Parametrization of SRAM sub-system

Major Project

Submitted in Partial Fulfilment of the Requirements
for the Degree of
Master of Technology(M.Tech.)
in
VLSI Design
by

Rahul Shah
11MECV20

Department of Electronics & Communication Engineering
Institute of Technology
Nirma University
Ahmedabad
December-2012

Parametrization of SRAM sub-system

Major Project
Submitted in Partial Fulfilment of the Requirements
for the Degree of
Master of Technology(M.Tech.)
in
VLSI Design
by

Rahul Shah
11MECV20

Dr. N. M. Devashrayee
Internal Guide

Mrs. Aparna Dixit
Mr. Narasimha Devineni
External Guide

Department of Electronics & Communication Engineering
Institute of Technology
Nirma University
Ahmedabad
December-2012

Declaration

This is to certify that

1. I, Rahul Shah, a student of semester III Master of Technology in VLSI Design, Nirma University, Ahmedabad hereby declare that the project work "Parametrization of SRAM sub-system"has been independently carried out by me under the guidance of Mrs. Aparna Dixit and Mr. Narasimha Devineni, Intel Technology India Private Limited, Bangalore and Dr. N. M. Devashrayee, Program Co-ordinator, Department of VLSI Design, Nirma University, Ahmedabad. This Project has been submitted in the partial fulfillment of the requirements for the award of degree Master of Technology(M.Tech.) in VLSI Design, Nirma University, Ahmedabad during the year 2012 - 2013.

2. I have not submitted this work in full or part to any other University or Institution for the award of any other degree.

Rahul Shah
11MECV20

Certificate

This is to certify that the Major Project entitled "Parametrization of SRAM sub-system "submitted by Rahul Shah (11MECV20), towards the partial fulfillment of the requirements for the degree of Master of Technology in VLSI Design of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project part-I, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Dr. N. M. Devashrayee
Internal Project Guide
Institute of Technology,
Nirma University, Ahmedabad

Mrs. Aparna Dixit
Mr. Narasimha Devineni
External Project Guide
Intel Technology India Pvt. Ltd.,
Bangalore

Dr. N. M. Devashrayee
Program Cordinator
Institute of Technology,
Nirma University, Ahmedabad

Dr. P. N. Tekwani
Head of EE Dept
Institute of Technology,
Nirma University, Ahmedabad

Date:

Place:Ahmedabad

Acknowledgement

First and foremost, sincere thanks to Mr. Santhosh Kumar Amanna, Manager, Intel Technology India Private Limited, Bangalore for assigning me such project and guide me through.

I would like to thank my Mentors, Mrs. Aparna Dixit and Mr. Narasimha Devineni, Intel Technology India Private Limited, Bangalore for their valuable guidance. Throughout the training, they have given me much valuable advice on project work which I am very lucky to benefit from.

I would like to thank teammates, Venkatraman Girish, Satkuri Rajkumar, Kamatham Sudheer, Intel Technology India Private Limited Bangalore, for giving valuable help in ISH environment ramp-up and solving my all queries.

I would also like to thank Dr. K.R.Kotecha, Director, Institute of Technology, Nirma University, Ahmedabad for providing me an opportunity to get an internship at Intel Technology India Private Limited, Bangalore.

I would also Thank to my Project Co-ordinator, Dr. N. M. Devashrayee, Professor, VLSI Design, Institute of Technology, Nirma University, Ahmedabad for giving valuable support for project work.

I would like to thank my all faculty members for providing encouragement, ex-changing knowledge during my post-graduate program.

I also owe my colleagues in the Intel, special thanks for helping me on this path and for making project at Intel more enjoyable.

Rahul Shah
11MECV20

Abstract

Creating reusable models typically requires that general-purpose models be written with re-definable parameters such as SIZE, WIDTH and DEPTH. With respect to coding parametrized Verilog models, two Verilog constructs that are over-used and abused are the global macro definition (‘define) and the infinitely abusable parameter redefinition statement (defparam). This report will detail techniques for coding proper parametrized model for SRAM Controller, detail the differences between parameters and macro definitions, present guidelines for using macros, parameters, parameter definitions and also some added features in existing design with RTL quality check like Lintra and LEC on whole design for making it easy to synthesize. And also verifying the updated design by creating new test case which will mostly concentrating on verification of updated features in the design

Contents

List of Figures

Chapter 1

Introduction

1.1 What is SoC?

A system on a chip or system on chip (SoC or SOC) is an Integrated circuit(IC) that integrates all components of a computer or other electronic system into a single chip. It may contain digital, analog or mixed - signal and often radio-frequency functions-all on a single chip substrate.
The VLSI manufacturing technology advances has made possible to put millions of transistors on a single die. This enables designers to put systems on a single chip thus moving everything from board to single chip resulting to growth of system on chip (SOC) technology. SOC design includes efforts to integrate heterogeneous or different types of silicon IPs (intellectual properties) on to the same chip, like memory, Micro Processor, random logics, and analog circuitry. SOC often incorporates analog components, and can also include opto/micro-electronic mechanical system components in the future.
An SOC design provide significant advantages in terms of speed , area ,reliability ,security and power however suffers from high system complexity , fabrication costs and increase verification requirements. However increasing reusability of IP and highly sophisticated tools (hardware/software) are making the SOC designs an extremely favorable and exciting option for modern applications.
SOC will design in two phases first SIP (Soft Intellectual Property) and then HIP (Hard Intellectual Property) we are working on SIP design. So the tradition flow for SIP design is as follows.

1. Specification and RTL coding
2. Verification
3. Synthesis
4. Formal Equivalence Verification
5. Engineering change order

1.1.1 Specification and RTL coding

Chip design commences with the conception of an idea dictated by the market. These ideas are then translated into architectural and electrical specifications. The architectural specifications define the functionality and partitioning of the chip into several manageable blocks, while the electrical specifications define the relationship between the blocks in terms of timing information. The next phase involves the implementation of these speci-

fications. In the past this was achieved by manually drawing the schematics, utilizing the components found in a cell library. This process was time consuming and was impractical for design reuse. To overcome this problem, hardware description languages (HDL) were developed. As the name suggests, the functionality of the design is coded using the HDL. There are two main HDLs in use today, Verilog and VHDL. Both languages perform the same function, each having their own advantages and disadvantages. There are three levels of abstraction that may be used to represent the design; Behavioral, RTL (Register Transfer Level) and Structural. The Behavioral level code is at a higher level of abstraction. It is used primarily for translating the architectural specification, to a code that can be simulated. Behavioral coding is initially performed to explore the authenticity and feasibility of the chosen implementation for the design. Conversely, the RTL coding actually describes and infers the structural components and their connections. This type of coding is used to describe the functionality of the design and is synthesizable to form a structural netlist. This netlist comprises of the components from a target library and their respective connections; very similar to the schematic based approach. The design is coded using the RTL style, in either Verilog or VHDL, or both. It can also be partitioned if necessary, into a number of smaller blocks to form a hierarchy, with a top-level block connecting all lower level blocks.
In the register transfer level model we split the complete system state up into registers and consider the flow of information 'in bulk' from one register to the next on each clock tick.

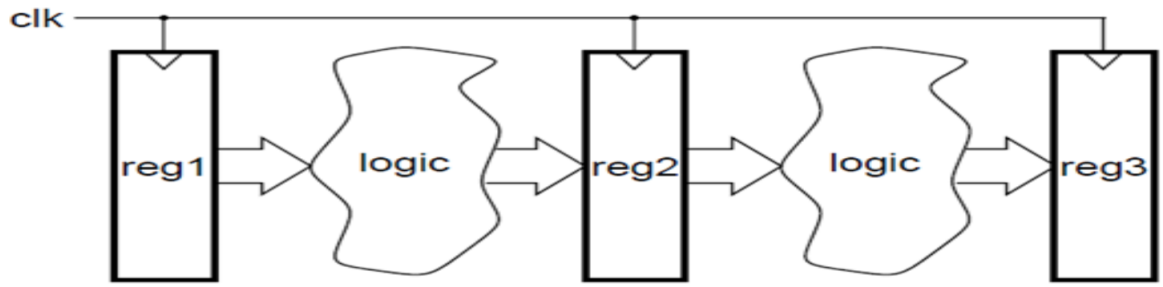

Figure 1.1: RTL Model

1.1.2 Verification

The next step is to check the functionality of the design by simulating the RTL code. All currently available simulators are capable of simulating the behavior level as well as RTL level coding styles. In addition, they are also used to simulate the mapped gate-level design. The test bench is normally written in behavior HDL while the actual design is coded in RTL. Usually the simulators are language dependent (either Verilog or VHDL), although there are a few simulators in the market, capable of simulating a mixed HDL design. The purpose of the test bench is to provide necessary stimuli to the design. It is important to note that the coverage of the design is totally dependent on the number of tests performed and the quality of the test bench. This is the reason why a sound test bench is extremely critical to the design. During the simulation of the RTL, the

component (or gate) timing is not considered. Therefore, to minimize the difference between the RTL simulation and the synthesized gate-level simulation at a later stage, the delays are usually coded within the RTL source, usually for sequential elements.

1.1.3 Synthesis

For a long time, the HDLs were used for logic verification. Designers would manually translate the HDL into schematics and draw the interconnections between the components to produce a gate-level netlist. With the advent of synthesis tools, this manual task has been rendered obsolete. The tool has taken over and performs the task of reducing the RTL to the gate-level netlist. This process is termed as synthesis. Synopsys's Design Compiler (DC) is the de-facto standard and by far the most popular synthesis tool in the SOC industry today. Synthesizing a design is an iterative process and begins with defining timing constraints for each block of the design. These timing constraints define the relationship of each signal with respect to the clock input for a particular block. In addition to the constraints, a file defining the synthesis environment is also needed. The environment file specifies the technology cell libraries and other relevant information that DC uses during synthesis. DC reads the RTL code of the design and using the timing constraints, synthesizes the code to structural level, thereby producing a mapped gate level netlist.

1.1.4 Formal Equivalence Verification (FEV)

The purpose of the formal equivalence verification in the design flow is to validate the RTL against RTL, gate-level netlist against the RTL code, or the comparison between gate-level to gate-level netlists. The RTL to RTL verification is used to validate the new RTL against the old functionally correct RTL. This is usually performed for designs that are subject to frequent changes in order to accommodate additional features. When these features are added to the source RTL, there is always a risk of breaking the old functionally correct feature. To prevent this, formal verification may be performed between the old RTL and the new RTL to check the validity of the old functionality.

