ECE 571
Project Report

NAND FLASH based Memory Controller

Manjush Padmamma Venkatesha
Lakshmi Manoja Kavuri
Sumeeth Budhi Mahaveer
Yash Hiremath

Table of Contents

Abstract

Flash memory, whether it is in NOR or NAND in structure, is a non-volatile memory that is used to replace traditional EEPROM and hard disks for its low cost and versatility. Because of the difference in the structure of interconnection of the memory cells, NOR Flash is known for its random-access capability, while the NAND Flash is known for its compact size and high speeds for page accesses. This is especially important in applications where the highest-density memory is offered in the smallest footprint.

Our project involves designing a NAND FLASH controller and the NAND FLASH memory itself for verification purposes. One of our main objectives were to explore Systemverilog for verification and to also explore what emulation flow offers.

Verification methodologies used:

- Deterministic approach
- Functional verification
- Emulation

Proposed Objectives

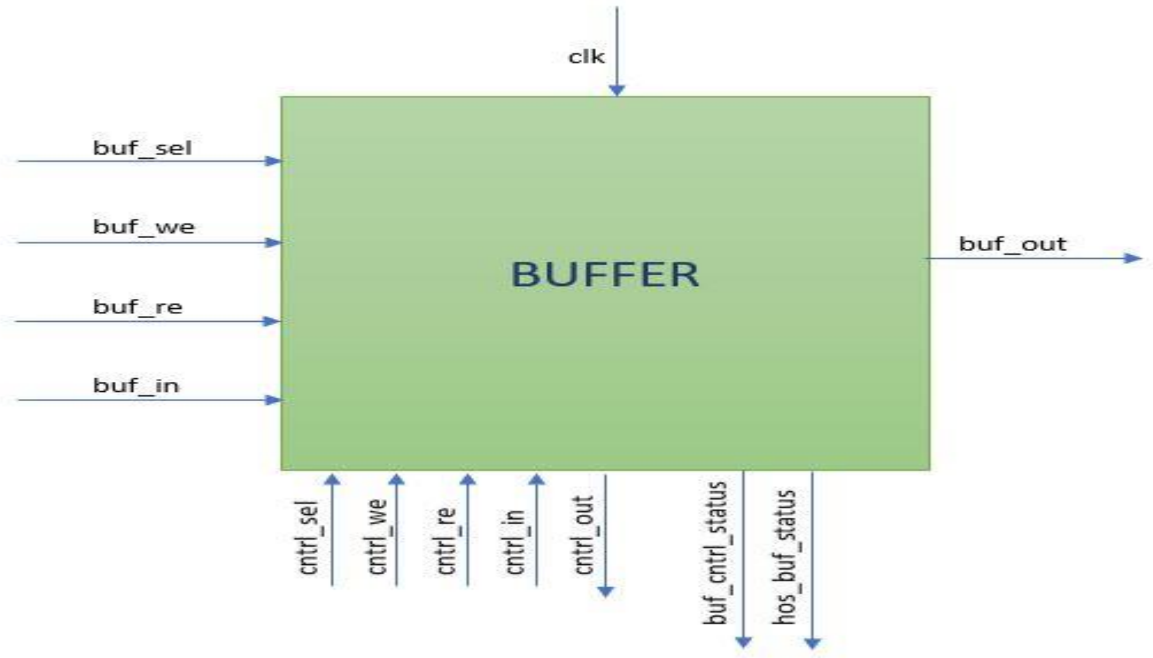
- Design, Verification & Emulation for NAND FLASH based Memory Controller
- Operations to be performed:
    - Reset
    - Program Page
    - Page Read

Additional Improvements

- Memory Design for Memory Controller
- Additional operations:
    - Block Erase
- DC Synthesis
    - Realized netlist
- Verification:
    - Implementation of scoreboard

Functional Description

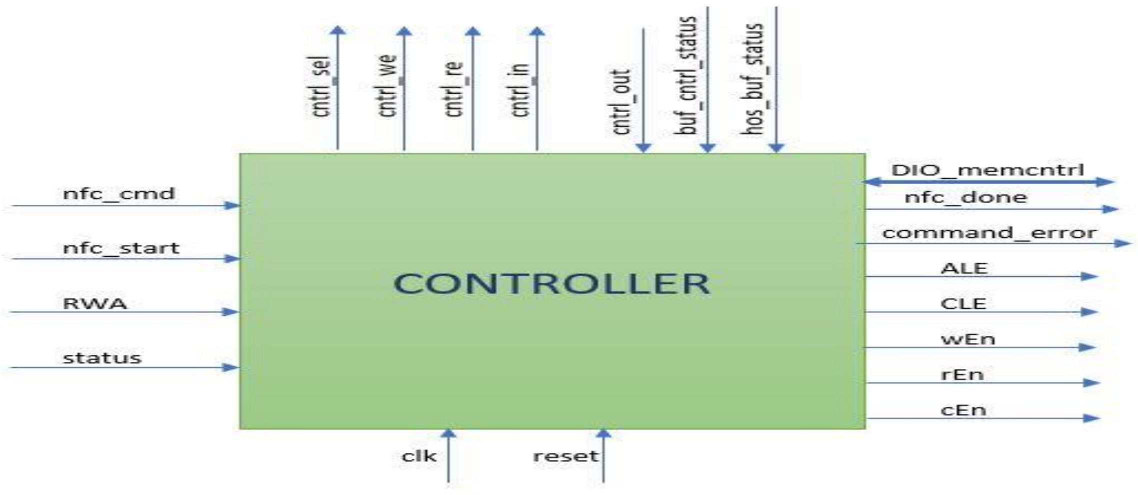
Buffer Design:



Buffer definitions:

