ELE00011H Digital Engineering
ELE00121M Digital Engineering for MSc

Laboratories: Session 4

Testing and verification

UG Students: **- ALL LABS SHOULD BE DONE IN GROUPS OF TWO STUDENTS**
- A mark penalty will apply to single-student submissions unless agreed in advance
- Any issues related to groups (conflicts) should be communicated as soon as possible

MSc Students: **- ALL LABS SHOULD BE INDIVIDUAL SUBMISSIONS**

Report formatting:

There is no formal report structure – you will be marked on the items listed within each lab script. Most reports will include code printout and simulation screenshots – see Lab 1 appendices for guidelines.

The reports for labs 1 - 4 must be handed in, **together in a single zip archive**, via the VLE by the deadline indicated on the front page of the script. A single submission should be handed in for each group (by any member of the group).

Each lab report, containing all the material required in the order specified, should be submitted as a **separate PDF file**. The exam numbers of all members of the group should be printed on the front page of each PDF.

The PDF files must be named **Yxxxxxxx-Yxxxxxxx_DE_Lab#.pdf**, where the Yxxxxxxx are the exam numbers of the group members (normally two for UG students, one for MSc students) and # is the lab number.

In all cases, read carefully the instructions on the VLE submission page. Failure to follow the instructions could lead to your assignment not being marked and in any case to a mark penalty.

Submission weight on module mark: 25% (50 marks)

Task 1: Test Pattern Generation [28 marks]

The aim in this task is to define a minimal (or almost minimal) set of test patterns to exhaustively test the circuit of Figure 1 for detectable single stuck-at faults.

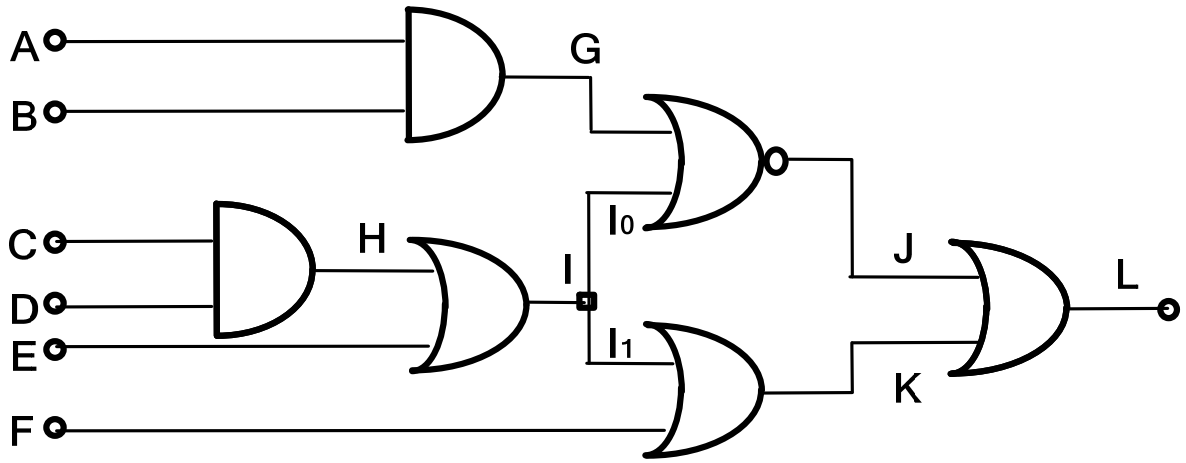


Figure 1

The procedure to be followed consists of the following steps:

Step 1: For each kind of logic gate in the circuit (AND, OR, NOR), perform fault collapsing to identify equivalent and dominated faults.

3.1.1: For each kind of logic gate in the circuit (AND, OR, NOR), specify equivalent and dominated/dominant faults. Show your derivation.

Step 2: Consider the remaining (non-dominated and non-equivalent) faults for each gate of the circuit of Figure 1. Considering that a fault on the output of a gate is equivalent to the fault on the input of the following gate, and considering the *fault collapsing* rules (see lecture slides and results of 3.1.1), this should lead to a significant reduction of the fault list.

The complete list of possible faults is:

Node A s-a-0	Node A s-a-1	Node B s-a-0	Node B s-a-1	Node C s-a-0	Node C s-a-1
Node D s-a-0	Node D s-a-1	Node E s-a-0	Node E s-a-1	Node F s-a-0	Node F s-a-1
Node G s-a-0	Node G s-a-1	Node H s-a-0	Node H s-a-1	Node I s-a-0	Node I s-a-1
Node I_0 s-a-0	Node I_0 s-a-1	Node I_1 s-a-0	Node I_1 s-a-1	Node J s-a-0	Node J s-a-1
Node K s-a-0	Node K s-a-1	Node L s-a-0	Node L s-a-1		

Note: Due to fanout, I_0 and I_1 will have to be considered as *different* from I. This is because a fault could affect only one of the two branches (metal lines) of the fanout. For example, a fault I_0 s-a-1 would cause the input to the NOR gate (I_0) to be stuck at Vcc, but both I and I_1 would be operating correctly. In practice, you can postulate the presence of a "gate" where I splits into I_0 and I_1. The fault collapsing rules for this "gate" are that there is NO equivalence or dominance between the three lines.

