

Modeling ASIC Memories in VHDL

*Ekambaram Balaji, Design Engineer
Prabhu Krishnamurthy, Design Engineer
LSI Logic Corporation
1501, McCarthy Blvd, M/S E-192
Milpitas, CA 95035
ebalaji@lsil.com, prabhu@lsil.com*

Abstract

Memories are an important component in ASIC designs. As million gate ASICs are becoming reality, there is an increasing need to model memory devices with a high level of accuracy and simulation efficiency. There are no modeling standards available currently to model memories in VHDL. This paper describes the functional/timing aspects of VHDL memory models, their implementation, and issues involved in various approaches for modeling memories in VHDL. This paper also presents a generic memory interface package used in the development of VHDL models of ASIC memories.

1. Introduction

Modeling memories in VHDL can be a difficult task, but it is important that they be modeled accurately with regard to functional/timing aspects and efficiently to minimize initialization, accessing overhead and computing resources. An ASIC foundry has to support multiple CAE tools and hence developing and maintaining these models can be a challenging task. This paper brings out the experiences and findings of the authors while developing simulation models in VHDL, for memory architectures available in LSI Logic's array and cell based technologies. **The memories modeled are Static RAMs and ROMs.** But the same principles can be extended to complex memories as well.

The main objective of this paper is to present an implementation strategy for VITAL compatible memory models. This strategy is modular and generic in nature and has evolved after prototyping few modeling solutions. Some of the principles for VHDL memory implementation have been derived from the Verilog environment, as the memory models in Verilog are already in use for design sign-off.

2. LSI Logic Memories

LSI Logic memories are typically designed as addressable latches. The memory latches are spread out on a rectangular grid and have specific row and column addresses. Under normal asynchronous operation, the RAM always reads the contents of the currently addressed memory location on the data output bus. Due to a address change on the read address bus, the new address gets decoded, and the data appears on the data output bus after a hold time, referred to as Toh. After the memory access time (Taa time), the data on the output bus is considered to be valid corresponding to the new address.

Write operations are triggered by activating write enable pins, which internally make the memory latches transparent, allowing new data to be written into the memory location. If the read address bus is pointing to the memory location that is being updated as a result of a write operation, then the data that is written into the memory location appears on the data output bus as well. For a write operation, the propagation time for the data to reach the output bus depends on who initiated the write operation: either the write enable pin going active or the data input bus changing while the write enable pin is active.

A synchronous RAM is much like an asynchronous RAM with flip-flops attached to all the RAM inputs. Proper clocking of data into the input flip-flops ensures correct operation of the internal asynchronous part of the RAM. The clock triggers all read and write operations for synchronous RAMs, but internal operations after the flip-flop stage are exactly the same for both synchronous and asynchronous RAMs. One main difference between asynchronous and synchronous RAMs is the ability to do consecutive read or write operations by keeping the write enable pin at a constant high or low and applying a series of clock pulses.

2.1 Multi-port Memory Architectures

Consider three different multi-port RAM architectures and their behavior during read/write operations:

In the case of a 2-port (1 read, 1 write) memory, the read port is totally asynchronous, and the contents of the currently addressed memory location appear at the data output bus. However, if a write operation is initiated by a write address port and both the read and write ports address the same memory location, then the current read operation to that address is disabled. Also, the most recently updated data appears at the data output bus after the **Cross Port Delay** (twwd) instead of **Address Access time** (taa).

In the case of a 2-port (1 read/write, 1 read) memory, there are two output buses, DOA and DOB. The B port is the read port and is asynchronous. When the A (write port) and B ports address the same location while the A port is writing data, the new data written into the memory location appears at the DOB output bus, similar to the situation above.

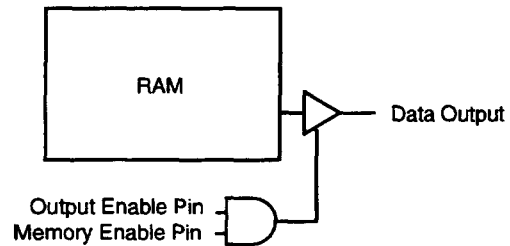
In the case of a 2-port (2 read/write) memory, the read/write operations proceed independently as long as the A and B addresses do not match. When the A address equals the B address and if port A writes to an address being read by port B, then the new data appears on the DOB bus after the **Cross Port Delay** (twwd). The data appears on DOA after the **Same Port Delay** (twws). Simultaneous write operations from two different ports onto the same location is illegal and causes the RAM to issue a warning message, abort the current read/write operations on both ports, and corrupt the memory location.

