A novel system-level methodology for the design and implementation of multiplexed master-slave system-on-chip components using object-oriented patterns
Menon Satish, S.; Jayadevappa, S.

Published in:

Published: 01/01/2011

Document Version
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
A Novel System-Level Methodology for the Design and Implementation of Multiplexed Master-Slave System-on-Chip Components using Object-Oriented Patterns

Sushil Menon and Suryaprasad Jayadevappa

Abstract—In a customer driven environment, systems pose a great challenge to designers in terms of complexity and time-to-market. In the past, designers have looked at Commercial-Off-The-Shelf (COTS) based development techniques that rely on integrating components produced by various manufacturers. However, the vast range of components from various manufacturers leads to obscurity of interfaces between them, making system integration a dreadful task. Hence, large warehouses of intellectual-property (IP) modules comprising of various software simulation models of system components at different levels of abstraction that can be readily integrated to form a Virtual Prototype are highly beneficial towards the exploration of various architectures.

In order to design and implement IP modules efficiently, we identify the Design Patterns that they follow. We propose a Design Pattern based methodology for modeling and implementing multiplexed Master-Slave devices at the system-level. As an effort to showcase the methodology, we choose to model a Level-2 Cache (L2Cache) using SystemC. A technical specification document of the Motorola MPC2605 L2Cache is procured and modeled using hierarchical state machines that we implement using SystemC. The developed model is an integrated look-aside, no write-allocate L2Cache supporting 4-way set-associative cache mapping and LRU replacement algorithm, compatible with the Simple Bus. The L2Cache module is then further parameterized to accommodate changes in L2Cache size, cache mapping strategies and replacement algorithms.

In this paper, we present the proposed methodology through a description of the process of modeling and verifying an L2Cache IP module at system-level, using the Master-Bus-Slave/Master-Bus-Slave Design Pattern. The test-bench involves a master (to generate transactions), a Simple Bus, the L2Cache Module and Slave Module (Simple memory). Various relevant test cases were generated to test the functionality of the developed module. The test-case outputs were routed to a debug file which is inspected to determine whether the output is as expected. We conclude by enunciating our learning from our experience with this methodology.

Index Terms—Electronic System Level, Level-2 cache, Intellectual Property, design patterns, Transaction Level Modeling, SystemC

I. INTRODUCTION

EVER inflating user requirements have been posing a great challenge to the designer in terms of increasing system functionality. Today’s typical cell-phones have additional features like cameras, MP3 players, Internet browsers and even televisions. To support these features they comprise of about more than 80 peripheral devices [2]. Further, cell-phones are estimated to contain over a million lines of code [1]. Increasing system complexity in terms of hardware and embedded software add to the difficulties of system design. System Architects are further strained with diminishing time-to-market. The estimated time to convert a modern day cell-phone from concept to product is typically about 18 months [2].

COTS has provided the interim relief to handle the current issue of complexity and time-to-market. However, the increasing sources of components lead to an increase in the probability of an architectural mismatch, thereby making systems difficult to integrate, and also increases the cost of system integration. Hence, System Architects need to move up the value chain, using newer tools and methodologies that are superior to the current method of madness of designing systems. Electronic System Level (ESL) design is an effective alternative for System Architects, enabling them to combat the complexities and requirements of systems to come, also supporting architectural exploration or “What if?” scenarios very early in the design cycle.

ESL design deals with designing systems at higher levels of abstraction. Higher levels of abstraction increase system comprehension by capturing the functionality of a component and eliminating unnecessary details that are required for system synthesis, thereby providing high-speed simulation models that are suitable for the verification of software applications. The Electronic Design Automation (EDA) industry uses several tools, including Rosetta, SystemVerilog and SystemC for modeling systems using ESL design.

SystemVerilog (IEEE standard 1800) is used in the industry for hardware design and verification at the system-level. It is an extension of the Verilog Hardware Description Language (HDL) which supports the design and constrained random verification of parameterizable hardware modules wherein random test-vectors are input as a stimulus to the design under test. The results of the simulation are automatically verified using language assertion features
called SystemVerilog Assertions. SystemVerilog also supports the usage of functional-coverage features that measure the completeness of the testing performed, and constraint features that direct test-vectors towards boundary test-cases, thereby increasing the test-coverage.