1.1.5 Engineering Change Order (ECO)

Many designers regard engineering change order (ECO) as the change required in the netlist at the very last stage of the SOC design flow. For instance, ECO is performed when there is a hardware bug encountered in the design at the very last stage (say, after tape-out), and it is necessary to perform a metal mask change by re-routing a small portion of the design. As a result ECO is performed on a small portion of the chip to prevent disturbing the placement and routing of the rest of the chip, thereby preserving the rest of the chips timing. Only the part that is affected is modified. This can be achieved, either by targeting the spare gates incorporated in the chip, or by routing only some of the metal layers. This process is termed as metal mask change. Normally, this procedure is executed for changes that require less than 10% modification of the whole chip (or a block, if doing hierarchical place and route). If the bug fix requires more than 10% change then it is best to repeat the whole procedure and re-route the chip (or the block). The latest version of DC incorporates the ECO compiler. It makes use of the mathematical algorithms (also used by the formal verification techniques), to automatically implement the required changes. Making use of the ECO compiler provides designers an alternative to the tedium of manually inserting the required changes in the netlist, thus minimizing the turn-around time of the chip. Some layout tools have incorporated the ECO algorithm

within their tool. The layout tool has a built-in advantage that it does not suffer from the limitation of crossing the hierarchical boundaries associated with a design. Also, the layout tool benefits from knowing the placement location of the spare cells (normally included by the designers in the design), thus can target the nearest location of spare cells in order to implement the required ECO changes and achieve minimized routing.

1.2 Conclusion

In this chapter SIP flow incorporating the latest tools and technology for very deep sub-micron (VDSM) technologies were reviewed. The flow started with the definition of specification, and ended with Formal Equivalence Verification.

Chapter 2

TOP LEVEL DIAGRAM

2.1 Block Diagram

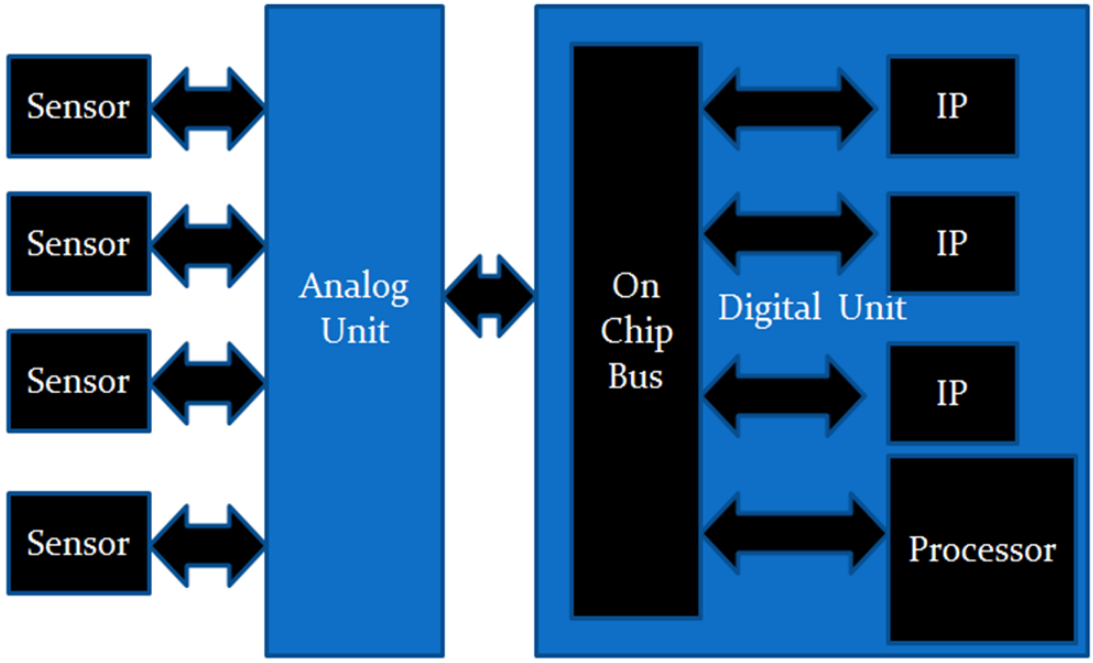

Figure 2.1: Block Diagram of ISH

2.2 Description

The above schematic shows the generalized block diagram for the project. It consists of 2 basic units

- Analog Unit
- Digital Unit

Sensors are attached to the both analog as well as digital side of the design and these sensor data is sampled and given to the different IP's to processing. The IP's talk to each other through an on chip bus which can be easily seen in the rtl code. Transactions take place using various protocols and interfaces present in the chip level design. The SoC

architecture contains various devices like DMA, Memory, microcontroller and different types of IPs with the connectivity bridges. Mainly these bridges are just connectivity between the various IPs for compatibility purpose. One IPs information may be not suit to communicate with the another IPs, so bridge can acts as mediator to accommodate the service.

Chapter 3

SRAM SUB-SYSTEM

3.1 Block Diagram

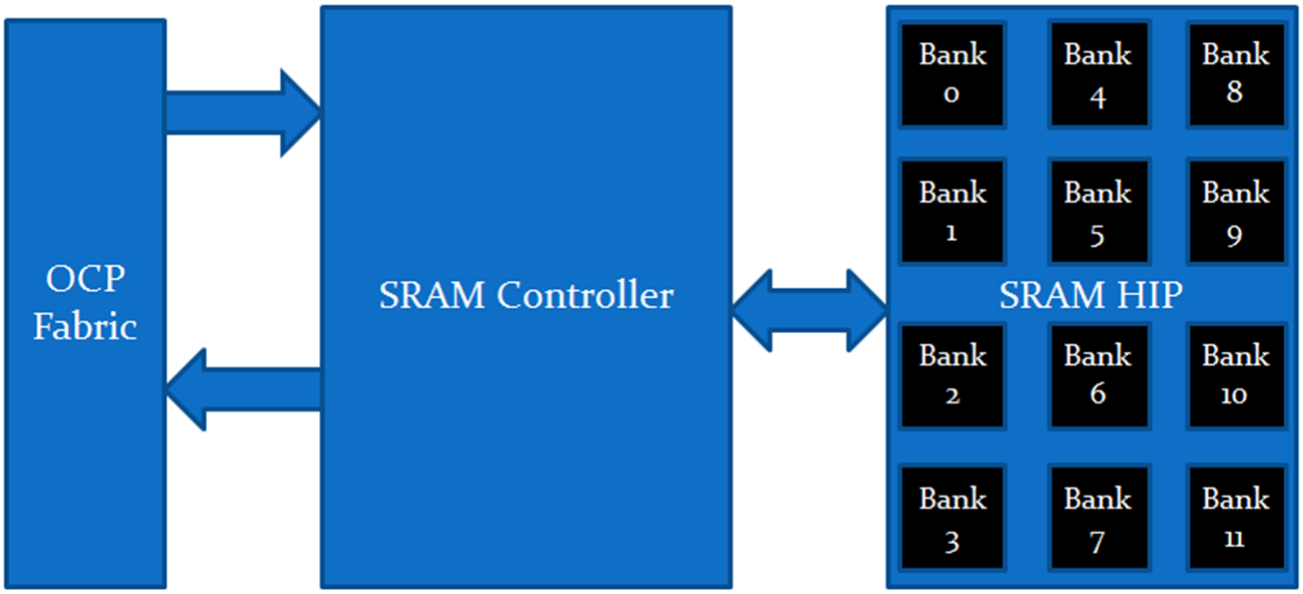

Figure 3.1: Block Diagram of SRAM sub-system

3.2 Open Core Protocol (OCP)

The Open Core Protocol (OCP) is an openly licensed, core-centric protocol intended to meet contemporary system level integration challenges. OCP defines a bus-independent, configurable and scalable interface for on-chip subsystem communications. OCP further extends capabilities in high performance multithreading, synchronization primitives and single-request/multiple-data transactions. OCP data transfer models range from simple request-grant handshaking through pipelined request-response to complex out-of-order operations.

Legacy IP cores can be adapted to OCP, while new implementations may take full advantage of advanced features: designers select only those features and signals encompassing a cores specific data, control and test configuration. Core definition using OCP encapsulates a complete system integration description enabling core and test bench reuse

without rework. Not only does OCP provide clear delineation of design responsibilities for core authors and System-on-Chip (SoC) integrators, but also institutes a key partitioning formalism for verification engineers and automation software.

3.2.1 Highlights

The OCP promotes IP core reusability and reduces design time, design risk and manufacturing costs for SoC designs. It focuses exclusively on IP core interfacing without preempting interconnect topology or other application-specific integration choices.

- Enables IP core creation to be independent of system architecture and application domain.
- Describes all inter-core communications.
- Optimizes die area by configuring into the OCP interface only those features needed by the core.
- Specified timing categories assure core interoperability.
- Facilitates rapid, plug-and-play IP integration.

3.2.2 Theory of operation

The Open Core Protocol interface addresses communications between the functional units (or IP cores) that comprise a system on a chip. The OCP provides independence from bus protocols without having to sacrifice high performance access to on-chip interconnects. By designing to the interface boundary defined by the OCP, you can develop reusable IP cores without regard for the ultimate target system. Given the wide range of IP core functionality, performance and interface requirements, a fixed definition interface protocol cannot address the full spectrum of requirements. The need to support verification and test requirements adds an even higher level of complexity to the interface. To address this spectrum of interface definitions, the OCP defines a highly configurable interface. The OCPs structured methodology includes all of the signals required to describe an IP cores communications including data flow, control, and verification and test signals. This chapter provides an overview of the concepts behind the Open Core Protocol, introduces the terminology used to describe the interface and offers a high-level view of the protocol.

Point-to-Point Synchronous Interface

To simplify timing analysis, physical design, and general comprehension, the OCP is composed of uni-directional signals driven with respect to, and sampled by the rising edge of the OCP clock. The OCP is fully synchronous and contains no multi-cycle timing paths. All signals other than the clock are strictly point-to-point.

Bus Independence

A core utilizing the OCP can be interfaced to any bus. A test of any bus-independent interface is to connect a master to a slave without an intervening on chip bus. This test not only drives the specification towards a fully symmetric interface but helps to clarify other issues. For instance, device selection techniques vary greatly among on-chip buses. Some use address decoders. Others generate independent device select signals (analogous

to a board level chip select). This complexity should be hidden from IP cores, especially since in the directly-connected case there is no decode/selection logic. OCP-compliant slaves receive device selection information integrated into the basic command field.
Arbitration schemes vary widely. Since there is virtually no arbitration in the directly-connected case, arbitration for any shared resource is the sole responsibility of the logic on the bus side of the OCP. This permits OCP compliant masters to pass a command field across the OCP that the bus interface logic converts into an arbitration request sequence.

Commands

There are two basic commands, Read and Write and five command extensions. The WriteNonPost and Broadcast commands have semantics that are similar to the Write command. A WriteNonPost explicitly instructs the slave not to post a write. For the Broadcast command, the master indicates that it is attempting to write to several or all remote target devices that are connected on the other side of the slave. As such, Broadcast is typically useful only for slaves that are in turn a master on another communication medium (such as an attached bus).
The other command extensions, ReadExclusive, ReadLinked and WriteConditional, are used for synchronization between system initiators. ReadExclusive is paired with Write or WriteNonPost, and has blocking semantics. ReadLinked, used in conjunction with WriteConditional has non-blocking (lazy) semantics. These synchronization primitives correspond to those available natively in the instruction sets of different processors.

