A Pattern based Methodology for the Design and Implementation of Multiplexed Master-Slave devices at the System-Level

Sushil Menon¹ & Dr. Suryaprasad J
Center for Electronic System Level Design & Verification
PES School of Engineering
Bangalore – India

¹Email: sushil.menon.1988@gmail.com
¹Cell: 1-215-407-1749
Outline

- IP Modules and their role in Virtual Prototyping
- Current technology/methodology
- Proposition of a “Design Pattern” based methodology
  - The “Master-Bus-Slave/Master-Bus-Slave” Design Pattern
- Modeling an L2Cache IP module using proposed methodology
- Conclusion
Virtual Prototyping and IP Modules

Advantages of IP-based Virtual Prototyping:
- "Plug-and-Play" IPs allow easy Virtual Prototyping
- "Parameterized IPs" allow architectural exploration

Figure: Virtio VPXS Virtual Platform - Intel XScale core
Current Technology/Methodology

- Industry focus more on RTL models, less on TLM models
  - RTL more suited for physical synthesis as compared to TLM
- However, RTL yields lower simulation speed and makes architectural exploration a nightmare!
  - Too many details! Too many signals/pins, cycle accurate designs.
  - Many component manufacturers, increased inter-component interface obscurity, hence increased integration difficulty!
- Thus, TLM more suitable for Virtual Prototyping.
  - Increased Level of Abstraction, simplified inter-component interfaces, reduced details, thus resulting in faster simulation!
- ESL and SystemC are starting to become prevalent in the Electronic System Design industry.
Modeling IPs using “Design Patterns”

- Design Patterns – are standard architectures, targeted at specific problem scenarios.
  - Advantage: re-usability of pre-defined and tested architectural solutions
  - Example: “Master-Bus-Slave” Design Pattern
    - Usually: Master – Processor, Bus – System Bus, Slave – Peripherals

System Block Diagram

Message Sequencing Chart
The “Master-Bus-Slave/Master-Bus-Slave” Design Pattern

- Usually adopted by “Multiplexed Master-Slave” components
  - Example: L2Caches, DMA Controllers etc.

System Block Diagram

Message Sequencing Chart
Modeling an L2Cache IP Module at Transaction Level

- L2Cache – small, high-speed memory between Processor and Memory, used to improve system performance

- Goal: Showcase proposed Methodology through the design of a L2Cache IP Module using the “Master-Bus-Slave/Master-Bus-Slave” Design Pattern
  - L2Cache IP Module is designed at Transaction Level

- Phases in proposed Methodology:
  - Phase 1: Elicitation of Requirements
  - Phase 2: Deriving Model Block Diagram and State Machines
  - Phase 3: Implementing the L2Cache Structure & Derived State Machines using SystemC
  - Phase 4: Implementing the System Test-bench
  - Phase 5: Extending the L2Cache for Parameterization
  - Phase 6: Verification of the L2Cache IP Module
Phase 1 – Elicitation of Requirements

- Data was collected from the technical specification of the “Motorola MPC2605 L2Cache”
- Requirements of the module are captured and illustrated in the following Use-Case diagram
Phase 2 – Deriving Model Block Diagram and State Machines

- Convert textual information from technical spec into a suitable Model Block Diagram and Hierarchical State Machines
- Hierarchical State Machines - State Machines that contain states that are sub-state machines (denoted by * in state)
  - Provides a highly modular framework
Phase 3.1 – Implementing the L2Cache Structure using SystemC

Block Diagram of L2Cache

L2Cache Structure – SystemC

class L2CACHE : public simple_bus_slave_if, public l2cache_registers_if, public sc_module
{
    private: /* Data-Ram and Tag-Ram arrays */
        #ifdef _SET_ASSOCIATIVE_
            sc_bv < DATA_RAM_SIZE > DataRam[SETS][BLOCKS];
            sc_bv < TAG_RAM_SIZE > TagRam[SETS][BLOCKS];
        #endif

    public: /* L2Cache Master Bus port */
        sc_port < simple_bus_direct_if > master_bus_port;

    public: /* L2Cache Constructor */
        .................
};
Phase 3.2 – Implementing the Derived State Machines using SystemC

