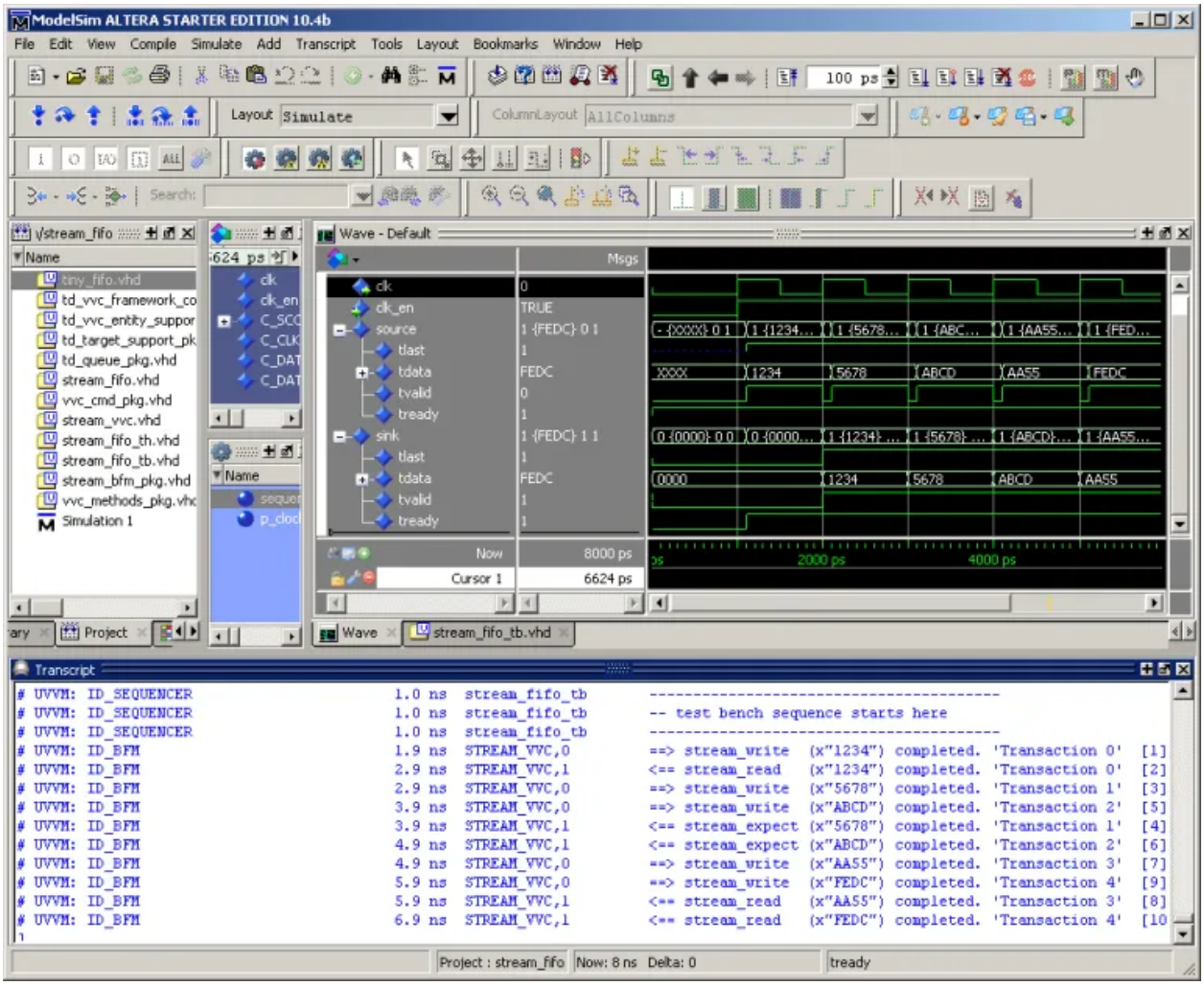
QUE



Tutorial: Create your own VVC for UVVM

VHDL designers are generally not spoiled with Verification IP (VIP). In fact, if you are using any sort of verification IP at all, then most likely it was developed in-house with the company you are working for. And most likely, such verification IP is a collection of BFM (Bus Functional Model) procedures which lacks any advanced features of a modern verification framework.

Part 1

Introduction

The de facto industry standard verification framework is based on SystemVerilog Assertions, but if you have ever checked the cost of a simulator supporting SystemVerilog Assertions, you will know that it is in the

range of an arm and a leg, possibly two. Just to mention it: If you are a VHDL developer, then you would also have to learn another language.

To the rescue comes OSVVM and UVVM! These VHDL Verification Methodologies gives any VHDL designer access to advanced verification features, and they both come for free!

In this tutorial, you will learn how to create your own VHDL Verification Component (VVC) for UVVM (Universal VHDL Verification Methodology).

UVVM Advantages

I assure you: After you master UVVM, you will never again write VHDL test benches in any old-fashioned way. Among the many advantages UVVM offers, your VHDL test benches will:

- Be less verbose, *achieving much more* with *less code*.
- Allow true BFM *concurrency* on all your DUT interfaces.
- *Protocol checkers* can be built into your VVCs, which will move your focus away from viewing waveforms (which can be *very* time consuming).
- Be *highly structural*. With UVVM, your test benches will look very much similar, and a lot of your test bench code can be *reused with little effort*.
- *Save time*, which can be used to write better test cases and locate all bugs in your modules before system integration.

With UVVM, you will also get a small number of VVCs that can handle a few different interfaces. If your particular interface is not included, then in order to unleash the full potential of UVVM you will need to learn to write your own VVCs for UVVM. Expect to spend a week or so to write your first VVC (or 1-2 days if you follow this tutorial).

DUT

To do a UVVM tutorial, we need a DUT (Device Under Test). In another article, I created a component called *tiny_fifo*. For this tutorial, I have now created a wrapper for *tiny_fifo,* where I have added a *tlast* signal. The entity for our new fifo looks like this:

```
1.  entity stream_fifo is
2.    generic (
3.      GC_DATA_WIDTH : natural;
4.      GC_DATA_DEPTH : natural);
5.    port (
6.      clk             : in  std_logic;
7.
8.      fifo_in_tdata   : in  std_logic_vector(GC_DATA_WIDTH-1 downto 0);
9.      fifo_in_tlast   : in  std_logic;
10.     fifo_in_tvalid  : in  std_logic;
11.     fifo_in_tready  : out std_logic;
12.
13.     fifo_out_tdata  : out std_logic_vector(GC_DATA_WIDTH-1 downto 0);
14.     fifo_out_tlast  : out std_logic;
```

```
15.       fifo_out_tvalid : out std_logic;
16.       fifo_out_tready : in  std_logic);
17.   end entity;
```

The DUT files are provided for your inspection.

In this tutorial we will:

- Create a VVC that can do write and read transactions to and from the *stream_fifo*.
- Create a test harness and a test bench which utilizes our new VVC.

Note: Included with UVVM, you also get an AXI-Stream VVC. This is a great VVC, and with some test bench tweaking it *could* be used to simulate our *stream_fifo*. But for our example, this VVC is not entirely suitable, because it has a rather high level of abstraction and is working on *arrays of bytes*. For our example, we want something lower level that works on *beats* (single-clock transactions) and an arbitrary number of bits.

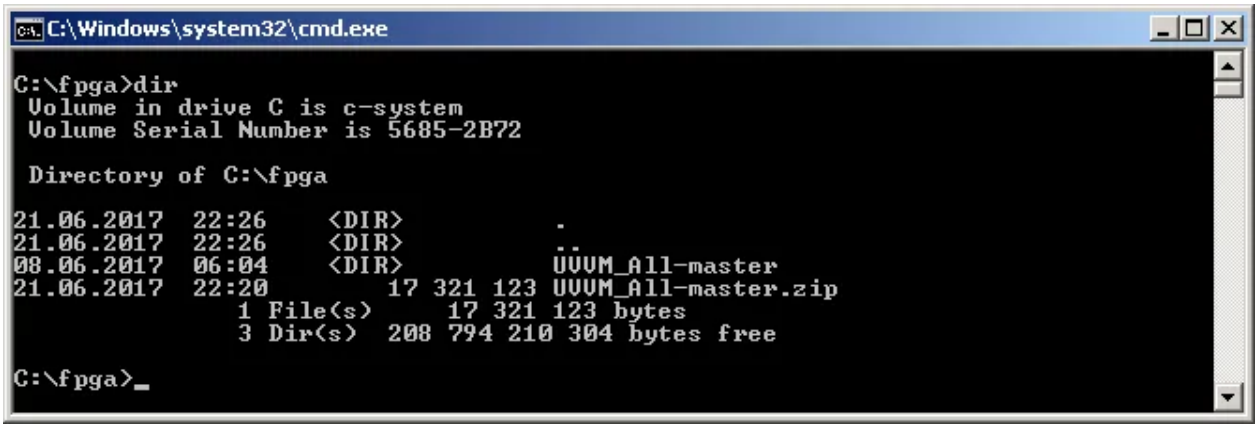
Preparations

- This tutorial assumes that you have a basic understanding of UVVM. If not, please read the documentation supplied with UVVM first.
- You need to have Python 3.x installed on your computer and in your PATH environment variable. (If you are a Linux user, I'm sure you know what to do).
- You need a VHDL simulator supporting VHDL-2008. The *free* Intel (Altera) version of ModelSim is supported. Xilinx Vivado Simulator is *not* supported.
- You need to know how to operate you VHDL simulator, know the concept of VHDL libraries, and you should know how to create and run scripts.

Download and install UVVM

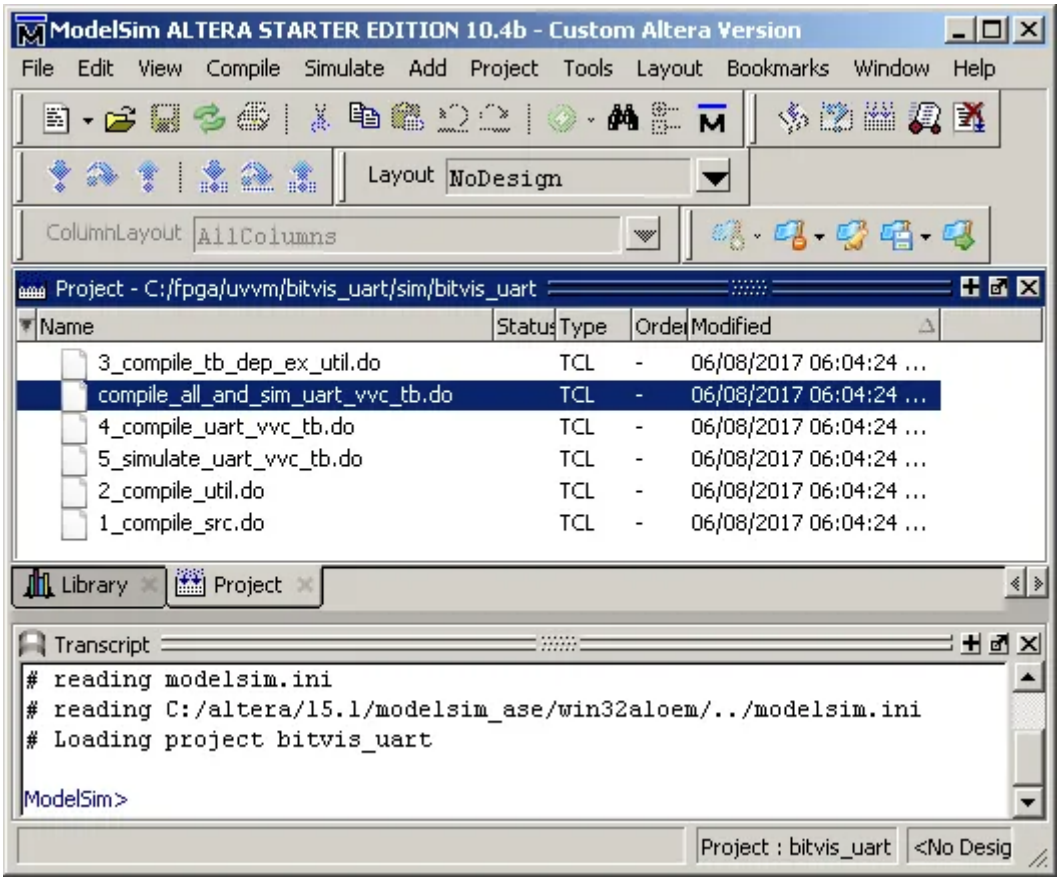
From https://github.com/UVVM/UVVM_All, download the latest ZIP archive. Install it anywhere you want, but for the remaining of this tutorial I will assume that you created a directory called *c:\fpga* and downloaded the .zip file there.

Then, using your favorite unzip program, choose *extract here*.