Address/Data

Wide widths, characteristic of shared on-chip address and data buses, make tuning the OCP address and data widths essential for area-efficient implementation. Only those address bits that are significant to the IP core should cross the OCP to the slave. The OCP address space is flat and composed of 8- bit bytes (octets). To increase transfer efficiencies, many IP cores have data field widths significantly greater than an octet. The OCP supports a configurable data width to allow multiple bytes to be transferred simultaneously. The OCP refers to the chosen data field width as the word size of the OCP. The term word is used in the traditional computer system context; that is, a word is the natural transfer unit of the block. OCP supports word sizes of power-of-two and non power- of-two as would be needed for a 12-bit DSP core. The OCP address is a byte address that is word aligned. Transfers of less than a full word of data are supported by providing byte enable information that specifies which octets are to be transferred. Byte enables are linked to specific data bits (byte lanes). Byte lanes are not associated with particular byte addresses. This makes the OCP endian-neutral, able to support both big and little-endian cores.

Pipelining

The OCP allows pipelining of transfers. To support this feature, the return of read data and the provision of write data may be delayed after the presentation of the associated request.

Response

The OCP separates requests from responses. A slave can accept a command request from a master on one cycle and respond in a later cycle. The division of request from response permits pipelining. The OCP provides the option of having responses for Write commands, or completing them immediately without an explicit response.

Burst

To provide high transfer efficiency, burst support is essential for many IP cores. The extended OCP supports annotation of transfers with burst information. Bursts can either include addressing information for each successive command (which simplifies the requirements for address sequencing/burst count processing in the slave), or include addressing information only once for the entire burst.

In-band Information

Cores can pass core-specific information in-band in company with the other information being exchanged. In-band extensions exist for requests and responses, as well as read and write data. A typical use of in-band extensions is to pass cacheable information or data parity.

Tags

Tags are available in the OCP interface to control the ordering of responses. Without tags, a slave must return responses in the order that the requests were issued by the master. Similarly, writes must be committed in order. With the addition of tags, responses can be returned out-of-order, and write data can be committed out-of-order with respect to requests, as long as the transactions target different addresses. The tag links the response back to the original request. Tagging is useful when a master core such as a processor can handle out of order return, because it allows a slave core such as a DRAM controller to service requests in the order that is most convenient, rather than the order in which requests were sent by the master. Out-of-order request and response delivery can also be enabled using multiple threads. The major differences between threads and tags are that threads can have independent flow control for each thread and have no ordering rules for transactions on different threads. Tags, on the other hand, exist within a single thread and are restricted to shared flow control. Tagged transactions cannot be re-ordered with respect to overlapping addresses. Implementing independent flow control requires independent buffering for each thread, leading to more complex implementations. Tags enable lower overhead implementations for out-of-order return of responses at the expense of some concurrency.

Threads and Connection

To support concurrency and out-of-order processing of transfers, the extended OCP supports the notion of multiple threads. Transactions within different threads have no ordering requirements, and independent flow control from one another. Within a single thread of data flow, all OCP transfers must remain ordered unless tags are in use. Transfers within a single thread must remain ordered unless tags are in use. The concepts of threads and tags are hierarchical: each thread has its own flow control, and ordering

within a thread either follows the request order strictly, or is governed by tags. While the notion of a thread is a local concept between a master and a slave communicating over an OCP, it is possible to globally pass thread information from initiator to target using connection identifiers. Connection information helps to identify the initiator and determine priorities or access permissions at the target.

Interrupts, Errors, and other Sideband Signaling

While moving data between devices is a central requirement of on-chip communication systems, other types of communications are also important. Different types of control signaling are required to coordinate data transfers (for instance, high-level flow control) or signal system events (such as interrupts). Dedicated point-to-point data communication is sometimes required. Many devices also require the ability to notify the system of errors that may be unrelated to address/data transfers.
The OCP refers to all such communication as sideband (or out-of-band) signaling, since it is not directly related to the protocol state machines of the dataflow portion of the OCP. The OCP provides support for such signals through sideband signaling extensions. Errors are reported across the OCP using two mechanisms. The error response code in the response field describes errors resulting from OCP transfers that provide responses. Write-type commands without responses cannot use the in-band reporting mechanism. The second method for reporting errors across the OCP uses out-of band error fields. These signals report more generic sideband errors, including those associated with posted write commands.

3.2.3 Basic signals of OCP

Name	Width	Function
MCmd	3	Transfer command
MAddr	32	Transfer address
MBurstLength	5	Burst length
MData	32	Write data
MDataByteEn	4	Data handshake phase write byte enables
MDataValid	1	Write data valid
MDataLast	1	Last write data in burst
MBurstSingleReq	1	Burst uses single request/ multiple data protocol
MRespAccept	1	Master accepts response
SResp	2	Transfer response
SData	32	Read data
SRespLast	1	Last response in burst
SCmdAccept	1	Slave accepts transfer
SDataAccept	1	Slave accepts write data

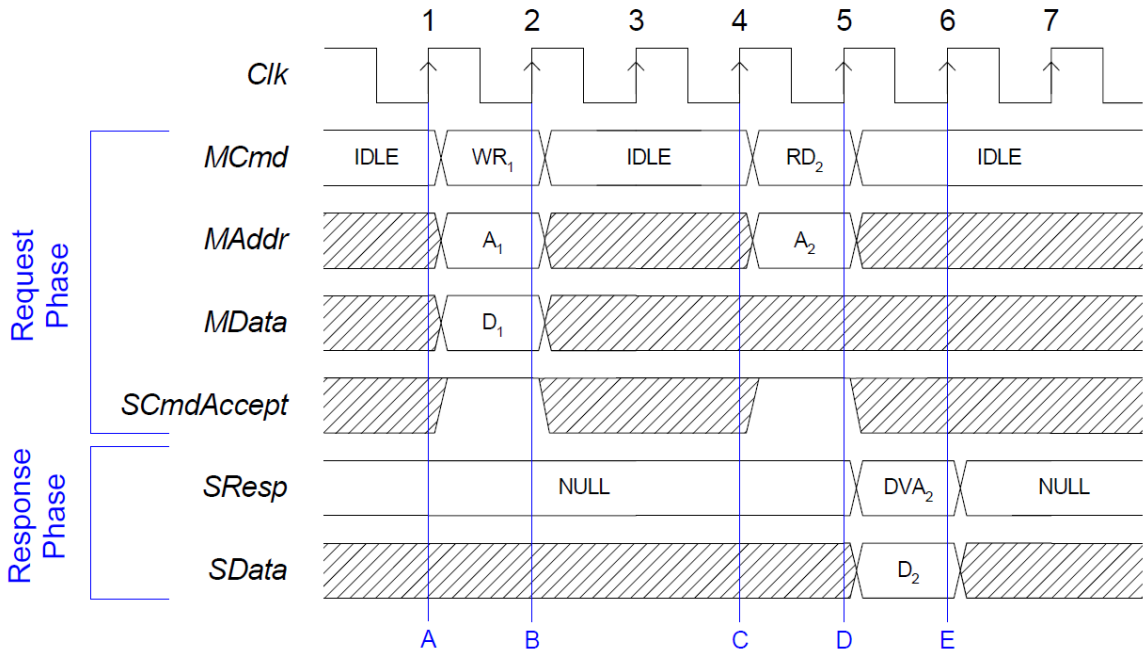

Figure 3.2: Waveform for simple Read and Write

3.2.4 Simple Read and Write Transfer

Sequence

[A]. The master starts a request phase on clock 1 by switching the MCmd field from IDLE to WR. At the same time, it presents a valid address (A1) on MAddr and valid data (D1) on MData. The slave asserts SCmdAccept in the same cycle, making this a 0-latency transfer.
[B]. The slave captures the values from MAddr and MData and uses them internally to perform the write. Since SCmdAccept is asserted, the request phase ends.
[C]. The master starts a read request by driving RD on MCmd. At the same time, it presents a valid address on MAddr. The slave asserts SCmdAccept in the same cycle for a request accept latency of 0.
[D]. The slave captures the value from MAddr and uses it internally to determine what data to present. The slave starts the response phase by switching SResp from NULL to DVA. The slave also drives the selected data on SData. Since SCmdAccept is asserted, the request phase ends.
[E]. The master recognizes that SResp indicates data valid and captures the read data from SData, completing the response phase. This transfer has a request-to-response latency of 1.

3.2.5 Burst Write

Sequence

[A]. The master starts the burst write by driving WR on MCmd, the first address of the burst on MAddr, valid data on MData, a burst length of four on MBurstLength, the burst code INCR on MBurstSeq, and asserts MBurstPrecise. MReqLast must be

