Hes·so VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

FlashMem VIP

Handbook

March 2012
Version 00.00

Oliver A. Gubler
Infotronics Research Unit
Institute of Systems Engineering
HES-SO Valais Wallis
Route du Rawyl 47
1950 Sion, Switzerland

Table of Contents

Illustration Index

Index of Tables

Abbreviations

DUV	Design Under Verification
ISI	Institute of Systems Engineering
ONFi	Open NAND Flash interface
OVM	Open Verification Methodology
SV	SystemVerilog
TB	Test Bench
UVE	Unified Verification Environment
UVM	Universal Verification Methodology
VC	Verification Component
VHDL	VHSIC (Very-High-Speed Integrated Circuit) Hardware Description Language
VIP	Verification Intellectual Property

Datasheet

FlashMem VIP is a Universal Verification Methodology (UVM) verification IP (VIP) that provides an Open NAND Flash interface (ONFi) compatible interface and should simulates the behavioral of a memory chip. The interface consists of the signals shown on the right side of Illustration 1. Please note that only the data port IO0 for single data rate mode is implemented. The data ports DQ0 and DQS0 are not used yet.

FlashMem VIP is able to **collect write accesses**, **prints** them **out** and **sends** them on to the rest of the TB as **transfers** with the elements shown on the left side of Illustration 1. On **read accesses** it delivers **random data**.

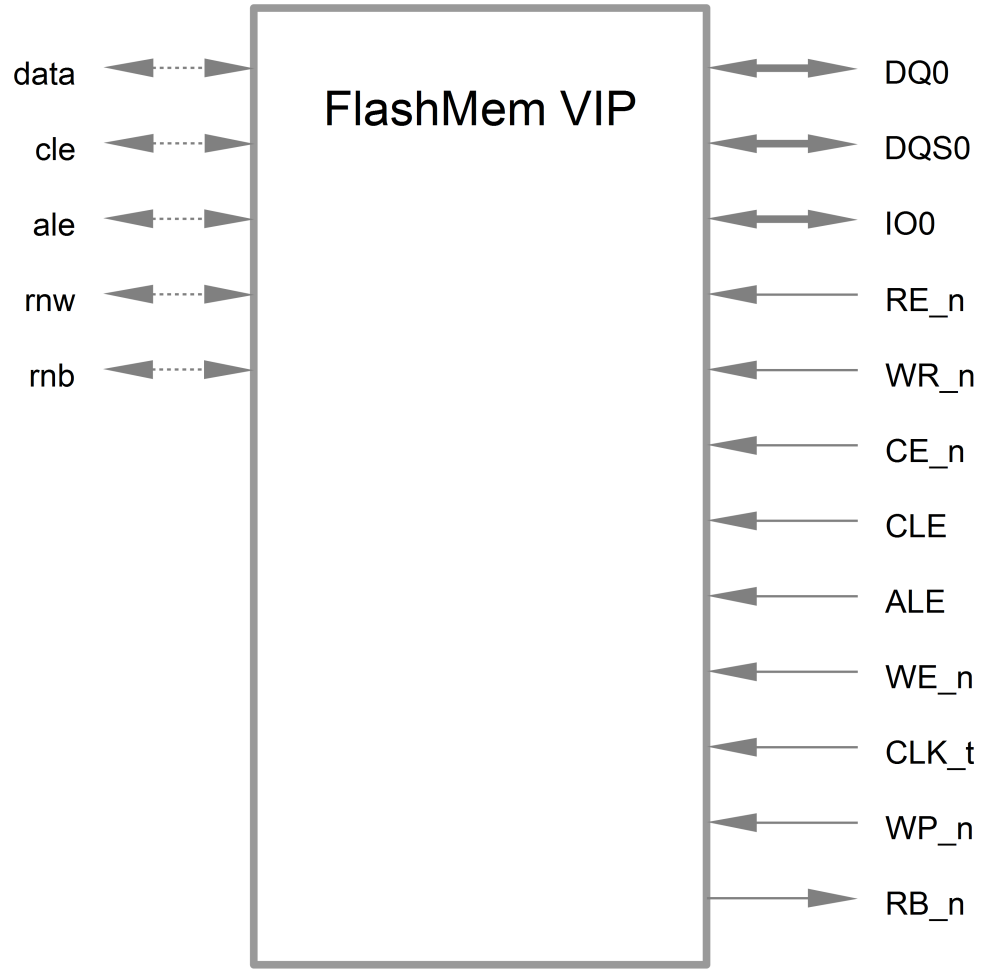

Illustration 1: FlashMem VIP Block Diagram

1 Introduction

The following chapters in this sections give a short introduction to flash memories and verification IPs. The section FlashMem VIP describes the present flash memory verification IP in detail. Finally section Test explains the tests executed to verify the VIP and gives some ideas how the FlashMem VIP could be improved.

1.1 Flash Memory

There exist basically two types of flash memory, which differ on their internal structure. They are therefore called NAND and NOR flash. It is left to the interested reader to inform himself about the structural details and differences of the two flash types. For this VIP, the important part of these flash memories is the interface they provide. While NOR memories provide an external address bus for read and program operations (and thus support random-access), unlocking and erasing NOR memories must proceed on a block-by-block basis. With NAND flash memories, read and programming operations must be performed one page at a time, while unlocking and erasing must happen in a block-wise fashion.[1] The FlashMem VIP is built imitating the NAND flash memories.

1.1.1 ONFi

An industry working group of more than 80 companies called Open NAND Flash interface (ONFi)[2] has developed an open standard for the low-level interface to raw NAND flash chips[3]. The goal of this specification is to simplify NAND flash integration[4]. To be compatible to a wide range of

1 http://en.wikipedia.org/wiki/Flash_memory#Low-level_access, 17:03, 25 February 2011
2 http://onfi.org/
3 http://en.wikipedia.org/wiki/Open_NAND_Flash_Interface_Working_Group, 16:07, 19 December 2010
4 http://www.onfi.org, 4 March 2011

memory chips, the FlashMem VIP is based on v3.0 of the ONFi standard, which is available for free on the ONFi web-page. However following restrictions will be applied to simplify the implementation:

- only the asynchronous interface will be implemented
- the target consists of only one Logical Unit
- the target has only one 8-bit I/O port used in single data rate

1.2 Verification IP (VIP)

When it comes to TBs in SV, an open source library called Open Verification Methodology (OVM) is widely adopted today. Nevertheless its successor, the Universal Verification Methodology (UVM), is gaining a lot of momentum. This VIP has been developed with OVM and then been adapted to UVM.