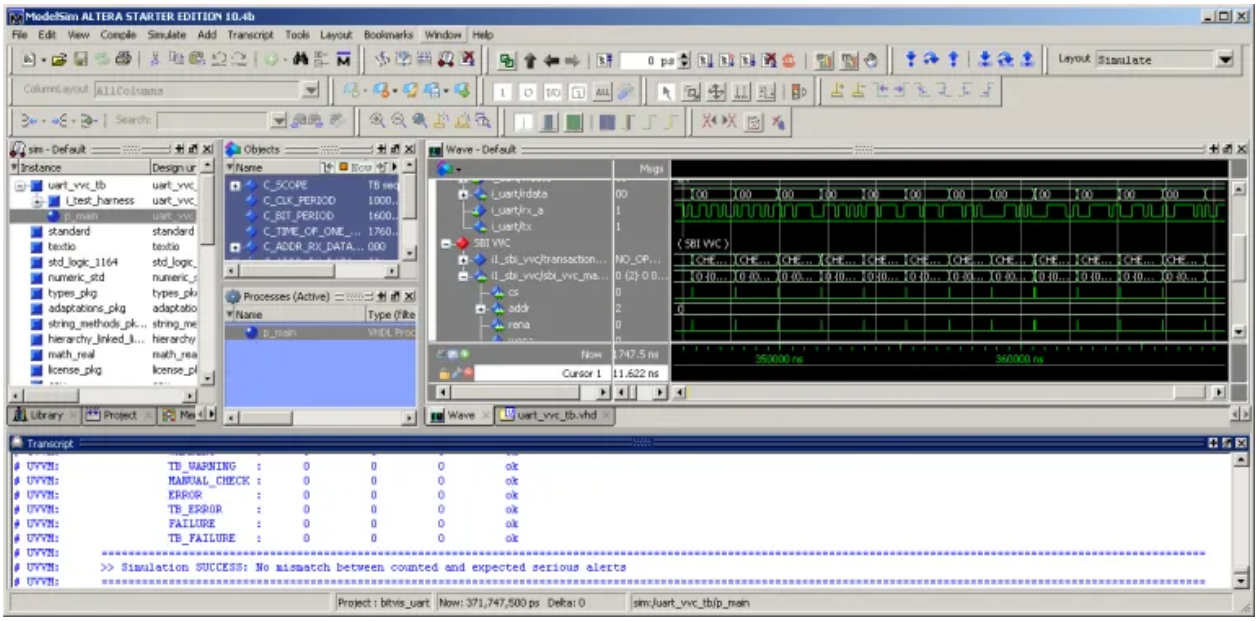
I prefer to rename the *UVVM_All-master* directory to just *uvvm*, like this:

Next, fire up ModelSim, and open the project *bitvis_uart.mpf* located in the *C:\fpga\uvvm\bitvis_uart\sim* directory.



Right-click on *compile_all_and_sim_uart_vvc_tb.do*, and select *Execute*

Congratulations – you should now have run a UVVM test bench for the first time! Check the the Transcript and Wave windows, and you should see a lot of activity there!



Prepare your VIP directory structure

For your new verification IP, create the following directory structure:

- *c:\fpga\vip*
- *c:\fpga\vip\vip_stream*
- *c:\fpga\vip\vip_stream\src*
- *c:\fpga\vip\vip_stream\sim*

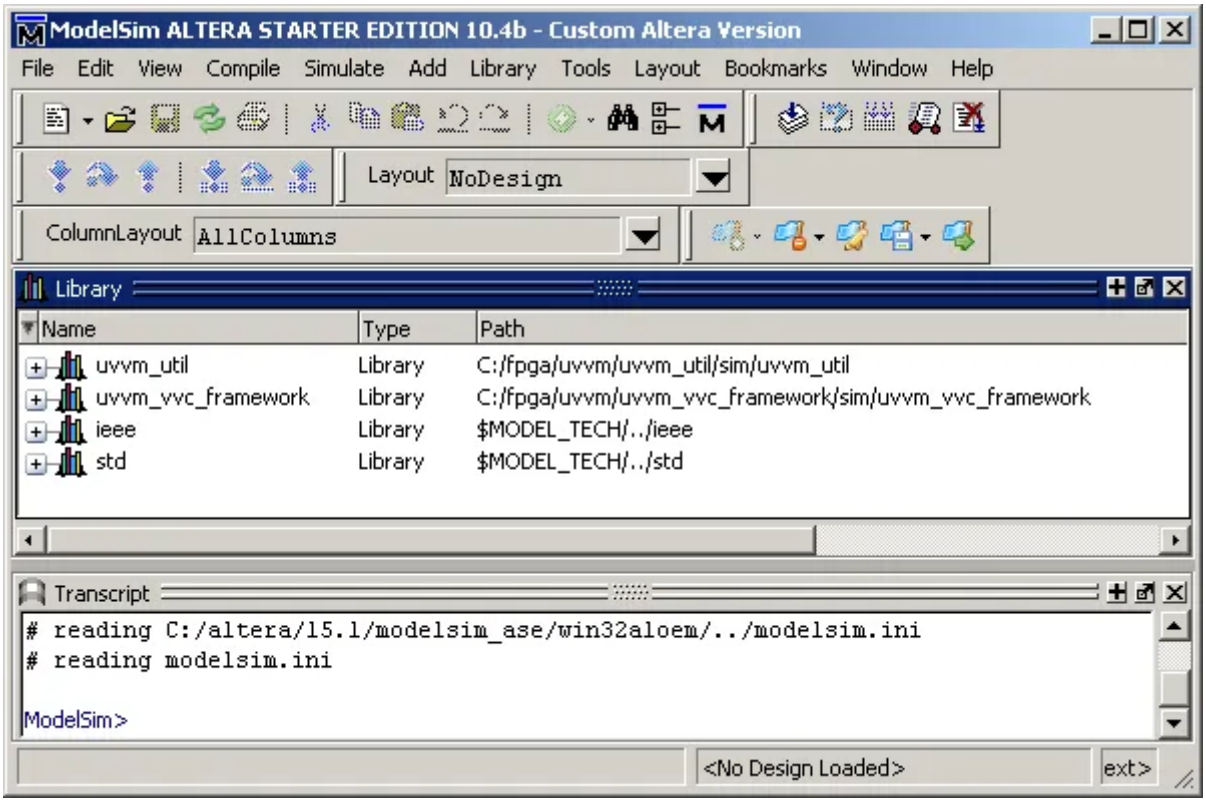
In this case, *stream* is going to be the name of the new VIP that we are creating.

In the *c:\fpga\vip\vip_stream\sim* directory, create a *modelsim.ini* file with the following contents:

```
[Library]
; STANDARD LIBRARIES
std                 = $MODEL_TECH/../std
ieee                = $MODEL_TECH/../ieee

; UVVM libraries
uvvm_util           = C:/fpga/uvvm/uvvm_util/sim/uvvm_util
uvvm_vvc_framework  = C:/fpga/uvvm/uvvm_vvc_framework/sim/uvvm_vvc_framework

[vcom]
VHDL93 = 2008

[vsim]
Resolution = ps
UnbufferedOutput = 1
```

Note: the UVVM libraries referenced in the *modelsim.ini* file were created when you ran the UART example above. If you didnt, then you must compile these libraries in some other way.

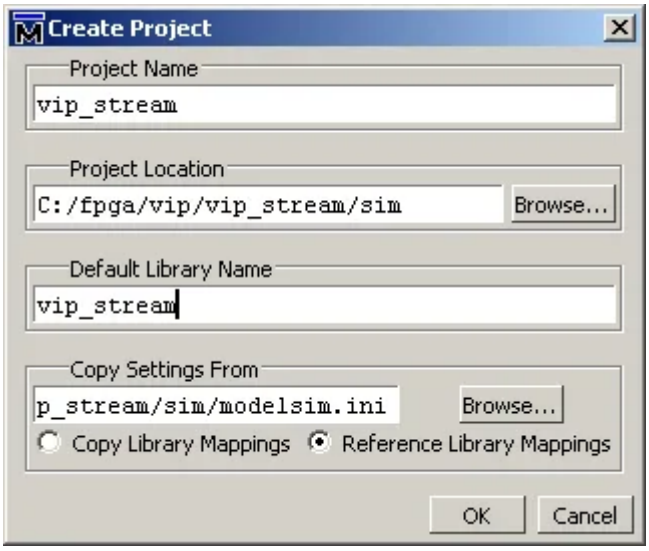
Quit the current simulation in ModelSim, and the navigate ModelSim (change directory) to the *c:\fpga\vip\vip_stream\sim* directory. Notice that the Libraries view in ModelSim now only contains the libraries that we need:



Create a ModelSim project

Later, you will probably perform all ModelSim tasks using a scripted flow. But for this tutorial, we will create a ModelSim project and perform all tasks from within the ModelSim GUI.

- Select File->New->Project and name your project *vip_stream.*
- Set the Default Library Name to vip_stream.

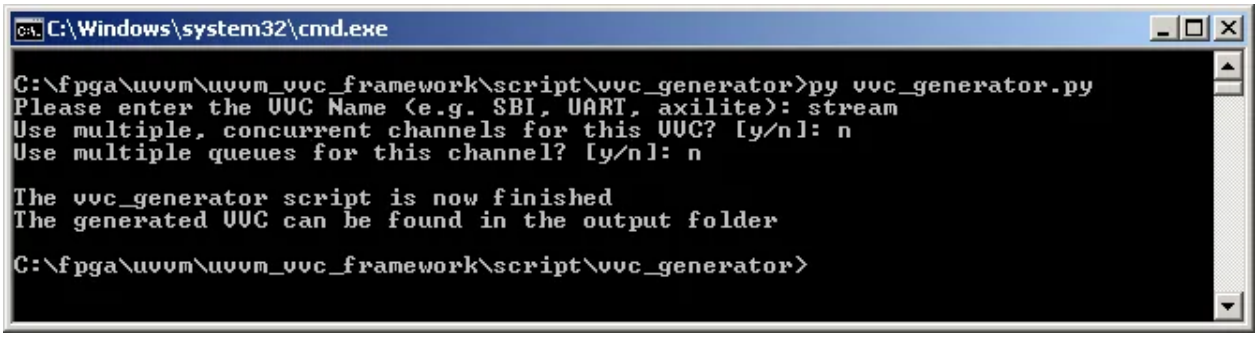


Press OK.

Create a template for your new VIP

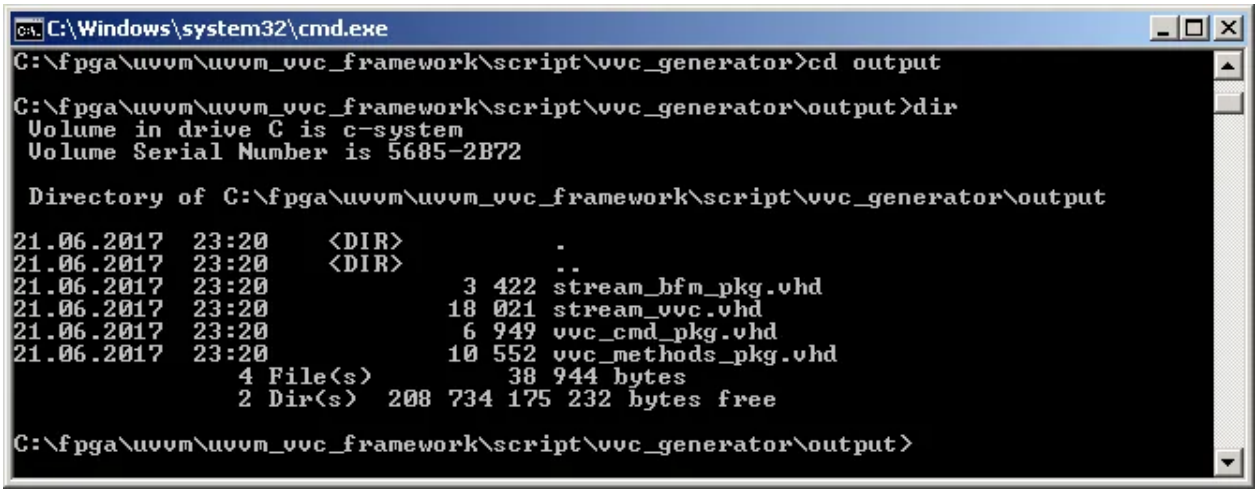
Open a command window, and navigate to the *c:\fpga\uvvm\uvvm_vvc_framework\script\vvc_generator* directory.

Type *py vvc_generator.py* and press *Enter* to run the script. Use a short and simple name for your VIP, like “uart”, “i2c” or similar. In this tutorial, we will use the name “stream”. Answer “n” to the next two questions.



Note: In earlier versions, UVVM would fail if a name with more than 14 characters was entered. Workaround: Use short names (like above) for your VIP.

An output directory was just created:



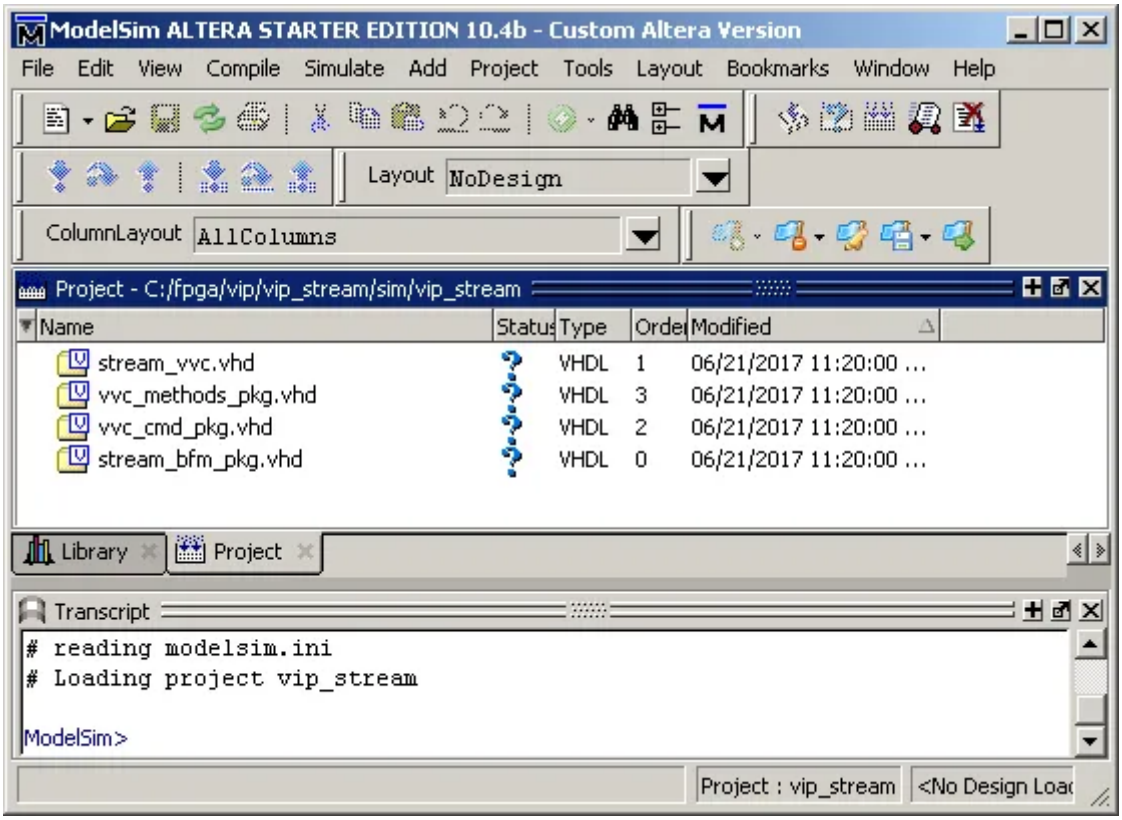
Observe the following:

- 4 files have been created which you will need to modify.
- File name suffixes *_bfm_pkg and *_vvc has been added automatically.
- Inside the files generated, there are names of types and objects that reflect the name you just entered above.