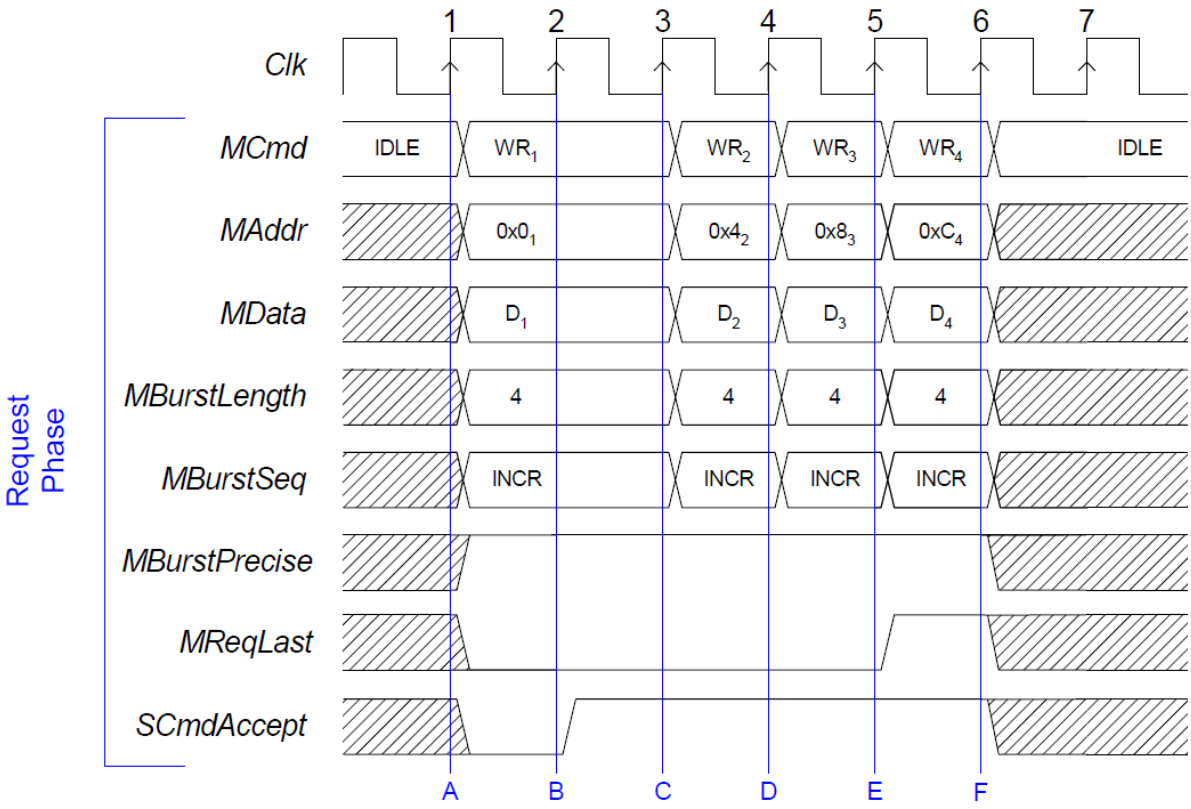

Figure 3.3: Waveform for Burst Write

deasserted until the last request in the burst. The burst signals indicate that this is an incrementing burst of precisely four transfers. The slave is not ready for anything, so it deasserts SCmdAccept.
[B]. The slave asserts SCmdAccept for a request accept latency of 1.
[C]. The master issues the next write in the burst. MAddr is set to the next word-aligned address. For 32-bit words, the address is incremented by 4. The slave captures the data and address of the first request.
[D]. The master issues the next write in the burst, incrementing MAddr. The slave captures the data and address of the second request.
[E]. The master issues the final write in the burst, incrementing MAddr, and asserting MBurstLast. The slave captures the data and address of the third request.
[F]. The slave captures the data and address of the last request.

3.3 SRAM Controller

The basic architecture of a SRAM includes one or more rectangular arrays of memory cells with support circuitry to decode addresses, and implement the required read and write operations. Additional support circuitry used to implement special features, such as burst operation, may also be present on the chip.

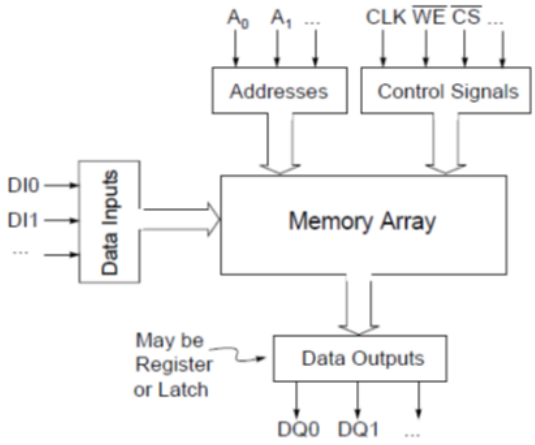

Figure 3.4: SRAM basic Architecture

3.3.1 Block diagram of SRAM controller

SRAM memory arrays are arranged in rows and columns of memory cells called word lines and bit lines, respectively. Each memory cell has a unique location or address defined by the intersection of a row and column. Each address is linked to a particular data input/output pin. The number of arrays on a memory chip is determined by the total size of the memory, the speed at which the memory must operate, layout and testing requirements, and the number of data I/Os on the chip.
An SRAM memory cell is a bi-stable flip-flop made up of four to six transistors. The flip-flop may be in either of two states that can be interpreted by the support circuitry to be a 1 or a 0.

3.3.2 Module wise hierarchy of SRAM sub-system

SRAM subsystem hierarchy is as shown in the fig 3.5.

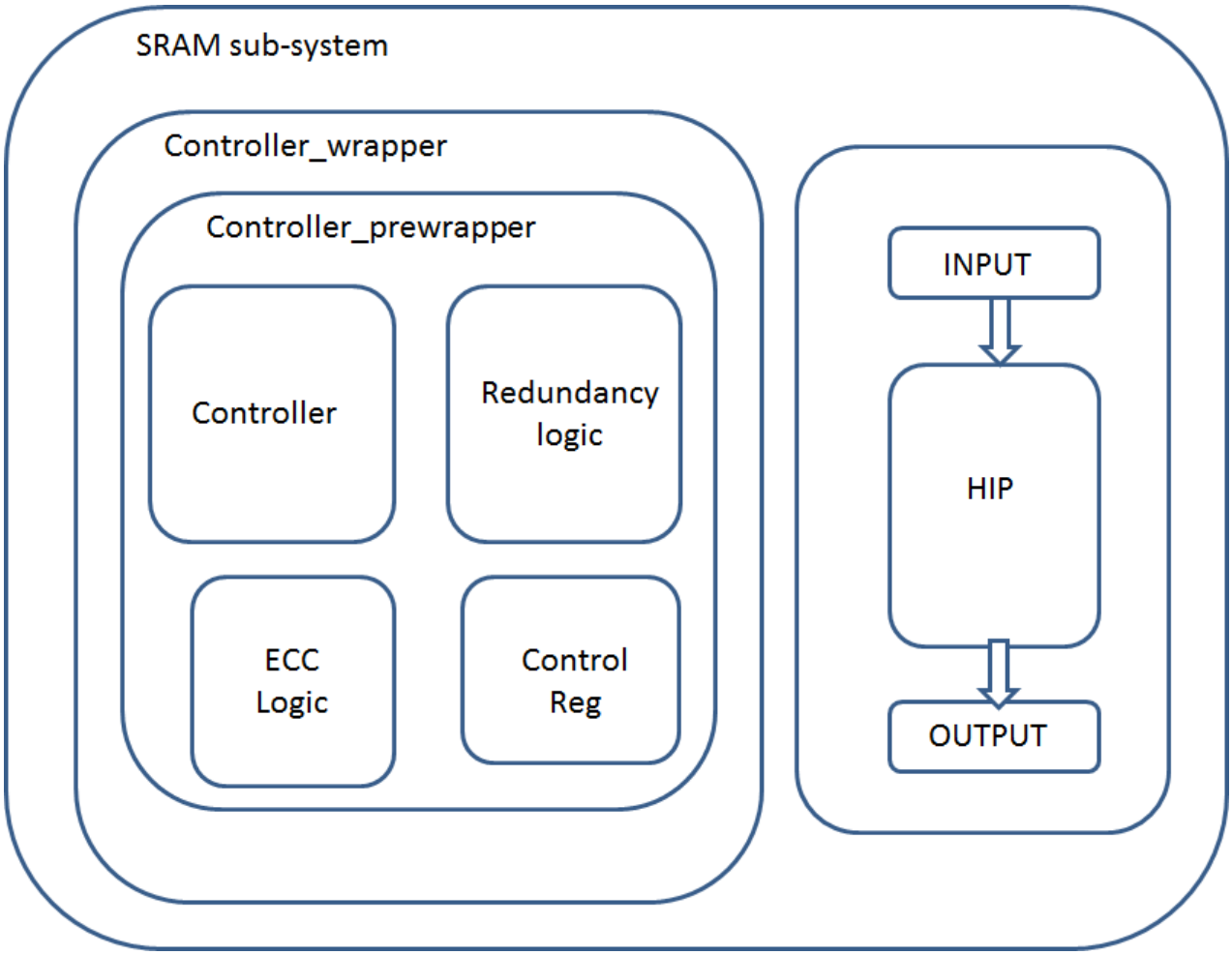

Figure 3.5: Hierarchy of SRAM sub-system

Controller:
SRAM memory controllers contain the logic necessary to read and write to SRAM. Reading and writing to SRAM is performed by selecting the row and column data addresses of the SRAM as the inputs to the multiplexer circuit, where the demultiplexer on the SRAM uses the converted inputs to select the correct memory location and return the data, which is then passed back through a multiplexer to consolidate the data in order to reduce the required bus width for the operation.

Redundancy Logic:
While reading or writing data from SRAM HIP by controller wrapper this block plays its role. When SRAM HIP is designed in Fab lab it has an accuracy of 99.99% so there may be chances that out of 1 million transistor one may not work well for that while fabrication itself they will have one extra column for redundancy. So the function of redundancy logic is to get the location of that particular damaged transistor from the top level module which is provided by the SRAM HIP provider and skip that entire column which will have faulty transistor in it and remaining function is same for accessing data.

ECC Logic:
This block will correct the error up to certain level of corruption in the data, if data is corrupted beyond that certain level than this block will raise and interrupt.

Control Reg:
This block contain different kind of control registers i.e., register for enable bit of bank, register for interrupt bit, interrupt mask register, erase address register.

3.3.3 Why use an SRAM?

There are many reasons to use an SRAM or a DRAM in a system design. Design tradeoffs include density, speed, volatility and cost. All of these factors should be considered before you select a RAM for your system design.

Speed:
The primary advantage of an SRAM over a DRAM is its speed. The fastest DRAMs on the market still require five to ten processor clock cycles to access the first bit of data. Although features such as EDO and Fast Page Mode have improved the speed with which subsequent bits of data can be accessed, bus performance and other limitations mean the processor must wait for data coming from DRAM. Fast, synchronous SRAMs can operate at processor speeds of 250 MHz and beyond, with access and cycle times equal to the clock cycle used by the microprocessor. With a well designed cache using ultra-fast SRAMs, conditions in which the processor has to wait for a DRAM access become rare.

Density:
Because of the way DRAM and SRAM memory cells are designed, readily available DRAMs have significantly higher densities than the largest SRAMs. Thus, when 64 Mb DRAMs are rolling off the production lines, the largest SRAMs are expected to be only 16 Mb.

Volatility:
While SRAM memory cells require more space on the silicon chip, they have other ad-

vantages that translate directly into improved performance. Unlike DRAMs, SRAM cells do not need to be efreshed. This means they are available for reading and writing data 100% of the time.

Cost:
If cost is the primary factor in a memory design, then DRAMs win hands down. If, on the other hand, performance is a critical factor, then a well-designed SRAM is an effective cost performance solution.

Chapter 4

Verilog for Parametrized module

In Verilog, there are two ways to define constants: the parameter, a constant that is local to a module and macro definitions, created using the ‘define compiler directive. A parameter, after it is declared, is referenced using the parameter name. A ‘define macro definition, after it is defined, is referenced using the macro name with a preceding ‘ (back-tic) character.
It is easy to distinguish between parameter and macro because macro have ‘identifier_name and parameter has identifier_name without ‘(back-tick).