2.2 Chip Select Controlled Memories

The chip-select signal is used to control the functions of the RAM in certain architectures. The RAM exhibits normal behavior when chip-select is high. When chip-select is low, it disables all RAM functions, except for retaining data and has the advantage of reducing power consumption. Multiple bits of the chip-select signal can control the higher- and lower-order bytes of RAMs. In case of an odd bits word size, the higher order byte consists of one bit more than the lower-order byte. For memories with a chip-select control signal, a read operation could also be initiated by the active edge of the chip-select signal. Data is read from the currently addressed memory location when the chip-select signal goes active. A write operation could be initiated/terminated by the chip-select signal going active/inactive, if the write enable pin is active.

2.3 Memory Enable Controlled Memories

A memory enable signal is used to control the functions in certain synchronous RAMs. This pin gates all inputs into the RAM, including the clock input. Normal memory operations (read/write) are performed when the memory enable pin is high (just as in a synchronous RAM without the memory enable pin) and when all setup and hold timing requirements with respect to the clock pin are satisfied. All timing checks are conditioned with the memory enable signal being high. If the signal is low, then address, data, write enable, and setup/hold violations are not reported. Enable (time for data output to go from Z to 0/1) and disable times (time for data output to go from 0/1 to Z) are modeled for the memory enable pin to output bus:



If either the output enable or memory enable pins are low, then the data output is in the high-impedance (Z) state. When both output enable and memory enable pins are high, the data output is valid and at 0 or 1.

2.4 Memory Compiler

The source of all memory models are memory templates. A template is a configuration independent representation of a model in VHDL written in a template language. This template language is a layer on top of the VHDL description and specifies the rules for expanding the various data, address and control functions. The memory compiler takes the user specified inputs and translates a memory template into a configured memory model by a process of variable substitution and for loop expansion. A single template is used to generate a bit or byte or word write configuration of the memory. The number of write enable and output enable pins are decided by the number of bits per byte. Each of these individual write enable pins then control the read and write operations for the corresponding byte. So, in case of a bit write memory model, the data output can reflect the result of a read and write to various bits of the same word at the same time.

2.5 Timing Considerations for Memories

Memory functionality depends on satisfying timing requirements. Any timing violation causes the output and/or the memory location to be corrupted/cleared with unknown values (that is, Xs). A subsequent valid operation causes the output to come out of the X state. The actions taken for different timing violations at different inputs and the state of the memory when some of the inputs are unknown is discussed here.

- If the data is X and the address and control inputs are valid, then the data is written into the addressed memory location. Data setup/hold violations cause an X to be written to the addressed memory location on a bit-by-bit basis. That is, only the affected bits cause the corresponding memory bit to be corrupted.
- If the address is unknown during a read operation, then the outputs go to the X state. If the address is unknown during a write operation, then the entire memory contents are corrupted (Xs written to the memory locations). In a byte-write configuration, only the appropriate byte gets corrupted in the memory. Address setup/hold violations corrupt the entire memory. A change in address during a write operation results in the entire memory getting cleared (to Xs).
- If the write enable is unknown, then both the data outputs and the currently addressed memory location are corrupted. The memory location is corrupted only if the data inputs are different from the contents of the memory location. Minimum pulse width violations on write enable causes X to be written on to the currently addressed memory location (the specific byte in a byte-write configuration). The memory location is corrupted independent of the state of the data inputs.
- If the chip-select signal is unknown, then data outputs go to an X state. During a write operation, an unknown chip-select signal causes the data word to be corrupted. Chip-select setup/hold violations cause X to be written to the currently addressed memory location during a write operation.
- If the memory enable signal changes state when clock signal is high (in cases of synchronous RAMs) and in cases of setup/hold violations of memory enable with respect to clock, the entire memory is cleared (to Xs). On a rising edge of the clock signal, if the memory enable signal is an X, the entire memory is cleared.

3. VHDL Implementation

The VHDL implementation of memories is based on the functional/timing aspects discussed earlier. In

addition, a modular approach, using user defined procedures/functions is followed to model the memories. These procedures are generic and are defined in a centralized VHDL package, so that they can be used across all templates consistently. The following sections discuss some of the issues in modeling memories in VHDL, followed by naming conventions, functional/timing implementation, model aesthetics, error reporting, VITAL SDF back-annotation and few generic guidelines to be followed.