There exists a Perl script provided together with the UVM library code[5] that converts any OVM VIP into an UVM compatible VIP. Nevertheless some manual corrections had to be done afterward.

The VIP implemented in this work is derived from a construct called *verification component (VC)* in UVM terminology. *VCs* typically contain one or more *agents*. An *agent* connects on one side to the DUV and on the other side to the rest of the TB (see Illustration 2). A standard UVM TB consists of several *VCs*.

An *agent* is divided into an active and a passive part. The active part acts and reacts on changes on the DUV pins and is build of the *sequencer* and the *driver*. The passive part collects and reports the activities on the DUV pins and contains at least a monitor.

Moreover, an *agent* can be a *slave agent* or a *master agent*. *Slave agents* respond to requests on the DUV IF, whereas *master agents* can initiate transactions between the TB and the DUV. As the Flash VIP has to react on data accesses form the DUV, it is naturally a *slave agent*.

5 http://www.accellera.org/activities/vip/uvm-1.0p1.tar.gz

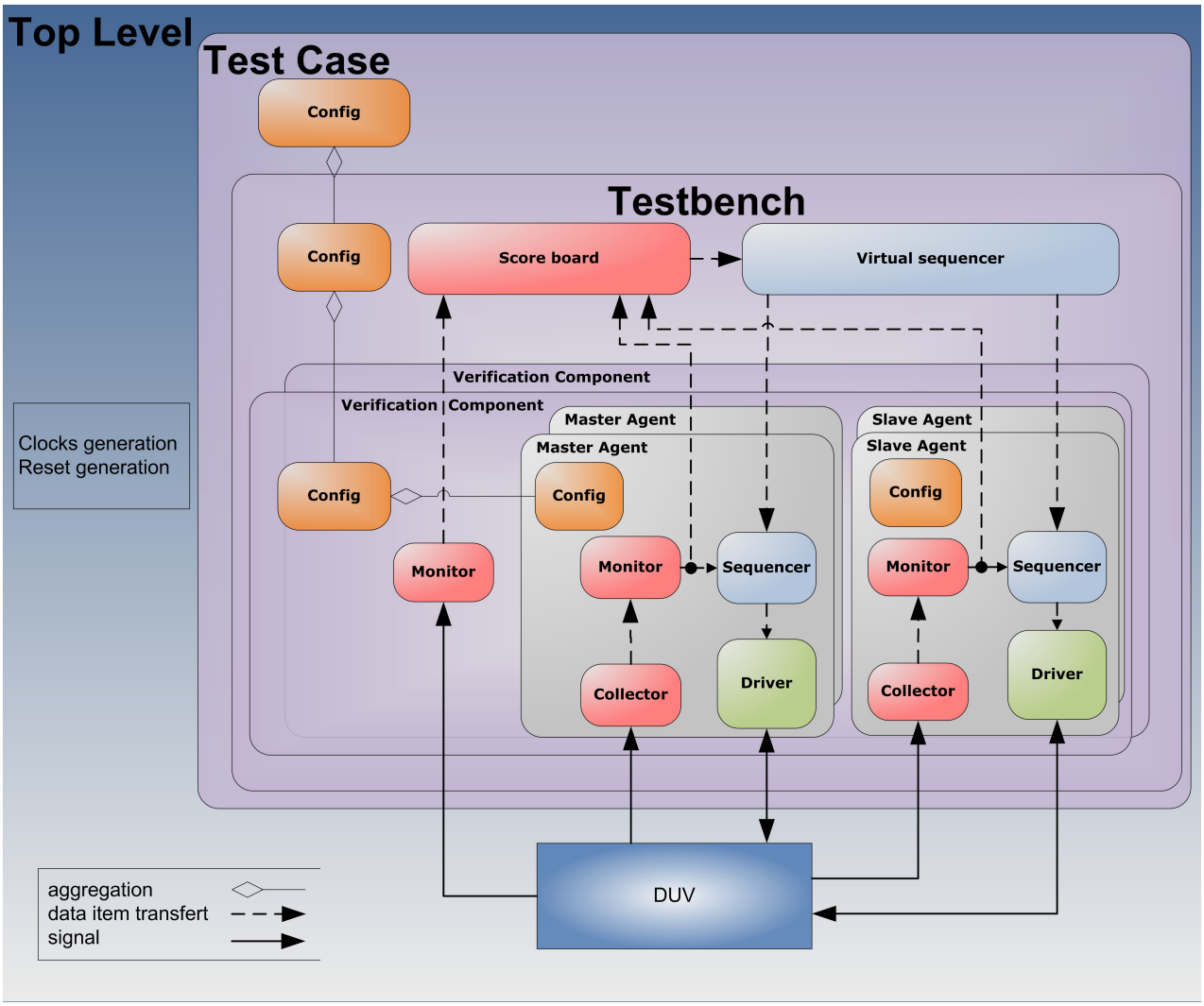

Illustration 2: UVM TB Environment

1.2.1 Interface

An *interface* serves as connection point between the DUV and the TB. It defines the signals used as well as their direction. Therefore the DUV and the *agent* do not connect directly. Instead they connect to the *interface*. This has the advantage, that the signals which are connected between DUV and TB are defined in a single place. The *interface* defines furthermore which signals are inputs or outputs for which component and at which specific moment in simulation time the signals are sampled or driven respectively.

To simplify the description of an *agent*, the *interface* is often neglected and it is assumed that the *agent* connects directly to the DUV.

1.2.2 Transfer

A *transfer* defines the objects exchanged inside the *agent* as well as between the *agent* and the rest of the TB. It serves as abstraction of the pin level *interface* and provides only the needed information to the rest of the TB. Like this, the higher level TB is not influenced by minor changes on the pin level. The higher level testbench can so also be reused for similar types of *interfaces*, by simply exchanging a given agent with another one whit the same transfer.

1.2.3 Driver

The active component acting on the DUV pins is called *driver*. It detects actions on the pins and requests then a transfer from the *sequencer*. The *driver* then converts this *transfer* into signals on the DUV pins.

1.2.4 Sequencer

A *sequencer* takes *sequences* from a list and sends them to the *driver*.

1.2.5 Monitor

The passive component is called *monitor*. It collects the changes on the DUV pins and verifies their correctness. Then it stores them into *transfers* and sends them out for further analysis.

2 FlashMem VIP

The VIP will model a flash memory and is implemented in SystemVerilog (SV). As mentioned above, it follows the UVM methodology and uses the UVM library. The TB environment was generated using the online OVM Template Generator[6] provided by Paradigm Works. In this chapter the lines of code added for the Flash VIP are shown.