4.1 Parameter

Parameters must be defined within module boundaries using the keyword parameter. A parameter is a constant that is local to a module that can optionally be redefined on an instance-by-instance basis. When instantiating modules with parameters, in Verilog there are two ways to change the parameters for some or all of the instantiated modules; parameter redefinition in the instantiation itself, or separate defparam statements. Parameter redefinition during instantiation of a module uses the # character to indicate that the parameters of the instantiated module are to be redefined.
In Example, two copies of the register are instantiated into the two_regs1 module. The SIZE parameter for both instances is set to 16 by the #(16)parameter redefinition values on the same lines as the register instantiations themselves.
The biggest problem with this type of parameter redefinition is that the parameters must be passed to the instantiated module in the order that they appear in the module being instantiated.
The defparam statement explicitly identifies the instance and the individual parameter that is to be redefined by each defparam statement. The defparam statement can be placed before the instance, after the instance or anywhere else in the file.

Unfortunately, the well-intentioned defparam statement is easily abused by:

1. Using defparam to hierarchically change the parameters of a module.
2. Placing the defparam statement in a separate file from the instance being modified.
3. Using multiple defparam statements in the same file to change the parameters of an instance.
4. Using multiple defparam statements in multiple different files to change the parameters of an instance.

In the case of multiple defparams for a single parameter, the parameter takes the value of the last defparam statement encountered in the source text.

```
module register (q, d, clk, rst_n);
  parameter SIZE=8;
  output [SIZE-1:0] q;
  input  [SIZE-1:0] d;
  input             clk, rst_n;
  reg    [SIZE-1:0] q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Figure 4.1: Parameterized register model

```
module register (q, d, clk, rst_n);
  parameter SIZE=8;
  output [SIZE-1:0] q;
  input  [SIZE-1:0] d;
  input             clk, rst_n;
  reg    [SIZE-1:0] q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Figure 4.2: Instantiation using parameter redefinition

4.2 Macro definition

The ̀define compiler directive is used to perform "global" macro substitution, similar to the C-language #define directive. Macro substitutions are global from the point of definition and remain active for all files read after the macro definition is made or until another macro definition changes the value of the defined macro or until the macro undefined using the ‘undef compiler directive.
Macro definitions can exist either inside or outside of a module declaration, and both are treated the same. parameter declarations can only be made inside of module boundaries. Since macros are defined for all files read after the macro definition, using macro definitions generally makes compiling a design file-order dependent. A typical problem associated with using macro definitions is that another file might also make a macro definition to the same macro name. When this occurs, Verilog compilers issue warnings related to "macro redefinition" but an unnoticed warning can be costly to the design or

to the debug effort.
Why is it bad to redefine macros? The Verilog language allows hierarchical referencing of identifiers. This proves to be very valuable for probing and debugging a design. If the same macro name has been given multiple definitions in a design, only the last definition will be available to the testbench for probing and debugging purposes. If you find yourself making multiple macro definitions to the same macro name, consider that the macro should probably be a local parameter as opposed to a global macro.
An enhancement added to the Verilog-2001 Standard is the localparam. Unlike a parameter, a localparam cannot be modified by parameter redefinition (positional or named redefinition) nor can a localparam be redefined by a defparam statement. The localparam can be defined in terms of parameters that can be redefined by positional parameter redefinition, named parameter redefinition (preferred) or defparam statements. The idea behind the localparam is to permit generation of some local parameter values based on other parameters while protecting the localparams from accidental or incorrect redefinition by an end-user.

4.3 Conclusion

In conclusion to this each time a new macro definition is made, that macro name cannot be safely used elsewhere in the design (name-space pollution). As more and more modules are compiled into large system simulations, the likelihood of macro-name collision increases. The practice of making macro definitions for constants such as port or data sizes and state names is an ill-advised practice. Macro definitions using the ‘define compiler directive should not be used to define constants that can be better localized to individual modules. Verilog parameters are intended to represent constants that are local to a module. A parameter has the added benefit that each different instance of the module can have different values for the parameters in each module.

Following is a summary of important guideline

1. do not use defparams in any Verilog designs.
2. only use macro definitions for identifiers that clearly require global definition of an identifier that will not be modified elsewhere in the design.
3. where possible, place all macro definitions into one ”definitions.vh” file and read the file first when compiling the design.
4. do not use macro definitions to define constants that are local to a module.

Chapter 5

Features added in the current project

- Adding a dedicated port for DMA to SRAM Controller which will support data width of 32.
- Parameterization of adding new banks to controller.
- Parameterization for adding different form factors, Example : 32k*4, 16k*8, 8k*16, 4k*32.

5.1 Dedicated port for DMA

To have a dedicated for DMA we can have two option either we can directly connect DMA to SRAM controller or we can have DMA connection via OCP fabric, these two configuration are shown in figure 4.1 and 4.2 respectively.

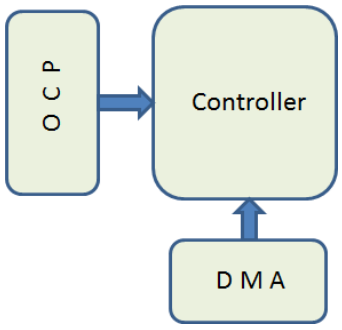

Figure 5.1: DMA port directly connected to SRAM controller

5.1.1 Arbiter

Many systems exist in which a large number of requesters must access a common resource. The common resource may be a shared memory, a networking switch fabric, a specialized state machine, or a complex computational element. An arbiter is required to determine how the resource is shared amongst the many requesters. When putting an arbiter into

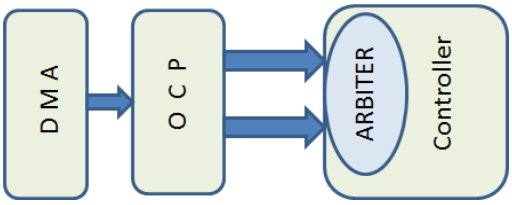

Figure 5.2: DMA port connected through OCP to SRAM controller

a design, many factors must be considered. The interface between the requesters and the arbiter must be appropriate for the size and speed of the arbiter. Also, the coding style used will usually impact the synthesis results.
Interfacing to an arbiter can appear very straight forward at first. The requester sends a request (req) signal, and the arbiter returns a grant. Requests to an arbiter are generally driven by either a FIFO queue or a state machine. A state machine requester is commonly used when the arbiter is used in a memory controller. If a portion of the memory is used for variable storage, a state machine may need to periodically read and/or write those variables.

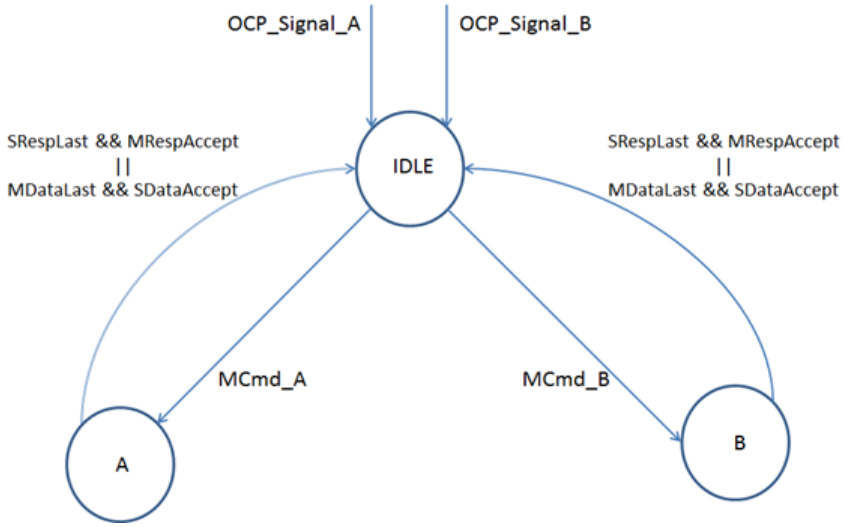

Figure 5.3: Arbiter FSM

As show in the FSM there are three states of it i.e., IDLE, A(uIA) and B(DMA). As long as the MCmd received from the OCP fabric from either DMA or uIA it will go in the respective state to complete the transaction. As soon as SRespLast and MRespAccep goes high at same instance of MDataLast and SdataAccept goes high at same instance it will return from the A or B state to IDLE state.
Following type of waveform observed in the DVE for the burst read mode and burst write mode. MRespAccept, SRespLast, MDataLast and SDataAccept these signals need to be under surveillance for designing FSM.

5.2 Parameterization for adding new banks to controller

Then main reason behind parameterizing is to have increase or decrease size of SRAM with few quick changes in Param file. So I have added on Parameter BANK_NUM in the