Although SystemVerilog supports system-level hardware design and verification, its HDL roots do not permit the co-design and development of hardware and software applications which are typically developed using Object-Oriented software languages such as C++. Hence, there is a need for a modeling platform which supports ESL design and enables the simultaneous design and development of hardware and software.

SystemC (IEEE standard 1666) is a commonly used platform that supports ESL design and enables hardware-software co-design. It is an open-source event-driven simulation kernel written in C++ that supports modeling at various levels of abstraction including Transaction Level Modeling (TLM). TLM allows designers to model systems at a higher level of abstraction, by separating component functionality from inter-component communication. Thus, TLM is highly suitable for implementing Virtual Prototypes at higher levels of abstraction.

Virtual Prototypes are software simulation models of the hardware components of a system that provide the complete functionality of the system, thereby enabling the usage of software simulation models during early phases of system architecture and design, hence aiding architectural exploration – “What if?” scenarios. Virtual Prototyping provides functionally accurate models of the hardware, allowing software development at early stages of system development, and also provides System Architects with a platform for analyzing performance statistics such as power estimation at the system-level [5]. Figure 1 illustrates a typical Virtual Prototype – Virtio’s VPXS Virtual Platform. It comprises of an instruction-accurate instruction-set simulator (ISS) of the Intel XScale core, functional models of the respective peripheral system-components and a cycle-accurate model of the on-chip bus which is implemented in SystemC [8].

However, there exists a class of integral system components, such as L2Caches, DMA Controllers etc., that exhibit multiplexed Master-Slave functionality. Such system components adopt the Master-Bus-Slave/Master-Bus-Slave pattern, where \( \text{Slave/Master} \) represents the multiplexed Master-Slave device. Figure 4 illustrates the Master-Bus-Slave/Master-Bus-Slave pattern and Figure 5 shows the sequence of messages exchanged in a typical Master-Bus-Slave pattern based system. In normal-mode, the multiplexed Master-Slave operates in a manner identical to that of the Master-Bus-Slave pattern. However, in multiplexed-mode, the multiplexed Master-Slave operates in a manner identical to that of a transaction router. First, the Master initiates a transaction.
which is forwarded over the Bus to the multiplexed Master-Slave. On receiving the transaction from the Master, the multiplexed Master-Slave routes the transaction over the Bus to the Slave. The Slave then forwards its response over the Bus to the multiplexed Master-Slave. On receiving the response from the Slave, the multiplexed Master-Slave routes the response over the Bus to the Master, thereby completing the transaction.

**Fig. 4. The Master-Bus-Slave/Master-Bus-Slave design pattern**

Modern electronic systems are characterized by increasing integration complexity, thereby necessitating novel design methodologies that allow architectural exploration at reduced integration costs. Platform Based Design (PBD) is one such design methodology wherein systems are composed of fully-parameterizable computational and communication elements, defined at an appropriate level of abstraction, thereby minimizing integration costs and aiding architectural exploration. Vincentelli in [1] defines a Platform as “a library of components that can be assembled to generate a design at that level of abstraction”. Thus, the success of PBD is highly dependent on the easy access to these large libraries of IP modules.

Several authors have considered the use of Design Patterns for efficient and extensible hardware design. Doucet and Gupta in [9] introduce the bus-protocol pattern for specifying on-chip bus structures and associated protocol behaviors, and the DLX processor pipeline pattern for modeling a DLX pipelined processor. Astrom et al. in [10] use the composite, object adaptor, abstract factory and decorator patterns to model a DSP library. Damasevicius et al. in [11] introduce the wrapper pattern that permits the behavioral adaptation of an IP module to the requirements of its operational environment, which they use to generate handshake wrappers using the single-rail-4-phase handshake protocol. MacKay in [12] discusses the application of the planar, grid-on-chip and symmetric-multiprocessing patterns in the design of multicore embedded systems.

Despite the plentiful work done in exploring the application of Design Patterns for hardware design, there still exists an unexplored yet promising domain – the proposition of a Design Pattern based methodology for the efficient design and implementation of IP modules. Although Doucet and Gupta [9] present a methodology based on the usage of Design Patterns for the translation of a model of computation into hardware, neither do they discuss its applicability to the design and implementation of IP modules, nor do they demonstrate its usage. Our work aims at addressing the need to design IP modules efficiently by exploring the usage of Design Patterns in the design cycle. Our contributions are as follows:

- We propose a methodology based on the usage of a pre-existing Design Pattern – the Master-Bus-Slave/Master-Bus-Slave pattern – whereby multiplexed Master-Slave IP modules can be efficiently modeled and implemented at the system-level.
- We demonstrate the proposed methodology by modeling and implementing an L2Cache IP module using SystemC as our choice of ESL design language.