Update the ModelSim project

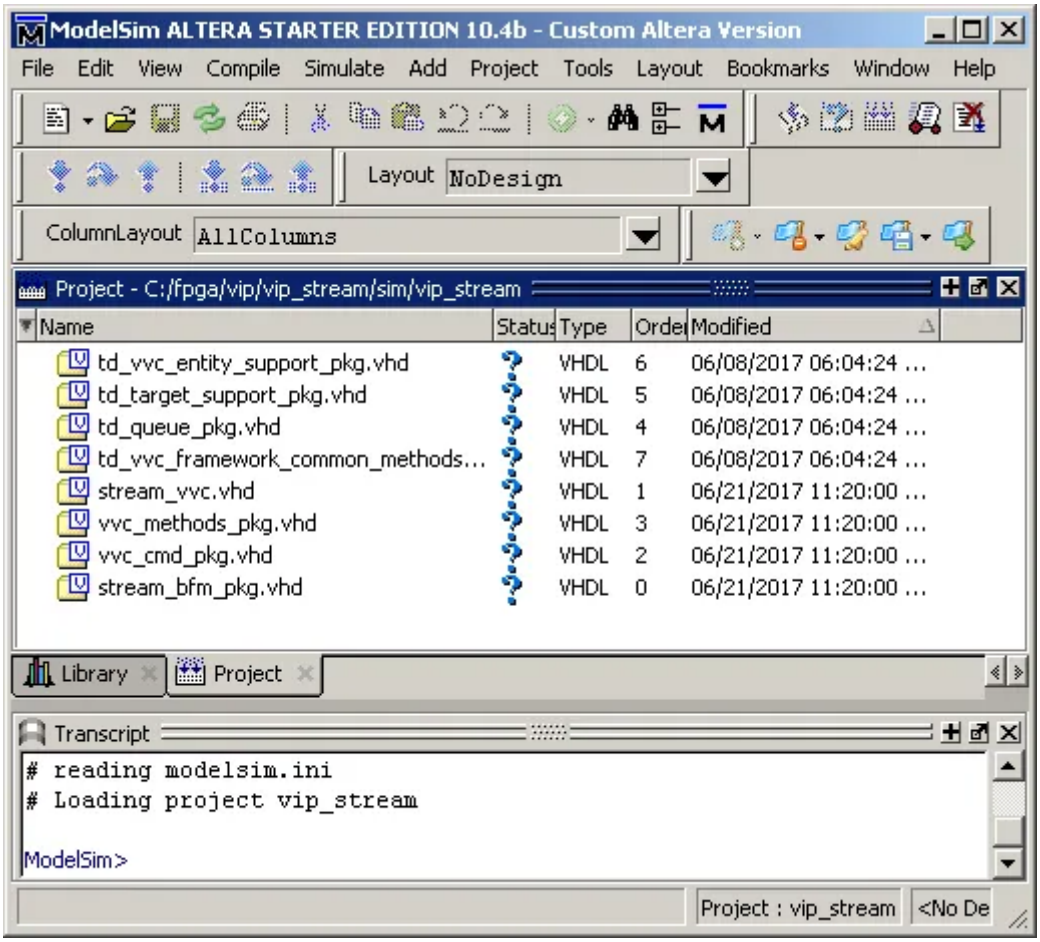
First, move the 4 files you just created above to the *c:\fpga\vip\vip_stream\src* directory.

Then – add these files into your ModelSim project.

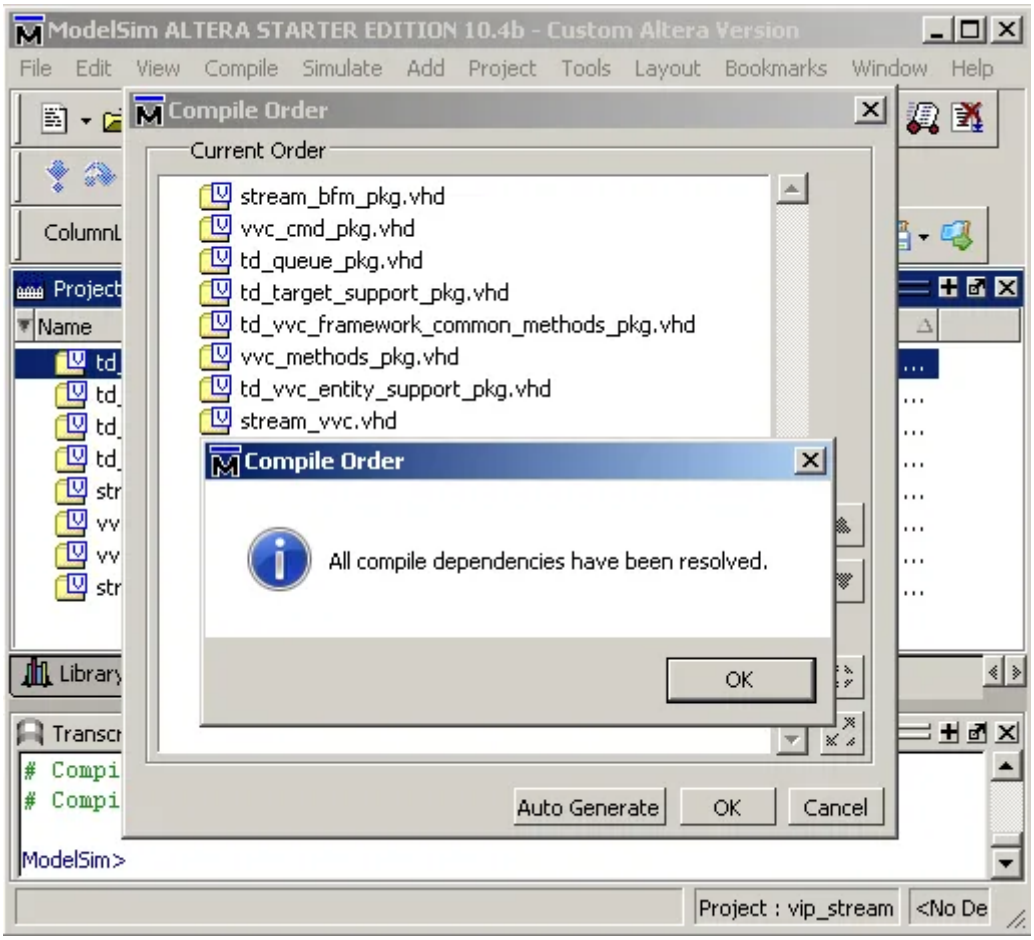


You need to add a few more files to your ModelSim project. Add the 4 files in the *c:\fpga\uvvm\uvvm_vvc_framework\src_target_dependent* directory to the project.

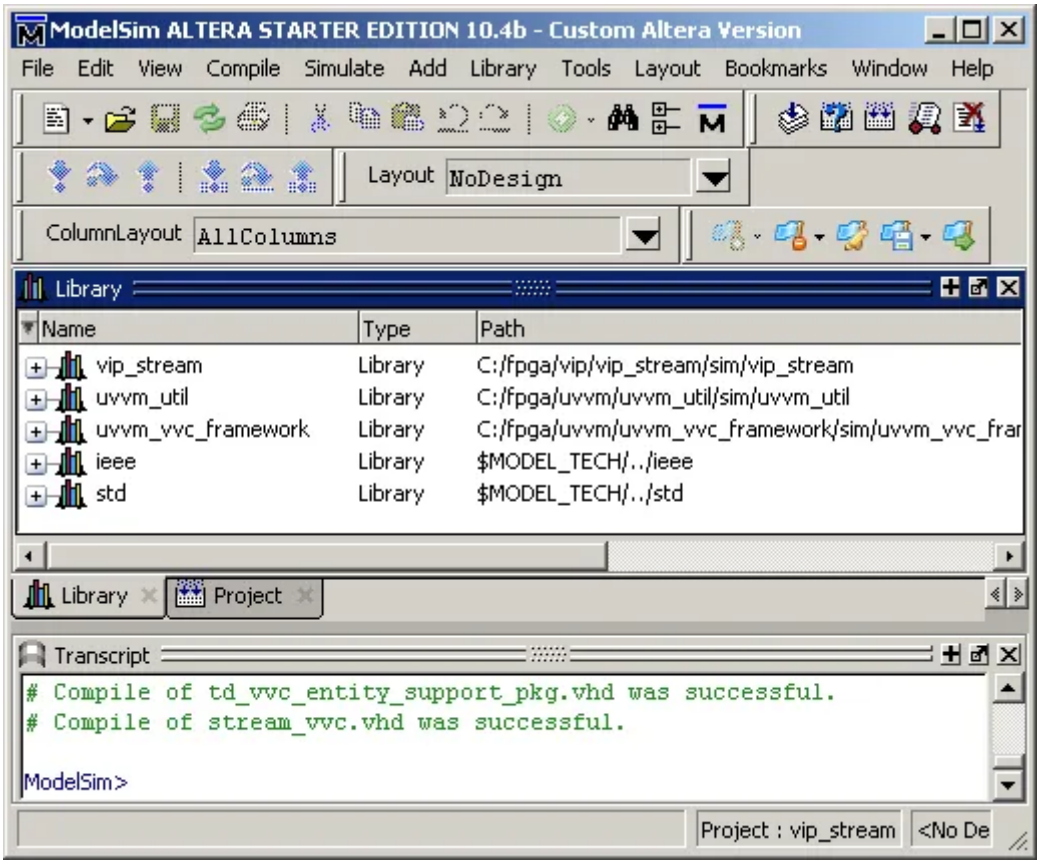
Your project should now look like this:



Right-click in the files section, and select *Compile->Compile Order..* and press *Auto Generate*



And now the *Library* view should look like this:



We have not modified our template files yet, but it is a great start to have all the required files compiling without errors. *(Note: this was not the case with the first versions of UVVM).*

Congratulations – we are now ready to start targeting our template VVC!

Part 2

Planning your new VVC

This is the most important stage in the creation of your new VVC.

In case you missed the previous sentence, i will repeat it:

- ***Planning is the most important stage in the creation of your new VVC.***

As an experienced VHDL designer, you would know how tedious it can be to make changes in an existing VHDL file hierarchy. If you make a change in one location, there are typically 10+ other changes that also need to be made, which is usually very time consuming. You can avoid that with careful planning.

User parameters

To operate the VVC, it needs user parameters. There are 3 sources of user parameters:

- Generic constants which are set when a VVC is instantiated. These may be forwarded to a BFM.
- Dynamic parameters given to a CDM, usually to be forwarded to a BFM.
- The BFM configuration record (in *_bfm_pkg.vhd) which may be initialized in the test harness or in the test bench.

The BFM configuration record

The purpose of the BFM configuration record is to hold various parameters needed for your BFM. Typically, the parameters will be assigned a default value, and if required then you modify them in your test bench *once* in the beginning of your simulation. Only one of the parameters are mandatory: the *clock_period* parameter.

For our stream_vvc, the definition of the BFM configuration record t_stream_bfm_config is located in the stream_bfm_pkg.vhd file. It also has a default value, C_STREAM_BFM_CONFIG_DEFAULT.

More on the *clock_period* BFM configuration parameter

Except for the *clock_period* parameter, I don't find myself using the BFM configuration record for other purposes. Instead, I prefer to configure my VVCs using *generic constant* parameters which I then forward to my BFM. It is therefore a little unfortunate that *clock_period* is the only mandatory parameter in the BFM configuration record, as you now need to configure your VVC with both *generic constant* parameters in the VVC instantiation *and* in the BFM configuration record.

A BFM configuration parameter like *clock_period* is important. You should have to set it *explicitly* for every VVC instantiation, and the simulation should flag an error if you forget to do so. In the generated template files, *clock_period* is set to a default value of 10 ns. If you forget to change it, then your test bench will run happily with *clock_period* = 10 ns, but then that could be very wrong in your particular design.

You may wonder why you would need the *clock_period* parameter for your BFM. In fact, if you only need to do *wait until rising_edge(clk)* and *wait until falling_edge(clk)* in your BFM, then you don't need this parameter in the BFM itself (it is still needed in the VVC though). But for more sophistication, your BFM may need to operate on fractions of a clock period. An example is a routine from the UVVM utility library that we will be using:

```
1.  -- wait 1/10 (10%) into the clock cycle
2.  wait_until_given_time_after_rising_edge(clk, (config.clock_period*1)/10);
```

Again: Do not forget to set *clock_period* in your test bench or in your test harness!

CDM dynamic parameters

During the execution of your test bench sequencer, you will make CDM procedure calls with parameters that you provide dynamically. Typical names of dynamic parameters could be *data_to_write* or *data_to_expect*.

VVC constant parameters

When you instantiate a VVC in the test harness, it needs to be configured with generic constant parameters. Typical names of generic parameters could be GC_VVC_TDATA_WIDTH and GC_VVC_IS_SOURCE. These generic parameters may be forward to your BFM.

VVC executable code

While making your VVC, there are several places that you may add your own executable code:

- *vvc_methods_pkg.vhd*: This is where you define your CDMs (Command Distribution Methods), which are the definitions of the procedures that you may call from your test bench. Various sanity checks, etc. for the CDM call could be placed here.
- **_vvc.vhd*: This is the component you instantiate in your test harness, which receives commands from the VVC command queue, and make procedure calls to the appropriate BFMs. You could write executable code here also, prior to or after the call to your BFM.
- **_bfm_pkg.vhd*: This is where you define your BFMs (Bus Functional Models), which are the procedures that do the actual *wiggling* of your DUT interface signals, through the VVC entity ports.