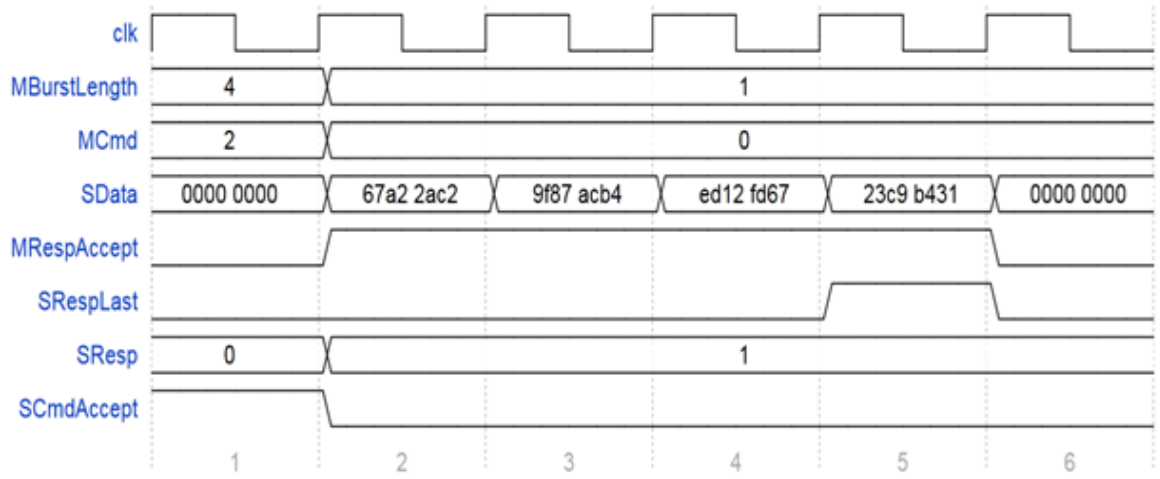

Figure 5.4: Waveform for burst read

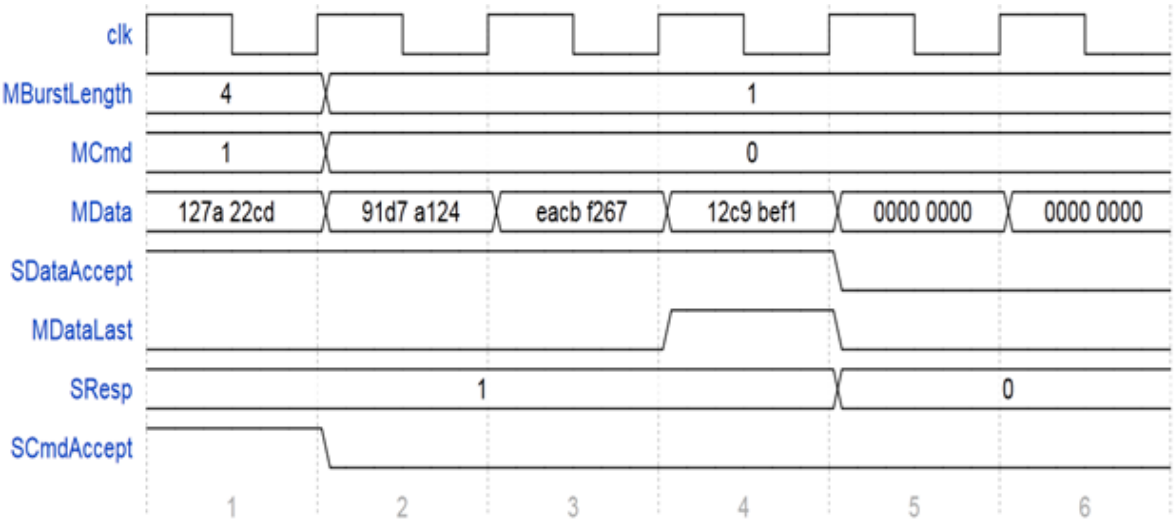

Figure 5.5: Waveform for burst write

current project which will changed depending on the requirement of the project.
One limitation for the design I have done for Parameterization of number of banks is that we cant increase the number of bank beyond 30 banks as there is one control register in the SRAM controller which will have 32 bit size out of them 2 bit are reserved for some interrupt status bit. This issue could also be solved if we have some more register for storing status bit.

5.3 Parameterization for adding different form factors

We can increase or decrease the size of SRAM by two approaches either we have increase the number of bank in current SRAM or we can increase size of each bank in current SRAM. So this second approach i.e., to increase size of each bank.

Chapter 6

RTL QUALITY CHECK (LINTRA TOOL)

Lintra is a tool for linting RTL code i.e., analysing RTL code to verify that it conforms to design rules and coding guidelines. Lintra is built for RTL languages, providing an RTL Data Model with a C++/Perl application program interface (API), and a built-in light synthesis capability. Lira supports iHDL, Verilog and System Verilog. The warnings and errors issued during Lira's compilation flow form Lintra's 'built in rules.

6.1 LINTRA INPUTS AND OUTPUTS

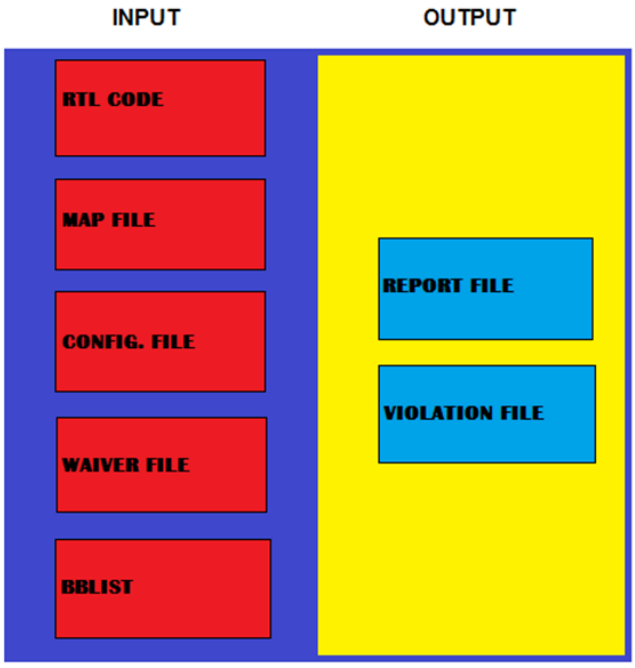

Figure 6.1: Lintra requirements

- **Lintra Inputs:** RTL code configuration fille (.f), waiver fille (.w), map file (.map), bblist.
- **Lintra Outputs:** Report file (.rep), Violation file (.xml).

6.1.1 Lintra Input

- **RTL code:** Lintra can be run on any .v and .sv file. This the input for lintra as we perform linta on RTL code for this file we will check the errors and warnings.

- **Map file :** It contains commands for setting environmental variables which are utilized by the lintra script. For the lintra script, the top module name and Lintra run directory are required.

- **BB list :** BB list is a black box list. Sometimes, we don't want to perform the lint on sub-modules with in a module. In such conditions, those sub module names should be kept in the bblist file. Eg: Since SPI is an external IP, we might not want to run Lintra on this module.

- **Configuration file :** It should contain the paths of directories where Lintra can search for the modules. The mode of usage of BB list and the path for BB list also need to be mentioned here. There are different modes in which Lintra process the BB list. Full mode implies Lintra won't analyze the entire module and modules mode implies it will analyze the given module but wont analyze the sub-modules within the given module.

- **Waiver file :** It can be used to mask the errors or warnings which are of low priority or which are invalid. Lintra supports two types of waiver mechanisms.
  Each waiver contains the following filter fields: rule id, message regexp, fileregexp, and limit. The waiver will filter out violations that match the given rule id, whose message matches the given message (whether as an exact string match or as a regular expression match) and whose location matches the given file expression. If any of the filters are left unspecified, messages will not be filtered by this attribute (e.g. if only the message is specified, the rule id and file will be ignored). In addition, if a number greater than zero is specified for the limit, the number of violations caught by the waiver will be thus be limited. This is particularly useful when the violation message is the same for all violations of the rule, and offers a way to waive a particular violation, yet be warned when additional violations of this rule occur (in the same file). The regular expression semantics used by Lintra is QT regular expression, which is similar to PERL, where QT's quantifiers are the same as Perl's greedy quantifiers. Note - when using regular expressions, it may be necessary to escape characters in the original message text (such as brackets) by adding / before them. In addition to the filter, the waiver contains a description field, a unique id, an owner, opening date and expiration date. The description field is optional, and contains free text. The unique id, owner, and opening date fields are mandatory, and are automatically set to the userid , system date, and combination of user plus timestamp when the waiver is created through Lintra GUI. The expiration date can be used to create temporary waivers.

6.1.2 Lintra Output

- **Report file :** It reports all the errors and warnings. It contains all the violation which are occurred in (.v) and (.sv) file. It gives all the details of the file like path of the file ,on which line error occurred ,severity of the violation, which lintra rule it follows and a message of the error.

```
<waiver rule="0209"
file=".*/ip-ish-.*/*/*.*"
message= ".*"
desc="Statements in loop will never execute in ish_sram_configreg"
limit="-1"
open_date=""
id="ish-lint-sram-subsystem-0209" />

<waiver rule="0393"
file=".*/ip-ish-.*/*/*.*"
message= ".*"
desc="expression bit length (14) is smaller than the bit length of the context (32) in ish_sr
am_configreg.v"
limit="-1"
open_date=""
id="ish-lint-sram-subsystem-0393" />

<waiver rule="0214"
file=".*/ip-ish-.*/*/*.*"
message= ".*"
desc="unconnected output port in instantiation of the module in ish_srams.v"
limit="-1"
open_date=""
id="ish-lint-sram-subsystem-0214" />
```

Figure 6.2: Waiver example

```
Lintra 11.2p2_shOpt64
FILE: /nfs/iind/disks/mg_disk0789/rkshah/project_synthesis/ip-ish-0p7/source/rtl/ctech/ish_map.v
Line    Sever.  Rule    Message
=====================================================================
  262   Error   8213    Use of a non synthesis level construct 'delays' - construct is ignored. (KEY : "delays" ,
HIERARCHY : "ish_sram_subsystem")
  263   Error   8213    Use of a non synthesis level construct 'delays' - construct is ignored. (KEY : "delays" ,
HIERARCHY : "ish_sram_subsystem")
  267   Error   8213    Use of a non synthesis level construct 'delays' - construct is ignored. (KEY : "delays" ,
HIERARCHY : "ish_sram_subsystem")
  268   Error   8213    Use of a non synthesis level construct 'delays' - construct is ignored. (KEY : "delays" ,
HIERARCHY : "ish_sram_subsystem")
FILE: /nfs/iind/disks/mg_disk0789/rkshah/project_synthesis/ip-ish-0p7/source/rtl/sram_subsystem/ish_clk_sample_gen
.sv
Line    Sever.  Rule    Message
=====================================================================
   99   Error   70023   asynchronous set/reset pin 'ish_clk_sample_gen_default_params.count_rst' of a flop is driv
en by combinational logic at /nfs/iind/disks/mg_disk0789/rkshah/project_synthesis/ip-ish-0p7/source/rtl/sram_subsy
stem/ish_clk_sample_gen.sv:66       (KEY : "ish_clk_sample_gen_default_params.count_rst,/nfs/iind/disks/mg_disk078
9/rkshah/project_synthesis/ip-ish-0p7/source/rtl/sram_subsystem/ish_clk_sample_gen.sv:66" , HIERARCHY : "ish_sram_
subsystem.ish_srm_gated_vnn_wrapper.ish_sram_prefw.ish_clocks_rst_sram.ish_clk_sample_gen_default_params")
  119   Error   70023   asynchronous set/reset pin 'ish_clk_sample_gen_default_params.clk_sample_i_set' of a flop
is driven by combinational logic at /nfs/iind/disks/mg_disk0789/rkshah/project_synthesis/ip-ish-0p7/source/rtl/sra
m_subsystem/ish_clk_sample_gen.sv:66        (KEY : "ish_clk_sample_gen_default_params.clk_sample_i_set,/nfs/iind/d
isks/mg_disk0789/rkshah/project_synthesis/ip-ish-0p7/source/rtl/sram_subsystem/ish_clk_sample_gen.sv:66"  , HIERAR
CHY : "ish_sram_subsystem.ish_srm_gated_vnn_wrapper.ish_sram_prefw.ish_clocks_rst_sram.ish_clk_sample_gen_default_
params")
FILE: /nfs/iind/disks/mg_disk0789/rkshah/project_synthesis/ip-ish-0p7/source/rtl/sram_subsystem/ish_sram_configreg
.v
Line    Sever.  Rule    Message
=====================================================================
  198   Error   0507    Signal 'ish_sram_configreg.sram_scfgr_be[13]' does not have a synchronous part in a block
controlled by edge-sensitive condition. Lira will model the signal as a latch.        (KEY : "ish_sram_configreg.s
ram_scfgr_be[13]"   , HIERARCHY : "ish_sram_subsystem")
  257   Warning 0209    Statements in loop will never execute. Loop initial: 'i = 22'; loop condition: 'i <= 19'.
     (KEY : " Loop initial: 'i = 22'; loop condition: 'i <= 19'."    , HIERARCHY : "ish_sram_subsystem")
```

Figure 6.3: Report file example

- **Violation file :** The violation file uses configuration file that allows customizing Lintra runs to the project requirements. It provides control over which severity levels will cause Lintra to terminate with a 'failed' exit status, and provides capabilities for determining which rules are active and tailoring the rules by changing the value of configurable parameters that are coded into the rules.