Signal Name	Direction w.r.t design	Description
clk	input	Clock signal
buf_in	input	Data into buffer from host
buf_sel	input	To select the buffer
buf_re	input	To read enable the buffer
buf_we	input	To write enable the buffer
cntrl_in	input	To get the data from the controller
cntrl_sel	input	To select the controller
cntrl_re	input	To read enable the controller
cntrl_we	input	To write enable the controller
buf_out	output	To send data to host
cntrl_out	output	To get the data out from the controller
buf_cntrl_status	output	Status flag between the buffer and the controller
hos_buf_status	output	Status flag between host and the buffer

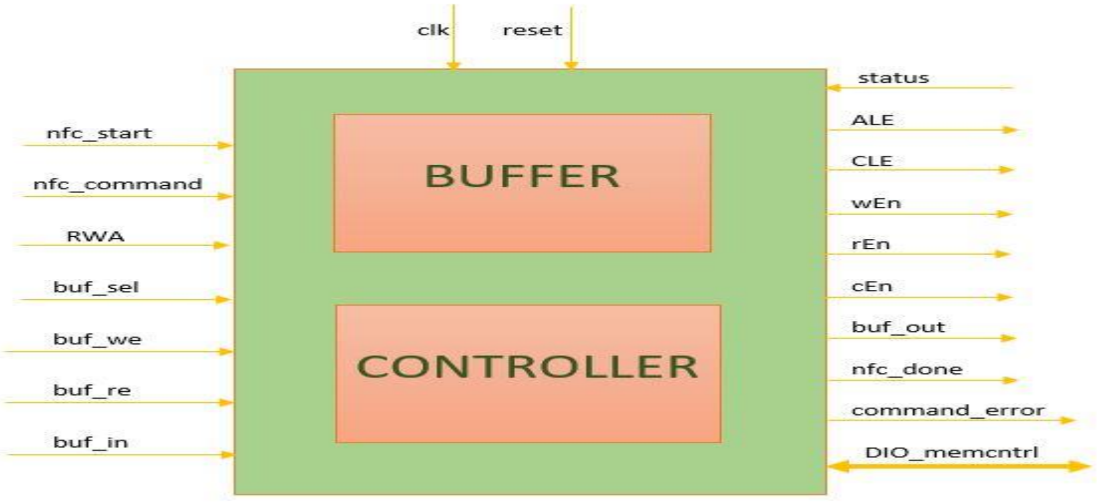
Controller Design:



Controller definitions:

Signal Name	Direction w.r.t design	Description
clk	input	Clock signal
reset	input	Reset signal
nfc_start	input	When asserted, indicates the host initiates an operation.
nfc_cmd	input	Command code signal.
RWA	input	Address signals used by device.
cntrl_in	output	To send the data to the controller
cntrl_sel	output	To select the controller
cntrl_re	output	To enable the controller for read opeartion
cntrl_we	output	To enable the controller for write operation
cntrl_out	input	To get the data from the controller
status	input	To set the status signal if memory is empty and ready for next operation.
buf_cntrl_status	input	Status flag between the buffer and the controller
hos_buf_status	input	Status flag between host and the buffer
ALE	output	Address Latch Enable signal
CLE	output	Command Latch Enable signal
wEn	output	To enable write on memory
rEn	output	To enable read on memory
cEn	output	To enable the chip
nfc_done	output	Signal to ensure the completion of operation
command_error	output	Generates error signal if command is not executed.
DIO_memcntrl	inout	Data transfer between memory and controller

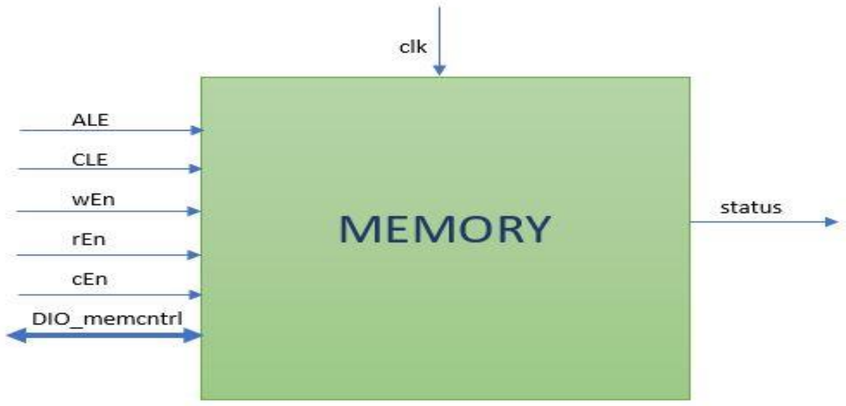
Memory Controller:



Memory Controller definitions:

Signal Name	Direction w.r.t design	Description
clk	input	Clock signal
reset	input	Reset signal
nfc_start	input	When asserted, indicates the host initiates an operation.
nfc_cmd	input	Command code signal
RWA	input	Address signals used by device.
buf_in	input	Data into buffer from host
buf_sel	input	To select the buffer
buf_re	input	To read enable the buffer
buf_we	input	To write enable the buffer
status	input	To set the status signal if memory is empty and ready for next operation.
buf_out	output	To send data to host
ALE	output	Address Latch Enable signal
CLE	output	Command Latch Enable signal
wEn	output	To enable write on memory
rEn	output	To enable read on memory
cEn	output	To enable the chip
nfc_done	output	When asserted, indicates an operation is done.
command_error	output	Generates error signal if command is not executed.
DIO_memcntrl	inout	Data transfer between memory and controller