3.1.2: List all non-equivalent and non-dominated faults for the circuit of Figure 1 (i.e. starting from the full list above, eliminate all equivalent and dominated faults). Show your work by identifying the reason for the eliminations (e.g., X s-a-0 is equivalent to Y s-a-1)

Step 3: For each fault on the list in step 2, use the D algorithm to define a test pattern that will detect that fault at the output L, or to show that it is undetectable. In the report, show your work using a table of the form (see example in module slides):

J s-a-1	**A**	**B**	**C**	**D**	**E**	**F**	G	H	I	I_0	I_1	J	K	**L**
Step 1												D′		
Step 2								1	X	X	X	D′		
Step 3	1	1						1	X	X	X	D′		
...														

3.1.3: List the test patterns (A-F inputs and expected output L, in this order) required to detect all detectable faults in the circuit and how they were derived using the D algorithm. Indicate any undetectable faults separately and show how you determined that they cannot be detected. At this stage, "don't care" input values should be labelled as 'X'.

Step 4: Compare the test patterns and determine whether any of the patterns can be merged. Two patterns can be merged when all resolved inputs (i.e., non-X) are equal. This will not necessarily be the case, depending on your fault list and the application of the D algorithm (in which case, you should specify that this is the case). Note that it might be useful to keep track of alternatives in the D algorithm output (e.g. 0-X or X-0 for C and D).

3.1.4: Indicate which test patterns in the list above can be merged (if any).

3.1.5: List the reduced test patterns required to exhaustively test the circuit. At this stage, all "don't care" values should be resolved to 0s and D/D' to 0s and 1s. Hint: at this stage, you should have reduced the set to 7 or 8 patterns.

3.1.6: For each test pattern, perform *fault simulation* to determine all faults that the pattern is able to detect. In other words, determine which of the faults on the original list (step 2) are detected by each of the vectors you found at 3.1.5. Use a table and list the faults for each vector alphabetically.

Task 2: Built-In Self-Test (BIST) [12 marks]

All files for this experiment are contained in the .zip archive available in the module website. If you extract this folder within a directory that contains no spaces, you should be able to work directly within the directory by opening the .xpr file.
The project contains a set of VHDL files that implement the circuit of Figure 1 with a fault injection mechanism that allows the simulation of the following faults:

- Node E stuck-at-1
- Node H stuck-at-0
- Node F stuck-at-0

The block diagram of the circuit is the following:

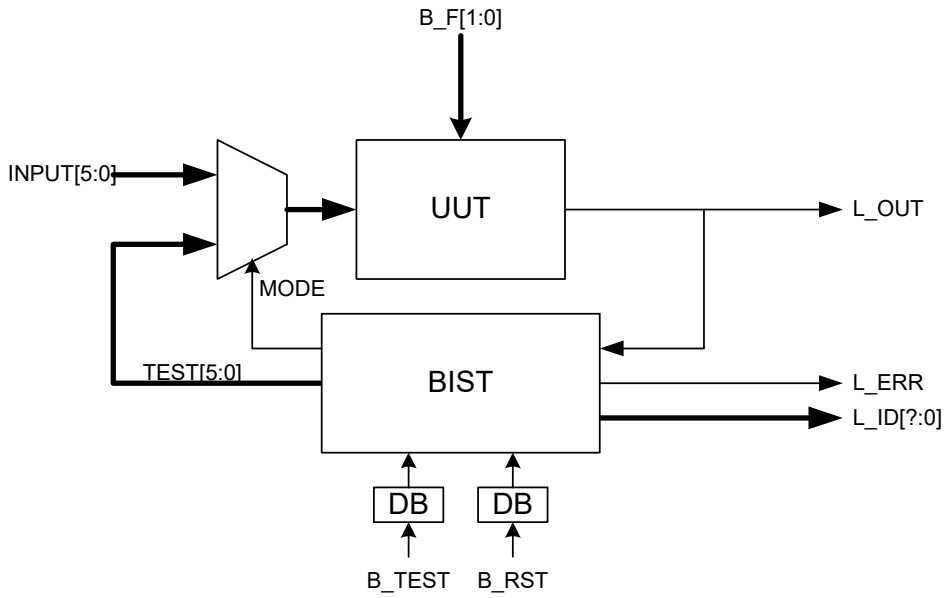


Figure 2

In practice, this circuit exploits some of the techniques described in the lecture for the test and verification of combinational circuits.

- The bus INPUT[5:0] carries the "standard" inputs to the logic circuit, inputs that are *de-gated* by the multiplexer when in test mode.
- L_OUT is the output of the combinational logic.
- B_F[1:0] is a 2-bit bus that encodes the fault be injected (00 for no fault, 01 for E s-a-1, 10 for H s-a-0, 11 for F s-a-0).
- B_TEST is a (de-bounced) signal driven by a push-button. When the push-button is pressed, the circuit will "lock" the fault set by B_F and enter test mode (see below).
- B_RST will reset the test logic (injected fault, state of the test).
- L_ERR is a single-bit output that should have value 1 if the test logic has detected a fault in the circuit. It should then remain 1 until the test logic is reset.
- L_ID will represent a unique ID for the test pattern that has allowed the fault to be detected.