The proposed memory implementation has been arrived at, after prototyping few modeling solutions for efficiency and ease of development. In order to ensure that models are developed consistently and in a way that allows them to be easily maintained, certain key objectives are set. They include complete modularity, code reuse, VHDL specific efficiency considerations, use of a generic strategy for all memory architectures and principles from the existing sign-off quality Verilog memory models.

4. Issues in Modeling Memories in VHDL

Modeling memory devices in VHDL has many efficiency considerations and is highly dependent on the type of memories being modeled. In order to develop an optimized implementation, it is necessary to look at all possible implementations and their corresponding trade-offs. Memory models are behavioral in nature and it is extremely difficult to use structural primitives exclusively, to define the functionality. Additionally, the functionality is typically delay dependent and the signal updates constitute waveform editing. This often precludes the use of a pin-to-pin delay model for memories. There are also requirements related to timing checks and X-handling. The next section summarizes some of the issues and findings as a result of prototyping a 2 PORT (1 READ, 1 WRITE) memory architecture in VHDL. The two main issues to consider are the representation of memories in VHDL and use of single versus multiple processes to model the memory. Following sections briefly discuss the trade-offs in each of these issues and then documents the suggested approach which meets the global objectives described in the previous section.

4.1 Representing Memories in VHDL (Signal Array Vs. Variable Array)

According to VHDL-87, the usage of variables is limited to the process in which they are declared. Global variables are not permitted in 87 version of VHDL language, however they are allowed in the 93 revision of VHDL, which the current VITAL standard does not

support. Using shared variables will make the model non-VITAL compliant.

In Verilog, memories are represented as global "variables" or a collection of "register" elements which are accessible globally. All procedural blocks (always and initial blocks) have access to this variable. It is thus convenient to have different blocks reading and writing on to this global entity. In VHDL, the only way to have a similar access is to declare the memory as a signal array. This however, has some serious implications.

Signal updates in VHDL happen in a way that is different from variable updates, and in fact it is different from the Verilog register update as well. This semantic difference prevents a one-to-one translation of a model or the modeling strategy from Verilog to VHDL. Signals in VHDL are typically more expensive to represent and operate than variables. In general, it is recommended to use variables instead of signals.

Using signal arrays as the means of propagating values is inefficient and has a significant overhead in terms of scheduling and de-scheduling of events. This is a critical factor, if higher memory densities are considered. Using signal objects for representing memories would require the use of resolved signals, if multiple processes were writing to memory. VHDL assumes that each signal being written to, has a driver of its own within that process. In light of the above facts we adopted a strategy of using a variable array for declaring memories.

4.2 Single Process Vs. Multiple Process Model

In order to use a variable array for memories, one option is to have the memory modeled completely within a single monolithic process. This is termed as a "**Single Process Model**". Another alternative is to have a separate memory access process and all accesses to the memory (declared as a variable array within this process) are made by passing the required parameters to this process. The typical parameters to be passed are, the address to which data is to be written or read, the operation function of the memory, and a bi-directional data bus for data to go in or come out. There will be other concurrent process blocks which need to synchronize the various events at the inputs and trigger a memory access. This is termed as a "**Multiple Process Model**". There are numerous pros and cons of using either process model. The following paragraphs briefly describe these issues.

4.2.1 Multiple Process Model

A multi process model exhibits true concurrency. For example consider a multi-port RAM in which all input signals do not affect all outputs. The use of a multiple process model is efficient because it will ensure that only the minimum of computation is done (for the relevant part of the RAM) whenever an input changes. One of the main drawbacks of this model is that it is very difficult to specify the order of events to be executed. Due to the semantics of VHDL language and definition of how the simulation cycle proceeds, it is nearly impossible to have a definite order of evaluation without causing synchronization problems. The alternative to this would be to introduce explicit delta cycles within the code. It is observed that the introduction of delta cycles is of the order N , where N is the number of processes to achieve the desired sequencing.

In addition to this synchronization issue, there is a problem in handling global communication signals. This issue relates to inter-process communication. In case of a single memory access process, the only way to pass the required information is through global signals. If multiple processes write to these signals, then it is a case of driver conflict. Defining resolution functions for these signals is highly complex and is expensive during simulation. These resolution functions should return the latest driven value which is difficult to compute.

4.2.2 Single Process Model

In a single process model the flow is completely sequential. It seems to have the desired concurrency when the execution of all blocks happen in the same delta cycle. Also, there is a logical order of execution of the various blocks which is enforced. Considering all our requirements, a single unified process for the entire memory has been found to be a better solution than a multiple process model.