It is not the intention of this report to introduce the reader to SV. Nevertheless there are some explanations about the syntax or concepts of SV where it seemed appropriate.

2.1 Architecture

The architecture of the FlashMem VIP is shown in Illustration 3 and is composed of an *environment* containing an *agent* and an *interface*. The *agent* is implemented as an *active slave*. This implies that it is built out of a *monitor*, a *sequencer* and a *driver*. In the following chapters you find more details about these components.

6 http://svf-tg.paradigm-works.com/svftg/ovm

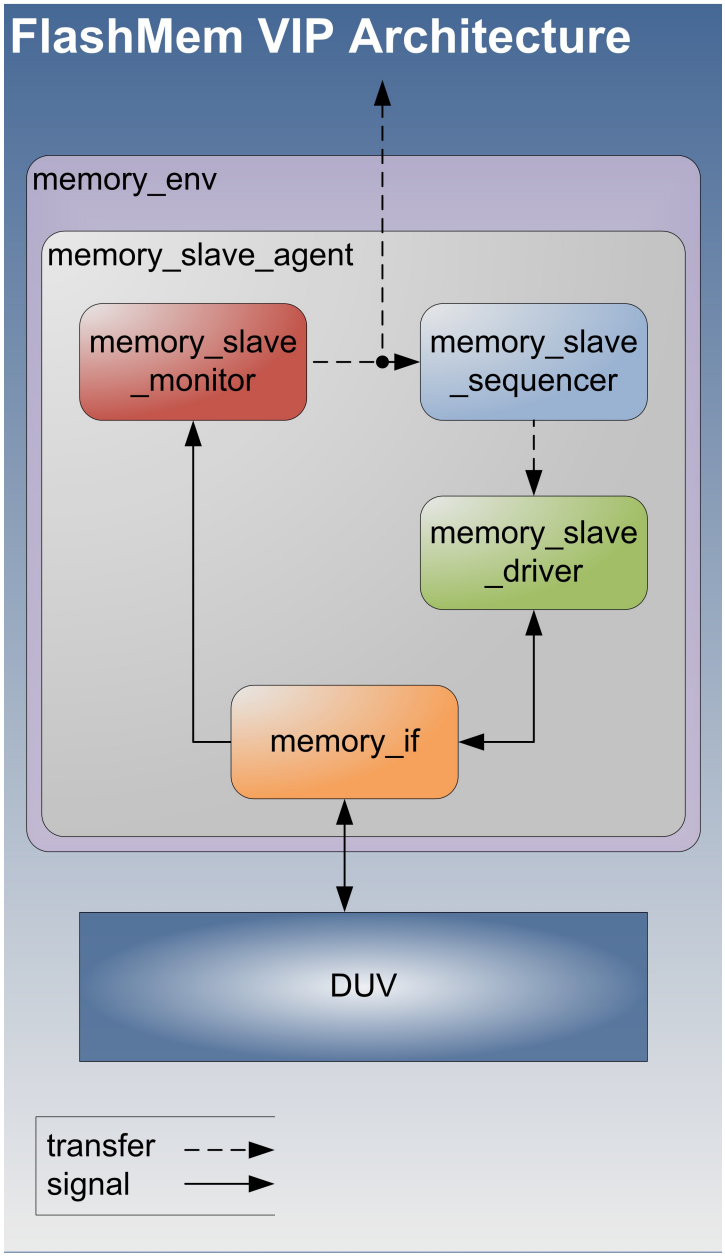

Illustration 3: FlashMem VIP Architecture

2.2 Interface

The *interface* contains the signals shown in Illustration 4 and the *clocking block* shown in Illustration 5. The interface is implemented in file *memory_if.sv*.

2.2.1 Signals

In SV there exists no specific data type for signals containing more than one bit, a bus is just an array of signals. Variables of the type **logic** can only be driven by a single *module.* Therefore the bidirectional signals have to be defined as **wire** to allow SV to resolve the final value[7].

In order to make the bidirectional **wire** IO0 (Line 4) accessible, a local signal IO0_drive (Line 17) has been added. This signal is driven by the *driver* and continuously assigned to the **wire** (Line 18).

There exists another solution to the problem of assigning bidirectional DUV pins. It can be solved with the help of **modport.** This approach would fit better into the UVM methodology and should be elaborated for a future TB or VIP.

```
48:  // flash inouts
49:  wire[(NCHIP*8)-1:0]   DQ0;    // IO Port 0:             for source synchronous
50:  wire[NCHIP-1:0]       DQS0;   // data strobe:           for source synchronous
51:  wire[(NCHIP*8)-1:0]   IO0;    // IO Port 0:                   for asynchronous
52:  // flash inputs
53:  logic                 RE_n;   // not read enable:             for asynchronous
54:  logic                 WR_n;   // not (write/read direction) = read not write
55:  logic                 CE_n;   // not chip enable
56:  logic                 CLE;    // command latch enable
57:  logic                 ALE;    // address latch enable
58:  logic                 WE_n;   // not write enable:            for asynchronous
59:  logic                 CLK_t;  // clock:                 for source synchronous
60:  logic                 WP_n;   // not write protect
61:  // flash outputs
62:  logic                 RB_n;   // not (ready not busy) = busy not ready
63:  // local signals
64:  logic[(NCHIP*8)-1:0]  IO0_drive = 'z;    // IO Port 0: logic to drive wire
65:  assign IO0 = IO0_drive;
```

Illustration 4: Interface Signals (memory_if.sv)

2.2.2 Clocking Block

The *clocking block* synchronizes the *interfaces* signals to a clock and servers therefore as divider between the event driven TB and the clock synchronous DUV. The *present clocking* block is synchronized to the rising edge of the clock signal (Line 69). The *TB top module* assigns the clock signal to the *interface.*

The *clocking block* also defines the exact time, when inputs are sampled and outputs are driven. On line 72 it is specified that the input signals have to be sampled just before the clock. The outputs to the DUV are driven one time unit after the clock (Line 74). The time unit is set in the *TB top module*, in this case it is 1ns.

7 http://en.wikipedia.org/wiki/SystemVerilog#New_data_types, 18:12, 21 May 2011

Furthermore the *clocking block* defines which signals are inputs, outputs or bidirectional signals (Line 75-77). The direction of the signals is defined as seen by the TB.

```
69:  clocking cb @(posedge clk);
70:    //input-output as seen by the TB
71:    //input (sample) skew, #1step = just before the clock
72:    default input   #1step;
73:    //output (drive) skew, #1 = one time unit after the clock
74:    default output  #1;
75:    input RE_n, WR_n, CE_n, CLE, ALE, WE_n, WP_n, CLK_t;
76:    inout DQ0, DQS0, IO0;
77:    output RB_n;
78:  endclocking : cb
```

Illustration 5: Interface Clocking Block (memory_if.sv)

2.3 Transfer

In the context of this project a *transfer* (Illustration 6) is defined as a data exchange (Line 57) together with some controlling signals (Line 58-61). The *transfer* is parametrized by the number of chips in parallel (Line 55, 57). The transfer's code can be found in the file *memory_transfer.sv*.

