software shall be run in the integrated operating system.
7 Reference