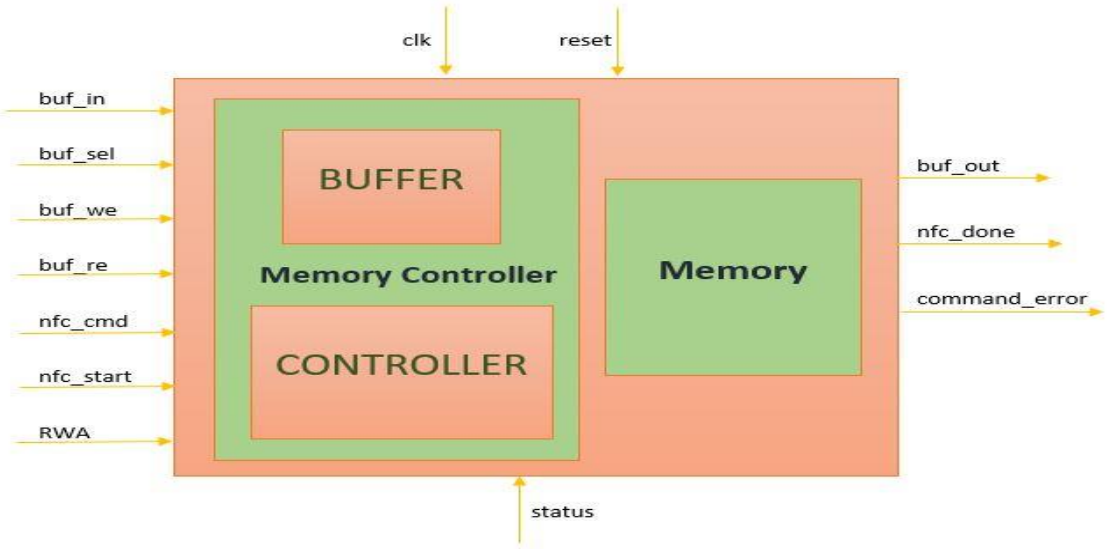
Memory:



Memory definitions:

Signal Name	Direction w.r.t design	Description
clk	input	Clock signal
ALE	input	Address Latch Enable signal
CLE	input	Command Latch Enable signal
wEn	input	To enable write
rEn	input	To enable read
cEn	input	To enable the chip
DIO_memcntrl	inout	Data transfer between memory and controller
status	output	Status signal to set if memory is full

NAND Flash based Controller:



NAND flash-based Controller Top definitions:

Signal Name	Direction w.r.t design	Description
clk	input	Clock signal
reset	input	Reset signal
nfc_start	input	When asserted, indicates the host initiates an operation.
nfc_cmd	input	Command code signal
RWA	input	Address signals used by device.
buf_in	input	Data into buffer from host
buf_sel	input	To select the buffer
buf_re	input	To read enable the buffer
buf_we	input	To write enable the buffer
status	input	To set the status signal if memory is empty and ready for next operation.
buf_out	output	To send data to host
nfc_done	output	When asserted, indicates an operation is done.
command_error	output	Generates error signal if command is not executed.

Design Aspects

- Buffer and Controller are the primary modules of the design.
- Buffer acts as data buffer which writes data to the flash and reads data from the flash.
- Controller is used to control the data transfer from memory to buffer depending on the status obtained from memory.
- Operations on the memory controller:
  - Erase (per block basis)
  - Program Page (copy content of data buffer into Flash memory)
  - Read Page (content of a Flash page is copied into the data buffer)
  - Reset

Block Erase operation : This operations is to erase memory content for a block. A block is defined to be 4 pages. A page is defined to be 2048 words (16 bits). Erasing them would set all the bits to logic ‘0’. The design initiates block erase operation when the host interface signal nfc_cmd is set to 0 (on the DIO bus) and the signal nfc_start is active. The state machine in the controller module switches from the Hold state to the Start state and then the Erase state. Then the address is latched onto and finally the Block erase begins. The state machine then detects the signal status. If the status is set to 1 by the Flash, it indicates the Flash finishes the block erase operation. And then nfc_done is asserted for a clock cycle.

Program Page operation : Writing a page of data. The design initiates page program operation when the host interface signal nfc_cmd is set to 1 and the signal nfc_strt is active. The state machine in the main FSM module switches from Hold state to the Start state. Then the address is latched and the memory write begins. At the same time the host sends relevant signals to the buffer to put the data into the buffer. This will be read later by the controller using buffer interface signals. The state machine then waits on the status signal. If the status is set to 1 by the Flash, it indicates the Flash finishes the page program operation. Then the nfc_done pulse is sent. This completes the page program operation.