Examine the internal logic of the BIST system. When the B_TEST button is pressed, the UUT will be *de-gated* from its standard inputs and subjected to the set of test patterns you have derived in task 1. If no fault is detected, the unit should then resume normal operation. The internal block diagram of the BIST unit is shown in Figure 3.

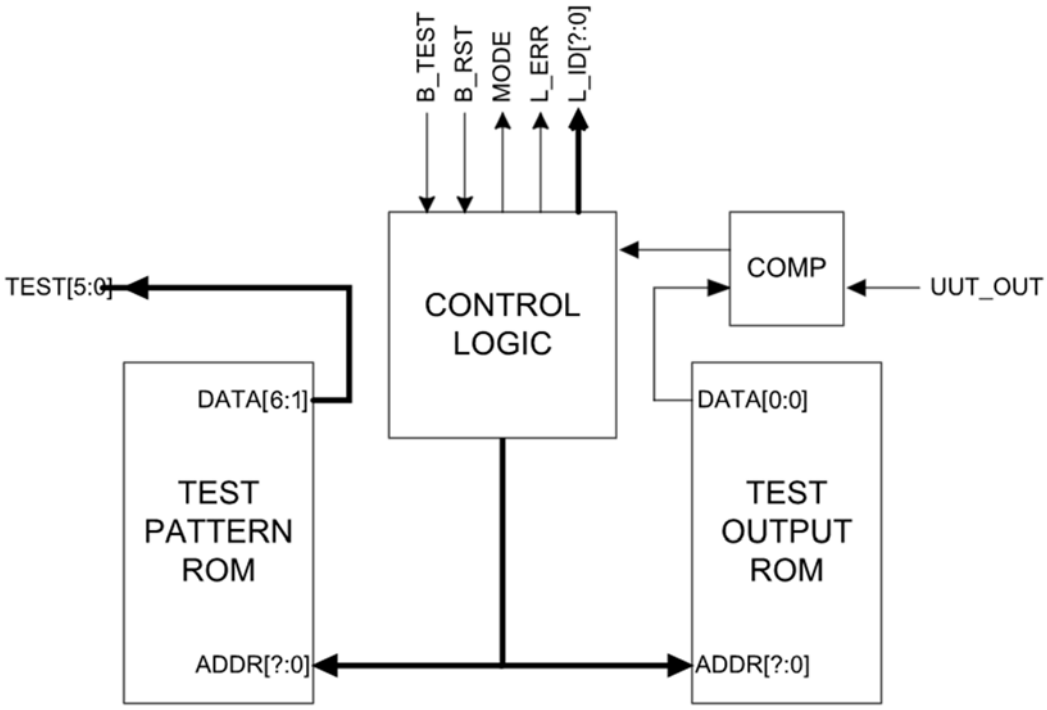


Figure 3

Your task will be to define and instantiate a *single-port distributed RAM using an IP component* (see Appendix) to store the test patterns defined in task 1. Two important things to note:

1. The figure above, with two separate memories to store inputs and expected outputs, is a functional description only: you should not instantiate two RAMs, but only a single one, storing the output as the LSB of a 7-bit data word.
2. Normally, a ROM would be the most appropriate memory type for this circuit, since its contents do not need to be modified. However, a bug in the Xilinx tools seems to affect ROMs, and therefore you should use a RAM with the write enable and data inputs set to 0.

If a fault is detected, the circuit freezes, the L_ERR output signals that a fault has been detected (value 1) and the L_ID output shows the ID of the test pattern that detected the fault (in practice, its address within the memory). The B_RST button is then used to reset the system to the original state.

Define a testbench that will verify the operation of the circuit, including all possible fault scenarios given the three faults specified above (remember that only a single fault can be injected each time).

3.2.1: Print out the memory content file.

3.2.2: Can you explain why the minimum depth of a Xilinx distributed RAM is 16?

3.2.3: Print out the VHDL testbench.

3.2.4: Print out screenshots of the behavioural simulation for the system. The screenshot(s) should show, in readable format, all inputs and outputs of the circuit as well as the internal signal MODE and the address and output data of all internal memories, for:

- **The operation of the fault-free circuit for at least two different combinations of inputs**
- **The complete test cycle when no fault is present**
- **The complete test cycle when each of the three faults is present**

Task 3: System-level IP – the Digital Clock Manager and Chipscope [10 marks]

In this task, you will implement the design above in the Xilinx boards, use the Digital Clock Manager to control its operational frequency and then observe its operation within the device using Chipscope.