All signals are of the type `rand logic`. This means that they are initialized with a random value for each *transfer* sent to the driver (§2.4.2). Nevertheless this random value can be overwritten in the *sequence* by a specific value.

```
55:  parameter int unsigned NCHIP = 3;
56:
57:  rand logic[(NCHIP*8)-1:0] data; // data
58:  rand logic                cle;  // command
59:  rand logic                ale;  // address
60:  rand logic                rnw;  // read not write
61:  rand logic                rnb;  // ready not busy
```

Illustration 6: Transfer (memory_transfer.sv)

2.4 Driver

The *driver* is modified to contain several custom *functions* or *tasks*. The first one serves to detect a data exchange on the DUV pins. Then another *task* requests a *transfer* from the *sequencer*, to respond to the data exchange. A third *task* executes the *transfer* on the DUV pins. Of these

functions, the first two happen in zero simulation time. Only the driving *task* advances the simulation time. When there's no data exchange, the output signals have to be driven to an idle state. This is done by a fourth *task.* A last *task* then guarantees that the *driver* advances in simulation time, even if there is no *transfer* to drive on the DUV pins. The starting point off all these *functions* is the *driver*'s *run_phase task.* The source code of this driver is written to the file *memory_slave_driver.sv.*

In SV there exist *tasks* and *functions.* Their syntax is quite similar, though there are some important differences in their usage, which are disclosed in Table 1.

Task	Function
can consume simulation time (posedge, negedge, # delay)	should be executed in zero simulation time
many inputs and outputs	many inputs, one output
no return value	return a value (with return or by assigning a value to the name of the function, whereas the return functionality has priority)
for modeling combinatorial and/or sequential logic	for modeling combinatorial logic
can call other tasks or functions	can call other functions

Table 1: Task vs. Function

2.4.1 Run Phase

The `run_phase` *task* (Illustration 7) of our VIP forks the *task* to drive the pins (Line 39 and §2.4.2) as well as the *task* to set the signals to the reset or idle state (Line 40 and §2.4.5). This means, that the two *functions* always run in separate threads independently.

```
37:  task memory_slave_driver::run_phase(uvm_phase phase);
38:     fork
39:       get_and_drive();
40:       reset_signals();
41:     join_none
42:  endtask : run_phase
```

Illustration 7: Driver Run Phase (memory_slave_driver.sv)

2.4.2 Get Transfer

The `get_and_drive` *task* (Illustration 8) tests if there is a data exchange request from the DUV by calling the *function* `detect_transfer` (Line 86 and §2.4.3). If this is the case, the *sequencer* is checked to have a *transfer* available (Line 89). The available *transfer* is then fetched from the *sequencer* (Line 92) and driven on the DUV pins (Line 100). Then it is reported back to the

sequencer that the *transfer* has been used (Line 101). Finally all signals are reset to idle state (Line 118).

```
 80:  task memory_slave_driver::get_and_drive();
 81:    uvm_sequence_item item;
 82:    memory_transfer t;
 83:
 84:    forever begin
 85:
 86:      if ( detect_transfer() ) begin
 87:        // if data exchange requested from DUV
 88:
 89:        if ( seq_item_port.has_do_available() ) begin
 90:          // transfer available
 91:
 92:          seq_item_port.get_next_item(item);
 93:
 94:          uvm_report_info(
 95:            get_type_name(),
 96:            "sequencer got next item",
 97:            UVM_DEBUG );
 98:
 99:          $cast(t, item);
100:          drive_transfer(t);
101:          seq_item_port.item_done();
102:
103:          uvm_report_info(
104:            get_type_name(),
105:            "sequencer item_done_triggered",
106:            UVM_DEBUG );
107:
108:        end
109:
110:        uvm_report_info(
111:          get_type_name(),
112:          "sequencer item_done_triggered",
113:          UVM_DEBUG );
114:
115:      end
116:
117:      // Advance clock
118:      send_idle();
119:    end
120:
121: endtask : get_and_drive
```

Illustration 8: Driver Get and Drive (memory_slave_driver.sv)

2.4.3 Detect Transfer

This *function* (Illustration 9) detects data exchanges on the DUV pins. It is called when needed by the `get_and_drive` *task* (see §2.4.2). As the current *driver* only drives the data on read accesses from the DUV, a data exchange is detected when the `RE_n` signal is driven low by the DUV (Line 55). If this is the case, a 1 is reported back to the caller of the *function* (Line 65), otherwise a 0 (Line 75).

```
50:  function int memory_slave_driver::detect_transfer();
51:
52:    if (
53:  //          intf.CLE  == 1 |
54:  //          intf.ALE  == 1 |
55:            intf.RE_n == 0 //|
56:  //          intf.WE_n == 0
57:         )
58:    begin
59:
60:      uvm_report_info(
61:        get_type_name(),
62:        "Flash memory transfer request detected",
63:        UVM_DEBUG
64:      );
65:      return 1;
66:
67:    end;
68:
69:    uvm_report_info(
70:      get_type_name(),
71:      "_NO_ Flash memory transfer request detected",
72:      UVM_DEBUG
73:    );
74:
75:    return 0;
76:
77:  endfunction: detect_transfer
```

Illustration 9: Driver Transfer Detection (memory_slave_driver.sv)

2.4.4 Drive Transfer

This *task* (Illustration 10) finally drives the *transfer* on the DUV pins. As all the previous *functions* don't use any simulation time, the system has to be synchronized to the system clock on the first access of this task (Lines 130-132). Afterwards the system remains synchronized to the system clock and the synchronization therefore does not has to be repeated. The synchronization simply waits on the next rising edge of the system clock by waiting on the next event of the *interfaces clocking block*.