4.3 Access Types and Dynamic Allocation

Another one of our requirements was to have a centralized VHDL package which consists of generic reusable code. This package contains the procedures/functions needed for generic read, write and corruption operations on the memory. All of these procedures are related to memory access and hence the memory element needs to be passed to them as a parameter. This could become a serious problem if we statically allocate a two-dimensional variable array. In each call to a procedure involving memory access, we would need to pass the complete array as a parameter, which is a sig-

nificant overhead in terms of evaluation. The alternative is to pass the memory as a reference pointer. In order to pass memory to the various read/write procedures and other functions, VHDL access types have been used. A memory type which is a set of access types pointing to a two dimensional array has been defined. Dynamic allocation is used to allocate the required number of words and bits depending on the desired configuration.

5. Functional Implementation

The functional implementation of the memory could be classified into several sections, such as the declaration section, logic section, timing check violation handling section etc. Each of these sections are described as follows:

The **header declaration** consists of header comments and library/use clause definitions. The header comments includes a descriptive comment about the technology, the design library, memory architecture, configuration, notes etc. The library and use clause definitions are as follows:

```
LIBRARY STD;
LIBRARY IEEE;

USE STD.TEXTIO.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.Vital_Primitives.ALL;
USE IEEE.Vital_Timing.ALL;
```

The **interface declaration** consists of an entity definition specifying input/output ports and timing generics to hold the back-annotated delay information. All ports are scalars and their type is std_ulogic. The timing generics include tpd_*, tpd_* and timing check generics. The naming convention for these generics are described in a later section. In addition, there are some control generics defined to control the usage of the model. These are defined as follows:

```
MsgOn      : BOOLEAN := DefTimingMsgOn;
XOn        : BOOLEAN := DefTimingXOn;
TimingChecksOn : BOOLEAN := DefTimingChecksOn;

MemLoadFileName : STRING := "<mem_data_file>";
MemDumpFileName : STRING := "<mem_dump_file>";
MemLoadFileFlag : BOOLEAN := DefMemLoadFileFlag;
MemDumpFileFlag : BOOLEAN := DefMemDumpFileFlag;
```

In addition to these, the entity declaration contains the VITAL Level0 compliance attribute. This attribute is set to TRUE in order for the simulation tool to perform VITAL compliance checks.

```
ATTRIBUTE Vital_level0 OF RAM: ENTITY IS TRUE;
```

The **architecture declaration** section consists of signal declarations for all internal buses. The data flow inside the memory model is accomplished using vector buses for data, address and control inputs. In addition, alias definitions are used wherever meaningful, for better readability of the model. A sample declaration is as shown below:

```
signal oea_s      : std_logic_vector (1 downto 0);
signal badr_s     : std_logic_vector (2 downto 0);

alias dib_byte0_a : std_logic_vector(1 downto 0) is dib_s(1 downto 0);
alias dib_byte1_a : std_logic_vector(1 downto 0) is dib_s(3 downto 2);
```

5.1 Memory Type Declarations

The memory is defined as a two dimensional unconstrained array. First, an unbounded array of UX01 is defined as a MemoryWordType. This forms a single memory word. A pointer to the word type is also defined. The MemoryArrayType is an unbounded array natural range of such pointers. This represents the number of words in the memory. Then a record is defined which holds the pointer to the memory array and dimension of the memory. A pointer to this record defines the actual memory type. These pointers are allocated using dynamic allocation. This is done using a call to PF_Declare_Memory () which returns a pointer to the final memory array. This pointer is stored in a variable and used throughout the model. The following type declarations are used in the package:

```
TYPE MemoryWordType IS ARRAY (NATURAL RANGE <>)
OF UX01;
TYPE MemoryWordPtrType IS ACCESS MemoryWordType;

TYPE MemoryWordRecType IS
RECORD
    MemoryWordPtr : MemoryWordPtrType;
END RECORD;

TYPE MemoryArrayType IS ARRAY (NATURAL RANGE <>)
of MemoryWordRecType;
TYPE MemoryArrayPtrType IS ACCESS MemoryArrayType;

TYPE MemoryArrayRecType IS
RECORD
    NoOfBits      : POSITIVE;
    NoOfWords     : POSITIVE;
    MemoryArrayPtr : MemoryArrayPtrType;
END RECORD;

TYPE MemoryType IS ACCESS MemoryArrayRecType;
```

Within the call to the PF_Declare_Memory () function, a memory array pointer is defined by allocating a MemoryArrayRecType using the 'new' construct in

VHDL. Then the number of words in a memory is defined by allocating space for the MemoryArrayType equal to the memory length. Then, for each word in the memory array, a MemoryWordType is allocated depending on the number of bits in a word.