**III. THE PROPOSED METHODOLOGY**

We now present a methodology based on the usage of Design Patterns whereby multiplexed Master-Slave IP modules can be modeled and implemented efficiently at the system-level. The methodology is based on a pre-existing Design Pattern – the Master-Bus-Slave/Master-Bus-Slave pattern – and uses this pattern to filter out a component’s structure and

**Fig. 5. Message sequencing chart for the Master-Bus-Slave/Master-Bus-Slave design pattern**

In this paper, we propose a methodology based on the usage of the Master-Bus-Slave/Master-Bus-Slave pattern, whereby multiplexed Master-Slave IP modules can be modeled efficiently at the system-level. We present the methodology through a description of the process of modeling and verifying an L2Cache IP module, using the specified Design Pattern. This paper is organized as follows: First, we provide a brief description of previously related work, highlighting our motivation and our contribution. Next, we formally present the proposed methodology. We then demonstrate the proposed methodology through the process of modeling, implementing and verifying an L2Cache at Transaction Level using SystemC. Finally, we conclude by enunciating the advantages of the proposed methodology and our learning from our experience with this methodology.
functionality from its communication interfaces. It consists of the following seven steps:

1. Capture the requirements tabulated in the module specification using a Use-case diagram.
2. Derive the structure of the module and its logic using a module block-diagram and finite state machines (FSMs).
3. Using the Master-Bus-Slave/Master-Bus-Slave pattern, filter out the module’s structure and functionality from its communication interfaces.
4. Implement the module block-diagram and FSMs using SystemC.
5. Implement the module’s communication interfaces and the system test-bench using SystemC.
6. Partition the requirements represented in the Use-case diagram into equivalence classes and draft suitable test-cases using the partition testing technique.
7. Using the system test-bench, verify the implemented module using the previously drafted test-cases in order to ensure that it is developed precisely.

In the following sections, we demonstrate the proposed methodology through the design, implementation and verification of an L2Cache IP module at Transaction Level, using SystemC as our choice of ESL design language.

IV. DESIGNING THE L2CACHE AT TRANSACTION LEVEL

An L2Cache is a small, high-speed memory located between the Processor and main system memory (i.e. RAM), that is used to improve system performance. Frequently accessed data is placed in the L2Cache, thereby allowing the Processor to access the data at a higher rate than RAM. However, when the requested data cannot be located in the L2Cache, the Processor is stalled for additional penalty clock-cycles equivalent to the number of clock-cycles to retrieve the data from RAM and store it into the L2Cache. Thus, System Architects are constantly researching the effects of L2Cache parameters (such as L2Cache size, replacement algorithms and cache mapping strategies) on system performance. Since L2Caches are located between the Processor and RAM, they are multiplexed Master-Slave devices (Slave to the Processor, Master to the RAM), making them complicated to model, as some transactions include both Master and Slave functionality. An L2Cache IP module is developed and tested as per the specifications of the Motorola MPC2605. This device is an integrated look-aside, no write-allocate L2Cache supporting 4-way set-associative cache mapping and Least Recently Used (LRU) replacement algorithm, compatible with the 60x Bus. The MPC2605 is an L2Cache, designed by Freescale Semiconductor, for use with processors that implement the PowerPC architecture.

A. Elicitation of Requirements

The L2Cache IP module has to support parameterization in order to aid architectural exploration. The requirements to be supported by the L2Cache IP module are:

1. L2Cache sizes: ranging from a minimum of 1KB to a maximum of 4GB.
3. Replacement algorithms: Least-Recently-Used (LRU), First-In-First-Out (FIFO) and Random.
4. Support only for Random replacement algorithm in-case Fully-associative strategy is chosen, since support for LRU or FIFO replacement would involve a huge overhead to implement the counters that track the LRU or FIFO blocks.
5. Ignore the choice of replacement algorithms in-case Direct-Mapping strategy is selected.
6. Specification of a replacement algorithm is mandatory in-case Set-associative strategy is selected.