After synchronizing, the DUV pins are driven with the values received from the sequencer (Line 135 and 143).

```
123:  task memory_slave_driver::drive_transfer (memory_transfer trans);
124:  string msg;
125:
126:    uvm_report_info(get_type_name(), "drive_transfer");
127:
128:    // sync to clock
129:    //   only needed on first access, further accesses are synced implicitly
130:    if (!synced) begin
131:      @(intf.cb);
132:    end
133:
134:    // drive data
135:    intf.IO0_drive  = trans.data;
136:    $sformat(
137:          msg,
138:          "access with data = 0x%h",
138:          trans.data
139:          );
140:    uvm_report_info(get_type_name(), {"Memory read ", msg});
141:
142:    // and busy
143:    intf.RB_n = !trans.rnb;
144:
145:  endtask : drive_transfer
```

Illustration 10: Driver Transfer Driving (memory_slave_driver.sv)

2.4.5 Idle and Reset

These two *tasks* (Illustration 11) serve, as their names suggest, to drive the signals when there's no data exchange or to reset the signals at the beginning of a *test.*

In addition the `send_idle` *task* also advances the system time by one clock cycle by waiting on an event on the *clocking block* of the *interface* (Line 159 and §2.2.2).

The signal `RB_n` is the only output only signal of the TB. It is therefore set to 0 at reset (Line 165). The bidirectional IO signals are automatically assigned to high impedance when there's no access from the TB.

```
157:  task memory_slave_driver::send_idle();
158:    uvm_report_info(get_type_name(), "send_idle ...", UVM_DEBUG);
159:    @(intf.cb);
160:  endtask:send_idle
161:
162:  task memory_slave_driver::reset_signals();
163:    uvm_report_info(get_type_name(), "reset_signals ...");
164:    // flash outputs
165:    intf.RB_n   = '0; // not (ready not busy) = busy not ready
166:  endtask : reset_signals
```

Illustration 11: Driver Idle and Reset (memory_slave_driver.sv)

2.5 Sequencer

For this project, the *sequencer* shall model the memorizing behavior of the flash memory. It shall store the received data from the DUV in an array and deliver them on request. It's file is called *memory_slave_sequencer.sv.*

There shall also be a backdoor access to the data in the array. This is useful to preset the data if it is needed to start from a given data set. It also allows printing or verifying the content of the array at any time. Furthermore it can be used to alter the data during run-time, for example to introduce errors.

The present *sequencer* does not contain all of this functionality yet.

2.5.1 Monitor Export

The export to get collected *transfers* from the *monitor* (Illustration 12) is implemented on Line 37. The *transfers* are fetched from the *monitor* by implementing the *function* write (Line 48-51). This *function* is called by the *monitor's analysis port* each time it has a *transfer* available. The received *transfer* is then passed on to the `transfer_action` *function* (Line 49 and §2.5.2).

```
37:  item_collected_eport = new("item_collected_eport", this);

47:  // get transactions from monitor
48:  function void memory_slave_sequencer::write(memory_transfer trans);
49:    transfer_action(trans);
50:    item_collected = trans;
51:  endfunction : write
```

Illustration 12: Sequencer Monitor Export (memory_slave_sequencer.sv)

2.5.2 Transfer Action

The `transfer_action` *function* (Illustration 13) shall act on the *transfer* sent to the driver with the data received from the *monitor*. Furthermore it shall implement a data array which will store the memory data. This functionality should be added in a future work. At the moment the function only verifies the good reception of the *monitors* collected *transfer* with a printout of its content.

```
53: // act on transfer
54: function void memory_slave_sequencer::transfer_action(input memory_transfer trans);
55:
56:   string msg;
57:
58:     $sformat(
59:       msg,
60:       {"  data  : 0x%h\n",
61:        "  cle   : %b\n",
62:        "  ale   : %b\n",
63:        "  rnw   : %b"},
64:       trans.data,
65:       trans.cle,
66:       trans.ale,
67:       trans.rnw
68:     );
69:     uvm_report_info(
70:       .id(get_type_name()),
71:       .message({"\nSequencer analysis transaction reception\n", msg}),
72:       .verbosity(UVM_DEBUG),
73:       .filename(`__FILE__),
74:       .line(`__LINE__)
75:     );
76:
77: endfunction : transfer_action
```

Illustration 13: Sequencer Transfer Action (memory_slave_sequencer.sv)

2.6 Monitor

The *monitor* is similarly constructed as the driver. Like the *driver*, the *monitor* has to detect data exchanges on the DUV pins. But instead of giving a response, the *monitor* only collects all necessary information of the data exchange. This information is then sent on to the rest of the TB in form of a *transfer*. The monitor is implemented in the file *memory_slave_monitor.sv*.

2.6.1 Run Phase

In the *monitor* the `run_phase` *task* (Illustration 14) only calls the `monitor_transactions` *task* (Line 80 and §2.6.2) in a separate thread.

```
78: task memory_slave_monitor::run_phase();
79:  fork
80:    monitor_transactions();
81:  join_none
82: endtask : run_phase
```

Illustration 14: Monitor Run Phase (memory_slave_monitor.sv)

2.6.2 Monitor Transactions

The `monitor_transactions` *task* (Illustration 15) tests if there is a data exchange request from the DUV by calling the *function* `detect_transfer` (Line 53 and §2.6.3). If this is the case, a *task* is called to collect the data exchange into a *transfer* (Line 56). Then there are two *function* calls to check the *transfer* (Line 60) and to perform coverage collection (Line 64). The execution of these two functions can be enabled or disabled by setting the corresponding variable (Line 59 and 63). These two *functions* are not implemented in the current VIP. The next important step is to call the `write` *function* of the *monitor's analysis port* (Line 67). Finally the simulation time is advanced by waiting on an event on the *interface's clocking block* (Line 71 and §2.2.2)

```
50:  task memory_slave_monitor::monitor_transactions();
51:
52:   forever begin
53:     if (detect_transfer()) begin
54:
55:       // Extract data from interface into transaction
56:       collect_transfer();
57:
58:       // Check transaction
59:       if (checks_enable)
60:         perform_transfer_checks();
61:
62:       // Update coverage
63:       if (coverage_enable)
64:         perform_transfer_coverage();
65:
66:       // Publish to subscribers
67:       item_collected_port.write(trans_collected);
68:     end
69:
70:     // Advance clock
71:     @(intf.cb);
72:   end
73:
74: endtask // monitor_transactions
```

Illustration 15: Monitor Monitor Transactions (memory_slave_monitor.sv)

2.6.3 Detect Transfer

This *function* (Illustration 16) detects a data exchange on the DUV pins whenever one of the flash memory control pins is in an active state (Line 87 – 90). On this occasion, it returns 1 (Line 97) else 0 (Line 106).

```
 85:  function int memory_slave_monitor::detect_transfer();
 86:
 87:   if (  intf.CLE  == 1 |
 88:         intf.ALE  == 1 |
 89:         intf.RE_n == 0 |
 90:         intf.WE_n == 0 )
 91:   begin
 92:       uvm_report_info(
 93:          get_type_name(),
 94:          "Flash memory transfer request detected",
 95:          UVM_DEBUG
 96:          );
 97:     return 1;
 98:   end;
 99:
100:     uvm_report_info(
101:       get_type_name(),
102:       "Flash memory _NO_ transfer request detected",
103:       UVM_DEBUG
104:       );
105:
106:   return 0;
107:
108: endfunction: detect_transfer
```