5.2 Memory Declaration Section

The memory array is declared by giving a call to the function PF_Declare_Memory() which returns a pointer to the memory and this is stored in a variable called memoryArray. A data type called memoryType is defined in the package. In short, a two dimensional unconstrained array is defined and then a set of records containing pointer to the array type have been defined. Dynamic allocation is used to allocate the required number of words and bits depending on the desired configuration. Sample code for memory declaration is shown below:

```
MEMORY_PROCESS : process
variable memoryArray      : memoryType;
variable Address          : NATURAL;
variable F_AddressUnknown : BOOLEAN := FALSE;
-- Timing violation flags and variables
.....
begin
    memoryArray := PF_Declare_Memory (
        NoOfBits => 4,
        NoOfWords => 8
    );
```

5.3 Logic Section

The logic section consists of an output enable block and a single monolithic process in which the complete memory functionality/timing (except wire delay) are defined. In the declarative section of the process there are variable and alias declarations for the memory. Inside this process, all the memory functions and timing check violation handling are carried out. Memory declaration precedes any statement in this process. After the memory declaration, an infinite loop is introduced with a sensitivity list of input signals. An explicit "wait on" statement is used to define the sensitivity list of all input signals. This is needed because the memoryArray pointer variable (returned after a call to PF_Declare_Memory) is supposed to exist for the complete simulation. Consequently, all functionality like load operations can be performed within this region after the memory initialization is done.

The first section after the wait statement are the calls to VITAL timing check procedures to perform the constraint checking. The section on **Timing Implementation** describes more about the usage of these procedures. Any memory access for read or write is

determined by checking the S'EVENT attribute and the active level of the appropriate input signal. Separate logical blocks (**not VHDL block statement**) are created within the same process for read and write operations. For a read operation, read blocks are created for every address port. The data appears at the output, if the output enable buffers are active. The output enable block is implemented using VitalBUFIF primitives. The enable and disable delays are computed using the tpd_* generics.

The read block performs an address-initiated read operation, provided the address bits are not unknowns. If the address bits are unknowns, then the output is made X. In order to perform the read operation, the PF_Read_Memory () procedure is called. As a result of any memory access, the output data bus is scheduled to obtain a data value at a future time. If a subsequent memory access is triggered before the previous one is completed, the new transaction scheduled for the output data bus cancels all pending transactions and the previous memory access is considered invalid. This is due to the transport delay model used for scheduling the outputs.

A write operation is initiated by the active edge of the write enable signal. A separate logical block is created for each write enable signal in a byte-write configuration so that appropriate bytes are written. The write operation is performed using a call to PF_Write_Memory() procedure. The scheduling is done by giving a call to the PF_Schedule_Output() procedure. In a dual port (1 read, 1 write) RAM, this scheduling is done only if the read and write ports address the same location. A write operation can also be initiated by data input changes when write enable is active (like a transparent latch); the data in the memory is also reflected at the output after the Twdd time transpires. Sample code of a memory ready cycle follows:

```
loop
    wait on WEN_i, DIN_i, AADR_i;
    -- Read Operation For AADR PORT
    if ( AADR_i'EVENT and now /= 0 ns )
then
    PF_Address_To_Int (AADR_i,
Address, F_AddressUnknown);
    if ( F_AddressUnknown = TRUE ) then
        PF_Schedule_X (DOA_i);
    else
        PF_Read_Memory (memoryArray,
DOA_i_var, Address, 9, 0);
        PF_Schedule_Output (DOA_i,
DOA_i_var, tpd_AADR0_DOA0_posedge);
    end if;
    end if;
end loop;
```

5.4 Timing Violation Handling Section

The functional portion is followed by the timing check violation handling block. In case of a timing violation, the appropriate violation flag is set by the VITAL timing check procedures. A check is made to see if these flags are set and then appropriate action is taken to corrupt the memory based on the X-handling rules specified earlier. The PF_Corrupt_* routines are used to corrupt the contents of the memory. The PF_Schedule_X routine is used to schedule an "X" at the output immediately after the timing violation. If there are no timing violations, the final data that was updated in the memory gets latched.

6. Timing Implementation

VITAL Wire delay blocks are used for modeling wire delays. The delayed signals are used in the functional and timing check portions of the model. The tpd_* generics of the type VitalDelayType01 are used to back-annotate wire delay information.