These requirements are represented with the help of a Use-case diagram shown in Figure 6.

B. Deriving the L2Cache Block Diagram and Modeling the State Machines

This phase involves understanding the technical specification and translating verbose, textual information from the specification, into a suitable module block diagram and hierarchical state machines that are desirable for implementation purposes. Figure 7 shows the module block diagram of the L2Cache that we derived from the specification.

As shown in Figure 7, the L2Cache module consists of the following components:
• **Data-Ram array:** this is used to store the data that is frequently accessed by the Processor.

• **Tag-Ram array:** this is used to store the tags of the addresses, whose data is stored in the Data-Ram array.

• **L2Cache Controller and Bus Interface unit:** this unit implements the state machines of L2Cache and the interface for inter-component communication.

As mentioned earlier, the L2Cache module is designed using hierarchical state machines. Hence, some state machines contain states that are sub-state machines, making the framework highly modular. Taking advantage of the modular framework, we modified minor portions of the state machines and extended the MPC2605 architecture to include the parameterization features, namely different cache mapping strategies, replacement algorithms and L2Cache sizes. The top-level state machine of the L2Cache is shown in Figure 8.

![Fig. 8. Top-Level state machine of the L2Cache module (states with a * denote sub-state machines)](image)

C. Capturing the Module Communication Interface

In this phase, we capture the L2Cache module communication interface using information acquired from the message sequencing chart of the Master-Bus-Slave/Master-Bus-Slave pattern. The communication interface, as shown in Figure 7, consists of a Control-Bus which channels the control-word that specifies the class of a transaction (read/write), an Address-Bus which channels the memory-address of the corresponding transaction, and a Data-Bus which channels the data associated with the corresponding transaction. Since the L2Cache is a multiplexed Master-Slave device (refer Figure 5 for the sequence of messages exchanged), the channels of the communication interface are modeled as bi-directional channels (represented as double-ended arrows in Figure 7), thereby allowing the L2Cache to generate and respond to transactions. An interesting observation to note is that although the module block-diagram (Figure 7) suggests an explicit Control-Bus channel, due to the knowledge of the class of transaction currently occurring in the system, we choose to model the Control-Bus channel implicitly as shown later.

D. Implementing the Derived State Machines using SystemC

This phase involves translating the hierarchical state machines into SystemC code. First, the general structure of the L2Cache IP module is captured and translated into a SystemC module. Figure 9 shows the SystemC code that captures the general structure of the L2Cache IP module.

As shown in Figure 9, the Data-Ram and Tag-Ram arrays are captured as arrays of SystemC Bit-Vectors. Each state machine of L2Cache is captured as a C++ function/method. Hierarchical state machines are implemented by invoking the respective sub-state machine function from within the current state machine function. The Bus-interface functions are responsible for linking the L2Cache module to the communication infrastructure provided by SystemC (which will be covered in the next sub-section). Finally, the L2Cache module is initialized through the invocation of the respective initialization method from within the module constructor.

```
class L2CACHE : public simple_bus_slave_if, public l2cache_registers_if, public sc_module
{
  private:  
    enum { MOVE_COB_CONTENTS_TO_BUS = 0,
          WRITE_BUS_CONTENTS_TO_MEMORY, 
          END_OF_TRANSACTION }; 

  public:
    simple_bus_status   write ( int *data , unsigned int address );
    simple_bus_status   read  ( int *data , unsigned int address );

   // L2Cache Constructor */
    SC_HAS_PROCESS ( L2CACHE );
    L2CACHE ( sc_module_name name, ……… ) :
      sc_module ( name ) 
       { /* Initialization method calls */  } ;

  void L2CACHE :: WriteCOBContentsToMemory ( void ) 
  { /* List of States */
    enum { MOVE_COB_CONTENTS_TO_BUS = 0,
          WRITE_BUS_CONTENTS_TO_MEMORY, 
          END_OF_TRANSACTION };

    /* Setting START state */
    char state = MOVE_COB_CONTENTS_TO_BUS;

    /* State Machine Logic */
    while ( true )
    { switch ( state )
      { case MOVE_COB_CONTENTS_TO_BUS:
        ;
        ;
        ;
        break;
      } //End - switch
    } //End – while 
    //End WriteCOBContentsToMemory

    Fig. 9. SystemC code capturing L2Cache Module structure
```

Next, each state machine modeled from the previous phase is captured using a C++ method/function as shown in Figure 10. Each method begins with a list of states in the respective state machine. The starting state is set by equating a state-marker variable to the initial state. The entire state machine is then implemented using SystemC code. Shifting between states is accomplished by equating the state-marker variable to the name of a new
state defined in the list at the beginning of the method. Sub-state machines are executed by invoking a C++ method/function of an internal state machine function listed in the module structure code.