Illustration 16: Monitor Detect Transfer (memory_slave_monitor.sv)

2.6.4 Collect Transfer

In Illustration 17 it can be seen how this task collects the state of the interesting DUV pins into a *transfer*. Line 117 and 166 contain some standard UVM code for *transaction* handling and are not further explained here. Then on the lines 130-137 the *task* waits until there is a rising edge on the WE_n signal. This is the moment to sample the data as well as the control signals (Line 140 – 144).

```
112:  task memory_slave_monitor::collect_transfer();
113:
114:   string msg;
115:
116:   // indicate start of transaction
117:   void'(this.begin_tr(trans_collected));
118:
119:   if (intf.cb.CLE == 1) begin
120:     // command latch
121:     msg = "command";
122:   end else if (intf.cb.ALE == 1) begin
123:     // address latch
124:     msg = "address";
125:   end else begin
126:     //data latch
127:     msg = "data";
128:   end
129:
130:   // wait on WE_n to be low
131:   if (intf.cb.WE_n == 1) begin
132:     @(intf.cb.WE_n);
133:   end
134:   // wait on rising edge WE_n
135:   if (intf.cb.WE_n == 0) begin
136:     @(intf.cb.WE_n);
137:   end
138:
139:   // collect data for transfer
140:   trans_collected.data  = intf.IO0;    // data
141:   trans_collected.cle   = intf.CLE;    // command
142:   trans_collected.ale   = intf.ALE;    // address
143:   trans_collected.rnw   = !intf.RE_n;  // read not write
144:   trans_collected.rnb   = !intf.RB_n;  // ready not busy
145:
146:   $sformat(
147:     msg,
148:     {"\nCollected ", msg, " transfer with\n",
149:      "  data: 0x%h\n",
150:      "  CLE : %b\n",
151:      "  ALE : %b\n",
152:      "  REn : %b\n",
153:      "  RBn : %b"},
154:      trans_collected.data,
155:      trans_collected.cle,
156:      trans_collected.ale,
157:      !trans_collected.rnw,
158:      !trans_collected.rnb);
159:   uvm_report_info( get_type_name(), msg, UVM_DEBUG );
160:
161:   // indicate end of transaction
162:   this.end_tr(trans_collected);
163:
164: endtask // collect_transfer
```

Illustration 17: Monitor Collect Transfer (memory_slave_monitor.sv)

2.6.5 Write

This function is not implemented by the *monitor*, but has to be implemented by any subscriber to the *analysis port*. It is called in the subscriber by the *monitor* each time there is a *transfer*. This has the advantage that the *monitor* does not has to know who his subscribers are. In the current TB there are two subscribers to the memory *monitor*, the memory *scoreboard* and the memory *sequencer*. An implementation of this function can therefore be seen in §2.5.1.

2.7 Agent

In the *agent* the connection from the *monitors analysis port* to the *sequencers export* is implemented (Illustration 18). This is very easy and straight-forward with the help of the UVM `connect` *method* (Line 50). The agent's source code can be found in the file *memory_slave_agent.sv*.

In UVM a *port* can only be connected to one *export*. It is always the *module* with the *port* that initiates a *transfer* on the target with the *export*. Data can then flow in both directions. When the data flows from the *initiator* to the *target* it is called a *put* operation. The inverse data flow from the *target* to the *initiator* is called a *get* operation. The *analysis port* is a special *port*, as it can be connected to any number of *exports*.

```
47: function void memory_slave_agent::connect_phase();
48:  if(is_active == UVM_ACTIVE) begin
49:    driver.seq_item_port.connect(sequencer.seq_item_export);
50:    monitor.item_collected_port.connect(sequencer.item_collected_eport);
51:  end
52: endfunction : connect_phase
```

Illustration 18: Monitor Detect Transfer (memory_slave_agent.sv)

3 Test

Debugging and test of the implemented VIP have been done by analyzing the signals on simulation waveforms and with printouts to the transcript. All simulations have been done with Mentor Graphics Questa[8].

3.1 Reporting

UVM provides a very convenient reporting mechanism. There are predefined *macros* to publish messages of different severity levels. Furthermore a verbosity level can be applied to these messages. A list off all predefined verbosity levels and their corresponding report function are laid out in Table 2. During a *test,* a verbosity level for the main reporter function can be defined. Any report whose verbosity level exceeds this level will not be published. For example if the *test* runs with verbosity level *UVM_LOW*, no messages of the severity *info* will be shown. It is however to note that the verbosity is ignored for *warning*, *error* and *fatal* to ensure that the user does not inadvertently filters them out. The verbosity parameter is still in the function body of these *macros* for backward compatibility.

8 http://www.mentor.com/products/fv/questa/

Verbosity Level Name	Verbosity Level Value	Default Report Function Severity
UVM_NONE	0	uvm_report_fatal
UVM_LOW	100	uvm_report_error
UVM_MEDIUM	200	*uvm_report_info* uvm_report_warning
UVM_HIGH	300	-
UVM_FULL	400	-
UVM_DEBUG	500	

Table 2: UVM Verbosity Levels and Report Functions

Only the *macro* provided to report *info* messages is used in this VIP. It is used with the default verbosity level *UVM_MEDIUM* to inform the user of the VIP about important events. Then it is also used with the verbosity level *UVM_DEBUG*. These messages are only intended for the developer of the VIP to confirm the functionality of the VIP.