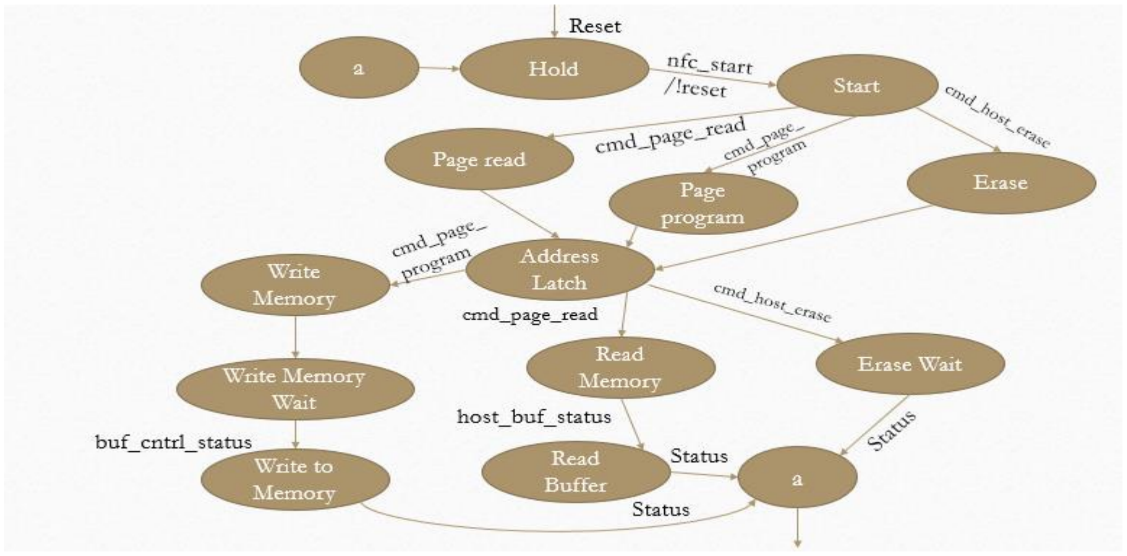
Page Read operation : Reading a page of data The design initiates page program operation when the host interface signal nfc_cmd is set to 2 and the signal nfc_strt is active. The state machine in the main FSM module switches from Hold state to the Start state. Then the address is latched and the memory read begins. At the same time the controller sends relevant signals to the buffer to put the data into the buffer. This will be read later by the host using buffer interface signals. The state machine then waits on the status signal. If the status is set to 1 by the Flash, it indicates the Flash finishes the page program operation. Then the nfc_done pulse is sent. This completes the page program operation.

Reset : Resetting the memory controller so that the pins between the memory controller and the memory are initialized.

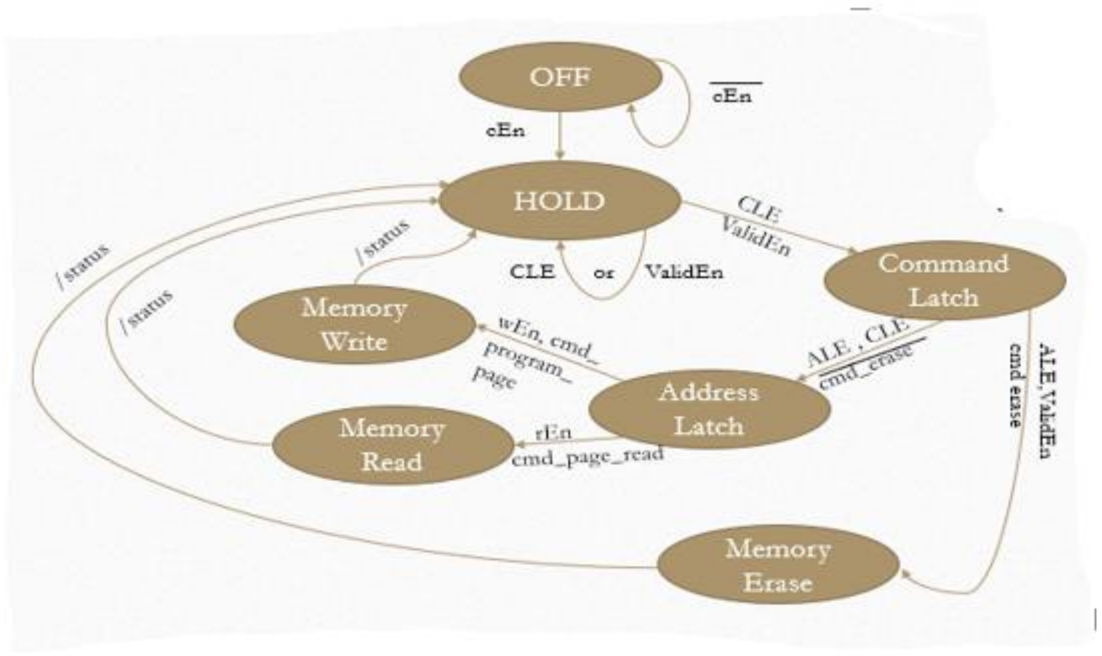
The design consists of the memory controller which includes the buffer and the controller logic itself. Then the DUT for verification is constituted by this memory controller and the memory module. This capacity of this memory is 64K words or 128 KB.

Buffer Design: The buffer depth is chosen to be 2048 words or 4096 bytes. This corresponds to the requirement for page reads or page program operations. As address is not required for the buffer, it is implemented using the counter logic. Also flags are used to impose restrictions on reading after the entire buffer is read, and similarly for writing to the buffer.

Controller FSM:



Memory FSM:



ValidEn & ALE/CLE/rEn/wEn=1
Status is asserted on transition of Memory read, Memory write, Memory erase states

Verification

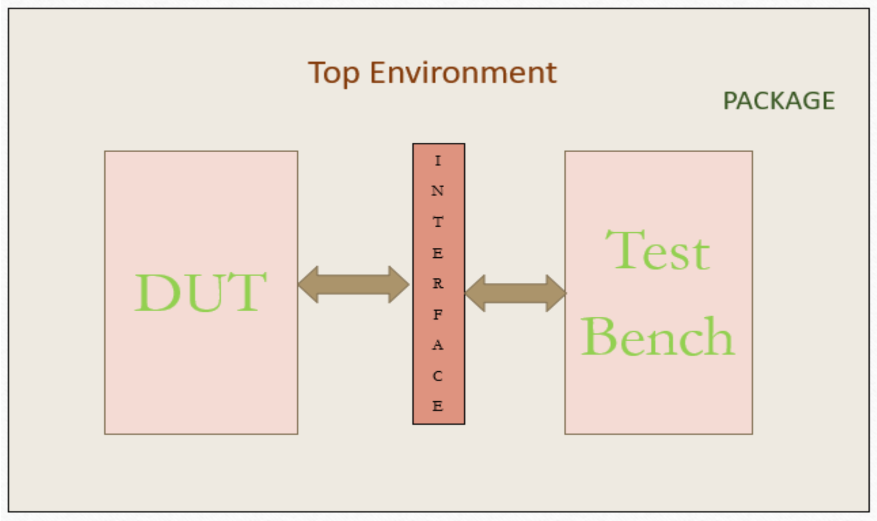
Verification is done in two ways

1)Deterministic Approach.