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:
                state = WRITE_BUS_CONTENTS_TO_MEMORY;
                break;
            case WRITE_BUS_CONTENTS_TO_MEMORY:
                WriteBusContentsToMemory( );
                break;
            }
        //End – while
    }
    //End WriteCOBContentsToMemory

Sample State Machine in L2Cache

State Machine Structure – SystemC
Phase 4 – Implementing the System Test-Bench

simple_bus_status L2CACHE :: read ( int *data, unsigned int address )
{
    /* Invoking ProcesorReadStateMachine() */
    ProcessorReadStateMachine();
    return TransactionStatus;
}

simple_bus_status L2CACHE :: write ( int *data, unsigned int address )
{
    /* Invoking ProcessorWriteStateMachine() */
    ProcessorWriteStateMachine();
    return TransactionStatus;
}

L2Cache Slave Interface – SystemC

class TOP : public sc_module
{
public:
    /* Pointers to the various objects in the testbench */
    L2CACHE *L2Cache;
    simple_bus *Bus; ........................................
    /* Constructor */
    SC_HAS_PROCESS ( TOP );
    TOP ( sc_module_name name ) : sc_module ( name ),
    { ................................/* Binding L2Cache to Bus */
        L2Cache -> master_bus_port (*Bus);
        Bus -> slave_port (*L2Cache);
    }
};
Phase 5 – Extending the L2Cache for Parameterization

- Existing State Machines are suitably re-modeled according to changes occurring due to selection of a different parameter
- Changes in State Machines are then translated into SystemC code by encapsulating them around conditional compilation directives

```systemc
class L2CACHE : public simple_bus_slave_if,
               public l2cache_registers_if, public sc_module
{
  private: /* Data-Ram and Tag-Ram arrays */
    #ifdef _SET_ASSOCIATIVE_
    sc_bv < DATA_RAM_SIZE >
    DataRam[SETS][BLOCKS];
    sc_bv < TAG_RAM_SIZE >
    TagRam[SETS][BLOCKS];
    #endif

  public:
    /* L2Cache Master Bus port */
    sc_port < simple_bus_direct_if > master_bus_port;
    /* Simple Bus interface functions */
    simple_bus_status read ( int *data , unsigned int address );
    simple_bus_status write ( int *data , unsigned int address );
    /* L2Cache Constructor */
};
```

Compiling Tag-RAM and Data-RAM arrays differently for Fully-Associative using Conditional Compilation Directives

L2Cache Structure – SystemC
Phase 6 – Verification of the L2Cache IP Module

- Implement a series of transactions, initiated by Processor, that trigger the appropriate functionality in L2Cache

Transactions generated by Processor:

1. /* Cache miss, hence write Data to RAM */
   Write (Addr=0x2A8, Data=0x12345678);

2. /* Cache miss, hence Data brought into block 0xAA of Cache */
   Read (Addr=0x2A8);

3. /* Cache miss, hence write Data to RAM */
   Write (Addr=0x6A8, Data=0x78243156);

4. /* Cache miss, hence replace block 0xAA in Cache with 0x78243156 */
   Read (Addr=0x6A8);

5. /* Cache hit for read from Addr=0x6A8 */
   Read (Addr=0x6A8);

6. /* Cache miss for read from Addr=0x2A8 */
   Read (Addr=0x2A8);

Verification of “Direct-Mapping” Mode
Conclusion

- Success of Virtual Prototyping depends on large warehouses of Pre-Designed, Developed and Parameterizable IP Modules
- Generation of IP Modules is accelerated through the usage of a "Systematic" and "Efficient" Methodology
- Proposed Methodology is:
  - Efficient – Usage of readily available “Design Patterns” that are pre-defined and tested
  - Systematic – Seamless flow between successive phases in the Methodology
Thank You! 😊

Questions/Concerns?