Part I: the Digital Clock Manager

Start the process by creating an XDC file that defines the pins to be used for the different signals. For this circuit, the following XDC file (included in the project) should provide all the required functionality (I have assumed 4 bits for the fault ID, so if you use fewer you will have to remove one entry, and of course you might have to change the I/O names if they do not match your implementation):

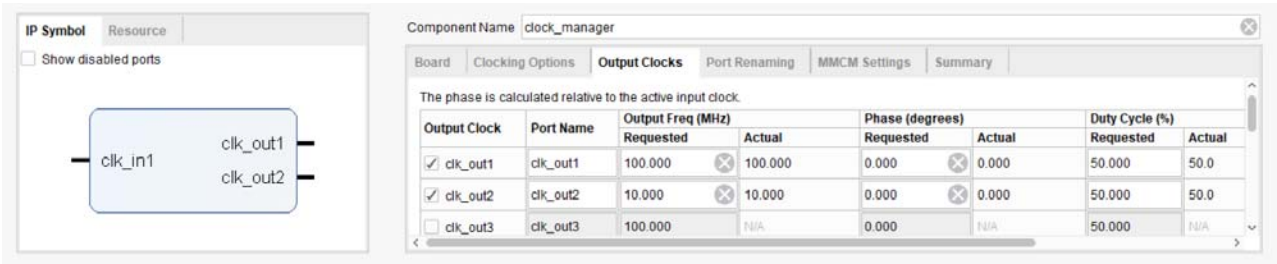
```
# CLK mapped to 100MHz clock source
set_property PACKAGE_PIN Y9 [get_ports {GCLK}]
set_property IOSTANDARD LVCMOS18 [get_ports {GCLK}]
# RST mapped to south button (BTNL)
set_property PACKAGE_PIN N15 [get_ports {B_RST}]
set_property IOSTANDARD LVCMOS18 [get_ports {B_RST}]
# INPUTS[5:0] mapped to switches 5:0
set_property PACKAGE_PIN F22 [get_ports {INPUTS[0]}]
set_property IOSTANDARD LVCMOS18 [get_ports {INPUTS[0]}]
set_property PACKAGE_PIN G22 [get_ports {INPUTS[1]}]
set_property IOSTANDARD LVCMOS18 [get_ports {INPUTS[1]}]
set_property PACKAGE_PIN H22 [get_ports {INPUTS[2]}]
set_property IOSTANDARD LVCMOS18 [get_ports {INPUTS[2]}]
set_property PACKAGE_PIN F21 [get_ports {INPUTS[3]}]
set_property IOSTANDARD LVCMOS18 [get_ports {INPUTS[3]}]
set_property PACKAGE_PIN H19 [get_ports {INPUTS[4]}]
set_property IOSTANDARD LVCMOS18 [get_ports {INPUTS[4]}]
set_property PACKAGE_PIN H18 [get_ports {INPUTS[5]}]
set_property IOSTANDARD LVCMOS18 [get_ports {INPUTS[5]}]
# TEST mode button mapped to centre button (BTNC)
set_property PACKAGE_PIN P16 [get_ports {B_TEST}]
set_property IOSTANDARD LVCMOS18 [get_ports {B_TEST}]
# Fault selection (B_F[1:0]) mapped to switches [7:6]
set_property PACKAGE_PIN H17 [get_ports {B_F[0]}]
set_property IOSTANDARD LVCMOS18 [get_ports {B_F[0]}]
set_property PACKAGE_PIN M15 [get_ports {B_F[1]}]
set_property IOSTANDARD LVCMOS18 [get_ports {B_F[1]}]
# L_OUT mapped to LED0
set_property PACKAGE_PIN T22 [get_ports {L_OUT}]
set_property IOSTANDARD LVCMOS18 [get_ports {L_OUT}]
# L_ERR mapped to LED2
set_property PACKAGE_PIN U22 [get_ports {L_ERR}]
set_property IOSTANDARD LVCMOS18 [get_ports {L_ERR}]
# L_ID[3:0] mapped to LED 7:4
set_property PACKAGE_PIN V22 [get_ports {L_ID[0]}]
set_property IOSTANDARD LVCMOS18 [get_ports {L_ID[0]}]
set_property PACKAGE_PIN W22 [get_ports {L_ID[1]}]
set_property IOSTANDARD LVCMOS18 [get_ports {L_ID[1]}]
set_property PACKAGE_PIN U19 [get_ports {L_ID[2]}]
set_property IOSTANDARD LVCMOS18 [get_ports {L_ID[2]}]
set_property PACKAGE_PIN U14 [get_ports {L_ID[3]}]
set_property IOSTANDARD LVCMOS18 [get_ports {L_ID[3]}]
```

The oscillator on the Xilinx board provides a 100MHz clock. However, we wish to run our design at **10MHz**. To obtain the correct frequency, you should define an internal clock signal, different from the input, generated using a Digital Clock Manager.

In the Project Manager, click on “IP Catalog”. This will open a new tab, containing a long list of IP components (organised in folders). Search for “clocking” and double-click on “Clocking Wizard”. This will launch a multi-tab graphical interface that allows you to configure the clock manager.

Start by giving the component a name (e.g. “clock_manager” or something similar). Select the “Clocking Options” tab. There are several (sometimes very complex) options related to the input clock. The most important value is the definition of the input clock frequency, which should be already set to100MHz as long as you have selected the correct devices and boards in your project. You can leave the default values as they are (feel free to have a browse).