E. Implementing the Module Communication Interface and the System Test-bench

The proposed test-bench consists of a Processor (a master that generates the necessary transactions), the L2Cache and RAM, communicating over the Processor and Memory buses, as shown in Figure 11. The Simple-Bus is an open-source communication infrastructure, shipped with SystemC that provides interface methods to attach communicating components. These interface methods accept the memory-address of a transaction and the data associated with the transaction as arguments, and hence we use them to model the Address-Bus and the Data-Bus channels of the L2Cache. The Control-Bus channel is implicitly captured by the provision of discrete read/write interface methods, thereby eliminating the requirement of an explicit channel. The class of the current transaction is determined by identifying the interface method that is invoked during the transaction.

Integrating the L2Cache into the test-bench is accomplished by attaching the L2Cache to the Processor and Memory Simple-Buses, which involves the following procedure:

1. First, the Simple-Bus slave-read and slave-write interfaces of the L2Cache are implemented to invoke the L2Cache read state machine and write state machine respectively, as shown in Figure 12.

2. Next, the Bus-master port of the L2Cache is connected to the Memory Simple-Bus and the Bus-slave port of the Processor Simple-Bus is connected to the L2Cache, as shown in Figures 13 and 14. The “Top” module instantiates the system test-bench objects and links them together to form the virtual system. Since all modules are modeled at Transaction Level, every transaction between the Processor and L2Cache takes exactly one clock-cycle of the SystemC engine, with the Processor executing on the positive edge of the clock and the L2Cache and RAM (if accessed) executing on the negative edge of the clock. Since the L2Cache module is a multiplexed Master-Slave device, it must perform all execution and return the results to the Processor (the master) on the negative clock edge. Hence, it is ensured that there are no wait-states during the L2Cache execution.

F. Extending the L2Cache for Parameterization

There exist a multitude of parameterization techniques that designers can choose from, some of which include the C++ Traits mechanism. However, since the focus is on methodology, we opt for a more primitive, conditional compilation technique; a brief description follows.

Each state machine is analyzed for changes that occur due to the selection of a different parameter. The changes made to the state machines are then translated into modifications in SystemC code by encapsulating each modification within suitable conditional compilation directives. Thus, by defining the respective pre-processor symbols, the module can be parameterized to select any of the different features supported i.e. changes in L2Cache size, changes in the cache mapping strategy and changes in the replacement algorithm. Figure 9
shows an example of how conditional compilation directives are used to compile the Data-Ram and Tag-Ram arrays differently for the N-way Set-associative mapping strategy, thereby supporting parameterization.

V. EQUIVALENCE CLASSING OF MODULE REQUIREMENTS AND DERIVING THE TEST-CASES

This phase involves the equivalence classing of the module requirements represented in the Use-case diagram (Figure 6) by partitioning the module configuration parameters into disjoint subsets, each of which exhibit the same behavior, thereby allowing the selection of a single representative test-case from each class, hence alleviating module verification by reducing the effective number of test-cases. Figure 15 illustrates the segregation of the equivalence classes of the L2Cache module requirements into three domains, namely Mapping Strategies, L2Cache Sizes and Replacement Algorithms.

Exhaustive testing involves the verification of all combinations of module configuration parameters, leading to the possibility of twelve test-cases, some of which are inappropriate due to the selection of an invalid combination of parameters, namely:

- Verification of the Fully-associative strategy with LRU or FIFO replacement algorithms or with no replacement algorithm is deemed illegal due to the violation of requirement 4, eliminating three test-cases.
- Verification of the Direct-mapping strategy with LRU, FIFO or Random replacement algorithms is deemed illegal due to the violation of requirement 5, eliminating three test-cases.
- Verification of Set-associative strategy with no replacement algorithm is deemed illegal due to the violation of requirement 6, eliminating one test-case.