Messages are coded in three different styles (Illustration 19). The simplest one is the one-liner (Line 1 and 2). It is used for short messages. The simple multi-liner (Line 11 – 15) is used for more complex messages, where putting them on one line would render the code unreadable. In the complex multi-liner (Line 21 – 52) the concept of a `string` variable (Line 21) to build the message is introduced. This is particularly useful when different information in the message is available on different locations in the method (Line 22 – 31). The `string` is used to build the message and published once all information is collected. As a further advantage signals can be included into the message. It is clear that this could also be included into the message *macro*, but it's nicer to separate it for better readability. It is also obvious that after the `string` forming, the message could be published in a one-liner format. However when the specified verbosity level is on a single line it can be temporarily commented out when you need to see just this message in a log without all the other debug messages.
The `$sformat` *system task* (Line 32 – 47) is used to include *variables* into a `string`. The first *argument* (Line 33) of the task gets the `string` constructed from the `string` in the second *argument* (Line 34 – 41) together with the *variables* in the third *argument* (Line 42 – 46).
Then concatenating `strings` in SV is also very easy. Just put a series of `strings` separated by comas between braces (Line 34 – 41).

```
// 1) one-liner, with default and specific verbosity level
 1: uvm_report_info(get_type_name(), "drive_transfer");
 2: uvm_report_info(get_type_name(), "send_idle ...", UVM_DEBUG);

// 2) simple multi-liner with specific verbosity level
11: uvm_report_info(
12:   get_type_name(),
13:   "sequencer item_done_triggered",
14:   UVM_DEBUG
15: );

// 3) complex multi-liner with string concatenation, parameters and specific verbosity level
21: string msg;
22: if (intf.cb.CLE == 1) begin
23:   // command latch
24:   msg = "command";
25: end else if (intf.cb.ALE == 1) begin
26:   // address latch
27:   msg = "address";
28: end else begin
29:   //data latch
30:   msg = "data";
31: end
32: $sformat(
33:   msg,
34:   {
35:     "\nCollected ", msg, " transfer with\n",
36:     "  data: 0x%h\n",
37:     "  CLE : %b\n",
38:     "  ALE : %b\n",
39:     "  REn : %b\n",
40:     "  RBn : %b"
41:   },
42:   trans_collected.data,
43:   trans_collected.cle,
44:   trans_collected.ale,
45:   !trans_collected.rnw,
46:   !trans_collected.rnb
47: );
48: uvm_report_info(
49:   get_type_name(),
50:   msg,
51:   UVM_DEBUG
52: );
```

Illustration 19: Reporting Styles

3.2 Transcript and Wave

All the published messages are logged in file called *transcript* by Questa. The complete *transcript* of a simulation might get some hundreds of thousands of lines. In the following, some excerpts of a *transcript* are shown to attest the functionality of all the implemented *functions*. The test *sequence* applied on the DUV is to execute a write access of one page of the flash memory, and then reread this page several times.

3.2.1 Test 1: Monitor and Sequencer

The first messages logged at simulation time 0 show that the *driver*, the *monitor* and the *sequencer* have been started as intended (Illustration 20).

```
1096: # UVM_INFO @ 0: uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].monitor
[memory_slave_monitor] Flash memory _NO_ transfer request detected
1097: # UVM_INFO @ 0: uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver [memory_slave_driver]
_NO_ Flash memory transfer request detected
1098: # UVM_INFO @ 0: uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver [memory_slave_driver]
send_idle ...
1099: UVM_INFO @ 0: uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver [memory_slave_driver]
reset_signals ...

667: # [0] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].sequencer: Starting default memory
slave sequence
668: # [0] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].sequencer: Blocking peek waiting in
default memory sequence
```

Illustration 20: Transcript First Messages

As there is no activity from the DUV, the messages shown in Illustration 21 are now repeated every clock cycle until the detection of the first data exchange.

```
800: # [130] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: _NO_ Flash memory
transfer request detected
801: # [130] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: send_idle ...
802: # [130] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].monitor: Flash memory _NO_
transfer request detected
```

Illustration 21: Transcript Repeated Messages

The first data exchange detected by the *monitor* is a command latch (Illustration 22). As the *driver* only waits on data reads, it does not detect the data exchange request (Line 7944 – 7945). The messages clearly show that the *monitor* collects the *transfer* in Line 7946 – 7952 and that the

sequencer gets the *transfer* instantly and correct (Line 7953 – 7958). The command 0x80 seen on Line 7948 indicates that the DUV is initiating a write process on the flash memory. The same command is repeated three times on the same data exchange, because the used DUV setup uses three flash memory chips in parallel sharing the same control signals. The next access to be expected would be the writing of the address where the data have to be stored.

```
7944: # [561630] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: _NO_ Flash memory
transfer request detected
7945: # [561630] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: send_idle ...
7946: # [561880] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].monitor:
7947: # Collected data transfer with
7948: #   data: 0x808080
7949: #   CLE : 1
7950: #   ALE : 0
7951: #   REn : 1
7952: #   RBn : 0
7953: # [561880] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].sequencer:
7954: # Sequencer analysis transaction reception
7955: #   data  : 0x808080
7956: #   cle   : 1
7957: #   ale   : 0
7958: #   rnw   : 0
```

Illustration 22: Transcript Command Latch

As expected, the next data exchange request detected is an address latch (Illustration 23). Again it is not detected by the *driver* (Line 7979 – 7980), successfully collected (Line 7981 – 7987) and sent to the *sequencer* (Line 7988 – 7993). In the complete *transcript* it can be observed that there are four address latch commands.

```
7979: # [562380] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: _NO_ Flash memory
transfer request detected
7980: # [562380] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: send_idle ...
7981: # [562630] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].monitor:
7982: # Collected data transfer with
7983: #   data: 0x000000
7984: #   CLE : 0
7985: #   ALE : 1
7986: #   REn : 1
7987: #   RBn : 0
7988: # [562630] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].sequencer:
7989: # Sequencer analysis transaction reception
7990: #   data  : 0x000000
7991: #   cle   : 0
7992: #   ale   : 1
7993: #   rnw   : 0
```

Illustration 23: Transcript Address Latch

The first real data exchange is detected some clock cycles later (Illustration 24). As it is a write operation it is again not detected by the *driver* (Line 8104 – 8105), successfully collected (Line 8106 – 8112) and sent to the *sequencer* (Line 8113 – 8118).

```
8104: # [564630] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: _NO_ Flash memory
transfer request detected
8105: # [564630] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: send_idle ...
8106: # [564880] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].monitor:
8107: # Collected data transfer with
8108: #   data: 0xf8f8f8
8109: #   CLE : 0
8110: #   ALE : 0
8111: #   REn : 1
8112: #   RBn : 0
8113: # [564880] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].sequencer:
8114: # Sequencer analysis transaction reception
8115: #   data  : 0xf8f8f8
8116: #   cle   : 0
8117: #   ale   : 0
8118: #   rnw   : 0
```

Illustration 24: Transcript Data Write

This test shows that the *driver*, the *monitor* and the *sequencer* are working as expected for a Command Latch, an Address Latch and a Data Write access from the DUV.