My recommendations

- **Put all your executable code in a *single* BFM.** If you start creating multiple BFMs (e.g. separate BFMs for reading, writing and checking), you will soon find yourself repeating fragments of code. Eventually, this is just harder to maintain. Instead – create a single generic all-sing-and-dancing BFM!
- **Place as much as possible of your executable code in your BFM**. Your code is much easier to maintain when it is not spread around multiple modules. Also, in you BFM you have access to all your constant generic *and* dynamic VVC parameters. In the CDM you only have access *only* to the dynamic CDM parameters which limits what you can do there.
- **Create a single generic all-sing-and-dancing CDM that does it all!** In our tutorial, we will make single CDM called *stream*. The BFM will be designed to know what to do, based on the dynamic and constant generic parameters provided.
- **The CDM may be overloaded later.** Get your complete VVC working properly first, then overload your CDM later with simplified and more specific CDMs. Remember that you have access to *all* parameters in your BFM, and you can do all sanity checks there – like not allowing writing to a VVC configured for reading, or the other way around.
- **Think of all possible parameters you may want to transfer to your BFM**, and determine which will be constant generic VVC parameters and which will be dynamic CDM parameters. (They will be harder to add later, because of the VHDL hierarchy update nightmare)

Constant generic parameters for the stream_vvc

For the stream_vvc, these are all the generic parameters i could think of:

```
1.  GC_VVC_IS_SOURCE              : boolean;
2.  GC_VVC_TDATA_WIDTH            : natural;
3.  GC_VVC_CHECK_TLAST            : boolean := true;
```

```
4.  GC_VVC_IS_MONITOR              : boolean := false;
5.  GC_VVC_VIOLATE_SRC_TVALID_RULE : boolean := false;
6.  GC_VVC_CHECK_SRC_TVALID_SIGNAL : boolean := true;
7.  GC_VVC_DONT_CHECK_JUST_REPORT  : boolean := false;
```

Most generic parameters have default values, meaning that they don't have to be explicitly set in the VVC instantiation, unless there is a particular reason to do so. Some of the parameters are only relevant when GC_VVC_IS_SOURCE = true, others are only relevant when GC_VVC_IS_SOURCE = false.

- GC_VVC_IS_SOURCE: Set to true if the VVC shall act as a source sending data into a stream_fifo. Set to false if the VVC shall act as a sink receiving data from a stream_fifo.
- GC_VVC_TDATA_WIDTH: The width of the VVC tdata port, in number of bits.
- GC_VVC_CHECK_TLAST: If false and VVC is sink, tlast checks are disabled.
- GC_VVC_IS_MONITOR: When true and VVC is sink, the VVC will not set the tready signal, only monitor it. This will allow you to have several interconnected DUTs in your test harness, and have a VVC configured as monitor to observe the transactions going between them.
- GC_VVC_VIOLATE_SRC_TVALID_RULE: When true and VVC is source, the VVC will wait for tready to be asserted before tvalid is asserted. This is not allowed according to our IP interconnect protocol rules, but may be used to check DUT behavior under these circumstances.
- GC_VVC_CHECK_SRC_TVALID_SIGNAL: When true and VVC is sink, the VVC will wiggle tready to check if the DUT tvalid follows the IP interconnect protocol rules.
- GC_VVC_DONT_CHECK_JUST_REPORT: If true and when the BFM is called with *expect data* parameters, the parameters will be ignored and the actual data read from the DUT will be reported instead. This allows monitoring a DUT output without knowing the expected result.

Dynamic parameters for the stream_vvc

These are the parameters that will be transferred in the *t_vvc_cmd_record* from the CDM to the BFM:

```
1.  tdata                : std_logic_vector(C_VVC_CMD_DATA_MAX_LENGTH-1 downto 0);
2.  tdata_width          : natural;
3.  tlast                : std_logic;
4.  dont_check           : boolean;
5.  called_as_source     : boolean;
```

- *tdata*: The data to be transferred from the CDM to the BFM. For a write operation, these will be the data to write to the DUT. For an expect operation, these are the data to expect from the DUT.
- *tdata_width*: The width of the data input to the CDM. In the BFM, *tdata_width* may be compared to GC_VVC_TDATA_WIDTH to check if they are equal. Without this check, it is possible to input a *tdata* value with incorrect length.
- *tlast*: The *tlast* bit to be transferred from the CDM to the BFM. Same behavior as for *tdata*.
- *dont_check*: This is provided as a second way (in addition to GC_VVC_DONT_CHECK_JUST_REPORT) to tell the VVC not to evaluate data read from the DUT, but just report them. The idea is that you may first run a *stream_read* CDM with *dont_check* = true just to observe DUT behavior, and later change the call to a *stream_expect* CDM with dont_check = false. The *stream_read* could be overloaded to accept all the *stream_expect* parameters, while also ignoring the parameters, just to be able to quickly change between the two in your test bench. Just a convenience feature – if you want it.

- *called_as_source*: In my recommendations above, I suggested to make a single all-sing-and-dancing generic *stream* CDM, and a single all-sing-and-dancing generic *wiggle* BFM. *called_as_source* tells the BFM if the test bench called the CDM as source or sink. As a sanity check, the BFM may compare *called_as_source* to to GC_VVC_IS_SOURCE to check if they are equal. Remember, the generic *stream* CDM may be overloaded later to provide simplified *stream_read*, *stream_write* and *stream_expect* CDMs.

That's it, we have now *planned* our VVC. Not it's time to implement it!

Part 3

Modify vvc_cmd_pkg.vhd

Open vvc_cmd_pkg.vhd in your favorite editor. First we need to add a new enumeration value in t_operation. You may call it anything, I suggest we call it CMD_STREAM. Remember to place a ',' after the TERMINATE_CURRENT_COMMAND.

```
1.  type t_operation is (
2.    NO_OPERATION,
3.    AWAIT_COMPLETION,
4.    AWAIT_ANY_COMPLETION,
5.    ENABLE_LOG_MSG,
6.    DISABLE_LOG_MSG,
7.    FLUSH_COMMAND_QUEUE,
8.    FETCH_RESULT,
9.    INSERT_DELAY,
10.   TERMINATE_CURRENT_COMMAND,
11.   --<USER_INPUT> Expand this type with enums for BFM procedures.
12.   CMD_STREAM
13.   -- Example:
14.   -- TRANSMIT, RECEIVE, EXPECT
15. );
```

A few lines below, change the value of the C_VVC_CMD_DATA_MAX_LENGTH to the *maximum* width of your tdata parameter that your CDM (and BFM) should accept. I suggest setting it to 1024, as I cannot think of a case where I have to go wider than that.

```
1.  constant C_VVC_CMD_DATA_MAX_LENGTH          : natural := 1024;
2.  -- constant C_VVC_CMD_ADDR_MAX_LENGTH          : natural := 8;
3.  constant C_VVC_CMD_STRING_MAX_LENGTH        : natural := 300;
```

A few lines below, you need to edit *t_vvc_cmd_record*. This is the record that is used to transfer the CDM call parameters to the BFM. We already listed those parameters above, now just add them to the record:

```
1.  type t_vvc_cmd_record is record
2.    -- VVC dedicated fields
3.    --<USER_INPUT> Insert all data types needed to transport data to the BFM here.
4.    tdata                 : std_logic_vector(C_VVC_CMD_DATA_MAX_LENGTH-1 downto 0);
5.    tdata_width           : natural;
```

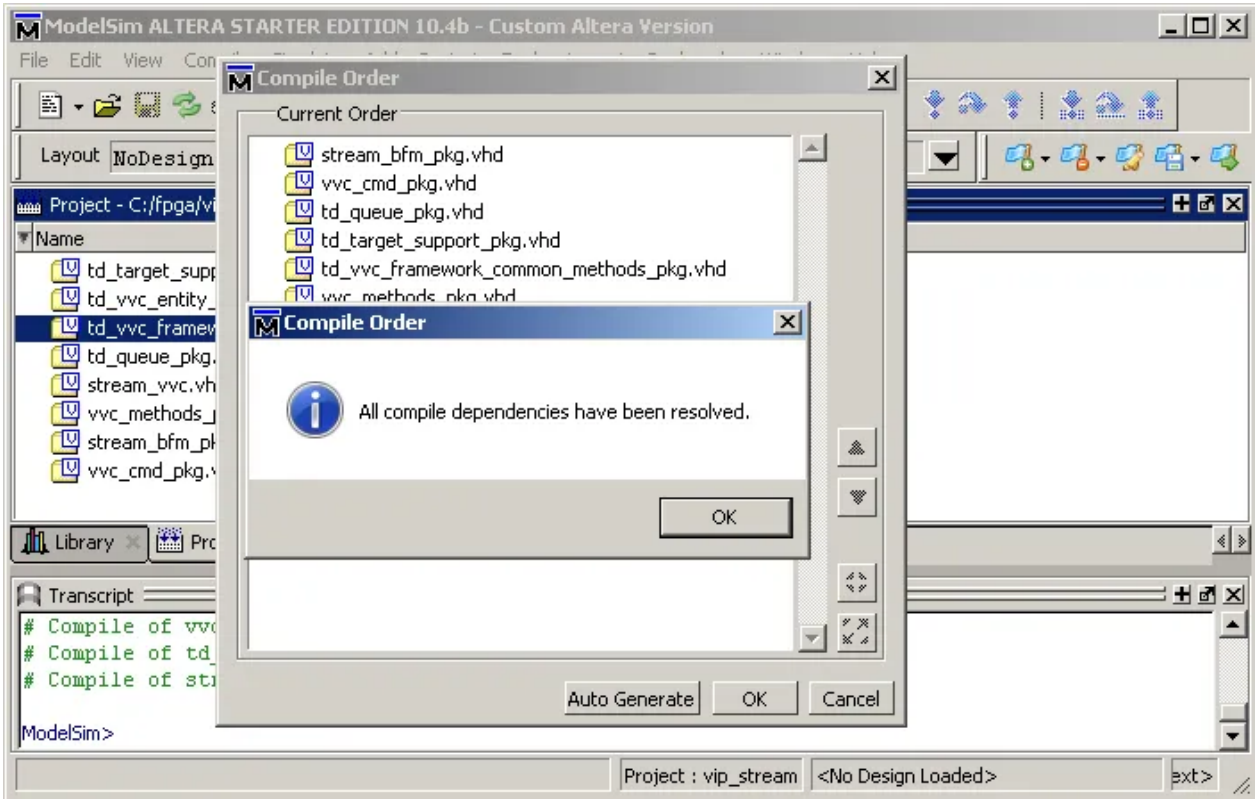
```
    tlast                : std_logic;
    dont_check           : boolean;
    called_as_source     : boolean;
    -- This includes data field, address field, constraints (e.g. timeout), etc.
    -- Example:
    -- data                 : std_logic_vector(C_VVC_CMD_DATA_MAX_LENGTH-1 downto 0);
    -- max_receptions       : integer;
    -- timeout              : time;
    -- Common VVC fields
    operation            : t_operation;
    proc_call            : string(1 to C_VVC_CMD_STRING_MAX_LENGTH);
    msg                  : string(1 to C_VVC_CMD_STRING_MAX_LENGTH);
    cmd_idx              : natural;
    command_type         : t_immediate_or_queued;
    msg_id               : t_msg_id;
    gen_integer_array    : t_integer_array(0 to 1); -- Increase array length if
needed
    gen_boolean          : boolean; -- Generic boolean
    timeout              : time;
    alert_level          : t_alert_level;
    delay                : time;
    quietness            : t_quietness;
  end record;
```

Observe that *tdata* has a width of C_VVC_CMD_DATA_MAX_LENGTH that we changed above. This is now the actual width of this parameter to be transferred to the BFM. The extra bits will have to be removed in the BFM later, using a normalization function.

Next, edit C_VVC_CMD_DEFAULT, which is a default record value to assign to instances of the record above:

```
  constant C_VVC_CMD_DEFAULT : t_vvc_cmd_record := (
    --<USER_INPUT> Set the fields you added to the t_vvc_cmd_record above to their
default value here
    tdata                => (others => '0'),
    tdata_width          => 0,
    tlast                => '-',
    dont_check           => false,
    called_as_source     => false,
    -- Example:
    -- data                 => (others => '0'),
    -- max_receptions       => 1,
    -- timeout              => 0 ns,
    -- Common VVC fields
    operation            => NO_OPERATION,
    proc_call            => (others => NUL),
    msg                  => (others => NUL),
    cmd_idx              => 0,
    command_type         => NO_COMMAND_TYPE,
    msg_id               => NO_ID,
    gen_integer_array    => (others => -1),
    gen_boolean          => false,
    timeout              => 0 ns,
    alert_level          => FAILURE,
    delay                => 0 ns,
    quietness            => NON_QUIET
  );
```

You are now done editing vvc_cmd_pkg.vhd. Save the file, go to ModelSim, Right-click in the files section, and select *Compile->Compile Order..* and press *Auto Generate*



Your project should compile without any errors. If that is not the case, correct the error(s) before continuing.

Modify stream_bfm_pkg.vhd

Un-comment the t_stream_if optional interface record and add signals required for our DUT interface.

```
1.  -- Optional interface record for BFM signals
2.  type t_stream_if is record
3.  --<USER_INPUT> Insert all BFM signals here
4.    tlast  : std_logic;
5.    tdata  : std_logic_vector;
6.    tvalid : std_logic;
7.    tready : std_logic;
8.  -- Example:
9.  -- cs      : std_logic;          -- to dut
10. -- addr    : unsigned;           -- to dut
11. -- rena    : std_logic;          -- to dut
12. -- wena    : std_logic;          -- to dut
13. -- wdata   : std_logic_vector;   -- to dut
14. -- ready   : std_logic;          -- from dut
15. -- rdata   : std_logic_vector;   -- from dut
16. end record;
```

Depending on the operating mode of our VVC (source or sink), these signals will have to be initialized with default values accordingly.