2)Functional Verification.

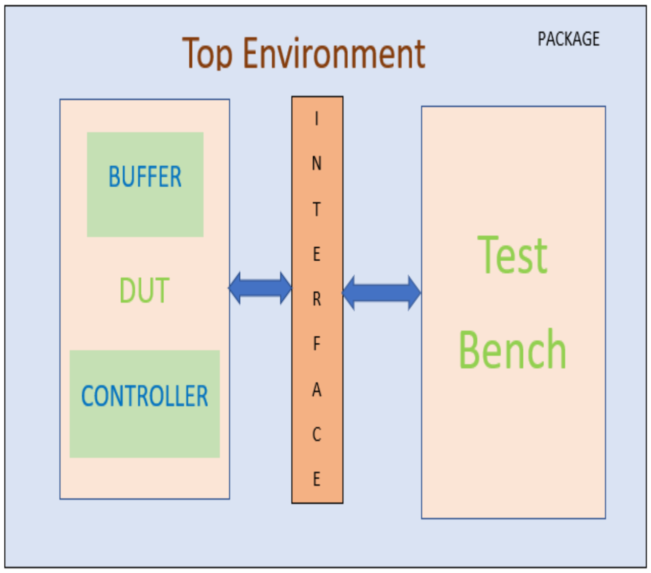
Deterministic Approach:

System Verilog uses the interface construct that has an intelligent bundle of with the connectivity, synchronization, and optionally, the functionality of the communication between two or more blocks. They connect design blocks and/or testbenches. It extends into the two blocks, representing the
drivers and receivers that are functionally part of both the test and the DUT.
The clock can be part of the interface or a separate port.

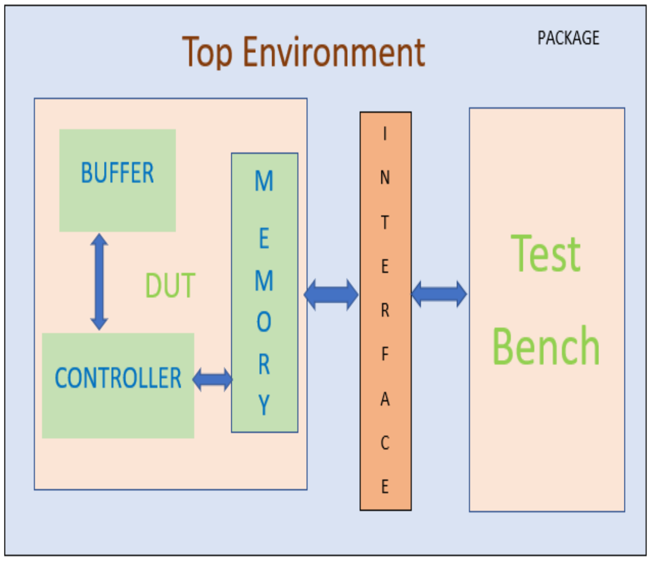


The simplest interface is just a bundle of nondirectional signals. By using logic, driving the signals from procedural statements is easy. Signal in an interface can be referred by making a hierarchical reference using the instance name. Interface signals should always be driven using nonblocking assignments. The modport construct in an interface helps in grouping signals and specify directions. The MONITOR modport allows you to connect a monitor module.

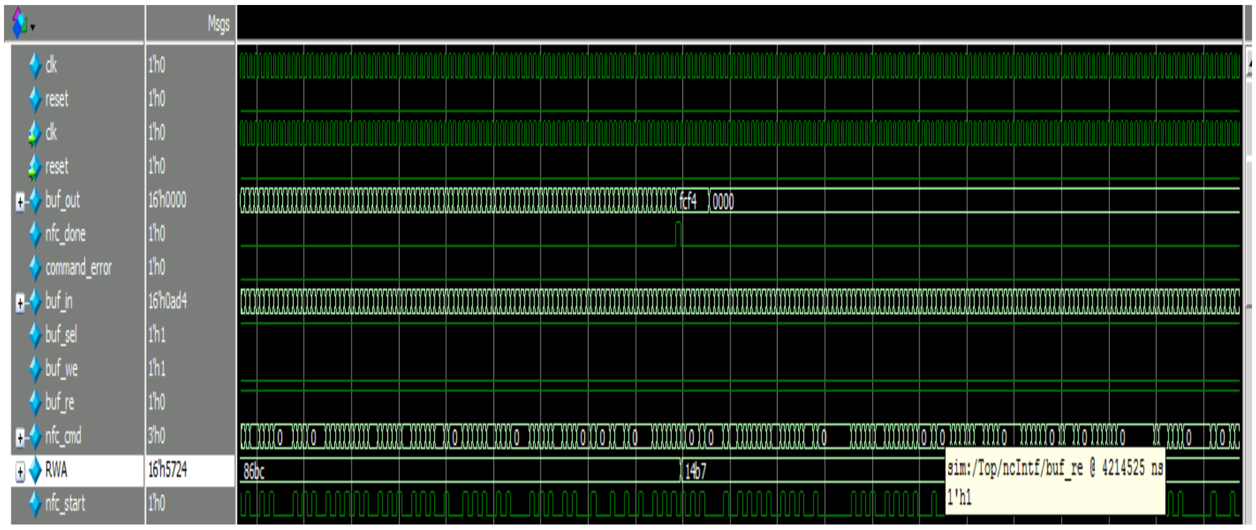
After interfacing the DUT and testbench input test vectors are forced onto the testbench and the output or response is monitored on waveforms and transcript. Now it is checked for correctness and now measured the progress against overall verification goals.