An intrinsic delay value pair represents a combination of the rise and fall values of the hold time (like Toh) and access time (like Taa). All the intrinsic pair delay values are modeled using a transport delay model. The delay scheduling is done using a PF_Schedule_Output (). The tpd_* generics hold the intrinsic delay value pairs. This is passed as a parameter to the PF_Schedule_Output () procedure. This procedure is used to schedule a data value onto the output data bus which is also passed in as a parameter to the procedure. It calculates the max time from the timing generic passed and then schedules the X value after OutputHoldTime and the valid value after the DataAccessTime. The following VHDL code below illustrates this implementation:

```
PROCEDURE PF_Schedule_Output (
    SIGNAL DataOut: OUT std_logic_vector;
    CONSTANT DataIn: IN std_logic_vector;
    CONSTANT IntrPairDelay: IN VitalDelayType01Z
) IS
    VARIABLE OutputHoldTime : VitalDelayType := 0 ns;
    VARIABLE DataAccessTime: VitalDelayType := 0 ns;
BEGIN
    OutputHoldTime := Max_Time(IntrPairDelay(tr0Z)
        , IntrPairDelay(tr1Z));
    DataAccessTime := Max_Time(IntrPairDelay(tr20)
        , IntrPairDelay(trZ1));
    DataOut <= transport (others => 'X') after
        OutputHoldTime, DataIn after DataAccessTime;
END PF_Schedule_Output;
```

All the timing checks are implemented using VITAL timing check procedures. The timing checks are

conditioned to reduce pessimism. The violation flags are declared as variables and passed to the timing check procedures. The timing checks are performed with delayed input signals which occur after the wire delay block. Following illustrates a typical call to the VITAL setup/hold check procedure:

```
VitalSetupHoldCheck (
    TestSignal => badr_i(1),
    TestSignalName => "BADR1",
    RefSignal => we_i(0),
    RefSignalName => "WE0",
    setupHigh =>
        tsetup_BADR0_WE0_posedge_posedge,
    setupLow =>
        tsetup_BADR0_WE0_negedge_posedge,
    RefTransition => 'R',
    TimingData => marker1,
    MsgOn => MsgOn,
    XOn => XOn,
    Violation => Viol_BADR_WE0
);
```

7. RAM Back Annotation using SDF

The timing information in the RAM model is introduced by using the VITAL-SDF back-annotation scheme. All the memory models are VITAL-Level0 compliant and hence follow the VITAL naming conventions for naming timing generics. A VITAL Level-0 compliant annotation tool should be used to back-annotate the SDF files. Intrinsic delay pairs for all timing arcs in memories are back-annotated using SDF IOPATH statements. In memories, there are multiple output transitions corresponding to a single input transition. For example, an address transition in an ADDRESS to DATAOUT path will cause two serial events to occur on the data output. First the data becomes invalid (DATAOUT goes to X state) and then the data becomes valid (DATAOUT goes from X to 1/0 state). The above is true for all propagation delay paths in memories. Following is the list of typical timing arcs for an asynchronous memory:

(i)	ADDRESS	=>	DATA OUTPUT
(ii)	WRITE ENABLE	=>	DATA OUTPUT
(iii)	DATA IN	=>	DATA OUTPUT
(iv)	OUTPUT ENABLE	=>	DATA OUTPUT (Disable and Enable Arcs)

In order to model the S-to-X and X-to-S transitions (where S is either a 0 or a 1) the place holders for tri-state delays in an SDF IOPATH statement are used. The following table shows the mapping of 'Z' delays to 'X' delays.

Table 1: Mapping of 'Z' to 'X' Delays

'Z' Delays	'X' Delays
0->1	0->1
1->0	1->0
0->Z	0->X
Z->1	X->1
1->Z	1->X
Z->0	X->0

The SDF IOPATH statements are used for specifying the memory intrinsic propagation delays. The generic type to be used for all intrinsic pair values is restricted to VitalDelayType01Z. Input edge specifiers are required in generic names. Given below is the SDF construct and the corresponding generics in the VITAL memory model for the above timing arc specifications.

```
SDF CONSTRUCT: IOPATH <edge> <InputPort>
                <OutputPort> <value>
GENERIC NAME:   tpd_<InputPort>_
                <OutputPort>_<edge>
GENERIC TYPE:  VitalDelayType01Z
```