Now select the “Output Clocks” tab. Here, you will define the clock frequencies that will be output from your clock manager. By default, clk_out1 is always a “copy” of the input clock. Even if you will not use it, leave it for good practice. Tick the box next to clk_out2 and set the “Requested Frequency” to 10.000 (MHz). Because 10 is a simple value to obtain from 100, the actual frequency should also be 10MHz (try something like 17.3MHz to see where approximation comes into play). You can leave the phase shift at 0 and the duty cycle at 50%. Scroll down and un-tick the boxes next to “Reset” and “Locked”.



Click OK, then “Generate”. This will add a new IP component to your project that you can use as any other component, by instantiating it in your code. Note that, with this and just about all other IP components, the Xilinx text editor generates a “false” error stating that the component cannot be found (you can verify that this is a false positive by checking that the manager has been correctly placed as a sub-component of your top level design in the Sources pane).

```
71  dcm_mgr : entity work.clock_manager
72  port map
73    (-- Clock in ports
74     CLK_IN1 => GCLK,
75     -- Clock out ports
76     -- CLK_OUT1 =>
77     CLK_OUT2 => CLK_10MHz);
```

Add the component into your design, making sure that the correct clocks are connected to the ports (you will need to create an internal STD_LOGIC signal for the slower clock and you can leave the clk_out1 port of the manager unconnected). Note that *all* your logic should use the new 10MHz clock (including the debouncers – which means that your simulations will now have to take into account a new duration for the button press!)

Run the behavioural simulation and ensure that the circuit still works. Note that the TB clock frequency should remain at 100MHz, but all your internal logic will work at $1/10^{th}$ of that speed! Also, the DCM needs more than 100ns to initialise, so increase the initial wait to at least 500ns.

3.3.1: Explain, using a few words and graphs, the concepts of phase shift and duty cycle in a clock signal.
3.3.2: Print out screenshots of the behavioural simulation for the system. The screenshot(s) should show, in readable format, all the events and signals of the previous task (3.2.5) but also clearly display the new internal clock and how it affects the circuit.

You are now ready to implement the design on the board. Run the implementation process in the design tab. Confirm that the desired clock frequency is achievable by looking at the Design Runs tab, then follow the steps outlined in lab 1 to configure the board.

You can now verify that the circuit operates as predicted. Experiment with the board to verify the correct operation of the circuit.

Part II: ChipScope Pro

The logic analyser is a powerful IP component available to you through the IP catalog. Launch the catalog (under Project Manager) and search for ILA. Click on ILA (Integrated Logic Analyzer).

An interface will open up to allow you to configure the IP. The first page allows you to select the monitor type (select Native), the number of probes (signals you want to observe) and the sample data depth (how many clock periods you want to be able to observe once the ILA is triggered). Ignore the other settings (check the Documentation if you are interested in knowing what they do).

In this task, you will want to observe 7 signals: the debounced input B_TEST_DB (which will trigger the analyser), VECTOR, UUT_OUT, B_F_SET, L_ERR, L_ID, and MODE. So, enter 7 in the "Number of probes" field. The default depth of 1024 should be sufficient.

Now click on the Probe_Ports tab and assign the correct sizes (number of bits) to the probes (in the order above: 1, 6, 1, 2, 1, 4 (or 3), and 1. Set the type to DATA AND TRIGGER for the first one (so you can at the same time use it to trigger the logic analyser and observe it as data) and DATA for all others. Click OK, then "Generate". A new component will appear in your sources.

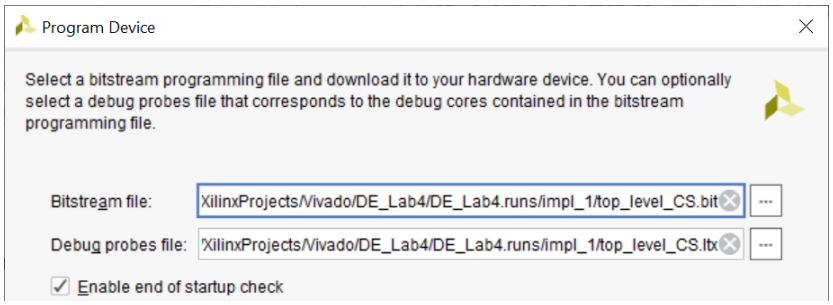
Expand it and open the .vhd file. This will give you an entity declaration that you can use to implement the ILA as a component in your top level VHDL. Connect the fast (100MHz) clock and the signals to the ports. Note that you will need internal signals to read your outputs and to turn your STD_LOGIC signals into STD_LOGIC_VECTOR of size 1 (odd, but there you go). There is a trick to that: your implementation should look something like this:

```
ILA: ENTITY work.ila_0
PORT MAP (
  clk => gclk,
  probe0(0) => B_TEST_DB,
  probe1 => VECTOR,
  probe2(0) => UUT_OUT,
  probe3 => B_F_SET,
  probe4(0) => L_ERR_INT,
  probe5 => L_ID_INT,
  probe6(0) => MODE_INT);
```