Memory Controller



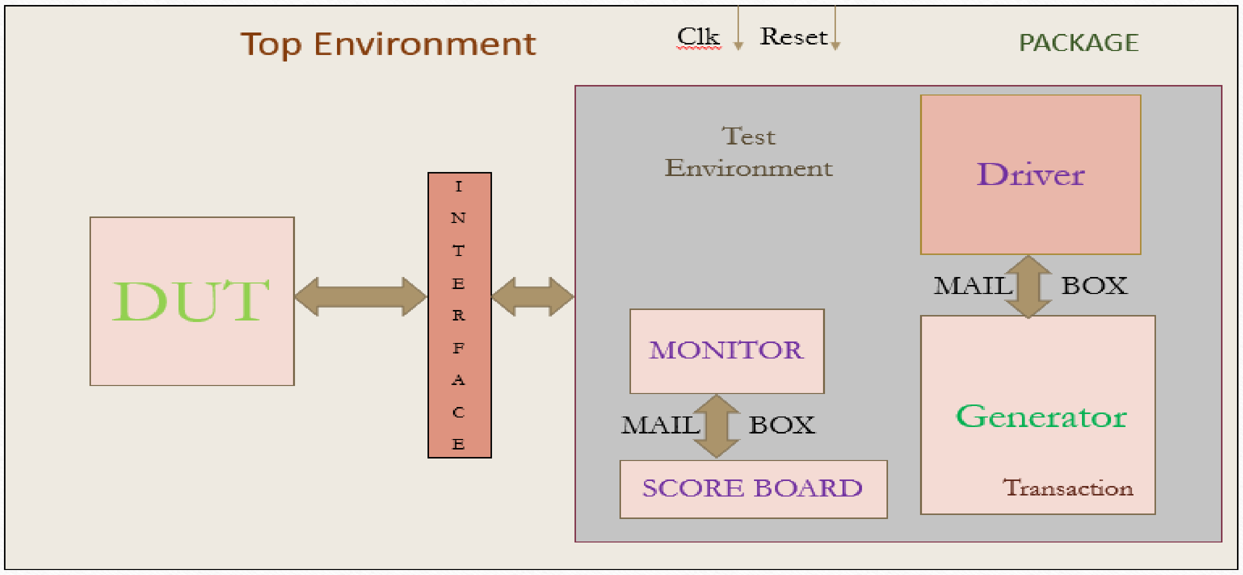
NAND FLASH Controller



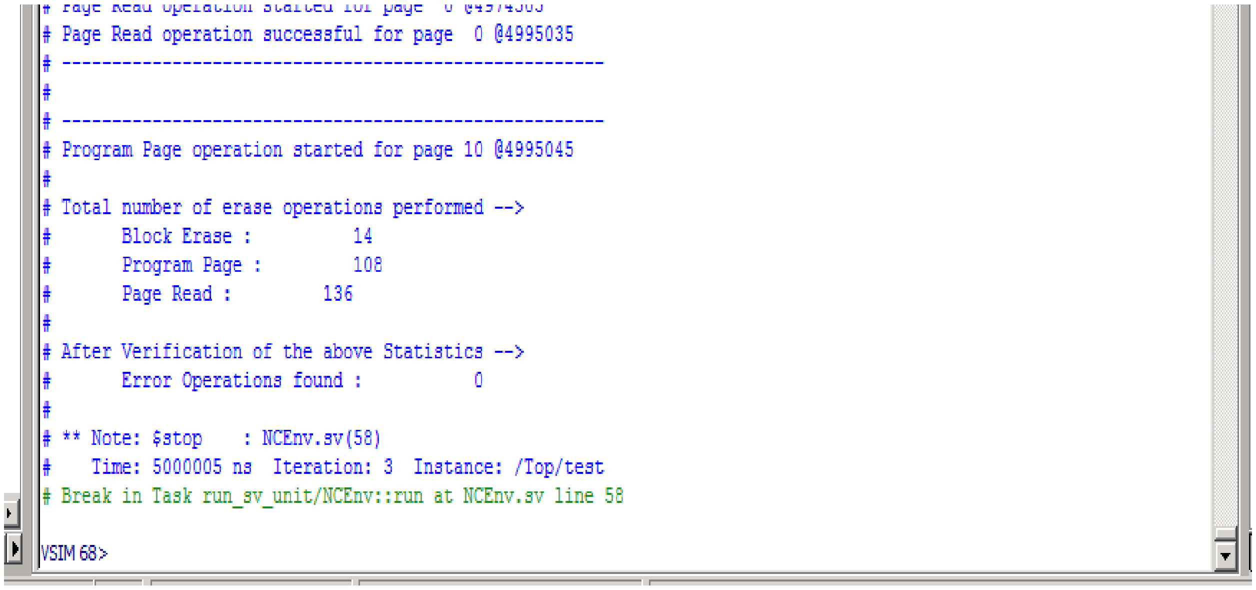
Deterministic Approach Verification Results

Functional Verification:

As it is difficult to create all or maximum test cases to check the functionality, we used constrained random tests (CRT) to automatically create testcases. This helps to find bugs which cannot be found by direct test. The construct used is $random, which is used to generate random input vectors. The randomize function returns 0 if a problem is found with the constraints. The testbench environment has a transactor with simple loop that receives a transaction object from a previous block, makes some transformations, and sends it to the following one, the Generator, that has no upstream block, the transactor constructs and randomizes every transaction, while others, such as the Driver, receive a transaction and send it into the DUT as signal transitions.