6.1.3 Lintra exit status

Lintra will terminate with one of the following

- **One:** If errors detected in the command line or when loading the configuration file or the UDRs prevented the lint run, Lintra will terminate with an exit status of 1

- **Two:** If during lint violations of a severity declared as 'cause failure' in the configuration file were reported, Lintra will report 'lint Failed' and terminate with an exist status of 2.

Types of Violation in Lintra

- **FATAL:** If fatal violation comes then the lintra will get terminiated.so, to remove the fatal error is the first priority of Lintra.

- **ERROR:** Depending on the type of error it can be fixed or waived.

- **WARNING:** It is decided by the designer whether he wants to fix the warning or to waive it.

Chapter 7

LEC or FEV

7.1 Introducation

The register transfer level (RTL) behavior of a digital chip is usually described with a hardware description language, such as Verilog or VHDL. This description is the golden reference model that describes in detail which operations will be executed during which clock cycle and by which pieces of hardware. Once the logic designers, by simulations and other verification methods, have verified register transfer description, the design is usually converted into a netlist by a logic synthesis tool. Equivalence is not to be confused with functional correctness, which must be determined by functional verification.
The initial netlist will usually undergo a number of transformations such as optimization, addition of Design For Test (DFT) structures, etc., before it is used as the basis for the placement of the logic elements into a physical layout. Contemporary physical design software will occasionally also make significant modifications (such as replacing logic elements with equivalent elements that have a higher or lower drive strength) to the netlist. Throughout every step of a very complex, multi-step procedure, the original functionality and the behavior described by the original code must be maintained. When the final tape-out is made of a digital chip, many different EDA programs and possibly some manual edits will have altered the netlist.
In theory, a logic synthesis tool guarantees that the first netlist is logically equivalent to the RTL source code. All the programs later in the process that make changes to the netlist also, in theory, ensure that these changes are logically equivalent to a previous version.
In practice, programs have bugs and it would be a major risk to assume that all steps from RTL through the final tape-out netlist have been performed without error. Also, in real life, it is common for designers to make manual changes to a netlist, commonly known as Engineering Change Orders, or ECOs, thereby introducing a major additional error factor. Therefore, instead of blindly assuming that no mistakes were made, a verification step is needed to check the logical equivalence of the final version of the netlist to the original description of the design (golden reference model).
Historically, one way to check the equivalence was to re-simulate, using the final netlist, the test cases that were developed for verifying the correctness of the RTL. This process is called gate level logic simulation. However, the problem with this is that the quality of the check is only as good as the quality of the test cases. Also, gate-level simulations are notoriously slow to execute, which is a major problem as the size of digital designs continues to grow exponentially.

An alternative way to solve this is to formally prove that the RTL code and the netlist synthesized from it have exactly the same behavior in all (relevant) cases. This process is called formal equivalence checking or logical equivalence checking and is a problem that is studied under the broader area of formal verification.
Since the RTL is dynamically simulated to be functionally correct, the formal verification of the design between the RTL and the scan inserted gate-level netlist assures us that the gate-level also has the same functionality. In this instance if we were to use the dynamic simulation method to verify the gate-level, it would have taken a long time (days and weeks, depending on the size of the design) to verify the design. In comparison, the formal method would take a few hours to perform a similar verification. The last part involves verifying the gate-level netlist against the gate-level netlist. This too is a significant step for the verification process, since it is mainly used to verify what has gone into the layout versus what has come out of the layout. What comes out of the layout is obviously the clock tree inserted netlist (flat or hierarchical). This means that the original netlist that goes into the layout tool is modified.

The figure shows the simplification of the general equivalence checking problem to logic equivalence.

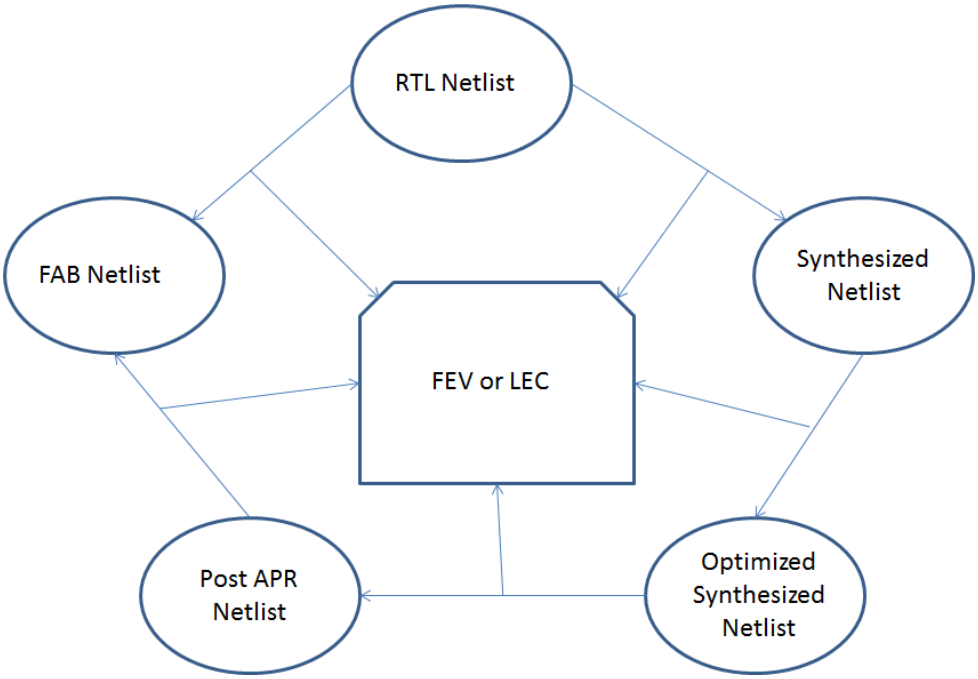

Figure 7.1: FEV at each stage

7.2 LEC do file

```
set log file lec.log -replace
read design -systemverilog -gold -f myrtl.filelist
read design -systemverilog -rev -f mynetlist.filelist
add renaming rule r1 foo bar -gold
set sys mode lec
report unmapped points
add compare points -all
```

compare
report compare data

7.3 Steps to run LEC

- Create work area for the design on which LEC is to be performed.
- Copy the necessary files i.e. golden and revised netlist and libraries into their corresponding path in the work area.
- Run command for LEC. This command will give the result for key points and compared points.
- Debug the key points and diagnose the not mapped key points.
- Make changes in local copy in order to solve the non-equivalent points.
- Report the compared points, if they are the major issues.
- Prepare Lec.do file which will automatically run the script to make LEC clean.

7.4 Inputs to LEC

- RTL/BMOD
- Schematic netlist (With correct tagging info)
- Black Block list

Chapter 8

Verification

8.1 Introduction to Open Verification Methodology (OVM)

8.1.1 OVM and Coverage-Driven Verification (CDV)

OVM provides the best framework to achieve coverage-driven verification (CDV). CDV combines automatic test generation, self-checking testbenches, and coverage metrics to significantly reduce the time spent verifying a design. The purpose of CDV is to:

- Eliminate the effort and time spent creating hundreds of tests.
- Ensure thorough verification using up-front goal setting.
- Receive early error notifications and deploy run-time checking and error analysis to simplify debugging.

The CDV flow is different than the traditional directed-testing flow. With CDV, you start by setting verification goals using an organized planning process. You then create a smart testbench that generates legal stimuli and sends it to the DUT. Coverage monitors are added to the environment to measure progress and identify non-exercised functionality. Checkers are added to identify undesired DUT behavior. Simulations are launched after both the coverage model and testbench have been implemented. Verification then can be achieved.
Using CDV, you can thoroughly verify your design by changing testbench parameters or changing the randomization seed. Test constraints can be added on top of the smart infrastructure to tune the simulation to meet verification goals sooner. Ranking technology allows you to identify the tests and seeds that contribute to the verification goals, and to remove redundant tests from a test-suite regression.
CDV environments support both directed and constrained-random testing. However, the preferred approach is to let constrained-random testing do most of the work before devoting effort to writing time-consuming, deterministic tests to reach specific scenarios that are too difficult to reach randomly. Significant efficiency and visibility into the verification process can be achieved by proper planning. Creating an executable plan with concrete metrics enables you to accurately measure progress and thoroughness throughout the design and verification project. By using this method, sources of coverage can be planned, observed, ranked, and reported at the feature level. Using an abstracted, feature-based approach (and not relying on implementation details) enables you to have a more readable, scalable, and reusable verification plan.

8.1.2 OVM Testbench and Environments

An OVM testbench is composed of reusable verification environments called OVM verification components (OVCs). An OVC is an encapsulated, ready-to-use, configurable verification environment for an interface protocol, a design submodule, or a full system. Each OVC follows a consistent architecture and consists of a complete set of elements for stimulating, checking, and collecting coverage information for a specific protocol or design. The OVC is applied to the device under test (DUT) to verify your implementation of the protocol or design architecture. OVCs expedite creation of efficient testbenches for your DUT and are structured to work with any hardware description language (HDL) and high-level verification language (HVL) including Verilog, VHDL, e, SystemVerilog, and SystemC.
OVCs might be stored in a company repository and reused for multiple verification environments. The interface OVC is instantiated and configured for a desired operational mode. The verification environment also contains a multi-channel sequence mechanism (that is, virtual sequencer) which synchronizes the timing and the data between the different interfaces and allows fine control of the test environment for a particular test.