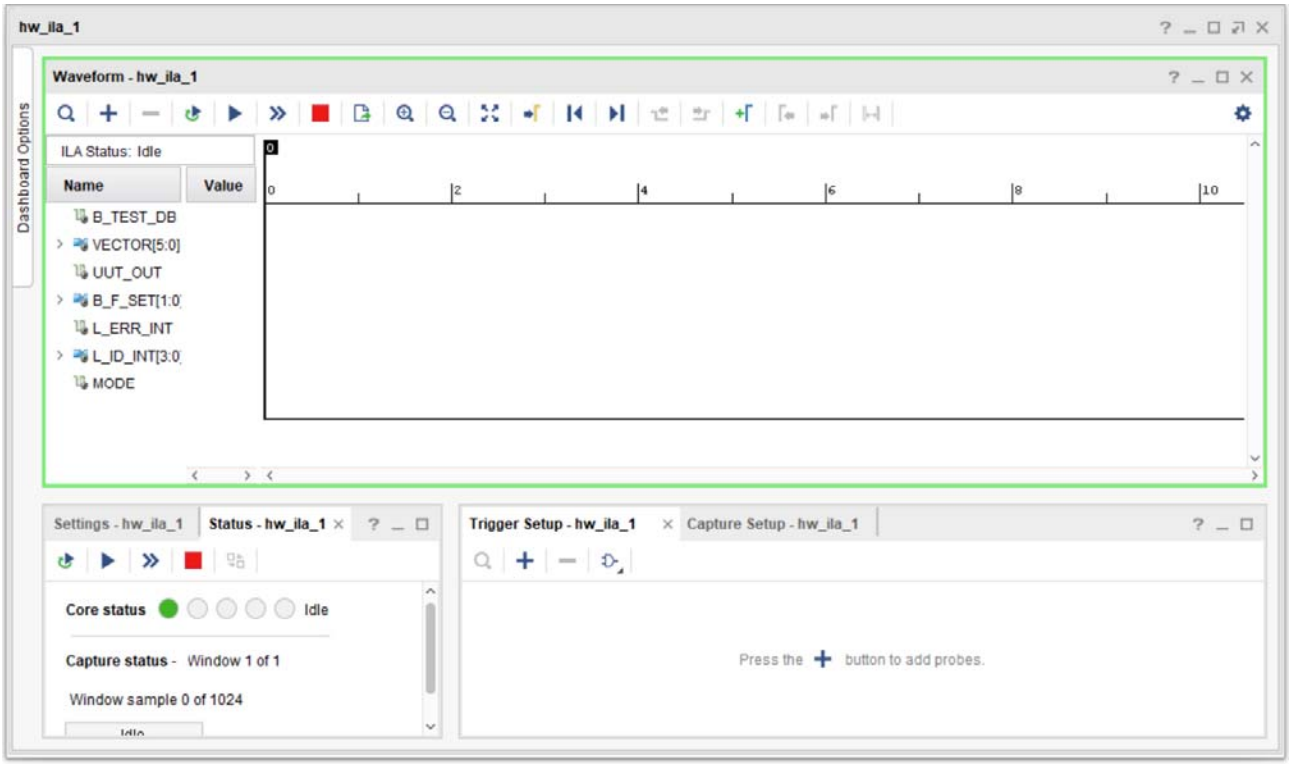
Last thing we need to do is to tell the tools not to touch those signals that we want to observe (normally, they would be removed or transformed during the synthesis process). To do so, add in the declaration section (between "architecture" and "begin") the following *compiler directives*:

```
attribute keep : string;
attribute keep of B_TEST_DB : signal is "true";
attribute keep of VECTOR : signal is "true";
attribute keep of UUT_OUT : signal is "true";
attribute keep of B_F_SET : signal is "true";
attribute keep of L_ERR_INT : signal is "true";
attribute keep of L_ID_INT : signal is "true";
attribute keep of MODE : signal is "true";
```

Now click on “Generate Bitstream”. If there are no errors in your code, the tools will re-synthesize and re-implement the design. Open the hardware manager and connect to the board. If all has gone well, you should now see that the programming pop-up window includes both a bitstream file and a debug file:

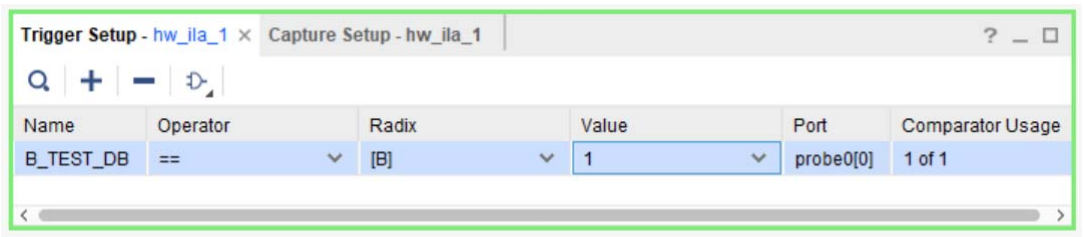


When you program the device, a new tab will appear, looking somewhat similar to a simulation window. But instead of running a simulation, you will be recording and displaying the value of the signals *inside* the device as it is running!



The first step is to define the *trigger conditions*, that is, the combination of values that will start the recording of the values on the lines you have defined. In our case, we have a very simple trigger: we want to see what happens when the user presses the TEST button. Therefore we need to set our trigger to meet that condition.