Mailbox is used to store data from source and load data to destination. Once data is put in mailbox it blocks till the data is removed. This is instantiated by calling the new function. Put task is used to put data into mailbox and get task is used to remove data from mailbox. Generator and driver exchange the transactions using mailbox, other mailbox is used between scoreboard and monitor. The scoreboard is used to compare output with expected data. The generator, scoreboard, monitor, driver classes are instantiated in environment class. The main verification or test is performed on the top-level program called test.



```
# Page Read operation successful for page  0 @4995035
# -------------------------------------------------------
#
# -------------------------------------------------------
# Program Page operation started for page 10 @4995045
#
# Total number of erase operations performed -->
#       Block Erase :          14
#       Program Page :         108
#       Page Read :         136
#
# After Verification of the above Statistics -->
#       Error Operations found :            0
#
# ** Note: $stop    : NCEnv.sv(58)
#    Time: 5000005 ns  Iteration: 3  Instance: /Top/test
# Break in Task run_sv_unit/NCEnv::run at NCEnv.sv line 58

VSIM 68>
```

Functional Verification Results

Emulation

Why Emulation?

The primary reason for the rapid growth of emulation is increasing design sizes that simulators can't handle.

Emulation done using *Veloce* by Mentor Graphics

Velocesolo at Portland State University:

- Capacity- 8 MG
- Number of users -1
- System memory- 0.5 to 1GB
- Assertion languages- SVA
- Design languages-System Verilog
- Testbench language- Systemverilog

Functional Verification is cheaper than the costs associated with post-silicon bug fixes.
Simulation is a software program that simulates the design and executes HDL code.
Compilation is faster than synthesis. To speedup, Simulators are event driven.

There are two links through which the host server and emulator are connected.
Host Link –This is a High Speed Link (HSL)cable which carries control signals, commands and is also used to download trace/emulation statistics from the emulator. This connection is the primary mode of connecting to the emulator.

CoModelLink –In this CoModel/ CoSimulation mode is used to run testbench in different usage modes.

1. Standalone – In this DUT and Testbench both are synthesized hence are to be synthesizable to run on the emulator. Once emulation begins there is no data transfer between Host and Emulator. Hence only uses Host Link.

   This mode can be achieved by –

   a. DUT + A Synthesizable Testbench.
   b. DUT + Test Cases loaded in Memory.
   c. Only DUT.
      Making synthesizable testbench/testcases is not always possible. Complex test cases are easier to generate through CoModel/CoSimulation.

2. Classical In-Circuit Emulation (ICE) -
An external target system is connected at the pin level to the emulator and is very similar to standalone mode.

3. TestbenchXpress (TBX) Mode Co-Simulation/Co-Modeling- Widely used usage mode
This has features of speedup from Veloce Emulator (parallelism) and testbench from SystemVerilog. Host link is used to connect, configure design and control emulation and

CoModel link is used to transfer data between Testbench(HVL) and HDL in forms of Transaction Level Modeling (TLM). HVL is untimed while executing in zero emulation time.when HVL called emulation clock pauses. HDL is timed part. Have to reduce unnecessary communication between HVL and HDL for best results. Transactor/BFM generates clock, reset and knows protocol.
Transactor = Proxy + Channel + BFM

This mode can be achieved by-

a) DPI-C Calls (CoModel)

b) SCEMI Pipes

c) Bus Functional Model (BFM) Hierarchical Access

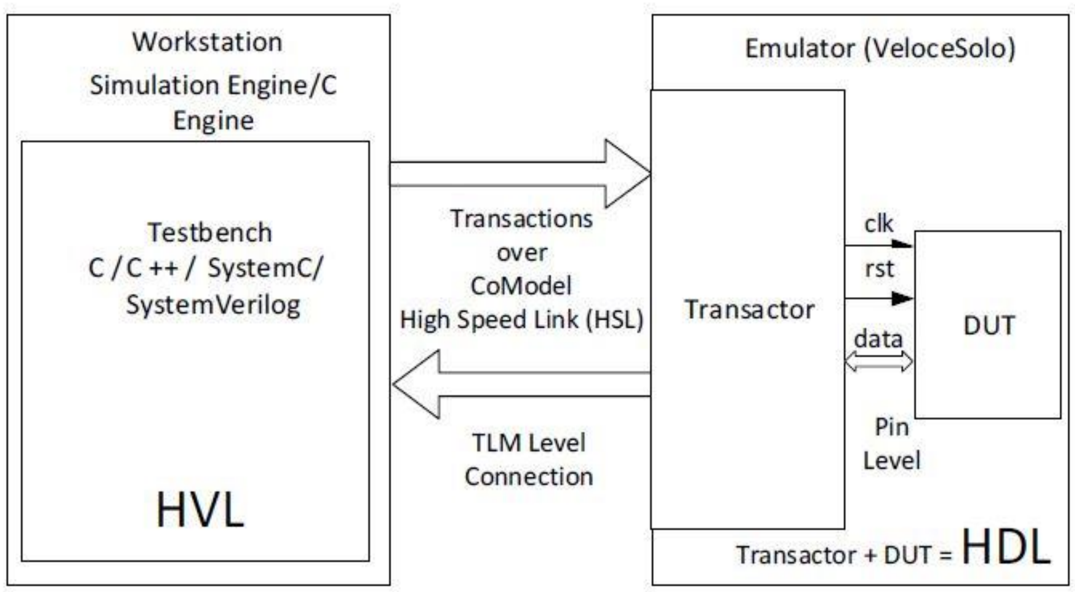
d) BFM Virtual Interface Access

e) UVM/OVM Environment

Advantages:
Interactive control of test environment
Hardware-Software co-verification
Accelerate UVM/OVM methodologies
Co-simulation speed is 2X to 10X
Co-Emulation speed is 50X to 1000X+