Modify the BFM configuration record like this, by enabling the *ID_FOR_BFM* record entry:

```
1.  -- Configuration record to be assigned in the test harness.
2.  type t_stream_bfm_config is
3.  record
4.    --<USER_INPUT> Insert all BFM config parameters here
5.    -- Example:
6.    -- max_wait_cycles          : integer;
7.    -- max_wait_cycles_severity : t_alert_level;
8.    id_for_bfm               : t_msg_id;
9.    -- id_for_bfm_wait          : t_msg_id;
10.   -- id_for_bfm_poll          : t_msg_id;
11.   clock_period             : time;  -- Needed in the VVC
12. end record;
```

This makes it possible later to provide a customized ID for *your* BFM, in order to filter out transaction log messages of no interest, and only enable logging of *your* BFM messages.

A few lines below, also modify the BFM configuration record default value. Also, change the *clock_period* to 0 ns. We can check for that in our BFM later and report an error if you did not set it explicitly to some other value.

```
1.  -- Define the default value for the BFM config
2.  constant C_STREAM_BFM_CONFIG_DEFAULT : t_stream_bfm_config := (
3.    --<USER_INPUT> Insert defaults for all BFM config parameters here
4.    -- Example:
5.    -- max_wait_cycles          => 10,
6.    -- max_wait_cycles_severity => failure,
7.    id_for_bfm               => ID_BFM,
8.    -- id_for_bfm_wait          => ID_BFM_WAIT,
9.    -- id_for_bfm_poll          => ID_BFM_POLL,
10.   clock_period             => 0 ns
11. );
```

Next, in the package body section, implement an DUT interface initialization function like this:

```
1.  function init_stream_if_signals (vvc_is_source  : boolean;
2.                                   vvc_data_width : natural;
3.                                   vvc_is_monitor : boolean) return t_stream_if is
4.    variable init_if : t_stream_if(tdata(vvc_data_width-1 downto 0));
5.  begin
6.    if vvc_is_source then
7.      init_if.tdata  := (others => '-');
8.      init_if.tlast  := '-';
9.      init_if.tvalid := '0';
10.     init_if.tready := 'Z';
11.   else
12.     init_if.tdata  := (others => 'Z');
13.     init_if.tlast  := 'Z';
```

```
    init_if.tvalid := 'Z';
    if vvc_is_monitor then
      init_if.tready := 'Z';
    else
      init_if.tready := '0';
    end if;
  end if;

  return init_if;
end function;
```

Remember to also add a declaration of this function in the package section:

```
function init_stream_if_signals (vvc_is_source  : boolean;
                                 vvc_data_width : natural;
                                 vvc_is_monitor : boolean) return t_stream_if;
```

Next we need to add our BFM procedure. For now, we will only add a *template* for it, and the come back an finish it later *after* we have our test harness and test bench up and running.

In the package body section, implement the BFM procedure like this:

```
--<USER_INPUT> Insert BFM procedure implementation here.
procedure wiggle (
  -- BFM generic parameters from VVC
  constant GC_VVC_IS_SOURCE               : in boolean;
  constant GC_VVC_CHECK_TLAST             : in boolean;
  constant GC_VVC_TDATA_WIDTH             : in positive;
  constant GC_VVC_IS_MONITOR              : in boolean;
  constant GC_VVC_VIOLATE_SRC_TVALID_RULE : in boolean;
  constant GC_VVC_CHECK_SRC_TVALID_SIGNAL : in boolean;
  constant GC_VVC_DONT_CHECK_JUST_REPORT  : in boolean;

  -- BFM dynamic execution parameters
  constant c_tdata            : in std_logic_vector;
  constant c_tdata_width      : in natural;
  constant c_tlast            : in std_logic;
  constant c_dont_check       : in boolean;
  constant c_called_as_source : in boolean;
  constant c_msg              : in string;

  -- BFM ports connecting to VVC ports
  signal clk            : in    std_logic;
  signal stream_if      : inout t_stream_if;

  -- BFM misc.
  constant scope        : in string              := C_SCOPE;
  constant msg_id_panel : in t_msg_id_panel      := shared_msg_id_panel;
  constant config       : in t_stream_bfm_config := C_STREAM_BFM_CONFIG_DEFAULT) is

begin
  log(ID_BFM, "=== I WAS HERE! : " & c_msg, scope, msg_id_panel);
end procedure; -- wiggle
```

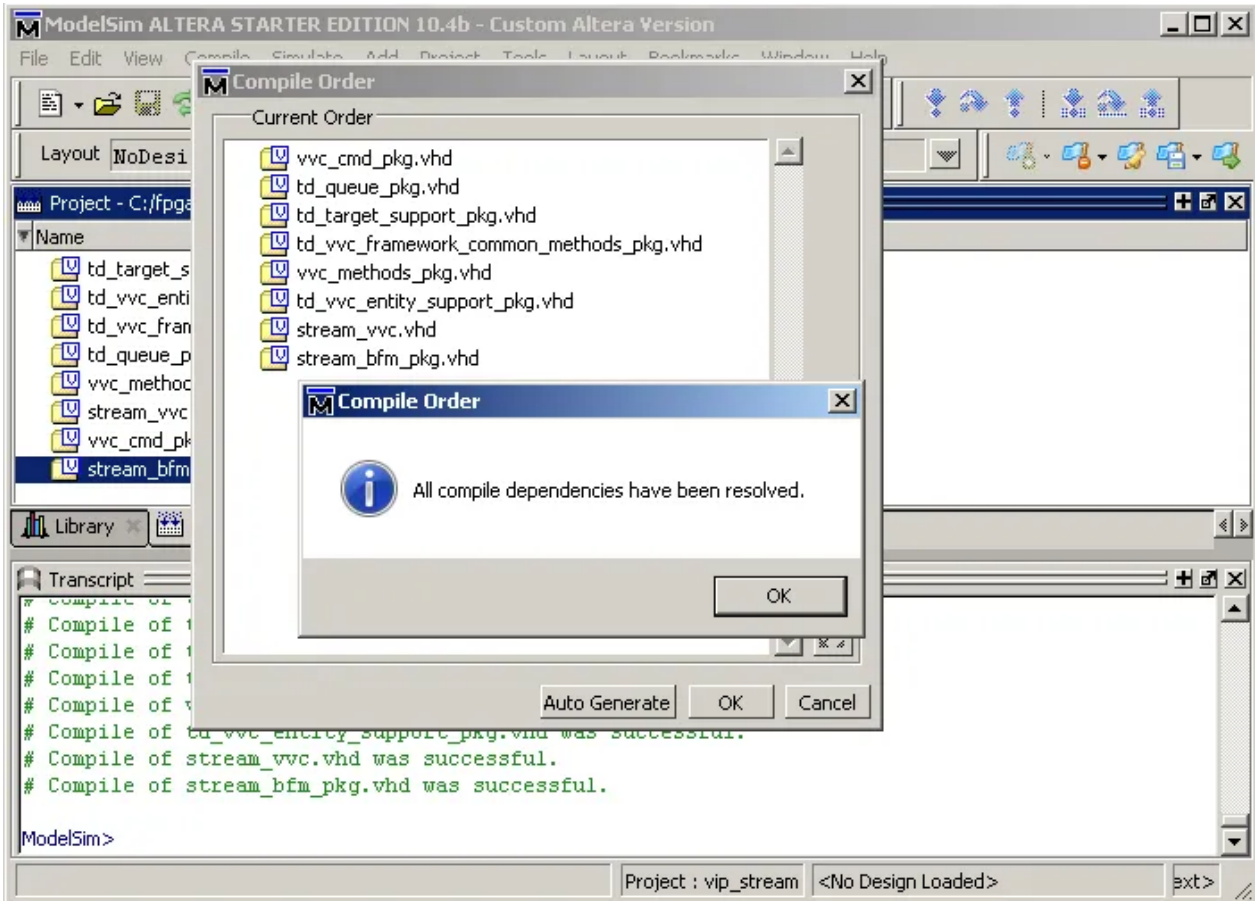
Observe that our BFM will accept all the parameters discussed above:

- all generic constants, inherited from the VVC instantiation.
- all dynamic parameters (c_*) forwarded from the CDM test bench calls.
- all signals connecting the VVC to our DUT.
- miscellaneous parameters, including the BFM configuration record which contains the *clock_period* parameter.

For now a call to this BFM will only return a log message. We will complete the BFM code later.

Remember to add the declaration of this procedure in the package section, just like we did with the initialization function above.

Finally – compile your files in ModelSim to check that everything compiles OK!



Modify stream_vvc.vhd

- Add the user specific generic constants to our VVC.
- Add the signals required for our DUT interface.
- Give the signals for the DUT interface a default value, using the initialization function we implemented earlier.

```
1.  entity stream_vvc is
2.    generic (
3.      --<USER_INPUT> Insert interface specific generic constants here
```

```
    GC_VVC_IS_SOURCE               : boolean;
    GC_VVC_TDATA_WIDTH             : natural;
    GC_VVC_CHECK_TLAST             : boolean := true;
    GC_VVC_IS_MONITOR              : boolean := false;
    GC_VVC_VIOLATE_SRC_TVALID_RULE : boolean := false;
    GC_VVC_CHECK_SRC_TVALID_SIGNAL : boolean := true;
    GC_VVC_DONT_CHECK_JUST_REPORT  : boolean := false;
    -- Example:
    -- GC_ADDR_WIDTH                          : integer range 1 to
C_VVC_CMD_ADDR_MAX_LENGTH;
    -- GC_DATA_WIDTH                          : integer range 1 to
C_VVC_CMD_DATA_MAX_LENGTH;
    GC_INSTANCE_IDX                        : natural;
    GC_STREAM_BFM_CONFIG                   : t_stream_bfm_config       :=
C_STREAM_BFM_CONFIG_DEFAULT;
    GC_CMD_QUEUE_COUNT_MAX                 : natural                    := 1000;
    GC_CMD_QUEUE_COUNT_THRESHOLD           : natural                    := 950;
    GC_CMD_QUEUE_COUNT_THRESHOLD_SEVERITY  : t_alert_level              := WARNING;
    GC_RESULT_QUEUE_COUNT_MAX              : natural                    := 1000;
    GC_RESULT_QUEUE_COUNT_THRESHOLD        : natural                    := 950;
    GC_RESULT_QUEUE_COUNT_THRESHOLD_SEVERITY : t_alert_level            := WARNING
  );
  port (
    --<USER_INPUT> Insert BFM interface signals here
    stream_vvc_if : inout t_stream_if := init_stream_if_signals (GC_VVC_IS_SOURCE,
                                                                 GC_VVC_TDATA_WIDTH,
                                                                 GC_VVC_IS_MONITOR);
    -- Example:
    -- stream_vvc_if               : inout t_stream_if :=
init_stream_if_signals(GC_ADDR_WIDTH, GC_DATA_WIDTH);
    -- VVC control signals:
    -- rst                         : in std_logic; -- Optional VVC Reset
    clk                         : in std_logic
  );
end entity stream_vvc;
```

Change the “if true” statement on line ~209 like this:

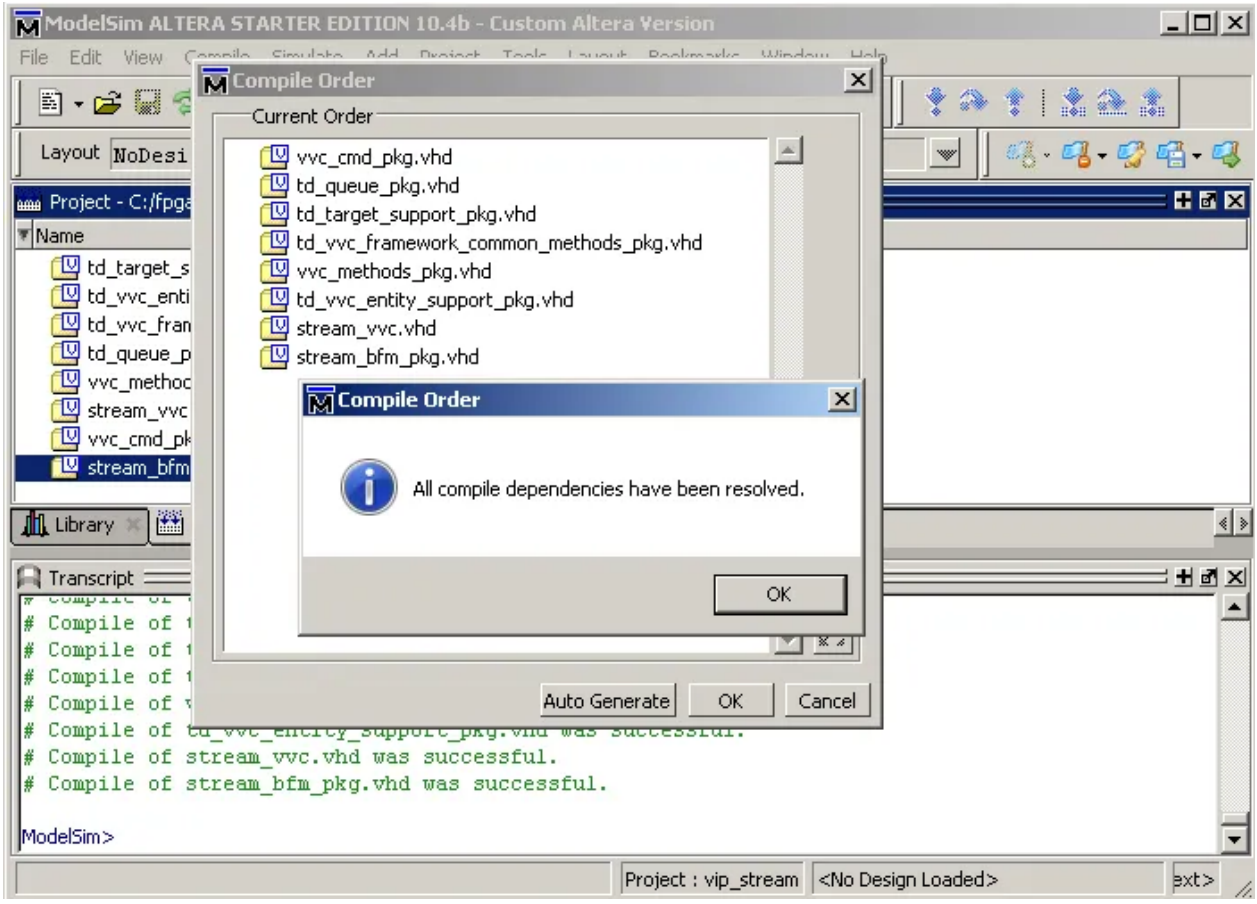
```
 -- Check if command is a BFM access
 --<USER_INPUT> Replace this if statement with a check of the current
v_cmd.operation, in order to set v_cmd_is_bfm_access to true if this is a BFM access
command
 -- Example:
 -- if v_cmd.operation = WRITE or v_cmd.operation = READ or v_cmd.operation = CHECK
or v_cmd.operation = POLL_UNTIL then

 if v_cmd.operation = CMD_STREAM then -- Replace this line with actual check
 v_command_is_bfm_access := true;
 else
 v_command_is_bfm_access := false;
 end if;
```