3.2.2 Test 2: Driver

To test the *driver*, a data read access on the flash memory would be needed. For this to happen with the above described test sequence, the *driver* needs to set the flash memory's busy signal correctly. This however is only possible when the *sequencer* controls this correctly. As the *sequencer* is not completed yet, another way has to be found to test the *driver* anyways. In consequence the *driver* has been modified as shown in Illustration 25 to act also on write accesses.

```
52:   if (
53: //        intf.CLE  == 1 |
54: //        intf.ALE  == 1 |
55:         intf.RE_n == 0 |
56:         intf.WE_n == 0     // for debug only: enabled to act also on write accesses
57:       )
58:   begin
```

Illustration 25: Driver Detect Transfer Write Access (memory_slave_driver.sv)

In the screenshot of the waveform in Illustration 27 it can be seen that the *flash_RnB* signal takes arbitrary values and is well aligned to the rising edge of the *Flash_WE_N* signal. Furthermore it can be seen that the data pins *Flash_ECC_IN* and *flash_IO_in* change to an undefined state, the moment the TB starts to drive the pins while the DUV is driving them too.

In the *transcript* (Illustration 26) the same action can be verified with the report messages. Line 10612 shows that the *driver* detects the *transfer* as well as the *monitor* (Line 10611). Then the *driver* drives the *transfer* on the DUV pins (Line 10615) and reports the random data it's going to put out (Line 10628). After that the *driver* advances the simulation time by calling the *send_idle task* (Line 10631) and that the sequencer prepares a new *transfer* (Line 10633). One clock cycle later, the data exchange is collected by the *monitor* (Line 10636 – 10642). As the DUV as well as the TB are driving the data pins, they are in an unknown state (Line 10638).

```
10611: # [744880] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].monitor: Flash memory
transfer request detected
10612: # [744880] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: Flash memory
transfer request detected

10614: # [744880] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: sequencer got next
item
10615: # [744880] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: drive_transfer

10628: # [745130] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: Memory read access
with data = 0x931f4d
10629: # [745130] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: sequencer
item_done_triggered
10630: # [745130] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: sequencer
item_done_triggered
10631: # [745130] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].driver: send_idle ...

10633: # [745130] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].sequencer: Blocking peek
waiting in default memory sequence

10636: # [745380] hier=uvm_test_top.VIPC_tb0.memory_0_inst.slaves[0].monitor:
10637: # Collected data transfer with
10638: #   data: 0xXXxXXX
10639: #   CLE : 0
10640: #   ALE : 0
10641: #   REn : 1
10642: #   RBn : 0
```

Illustration 26: Transcript First Messages

This is the confirmation that the *driver* is really driving the DUV pins on request. Further tests have to be done once the *sequencer* is completed.

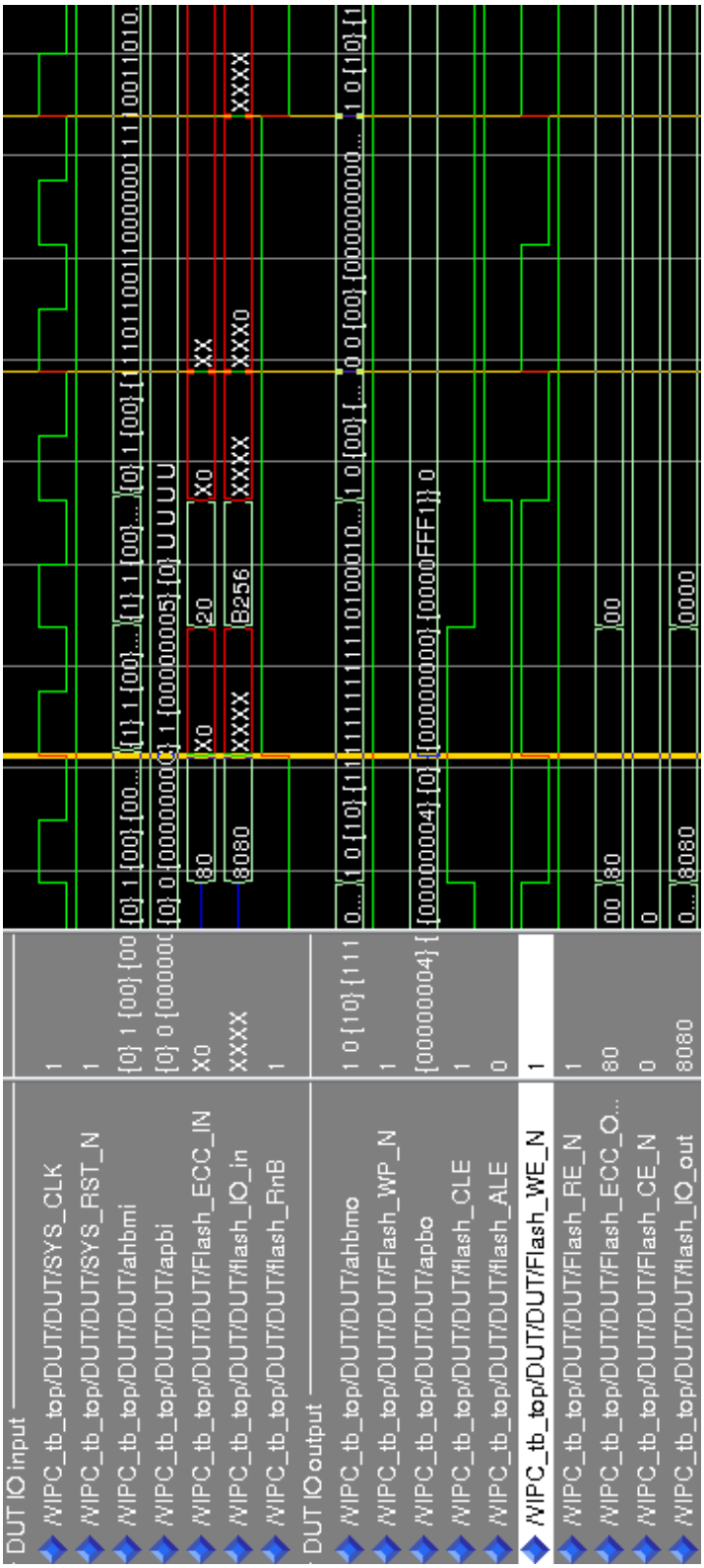

Illustration 27: Wave Test2 Driver

3.3 Future Work

In a next step the sequencer should be completed to store the received data and provide them on request. Then sequences to respond correctly to page write and page read accesses could be written, including the correct handling of the flash busy signal. Once this is done, a basic flash memory VIP will have been developed.

This basic VIP could then be improved by adding more sequences e.g. for reading the memory status, erasing a block, copy-back programming a page and so on. It could also be improved to be fully compatible with the ONFi specification without restrictions and for all types and sizes of flash memories.

Then also the case of the bidirectional signals could be solved in more genuine way by using a modport.