Having identified the inappropriate test-cases, we ignore these seven invalid combinations of configuration parameters and utilize the partition testing technique, thereby reducing our verification domain to the remaining five valid combinations shown in Table 1. The following section showcases the process of verification of each of these five valid test-cases.

<table>
<thead>
<tr>
<th>Mapping Scheme</th>
<th>L2Cache Size</th>
<th>Replacement Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>Any</td>
<td>None</td>
</tr>
<tr>
<td>Fully-Associative</td>
<td>Any</td>
<td>Random</td>
</tr>
<tr>
<td>Set-Associative</td>
<td>Any</td>
<td>LRU</td>
</tr>
<tr>
<td>Set-Associative</td>
<td>Any</td>
<td>FIFO</td>
</tr>
<tr>
<td>Set-Associative</td>
<td>Any</td>
<td>Random</td>
</tr>
</tbody>
</table>

Table 1. Valid module Configuration Parameter combinations

VI. VERIFYING THE L2CACHE IP MODULE

The process of verification involves implementing the previously derived test-cases as a series of transactions initiated by the Processor, triggering the appropriate functionality of the L2Cache. In the following sub-sections, we present each of the five derived test-cases that showcase the process of module verification.

A. Test-case #1: Verification of the Direct-Mapping Scheme

We set the L2Cache size to 1KB and select the Direct-mapping scheme. The logic of the test-case is to replace the same block in L2Cache by only changing the tag. Reading from the replaced address should result in a Read-miss since the L2Cache is Direct-mapped. In order to implement the test-case, we design the Processor to generate the following transactions in the given order:

1. 1st Transaction: Write Data 0x12345678 to Address 0x2A8. This will result in a Cache-miss, thereby writing the data to RAM.
2. 2nd Transaction: Read from Address 0x2A8. This will result in a Cache-miss, thereby fetching the data from RAM and storing it into Block 0xAA in L2Cache.
3. 3rd Transaction: Write Data 0x78243156 to Address 0x6A8. This will result in a Cache-miss, thereby replacing Block 0xAA with data 0x78243156.
4. 4th Transaction: Read from Address 0x6A8. This will result in a Cache-hit, since Address 0x6A8 corresponds to tag 0x1.
5. 5th Transaction: Read from Address 0x6A8. This will result in a Cache-hit, since Address 0x6A8 corresponds to tag 0x1.
6. 6th Transaction: Read from Address 0x2A8. This will...
result in a Cache-miss since Block 0xAA now holds data with respect to tag 0x1 whereas Address 0x2A8 results in tag 0x0.

Figure 16 shows the waveform generated from the previous set of transactions. Notice that from time 15ns onwards, the Replaced Block # has been set to 0xAA, indicating that the L2Cache has replaced the same block. Also notice that at time 55ns, L2Cache signals that there has been a hit (the 5th transaction) and at time 65ns, L2Cache signals that there has been a miss (the 6th transaction). This successfully verifies the behavior of the Direct-mapped L2Cache.

B. Test-case #2: Verification of the Set-Associative-Mapping Scheme with LRU Replacement Algorithm

We set the L2Cache size to 1MB and select the 2-way set-associative mapping scheme, along with the LRU replacement algorithm. The logic of the test-case is to bring in data into the 2-blocks of a given set and access the 1st block, hence reflecting that the LRU Block is the 2nd block. In order to implement the test-case, we design the Processor to generate the following transactions in the given order:

1. **1st Transaction**: Write Data 0x12345678 to Address 0x2AAA8. This will result in a Cache-miss, thereby writing the data to RAM.
2. **2nd Transaction**: Read from Address 0x2AAA8. This will result in a Cache-miss, thereby replacing Block #0 of Set 0xAAAA in L2Cache.
3. **3rd Transaction**: Write Data 0x23468971 to Address 0x12AAA8. This will result in a Cache-miss, thereby writing the data to RAM.
4. **4th Transaction**: Read from Address 0x12AAA8. This will result in a Cache-miss, thereby replacing Block #1 of Set 0xAAAA in L2Cache.
5. **5th Transaction**: Read from Address 0x2AAA8. This will result in a Cache-hit, hence resetting the LRU counter of Block #0 in Set 0xAAAA in L2Cache.
6. **6th Transaction**: Read from Address 0x22AAA8. This will result in a Cache-miss, thereby replacing LRU Block #1 in Set 0xAAAA in L2Cache.