In the *case v_cmd.operation* statement on line ~229, add a condition to the *case* statement that will call our BFM:

```
--<USER_INPUT>: Insert BFM procedure calls here
```

```
when CMD_STREAM =>
  wiggle (GC_VVC_IS_SOURCE                => GC_VVC_IS_SOURCE              ,
          GC_VVC_CHECK_TLAST              => GC_VVC_CHECK_TLAST            ,
          GC_VVC_TDATA_WIDTH              => GC_VVC_TDATA_WIDTH            ,
          GC_VVC_IS_MONITOR               => GC_VVC_IS_MONITOR             ,
          GC_VVC_VIOLATE_SRC_TVALID_RULE => GC_VVC_VIOLATE_SRC_TVALID_RULE,
          GC_VVC_CHECK_SRC_TVALID_SIGNAL => GC_VVC_CHECK_SRC_TVALID_SIGNAL,
          GC_VVC_DONT_CHECK_JUST_REPORT  => GC_VVC_DONT_CHECK_JUST_REPORT ,
          c_tdata                         => v_cmd.tdata                   ,
          c_tdata_width                   => v_cmd.tdata_width             ,
          c_tlast                         => v_cmd.tlast                   ,
          c_dont_check                    => v_cmd.dont_check              ,
          c_called_as_source              => v_cmd.called_as_source        ,
          c_msg                           => format_msg(v_cmd)             ,
          clk                             => clk                           ,
          stream_if                       => stream_vvc_if                 ,
          scope                           => C_SCOPE                       ,
          msg_id_panel                    => vvc_config.msg_id_panel       ,
          config                          => vvc_config.bfm_config         );
```

Again – compile your files in ModelSim to check that everything compiles OK!



Modify vvc_methods_pkg.vhd

We will now define the CDM to be called from our test bench.

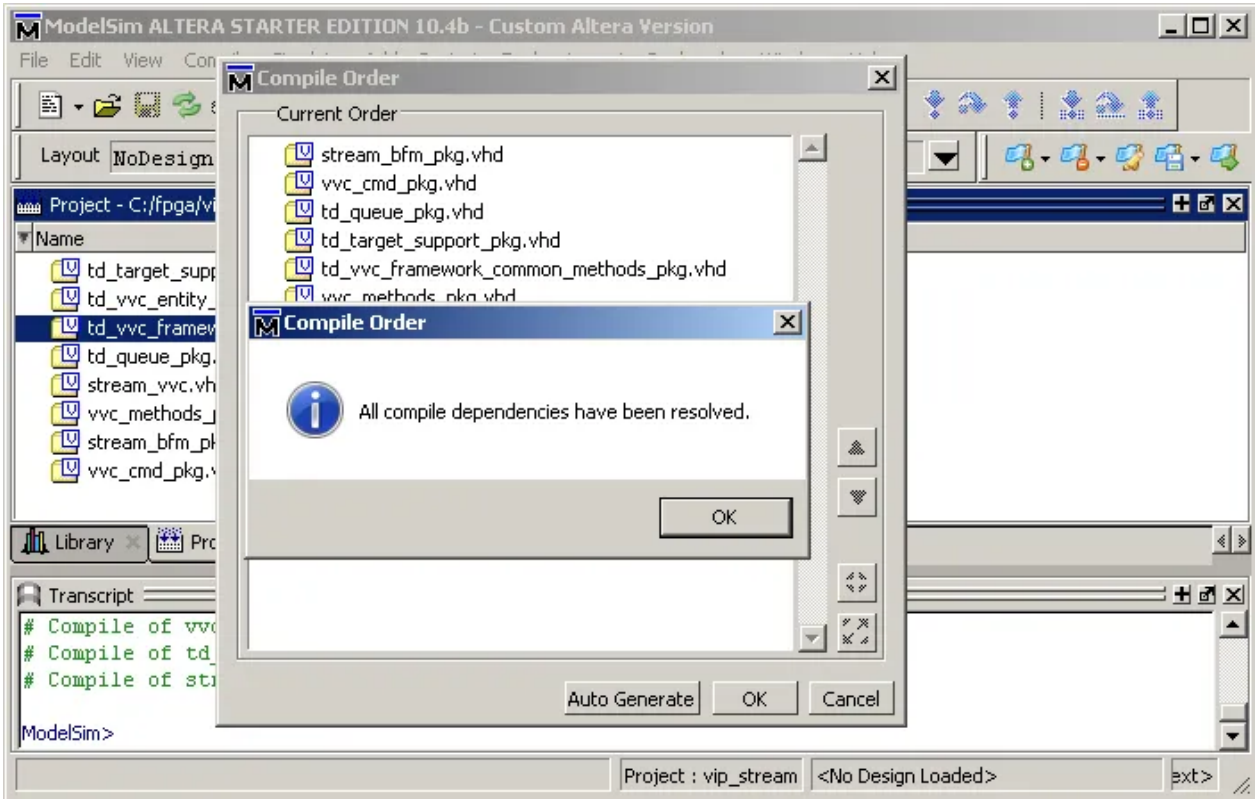
In *package body*, implement our *stream* CDM like this:

```
--<USER_INPUT> Please insert the VVC procedure implementations here.
procedure stream (
  signal   VVCT             : inout t_vvc_target_record;
  constant vvc_instance_idx : in integer;
  constant tdata            : in std_logic_vector;
  constant tlast            : in std_logic;
  constant msg              : in string;
  constant called_as_source : in boolean;
  constant dont_check       : in boolean
) is
  constant proc_name : string := get_procedure_name_from_instance_name(vvc_instance_idx'instance_name);
  constant proc_call : string := proc_name & "(" & to_string(VVCT, vvc_instance_idx) -- First part common for all
      & ", " & to_string(tdata, HEX, AS_IS, INCL_RADIX) & ")";
  variable v_normalised_data  : std_logic_vector(shared_vvc_cmd.tdata'length-1 downto 0)
    := normalize_and_check(tdata, shared_vvc_cmd.tdata, ALLOW_NARROWER, "tdata", "shared_vvc_cmd.tdata",
    proc_call & " called with to wide tdata. " & add_msg_delimiter(msg));
begin
  set_general_target_and_command_fields(VVCT, vvc_instance_idx, proc_call, msg, QUEUED, CMD_STREAM);
  shared_vvc_cmd.tdata            := v_normalised_data;
  shared_vvc_cmd.called_as_source := called_as_source;
  shared_vvc_cmd.tdata_width      := tdata'length;
  shared_vvc_cmd.tlast            := tlast;
  shared_vvc_cmd.dont_check       := dont_check;
  send_command_to_vvc(VVCT);
end procedure;
```

Also – remember to add a declaration of the CDM in the *package* section.

```
--<USER_INPUT> Please insert the VVC procedure declarations here
procedure stream (
  signal   VVCT             : inout t_vvc_target_record;
  constant vvc_instance_idx : in integer;
  constant tdata            : in std_logic_vector;
  constant tlast            : in std_logic;
  constant msg              : in string;
  constant called_as_source : in boolean;
  constant dont_check       : in boolean);
```

Compile your files in ModelSim to check that everything compiles OK.



Congratulations – we now have a skeleton for our *stream* VVC that compiles OK!

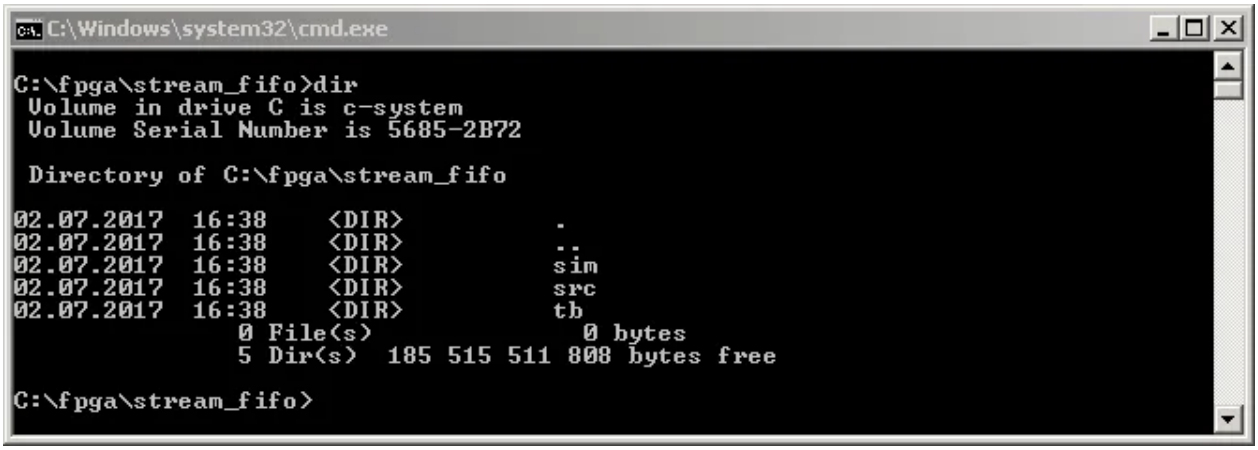
What remains to be done:

1. Create new ModelSim project for our DUT.
2. Create a test harness for our DUT.
3. Create a test bench for our DUT.
4. Modify the BFM to actually do something useful.
5. Overload the *stream* CDM with simpler and more intuitive version.

Part 4

Create a new ModelSim project for our DUT.

Create a new directory for our DUT called stream_fifo, and then create the *src*, *tb* and *sim* sub directories.

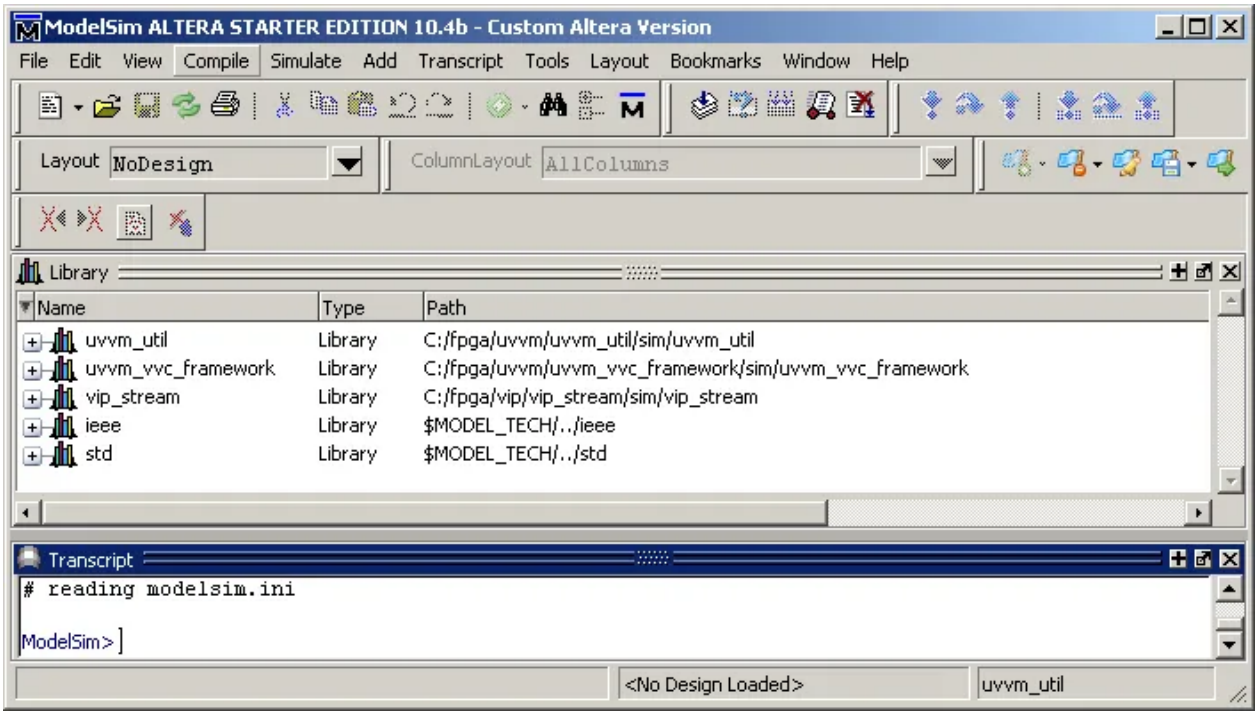


Copy the DUT source files to the *src* directory.