7.1 Intrinsic and Load Dependant Delays

Considering the number of bit by bit delay paths from address to data out bus, having IOPATH statements corresponding to each path will result in huge SDF files for reasonably large number of memory instantiations in the design. In addition, all the generics derived out of these IOPATH statements have to be collapsed into a single generic to be used inside behavioral memory descriptions. The intrinsic delays are load-independent, thus the delay values are the same for all timing arcs of the same nature. The load-dependent delays are lumped into the output port and are back-annotated using the DEVICE delay construct in SDF onto the 'tdevice_' generics in the model. Hence in the SDF file, the delay calculator writes out only one path delay statement, corresponding to the path from the first bit of the input pin to the first bit of the output pin. The memory model is coded in such a way that this path delay will be used for all bits of the input pin to all bits of the output pin. An example follows:

```
SDF CONSTRUCT: (INSTANCE
                <cell_instance>.<instance>)
( DELAY (ABSOLUTE ( DEVICE <OutputPort>
                  <values>)))
GENERIC NAME:
                tdevice_<instance>_<OutputPort>
```

```
GENERIC TYPE: VitalDelayType01Z
EXAMPLE: ( IOPATH (posedge ADR0) DO0
          () (1:2:3) (4:5:6) (7:8:9) (10:11:12))
          ( INSTANCE cell11.DOA0INST )
            ( DEVICE (1:2:3) (4:5:6))
```

```
tpd_ADR0_DO0_posedge : VitalDelayType01Z;
tdevice_DO0inst : VitalDelayType01;
```

The instance on whose output the device delay is specified must be a label of a concurrent procedure call to one of the VitalPrimitives. This is unlike other occurrences in SDF, where the instance name is the label of the component instantiation statement. The naming convention for the instance name should follow the rule <OutputPortName><Bit0>INST.

8. LSI Memory Package

The LSI Memory Package has been developed with the objective of code reuse across all VHDL memory templates. This package contains the memory type declarations which define a memory data type and procedures/functions which operate on the memory. The package also consists of some utility functions like calculating the maximum time from two time values and procedures for displaying messages. The usage of these procedures will guarantee consistency across all memory models. All the functions involving memory access take the pointer to the memory as a parameter. These procedures should be called after a call to the PF_Declare_Memory procedure which returns the memoryId to be used for future references. The utility procedures like PF_Memory_Print () is used to display messages from the memory model consistently. Following is a description of some functions and procedures available in the LSI Memory Package:

```
1. Function: FUNCTION PF_Declare_Memory(
                CONSTANT NoOfBits :IN POSITIVE;
                CONSTANT NoOfWords:IN POSITIVE;
                )
                return MemoryType;
```

Synopsis: Procedure used for declaring a two dimensional memory array dynamically, depending on NoOfBits and NoOfWords. It returns the pointer to the allocated memory array.

```
2. Procedure : PROCEDURE PF_Address_To_Int(
                VARIABLE IntAddress : OUT NATURAL;
                VARIABLE UnknownFlag : OUT BOOLEAN
                CONSTANT Address :IN std_logic_vector;
                );
```

Synopsis: Procedure used for transforming a valid address value to an integer. The presence of "X" values on the address bits sets the UnknownFlag.

```

3. Procedure: PROCEDURE PF_Write_Memory(
    VARIABLE MemoryPtr : INOUT MemoryType;
    CONSTANT DataIn : IN std_logic_vector;
    CONSTANT Address : IN NATURAL;
    CONSTANT HighBit : IN NATURAL;
    CONSTANT LowBit : IN NATURAL
);

```

Synopsis: Procedure used to write to an addressed memory location based on a bit/byte/word basis. The high bit and low bit offsets are used for byte write operations.

```

4. Procedure : PROCEDURE PF_Read_Memory(
    VARIABLE MemoryPtr : INOUT MemoryType;
    VARIABLE DataOut: OUT std_logic_vector;
    CONSTANT Address : IN NATURAL;
    CONSTANT HighBit : IN NATURAL;
    CONSTANT LowBit : IN NATURAL
);

```

Synopsis: Procedure used to read an addressed memory location based on a bit/byte/word basis. The high bit and low bit offsets are used for byte write operations.

```

5. Procedure: PROCEDURE PF_Load_Memory(
    VARIABLE MemoryPtr : INOUT MemoryType;
    CONSTANT FileName : IN STRING;
    CONSTANT HighBit : IN NATURAL;
    CONSTANT LowBit : IN NATURAL
);

```

Synopsis: Procedure used to load the contents of the memory from the specified memory data file and it is overloaded for byte and word write memory operations.

```

6. Procedure : PROCEDURE PF_Dump_Memory (
    VARIABLE MemoryPtr : INOUT MemoryType;
    CONSTANT FileName : IN STRING;
    CONSTANT AddressBgn: IN NATURAL;
    CONSTANT AddressEnd: IN NATURAL;
    CONSTANT HighBit : IN NATURAL;
    CONSTANT LowBit : IN NATURAL
);