Verification environment for emulation in TBX Mode of Flash Memory Controller:

Unit testing for memory and the buffer was done. The block diagram shows the verification environment set up for emulation (BFM Hierarchical Access):



<An image which is a replica of the above image but with module names as per final_test_controller>

XRTL – Extended RTL:

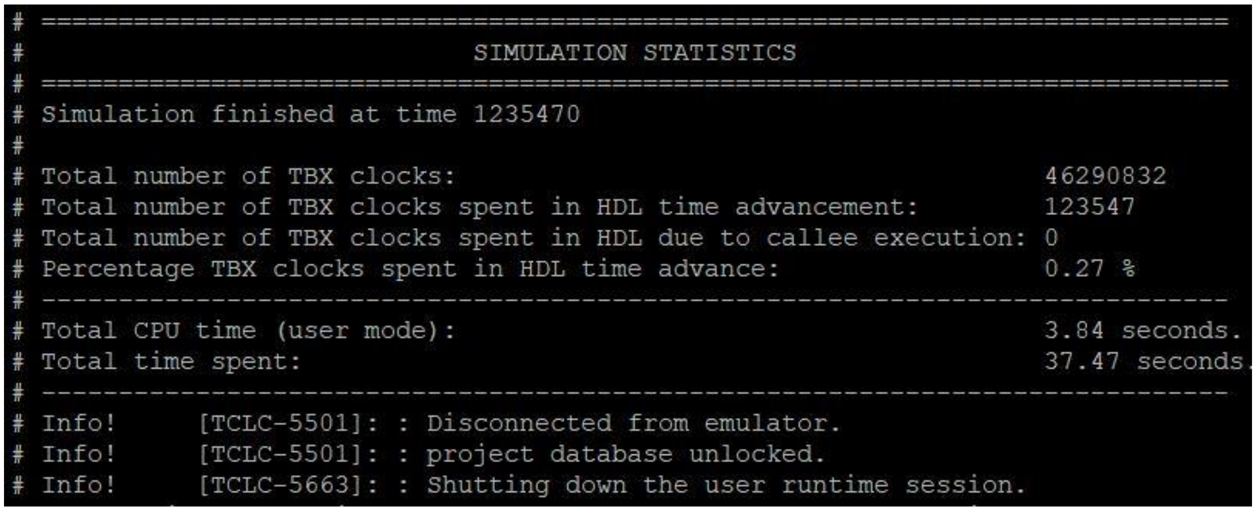
- The interface is declared as Transactor Interface (xif) with //pragma attribute <dut_if> partition_interface_xif
- The interface tasks are declared as Transactor Tasks/Functions (xtf) with //pragma tbx xtf
- Clock and reset generation //tbx clkgen

So the Top HDL and the interface module (Transactor) must be marked XRTL in veloce.config.

Compile Flow:
Create Libraries ---------vellib/vlog
Map Libraries ------------velmap/vmap
Analyze RTL Files--------velanalayze
Compile DUT files--------velcomp
HVL Compile--------------vlog + velhvl
Debug ---------------------velview
This compile flow has been automated using a makefile.

TBX Simulation Statistics:

TBX Clock (Uclock) is the emulation clock and all other clocks are derived from Uclock.
Clock spent in HDL time advancement is design run period.
Remaining TBX clocks is time for Transfer of Data (between HVL and HDL).
TBX clocks spent in HDL due to call are the clocks consumed by export calls.
Percentage HDL clocks spent in HDL time advance portrays how often design clock pauses.



```
# ============================================================================
#                            SIMULATION STATISTICS
# ============================================================================
# Simulation finished at time 1235470
#
# Total number of TBX clocks:                                    46290832
# Total number of TBX clocks spent in HDL time advancement:      123547
# Total number of TBX clocks spent in HDL due to callee execution: 0
# Percentage TBX clocks spent in HDL time advance:               0.27 %
# ----------------------------------------------------------------------------
# Total CPU time (user mode):                                    3.84 seconds.
# Total time spent:                                              37.47 seconds
# ----------------------------------------------------------------------------
# Info!     [TCLC-5501]: : Disconnected from emulator.
# Info!     [TCLC-5501]: : project database unlocked.
# Info!     [TCLC-5663]: : Shutting down the user runtime session.
```

Speed Up achieved : A mere 5% improvement was seen; in comparison to simulation

Challenges Faced

Design:

- Timing control
- Region based event handling
- Designing using synthesizable constructs

Verification:

- Hierarchical test environment creation
- Hierarchical calls using handles
- Concurrent thread execution of test vectors
- Region based event handling
- Designing using synthesizable constructs

Emulation:

- Making the top hvl module untimed. (Implicit FSM approach)
- Complying with XRTL constraints on modeling tasks.

Future Scope

Operations to be implemented

- Read ID
- Random Access
    - Program Page
    - Page Read

Functional Verification

- Reference Model Design
- Assertion based BFM
- Checker
- Functional Coverage

Conclusion

- Systemverilog introduces OOPS into RTL verification. This makes it a very powerful language for verification as well.
- Emulation reduces time taken for verification. It also ensures the use of synthesizable systemverilog constructs. This is particularly helpful, and shows why emulation is becoming a popular choice for verification.

References

- LATTICE Semiconductor NAND Flash Controller reference design manual.
- Data sheet of Samsung K9F1G08R0A NAND Flash.
- Design Compiler user guide.
- Veloce user guide.
- Veloce Languages and Communication Channels User Guide.
- Veloce Reference manual.
- System Verilog for verification, CHRIS SPEAR.