Copy the *modelsim.ini* file we created earlier to the *sim* directory, and then add our *vip_fifo* to the [Library] section of *modelsim.ini*:

```
[Library]
; STANDARD LIBRARIES
std                 = $MODEL_TECH/../std
ieee                = $MODEL_TECH/../ieee

; UVVM libraries
uvvm_util           = C:/fpga/uvvm/uvvm_util/sim/uvvm_util
uvvm_vvc_framework  = C:/fpga/uvvm/uvvm_vvc_framework/sim/uvvm_vvc_framework
vip_stream          = C:/fpga/vip/vip_stream/sim/vip_stream

[vcom]
VHDL93 = 2008

[vsim]
Resolution = ps
UnbufferedOutput = 1
```

In ModelSim, change directory to our new *sim* directory and notice the existence of our new vip_stream library:



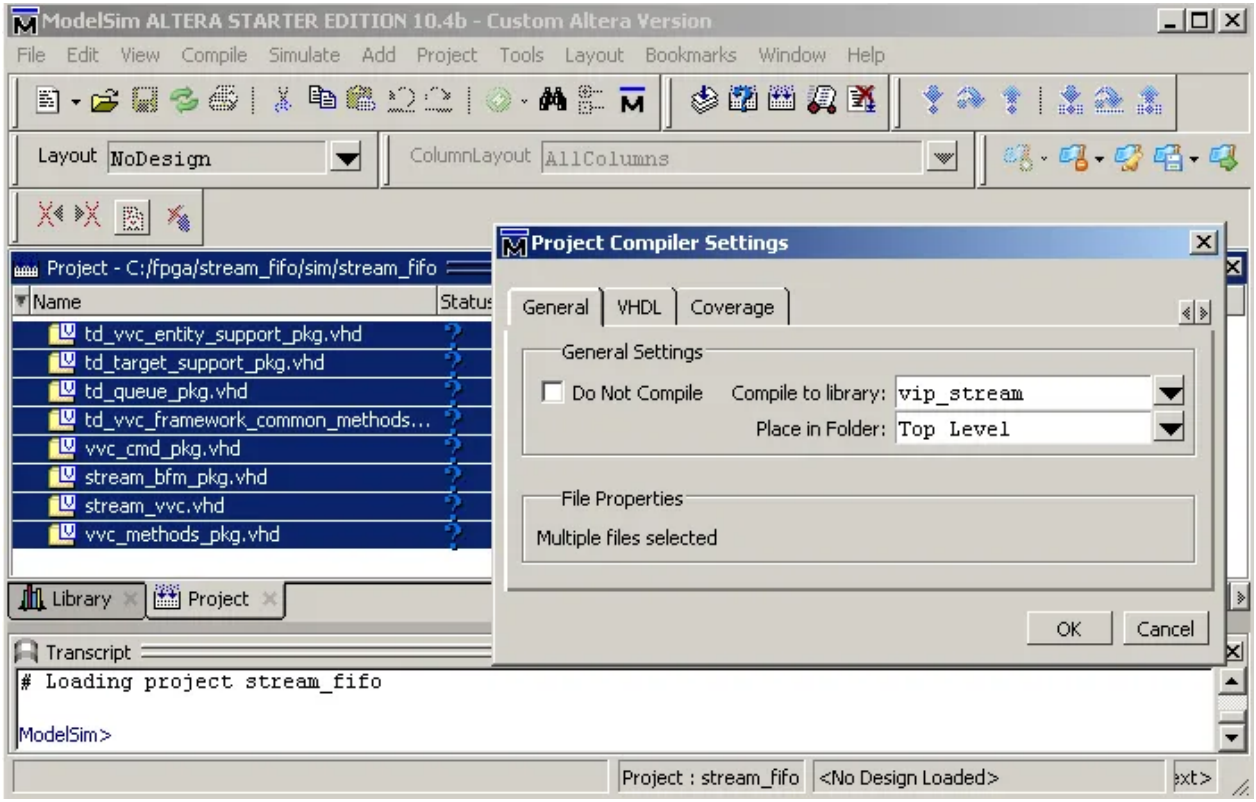
Create a new ModelSim project called *stream_fifo*. We will use this project to:

- Work on the test harness and test bench for for the *stream_fifo.*

- Update and add functionality to our vip_stream VVC.

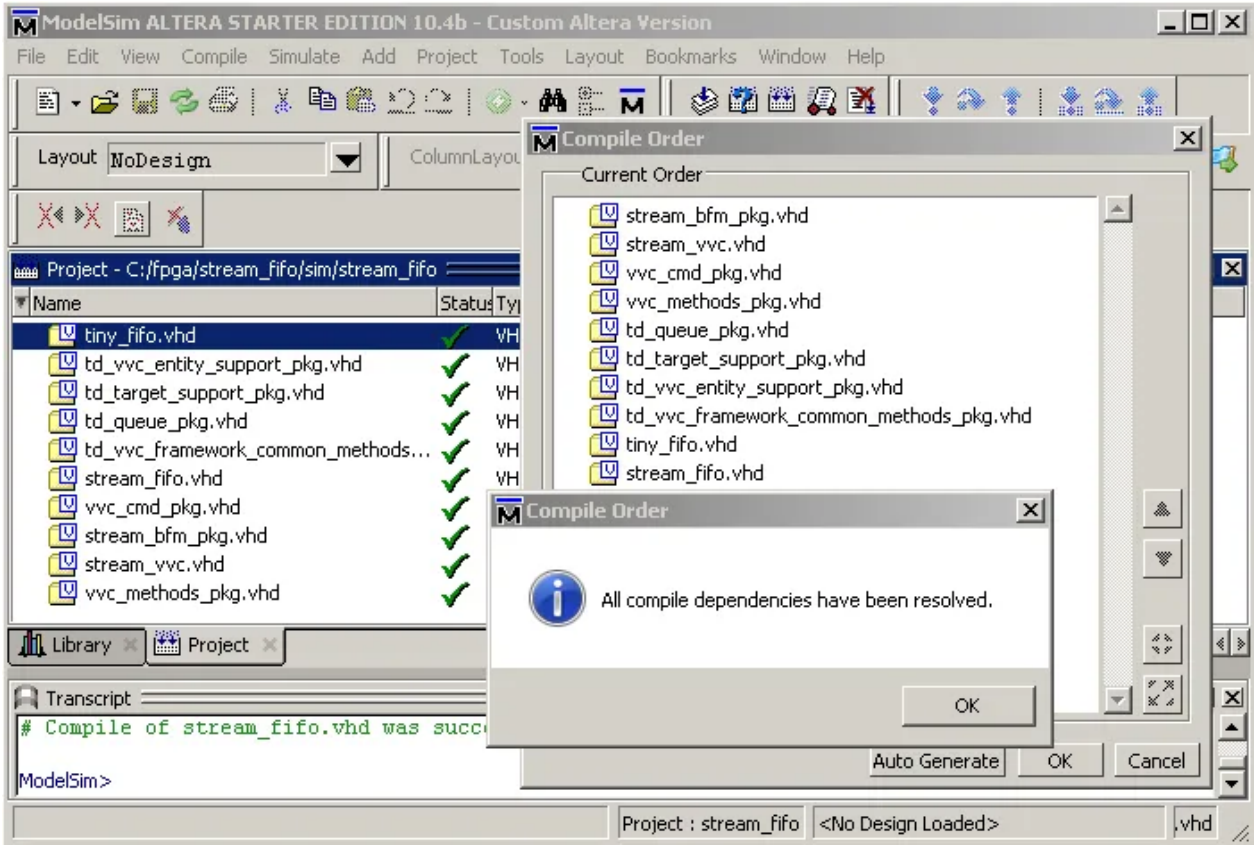
First – add all the files that we worked on earlier. Mark all of them – then right-click and choose *Properties.*

For these files, change the *Compile to library* to *vip_stream*:



Next, add our DUT files *tiny_fifo.vhd* and *stream_fifo.vhd*. They will compile to default library work, so no need to change properties for them.

Right-click, choose Compile->Compile Order… and press Auto Generate. Everything should compile OK like this:

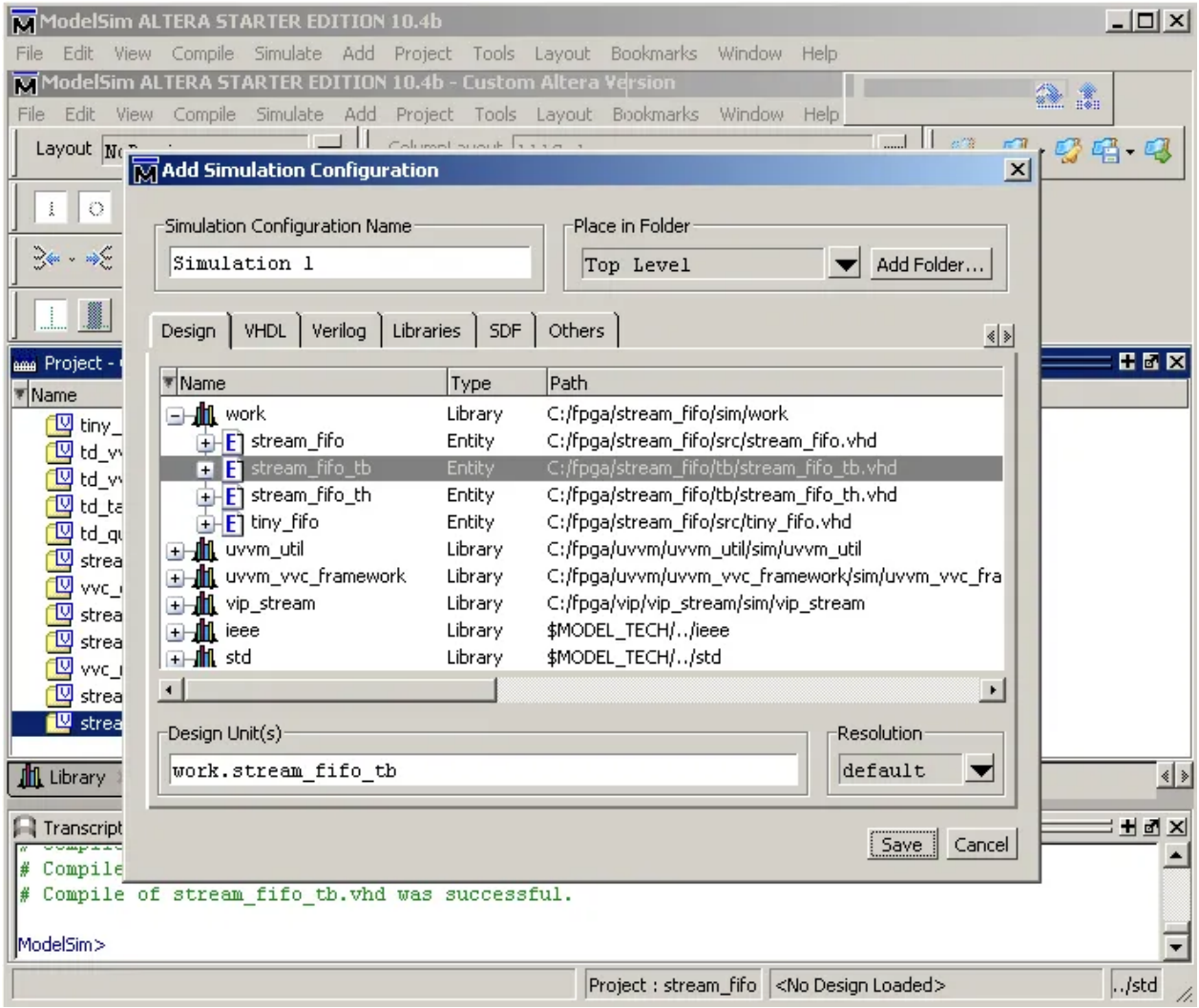


Create a test harness and test bench for your DUT

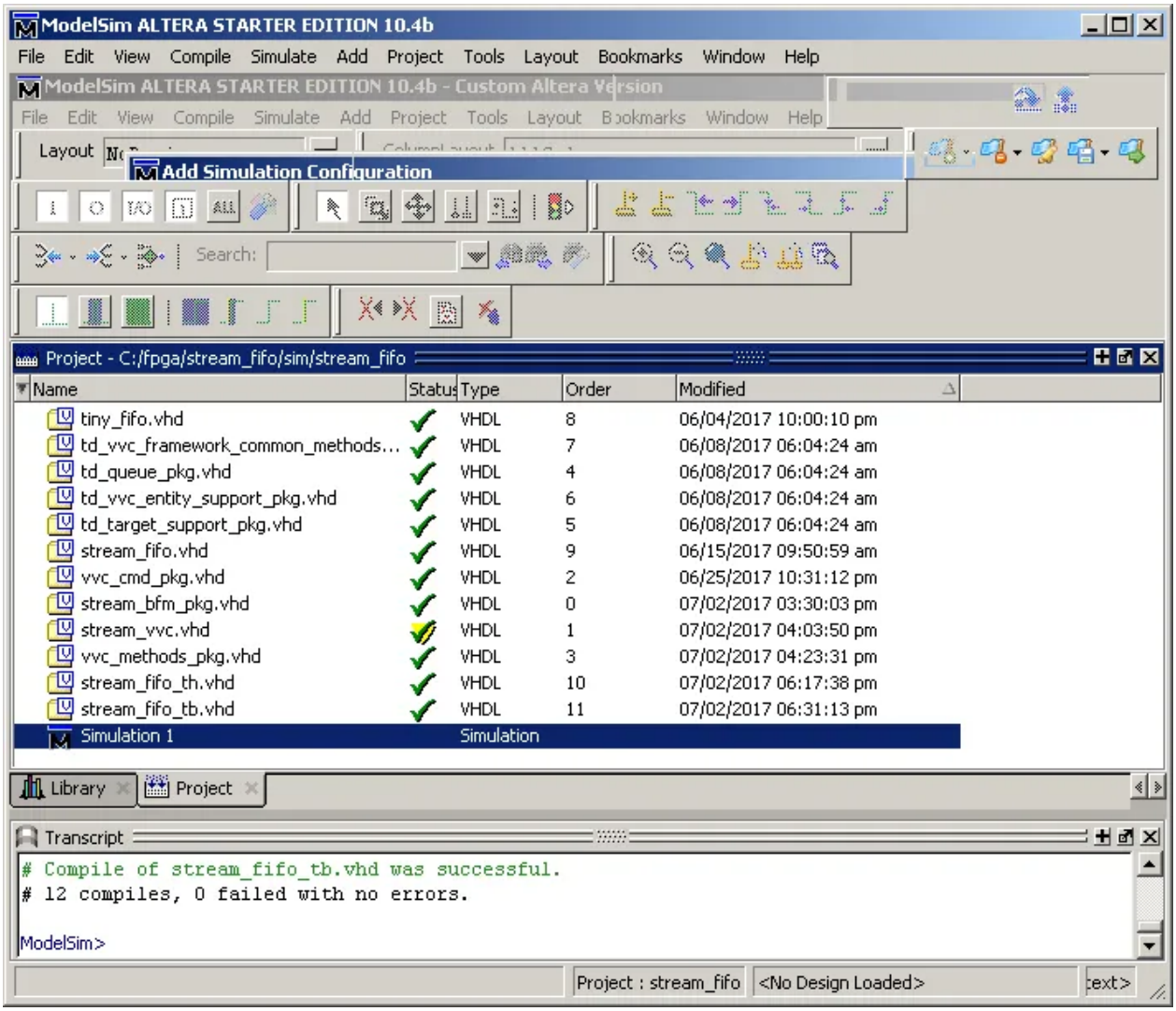
A test harness is provided for this tutorial, copy *stream_fifo_th.vhd* into the *tb* directory. A test bench is also provided, copy *stream_fifo_tb* into the *tb* directory also.