```

Synopsis: Procedure used to dump the contents of the memory to the specified memory data file and it is overloaded for byte and word write memory operations.

```

7. Procedure : PROCEDURE PF_Corrupt_Memory(
    VARIABLE MemoryPtr : INOUT MemoryType;
    CONSTANT BitPosition: IN NATURAL;
    CONSTANT HighBit : IN NATURAL;
    CONSTANT LowBit : IN NATURAL
);

```

Synopsis: Procedure used to corrupt (by writing 'X') all the locations of memory corresponding to the specified bit/byte/word.

```

8. Procedure: PROCEDURE
PF_Corrupt_Memory_Location(
    VARIABLE MemoryPtr : INOUT MemoryType;
    CONSTANT AddrBus : IN std_logic_vector;
    CONSTANT HighBit : IN NATURAL;
    CONSTANT LowBit : IN NATURAL
);

```

Synopsis: Procedure used to corrupt (by writing 'X') a specific location of memory corresponding to the specified byte/word.

```

9. Procedure: PROCEDURE
PF_Corrupt_MemLoc_BasedOn_DataIn(
    VARIABLE MemoryPtr : INOUT MemoryType;
    CONSTANT AddrBus : IN std_logic_vector;
    CONSTANT DataIn : IN std_logic_vector;
    CONSTANT HighBit : IN NATURAL;
    CONSTANT LowBit : IN NATURAL
);

```

Synopsis: Procedure used to corrupt (by writing 'X') a specific location of memory depending on the input data. If the DataIn happens to be the same as the contents of addressed memory location then the memory location is not corrupted.

```

10. Procedure : PROCEDURE PF_Memory_Print(
    CONSTANT Location : IN STRING;
    CONSTANT ErrorId : IN MemoryErrorType;
    CONSTANT Info : IN STRING
);

```

Synopsis: Procedure used to display information messages/errors/warnings from the memory model. It is used in order for the messages to be consistent with the message reporting format used by VITAL procedures.

```

11. Procedure : PROCEDURE PF_Schedule_Output (
    SIGNAL DataOut : OUT std_logic_vector;
    CONSTANT DataIn : IN std_logic_vector;
    CONSTANT IntrPairDelay: IN
        VitalDelayType01z
);

```

Synopsis: Procedure used to schedule a data value onto the output data bus passed in as a parameter to the procedure. It calculates the max time from the timing generic passed and then schedules the X value after OutputHoldTime and the valid value after the DataAccessTime.

```

12. Procedure : PROCEDURE PF_Schedule_X (
    SIGNAL DataOut : OUT std_logic_vector
);

```

Synopsis: Procedure used to schedule explicit "X" values onto the output data bus passed in as a parameter to the procedure. The scheduling of the X value is done after 0 ns.

9. VITAL Memory Testing

Since the Verilog memory models are used for design sign-off, the VITAL memory models are correlated against the verilog models. A verilog testbench written using tasks and functions is used to simulate the required memory configuration using Verilog-XL and the results of the simulation are converted to a generic vector format. This generic vector file is then post-processed and used for simulation of the VITAL memory model. Simulation differences exist due to one or more of the following reasons:

- The delay models in Verilog and VHDL being inertial and transport respectively.
- Simultaneous data and clock changes report only a setup violation in VITAL and hold violation in Verilog.
- In multi-port RAMs with multiple read and write blocks, the read blocks need to precede the write blocks in order to consistently read the correct data. This is due using a single process model (in VHDL memories) where the flow of control is sequential and the desired order of execution needs to be coded.

10. Conclusion

In conclusion, the paper summarizes the approach taken by the authors in modeling LSI Logic's ASIC memories in VHDL, using the capabilities provided by VITAL. Using the techniques described in the paper, the models are developed in a way that promote consistency and maintainability across all memory architectures. The same principles of modularity, and code reuse can be extended to more complex memories like FIFOs, CAMs, DRAMs in the future.

Acknowledgments

The authors would like to thank Raghuram Belur for his invaluable suggestions and constant review of technical content and Rajesh Mehta for his helpful tips during the preparation of the paper.

References

- [1] IEEE Standard VITAL ASIC Modeling Specification, IEEE/ANSI Standard 1074.4
- [2] IEEE Standard VHDL Language Reference Manual, IEEE/ANSI Standard 1076-1987