Figure 17 shows the waveform generated from the previous set of transactions. Notice that from time 45ns onwards, L2Cache signals that the LRU Block is Block #1, hence replacing Block #1 during the 6th transaction. Also, notice that at time 55ns, L2Cache signals that a read-hit has occurred (5th transaction). This successfully verifies the behavior of the 2-way set-associative, LRU replacement L2Cache.

C. Test-case #3: Verification of the Set-Associative-Mapping Scheme with FIFO Replacement Algorithm

We set the L2Cache size to 1MB and select the 2-way set-associative mapping scheme, along with the FIFO replacement algorithm. The logic of the test-case is to bring in data into the 2-blocks of a given set and access the 1st block, showcasing that, unlike the LRU replacement algorithm, since the FIFO Block is the 1st block, it will always be the choice of replacement, irrespective of the access to a block. In order to implement the test-case, we design the Processor to generate the following transactions in the given order:

1. **1st Transaction**: Write Data 0x12345678 to Address 0x2AAA8. This will result in a Cache-miss, thereby writing the data to RAM.
2. **2nd Transaction**: Read from Address 0x2AAA8. This will result in a Cache-miss, thereby replacing Block #0 of Set 0xAAAA in L2Cache.
3. **3rd Transaction**: Write Data 0x23468971 to Address 0x12AAA8. This will result in a Cache-miss, thereby writing the data to RAM.

4. **4th Transaction**: Read from Address 0x12AAA8. This will result in a Cache-miss, thereby replacing Block #1 of Set 0xAAAA in L2Cache.

5. **5th Transaction**: Read from Address 0x2AAA8. This will result in a Cache-hit, which would reset the LRU counter of Block #0 in Set 0xAAAA of an L2Cache implementing the LRU replacement algorithm, but will have no effect on an L2Cache implementing the FIFO replacement algorithm.

6. **6th Transaction**: Read from Address 0x22AAA8. This will result in a Cache-miss, which would replace LRU Block #1 in Set 0xAAAA of an L2Cache implementing the LRU replacement algorithm, but will replace Block #0 in Set 0xAAAA of an L2Cache implementing the FIFO replacement algorithm.

Figure 18 shows the waveform generated from the previous set of transactions. Notice that from time 65ns onwards, L2Cache signals that the FIFO Block is Block #0, hence replacing Block #0 during the 6th transaction. In contrast, we see that in test-case #2, the LRU Block #1 is replaced during the 6th transaction. Also, notice that at time 55ns, L2Cache signals that a read-hit has occurred (5th transaction). This successfully verifies the behavior of the 2-way set-associative L2Cache.

**D. Test-case #4: Verification of Fully-Associative Mapping**

We set the L2Cache size to 4KB and configure it appropriately so that it is organized as a heap of 1024 cache-blocks. We also select the fully-associative mapping scheme, thereby allowing accessed data to be located in any of the 1024 cache-blocks. The logic of the test-case is to show that multiple memory-addresses that map onto the same block in a 2-way set-associative L2Cache do not introduce conflict misses, indicating that the L2Cache is fully-associative. In order to implement the test-case, we first initialize all 1024 blocks of the L2Cache to store valid data for 1024 unique memory-addresses that would map onto the same set in a 2-way set-associative L2Cache. We then design the Processor to generate 1024 read transactions to the previous memory-addresses in order to query the Hit-Status of the L2Cache.

Figure 19 shows the waveform generated from the previous set of transactions. Notice that each of the 1024 read transactions that query the Hit-Status of the L2Cache results in a Cache-hit, indicating the L2Cache is fully-associative. In contrast, for a 2-way set-associative L2Cache, since each of the memory-addresses maps onto the same set, only the last 2 read transactions that query the Hit-Status would result in a Cache-hit. This successfully verifies the behavior of the fully-associative L2Cache.