Press the “+” button on the trigger setup pane. We only specified one trigger signal (B_TEST_DB) so you will only have one choice (if you had defined multiple signals, you could define complex combinations of values). We very simply want to trigger when the line has logic value 1:



The ILA is ready. Launch it by pressing the play button. The scope is now waiting for the trigger to be activated. On the board, select a fault and start test mode. You should now be able to observe the signals as they change on the actual board. Play around with the interface to explore some of the options (keep in mind that you can save the settings and re-use your changes in future runs).

3.3.3: Print out the VHDL code for the top level design (and only the top level), including the clock divider and the ILA component instantiations.

3.3.4: Print out the ChipScope Pro window showing the values of the internal signals when the analyser is triggered for each of the three faults. Make sure that the signal names (including vectors) are the same as the original VHDL file and that the output matches your simulations.

3.3.5: What do you think would have changed if we had used the 10MHz clock to clock the logic analyser instead of the 100MHz clock? Would you still expect to be able to observe the internal data signals, which also switch at 10MHz? Explain your reasoning and feel free to experiment.

APPENDIX: Implementing memories in Xilinx devices using IP cores

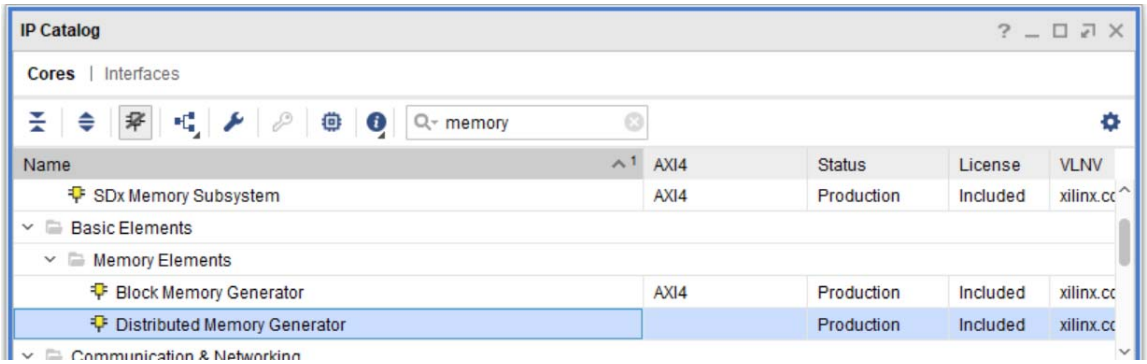
To synthesize memories for a Xilinx device in the most efficient way, a set of IP cores is specifically devoted to this task. In particular, using IPs allows the designer to select between the two kinds of memories available in Xilinx devices:

- Distributed Memory, built from the LUTs within the FPGA, or
- Block Memory, which uses custom memory blocks of 18Kb.

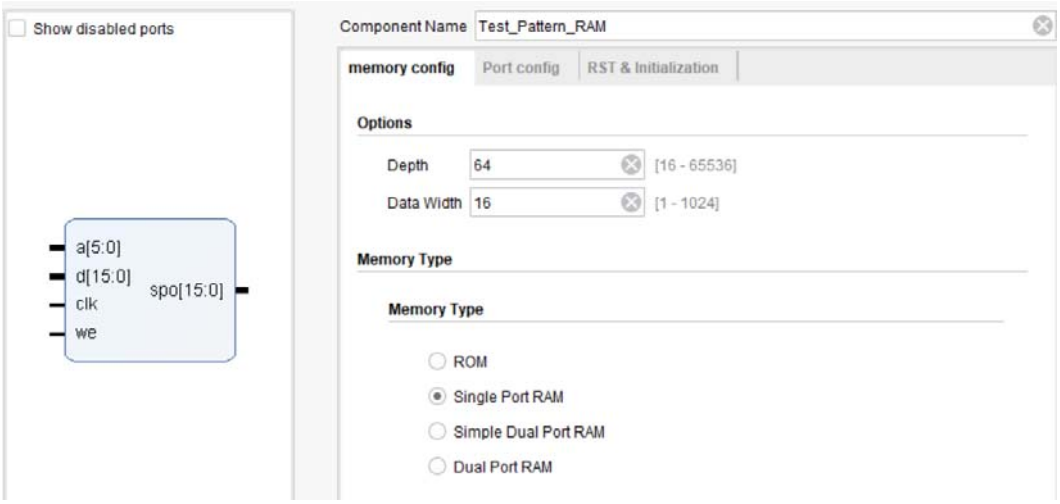
In this lab, given the small size of the required memory, we will be using distributed memory. Keep in mind that larger memories would use too many LUTs which could more efficiently be used for logic. To implement larger memories, Xilinx provides some dedicated memory blocks (BRAM) within its devices. Note that these blocks are of a fixed size (18Kb) and that while larger memories can be created by joining several blocks together, smaller memories cannot be defined (that is, a small memory would still require a complete 18Kb block even if it only uses a fraction of it).