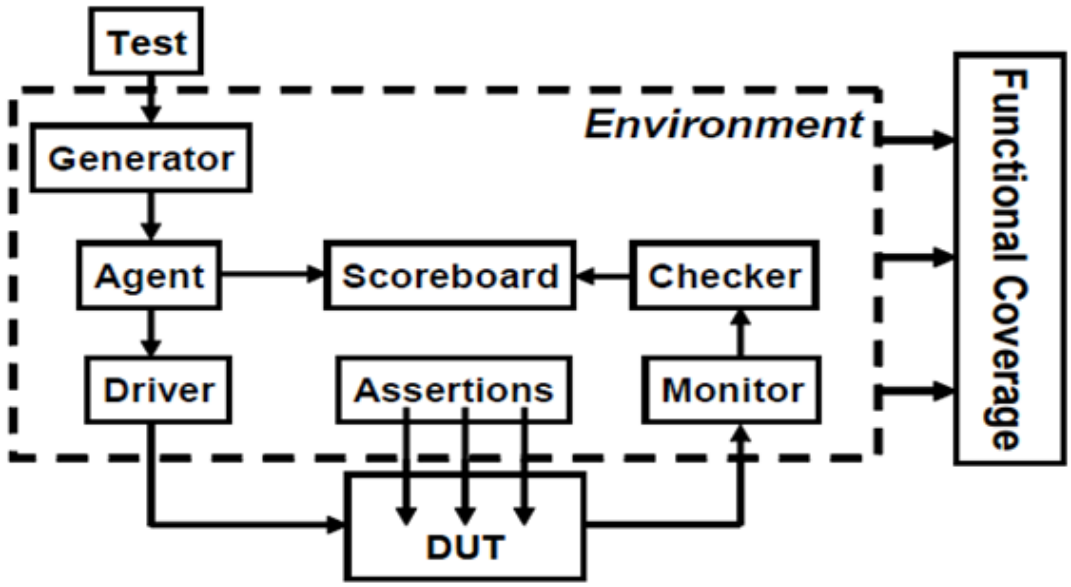

Figure 8.1: OVM Environment

- **Generator**

The generator component generates stimulus which are sent to DUT by driver. Stimulus generation is modeled to generate the stimulus based on the specification. For simple memory stimulus generator generates read, write operations, address and data to be stored in the address if its write operation. Scenarios like generate alternate read/write operations are specified in scenario generator. SystemVerilog provided construct to control the random generation distribution and order. Constraints defined in stimulus are combinatioural in nature where as constraints defined in stimulus generators are sequential in nature.
Stimulus generation can be directed or directed random or automatic and user should

have proper controllability from test case. It should also consider the generation of stimulus which depends on the state of the DUT for example, Generating read cycle as soon as interrupt is seen. Error injection is a mechanism in which the DUT is verified by sending error input stimulus. Generally it is also taken care in this module. Generally generator should be able to generate every possible scenario and the user should be able to control the gener;ation from directed and directed random testcases.

- **Driver**

The drivers translate the operations produced by the generator into the actual inputs for the design under verification. Generators create inputs at a high level of abstraction namely, as transactions like read write operation. The drivers convert this input into actual design inputs, as defined in the specification of the designs interface. If the generator generates read operation, then read task is called, in that, the DUT input pin "read_write" is asserted.

- **Monitor**

Monitor reports the protocol violation and identifies all the transactions. Monitors are two types, Passive and active. Passive monitors do not drive any signals. Active monitors can drive the DUT signals. Sometimes this is also refered as receiver. Monitor converts the state of the design and its outputs to a transaction abstraction level so it can be stored in a 'score-boards' database to be checked later on. Monitor converts the pin level activities in to high level.

- **Checker**

The monitor only monitors the interface protocol. It doesn't check the whether the data is same as expected data or not, as interface has nothing to do with the date. Checker converts the low level data to high level data and validated the data. This operation of converting low level data to high level data is called Unpacking which is reverse of packing operation. For example, if data is collected from all the commands of the burst operation and then the data is converted in to raw data , and all the sub fields information are extracted from the data and compared against the expected values. The comparison state is sent to scoreboard.

- **Scoreboard**

Scoreboard is sometimes referred as tracker. Scoreboard stores the expected DUT output. Scoreboard in Verilog tends to be cumbersome, rigid, and may use up much memory due to the lack of dynamic data types and memory allocation. Dynamic data types and Dynamic memory allocation makes it much easier to write a scoreboard in SystemVerilog. The stimulus generator generated the random vectors and is sent to the DUT using drivers. These stimuli are stored in scoreboard until the output comes out of the DUT. When a write operation is done on a memory with address 101 and data 202, after some cycles, if a read is done at address 101, what should be the data? The score board recorded the address and data when write operation is done. Get the data stored at address of 101 in scoreboard and compare with the output of the DUT in checker module. Scoreboard also has expected logic if needed. Take a 2 inputs and gate. The expect logic does the "and " operation on the two inputs and stores the output".

- **Enviornemnt**

Environment contains the instances of all the verification component and Component connectivity is also done. Steps required for execution of each component is done in this.

- **Tests**

Tests contain the code to control the TestBench features. Tests can communicate with all the TestBench components. Once the TestBench is in place, the verification engineer now needs to focus on writing tests to verify that the device behaves according to specification.

- **Functional coverage**

This component has all the coverage related to the functional coverage groups.

After making RTL changes in the design for verification of I have designed the test case which will access the 4 random address location from each back and write the address value itself in the register after writing the data in all the banks, reading of data is performed and as shown in the flowchart if the write and read data does not matches then OVM will give error and test case will get terminate.

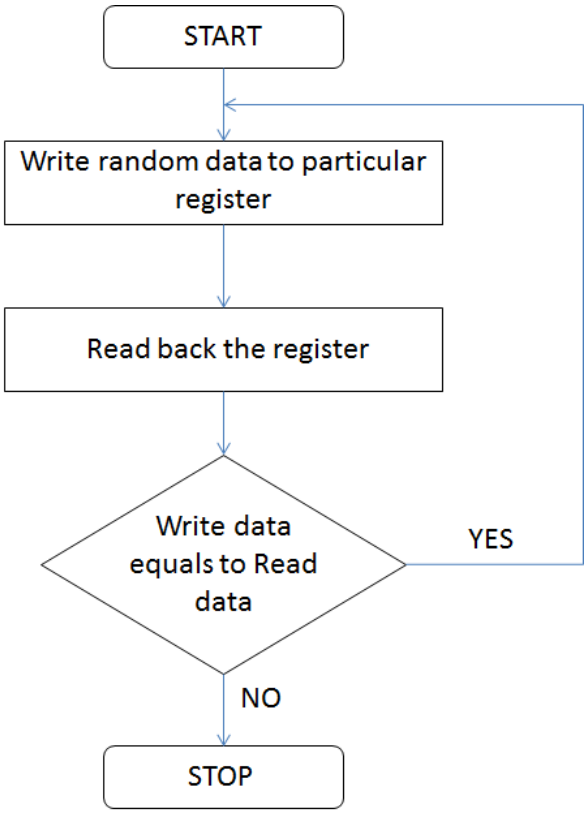

Figure 8.2: Test case flow

Following are the output waveform observed while running the above test case in the environment which will verify the design.

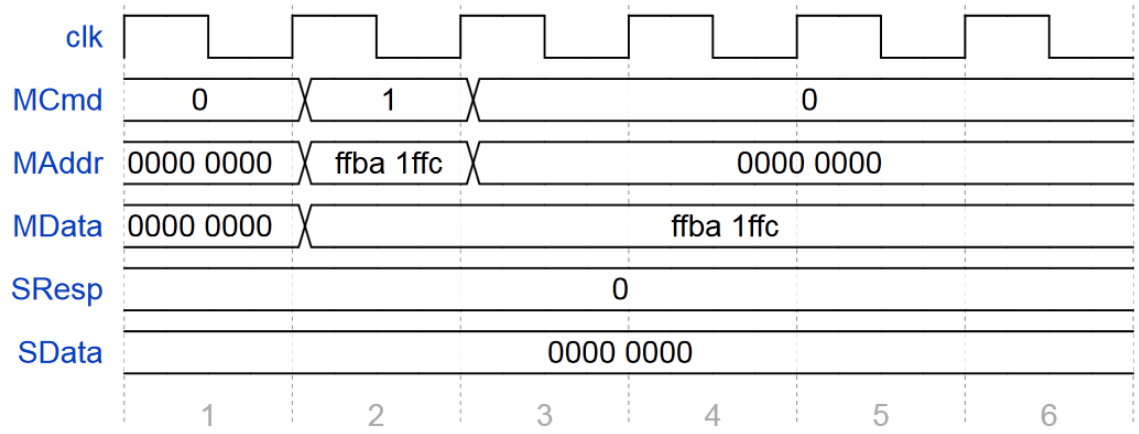

Figure 8.3: Waveform while writing into specific register

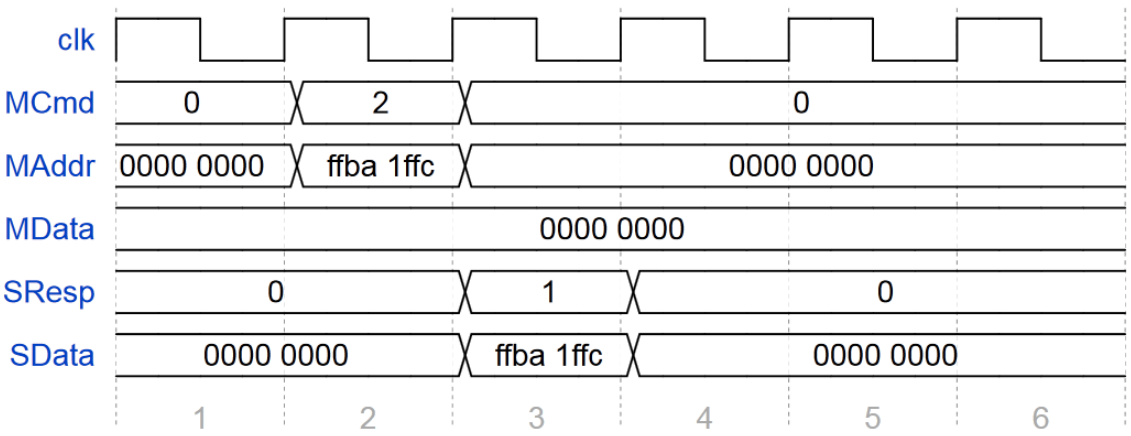

Figure 8.4: Waveform while reading from specific register

Chapter 9

Future Enhancement

- Support for 64 bit addressing.
- Support for various SRAM latency option.
- Support for non uniform bank size.

Chapter 10

Conclusion

- As there is requirement to increase or to decrease the size of memory in the new project design, it is good to parameterize the memory controller to meet the required objective as fast as possible.
- Lintra is used to check RTL quality check of the design, I have performed lintra on each module of SRAM and also at the top level. All the fatal errors are fixed from RTL design. Depending upon the design requirement I have waived many errors and warning.
- Formal Equivalence Verification is very useful and important for VLSI design flow.It is also helpful for verification of ECO (Engineering Change Order) design with exhaustive verification.
- Updated design has been verified in the OVM environment by making changes in present OVM environment as well to make it compatible with design.

Chapter 11

Reference

[1] Clifford E. Cummings, “Verilog–2001 Techniques for creating parameterized models”.

[2] Samir Palnitkar, “Verilog HDL: A Guide to Digital Design and Synthesis”.

[3] Bhasker J, “Verilog HDL Synthesis, A practical Primer, Star Galaxy publishing”.

[4] Moris Mano, “Digital design”.

[5] Open core Protocol Specification 2.1

[6] OVM user guide version 2.1.1, March 2010

[7] Intelpedia.

[8] Matt Wabber, “Arbiters: Design Ideas and Coding Styles”.