Add these files to the ModelSim project, and compile again to make sure everything compiles correctly.

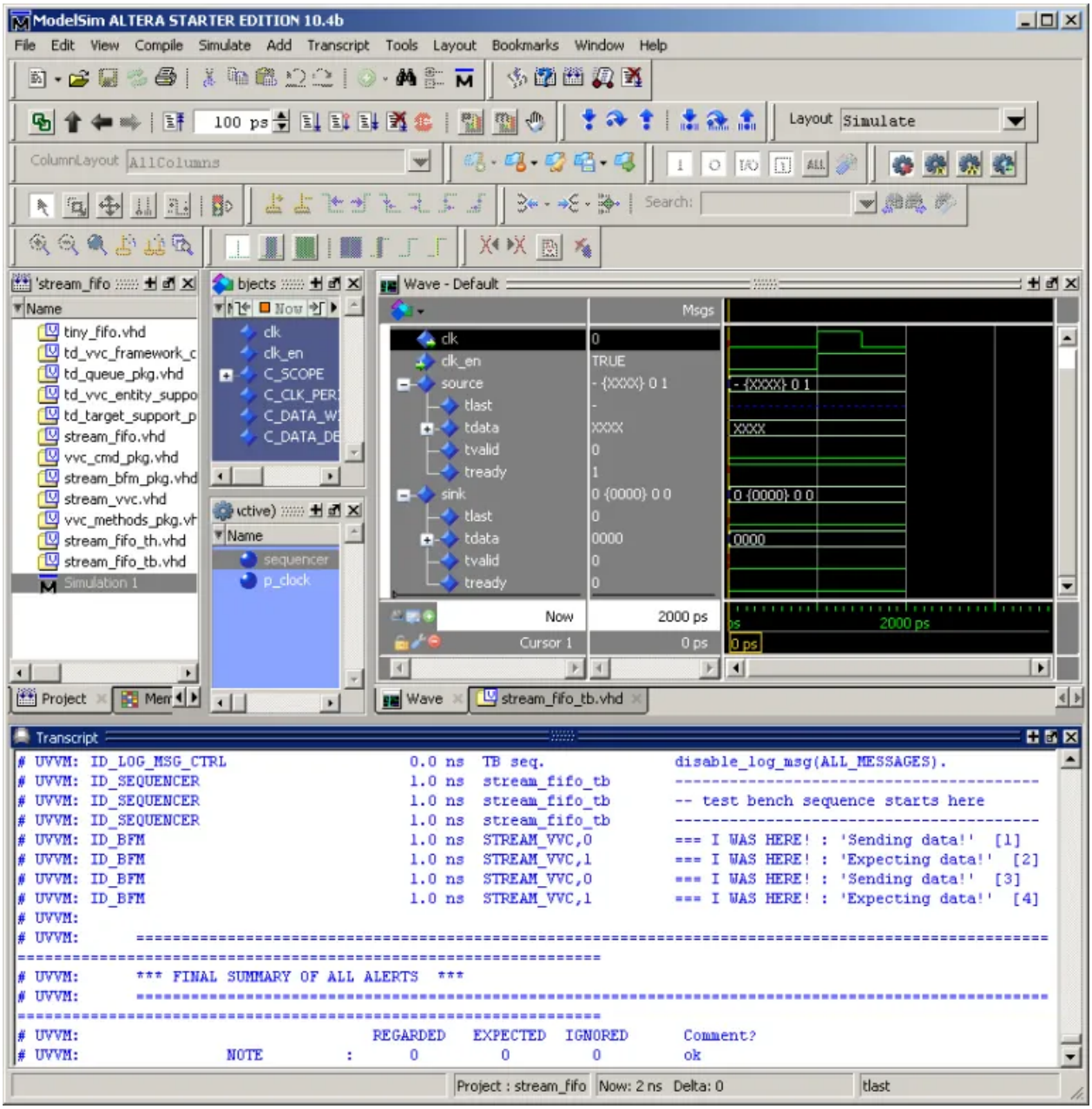
Again – right-click in the files window, choose *Add to Project -> Simulation Configuration*. Then select *stream_fifo_tb* as the simulation target. Press *Save*.



To start the simulation, you can now double-click *Simulation 1*.



Add the signals of interest to the waveform viewer, and type in *run -all* to run the simulation and observe the output.



Note the following:

- The test bench ran successfully, because no errors were reported from the BFM.
- The BFM was called multiple times – printing “I WAS HERE”.
- The BFM wasn’t really doing anything, because we have not implemented any proper functionality there yet.

Review *stream_fifo_th* and *stream_fifo_tb*

Review these files and consult the UVVM documentation, and make sure that you understand everything that is going on.

In *stream_fifo_th*, note the following:

- At time zero, processes within UVVM sets the BFM configuration record to a default value of C_STREAM_BFM_CONFIG_DEFAULT. To override this, we need to let UVVM set this default value

first, and then after a delta time delay (*wait for 0ns;*) we can change this parameter to our own value. Within process *p_th*, without the *wait for 0ns* statement, the result of assigning the BFM configuration records would be unpredictable.
- The *p_th* process in the test harness is used to loop through all the instantiated VVCs, and set the *clock_period* parameter in all BFM configuration records. Remember to use a *wait for 0 ns* statement first!
- In *p_th*, try to comment out the inner loop, and watch the simulation log. You will see that the log is crowded with *lots* of messages that are of no interest. By setting these messages to *DISABLED*, and only enable the *ID_BFM* message, your UVVM log will be a lot less crowded and much more readable.
- If you like, the *p_th* process which configures the BFM configuration record may be included in the test bench instead. I prefer to keep it in the test harness, close to the VVC instantiations.
- The instantiation of *i_ti_uvvm_engine* is mandatory.
- The instantiation of *p_clock* is a convenient way to generate a clock.

In *stream_fifo_tb*, note the following:

- The *disable_log_msg* and *enable_log_msg* calls relates to the *UVVM Utility Library* and not the *UVVM VVC Framework*. This is somewhat confusing, but leaving these as shown ensures that you don't clutter up your simulation log. Change these if you want.
- The *await_uvvm_initialization* procedure call is mandatory.

Obviously, all of these files need better commenting! I will leave that to you! 😉

At last – look at how simple and clean the test harness and test bench is. You can now do a lot more with less code! And most importantly: Most of the content of the test bench and test harness can be reused for other projects, just copy them and make modifications as necessary!

Modify the BFM to do something useful

In version 1.0 of the *wiggle* BFM, the following checks has been implemented:

- Check if *clock_period* = 0 ns. If this is the case, you forgot to assign *clock_period* and the test bench will stop with an error.
- In the *stream* CDM call, a boolean determines if the operation is a write or a check(read). This must match the mode of the VVC, *GC_VVC_IS_SOURCE*, otherwise the test bench will stop with an error.
- Unless *GC_VVC_DONT_CHECK_JUST_REPORT* is true or *c_dont_check* = true, then the width of the CDM data parameter is compared to *GC_VVC_TDATA_WIDTH*. If they do not match, the test bench will stop with an error.
- If *GC_VVC_IS_SOURCE* = false and *GC_VVC_IS_MONITOR* is true, then the VVC will not drive the *tready* signal, only monitor it. This way, you may have multiple DUTs instantiated in your test harness, with monitor VVCs just monitoring the signals going between them.
- If *GC_VVC_IS_SOURCE* = false and *GC_VVC_DONT_CHECK_JUST_REPORT* is true or *c_dont_check* = true, then the value read from the DUT will not be checked, only reported.
- If *GC_VVC_IS_SOURCE* = false and *GC_VVC_CHECK_TLAST* = true, then *tlast* will be checked for correctness, otherwise it will be ignored.

In a future version, I intend to include a protocol checker that utilizes the *GC_VVC_VIOLATE_SRC_TVALID_RULE* and *GC_VVC_CHECK_SRC_TVALID_SIGNAL* generic parameters.

Overload the *stream* CDM with simplified versions

The *stream* CDM is not intuitive. It was implemented in this way in order to have a single CDM that does it all. In order to make more intuitive CDMs, we need to overload the *stream* CDM.

In vvc_methods_pkg.vhd, I have created the following overloads of *stream*:

- *stream_write* (VVCT, vvc_instance_idx, tdata, tlast, msg)
- *stream_write* (VVCT, vvc_instance_idx, tdata, msg)
- *stream_expect* (VVCT, vvc_instance_idx, tdata, tlast, msg)
- *stream_expect* (VVCT, vvc_instance_idx, tdata, msg)
- *stream_read* (VVCT, vvc_instance_idx, tdata, tlast, msg)
- *stream_read* (VVCT, vvc_instance_idx, tdata, msg)
- *stream_read* (VVCT, vvc_instance_idx, msg)

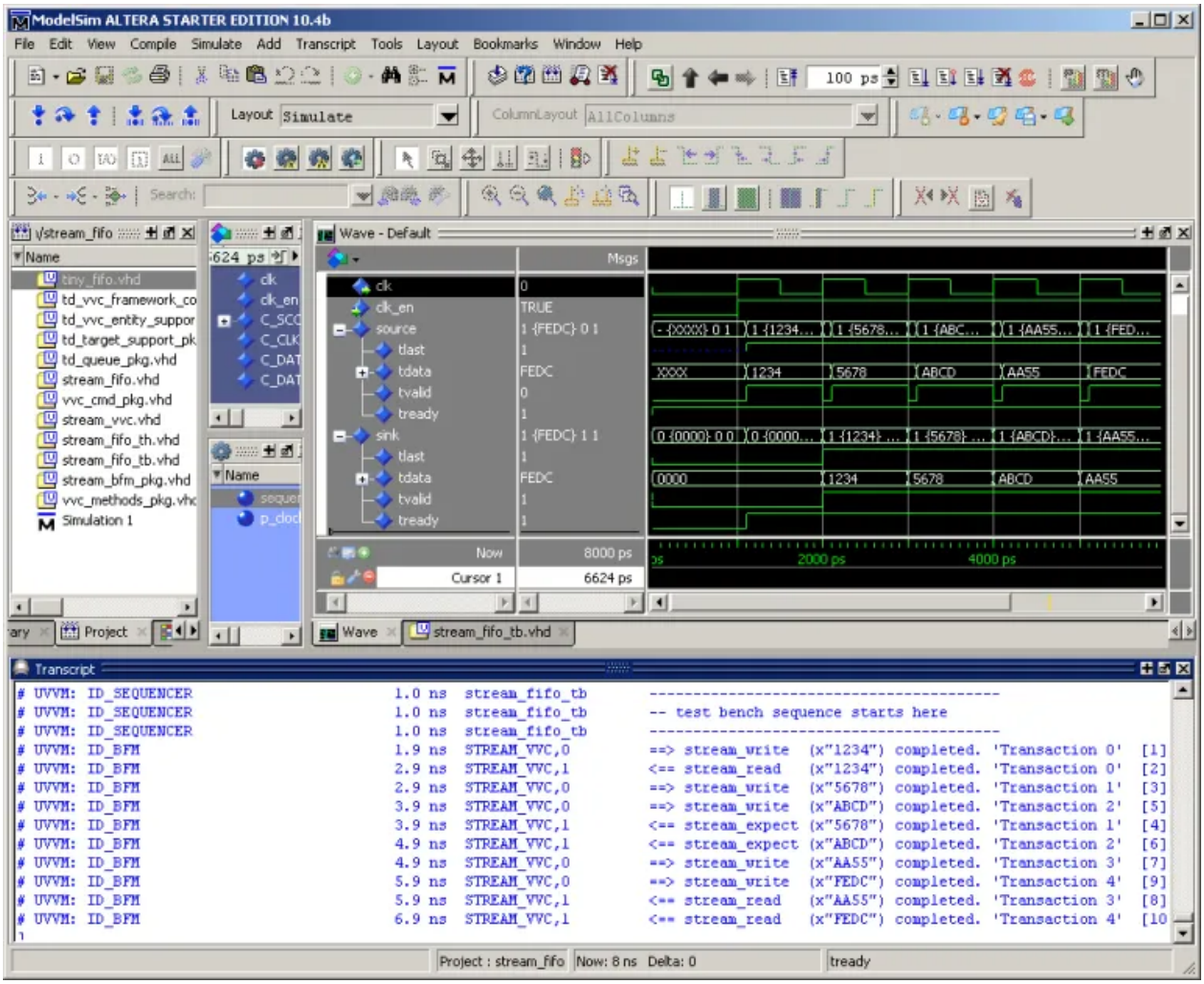
Sometimes, you may want to check an output value, without checking it for actual correctness. This is the purpose of the *stream_read* overload.

For *stream_read*, then *tdata* and *tlast* parameters are ignored. These overloads are provided as a quick way of converting *stream_expect* statements to *stream_read* statements without having to change the parameters. Then they will be quick to change back to *stream_expect* later after the parameters have been verified. The same thing thing can be achieved with the GC_VVC_DONT_CHECK_JUST_REPORT generic parameter.

Summary

In the final simulation, notice:

- The density of transactions in the waveform viewer. All DUT interfaces operates with true concurrency.
- The transaction log in the Transcript window, in particular that transactions on the DUT interfaces occurs in a different order than the CDMs that invoked them.



That's it! A .zip file containing the files of this tutorial is provided here:

UVVM_Tutorial_v1.0

Please report any errors, and I will fix them. Enjoy.

Share this:

This entry was posted in FPGA - Field Programmable Gate Array on July 3, 2017 [http://que.no/index.php/2017/07/03/uvvm-tutorial/] .