**E. Test-case #5: Verification of Set-Associative Mapping with Random Algorithm**

We set the L2Cache size to 4KB and select the 16-way set-associative mapping scheme, along with the Random replacement algorithm. The logic of the test-case is to show that for every memory-address that conflicts with a valid block in the respective set of the L2Cache, a random block within the set will be chosen for replacement. In order to implement the test-case, we first initialize the 16 blocks of set 0x2A to store valid data for 16 unique memory-addresses that map onto the same set of the L2Cache. We then design the Processor to generate 16 read transactions to unique memory-addresses that map onto set 0x2A, resulting in a Cache-miss, hence replacing a random block each time within set 0x2A.

Figure 20 shows the waveform generated from the previous set of transactions. Notice that each of the 16 conflicting read
transactions results in the replacement of a random block within set 0x2A. This successfully verifies the behavior of the 16-way set-associative, random replacement L2Cache.

VII. CONCLUSION

The application of Design Patterns in the design of electronic systems promotes architectural re-usability via the isolation of computational logic from communication infrastructure, thereby allowing designers and systems architects to re-use communication affiliated architectural infrastructure, enabling them to focus exclusively on modeling the required computational elements that comprise the system. In addition, the usage of Object-Oriented design and testing methodologies and engineering tools such as UML use-cases and state-machines empowers designers and System Architects with an efficient and effective engineering alternative, resulting in the production of high-quality electronic systems.

In this paper, we present a methodology whereby electronic systems can be efficiently designed and implemented using Design Patterns, which we then demonstrate through the design and implementation of a multiplexed Master-Slave IP module at the system-level. The increasing utilization of Virtual Prototyping in the manufacture of electronic systems coerces the design of parameterizable IP modules and the creation of IP warehouses that provide designers and System Architects with a variety of components designed at various levels of abstraction that can be readily integrated. We believe that the generation of IP modules would be accelerated through the adoption of a systematic and efficient methodology such as the one proposed in this paper. The efficiency of the proposed methodology is attributed to the usage of readily available, pre-defined and pre-verified Design Patterns that permit architectural re-usability. The systematic nature of the proposed methodology is attributed to the seamless flow offered between successive phases in the methodology. The usage of such a methodology provides designers and system architects with effective tools that would allow them to combat the complexities and time-to-market requirements of systems of the future.

REFERENCES


Sushil Menon received the Bachelors of Engineering degree in Computer Science & Engineering from PES School of Engineering, Bangalore, India, in 2010. He is currently pursuing a Master of Science degree in Embedded Systems at University of Pennsylvania. He was invited to the North American SystemC User Group (NASCUG) XIV meeting, Phoenix, Arizona, USA, for a seminar presentation on this topic. His research interests include System Design, Electronic System Level Design, Hardware-Software Co-Design, System-on-Chip Design, Electronic Design Automation and Computer Architecture.

Dr. Suryaprasad Jayadevappa has all his degrees in Computers, a Bachelors of Engineering from Bangalore University, a Masters from Mysore University and a Ph.D. from Florida Atlantic University (FAU), Boca Raton, Florida. He is currently the Director and Principal of PES School of Engineering (PESSE), since its inception in 2005. Prior to joining PESSE, he served at Motorola, Sunrise Plantation Fl. Lauderdale, USA for 2 years as Technical Lead in iDEN – Advance Technology Group. He was instrumental in implementing a new methodology based on his Ph.D. dissertation for reducing the design cycle time in the manufacture of mobile phones. He has also worked at Cadence Design Systems, San Jose, California as Senior Technical Member during 2001-2003. He has over 18 years of teaching experience, including at Florida Atlantic University during 1998-2001. His research interests include System Level Design methodologies, Hardware-Software Co-Design, Embedded System Design, Power Optimal Design, Verification methodologies, Advanced Programming methodologies and Promoting research at the undergraduate level.

Dr. Suryaprasad along with Dr. Ravi Shankar has developed a new teaching curriculum – “CAD Based Computer Design using state of art System Level Design Language – SystemC”. A paper based on the above was presented at the American Society for Engineering Education (ASEE) and was nominated for the best paper award. He has also published and presented many papers in leading International Conferences abroad, and has also delivered invited lectures, including at IEEE chapters. He also has a patent filed in USA and India on ‘Vehicle Management System’. Ever since his return to India in 2005, he has been rigorously involved in building a quality institution of international repute. He is a member of the Institute of Electrical and Electronics Engineers (IEEE), Association for Computing Machinery (ACM), International Society for Technology in Education (ISTE) and Computer Society of India (CSI).