The following is a guide on how to use the Xilinx Distributed Memory 8.0 core generator to create a single-port RAM of 64 16-bit words with user-defined initial contents. Note that the same interface can be used to generate ROMs or dual-port RAMs. **This guide is for illustration only and the specific values, sizes, and parameters do not necessarily correspond to the requirements of the tasks in this laboratory.**

1. Open the "IP Catalog" in the Project Manager. Search for "Memory" and select "Distributed Memory Generator".



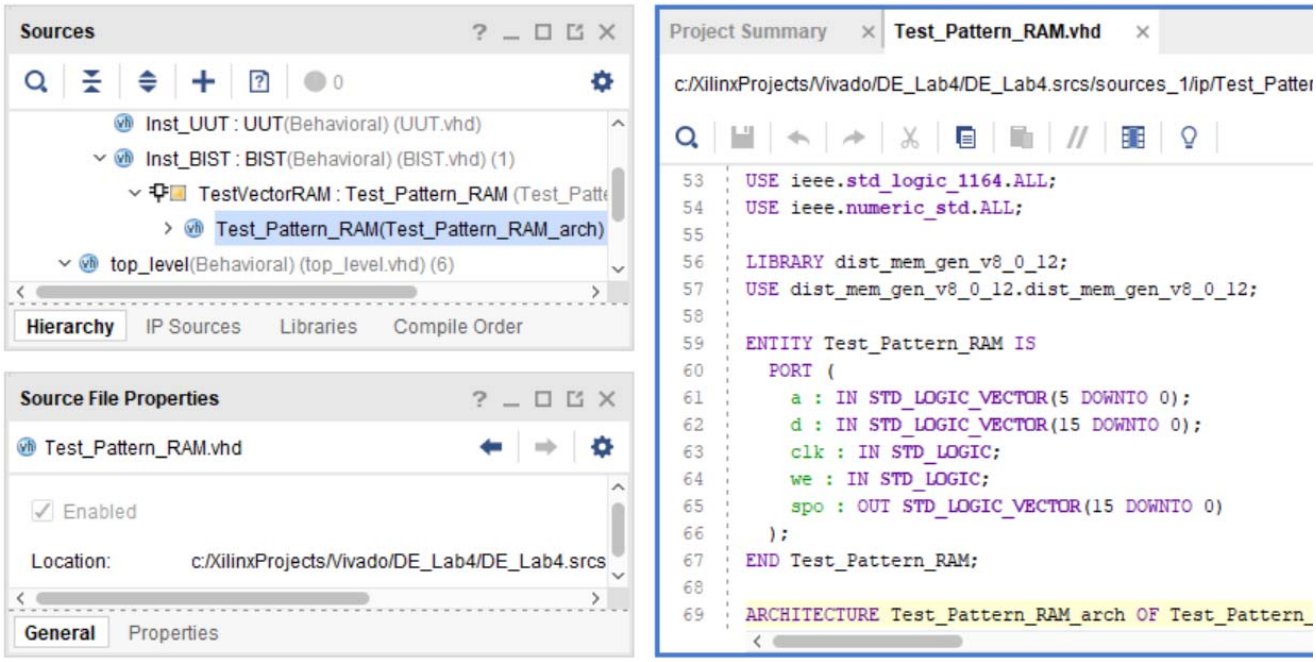
2. After a few seconds, a new window will open. Here you can define the name of the component ("Test_Pattern_RAM" to match the provided code, but that can of course be changed if you use a different name). You can also set the *depth* and *width* of the memory, and the memory type (in this example, we will keep the defaults, but you will need to set the correct values for your circuit, but **note that the minimum depth is 16** – keep that in mind for the address). Select the next tab ("Port config").



3. In this tab, you can define whether registers are present at the inputs and/or outputs of the memory and their features. In this lab, none are needed. Go to the next tab ("RST & Initialization").
4. In this tab, you are asked if you want to load a coefficient (.coe) file. For memories, the coefficient file is a plain text file that contains the initialization values for the memories. The format of the file is the following:

```
memory_initialization_radix=2;
memory_initialization_vector=
0000000,
0000001,
-- insert as many patterns as needed
1101010,
1100101;
```

   where the memory initialization radix is the radix of the numbers in the vector (2 for binary – probably the one you will use in this lab – , 10 for decimal, or 16 for hex) and the memory initialization vector specifies the contents of the first N memory locations (where N can be any number between 0 and the maximum address in the memory). Make sure that the width of the entries matches the width of the memory!
   Using a text editor, create the .coe file required for your task (using the create/edit button allows you to create the file but then opening it through the interface seems to cause an error in the tools - *groan* - you are welcome to try, otherwise any error in the coefficient file will have to be detected in simulation). Click on "Load coefficients" and locate your .coe file. Then click on "OK", then "Generate".
5. After a short time, a new source will appear in your project. If you expand the entry, you will see a VHDL architecture. Open it to see the entity declaration.



6. Use the entity description to instantiate the memory as a component in your design (in the same way and with the same syntax as you would use for any other component in your design) using the name you have chosen earlier (if you have used "Test_Pattern_RAM", then the provided code can be used as is). The component will now be compatible both for simulation and for synthesis.
7. Note that, to change the contents or any of the parameters of your memory, it will be necessary to *re-generate* the memory core (by double-clicking on it, which re-opens the configuration interface).