SLE 3^{rd} year Project

Digital Test Platform

Final Report

Students:
Dienstmann Felipe
Meireles João Pedro
Schuh Matheus

Professors:
Portolan Michele
Rolland Robin

February 2016

Contents

List of Figures

1 Introduction

This document reports on the case study of an embedded system implementation, which was carried out during the first semester of the 3^{rd} year of the Embedded Software and Systems specialization at Ensimag. The objective of this project was to design and validate an integrated hardware/software embedded system, starting from the architectural specification down to the prototype on a programmable device.

This report contains the specification of the system (a lower-cost automatic test unit), the process of development (for both hardware and software) and the integration of the final solution, with demonstrations and simulations showing how the system reacts through the different iterations developed.

2 Context

With the growth of the integrated circuits industry the need to create better test mechanisms is vital as the production time and cost must be reduced in order to be competitive in the market. Inside this context one of the solutions most used in the electronics scene is the ATE or *Automated Test Equipment*, that allow the possibility of testing a DUT (Device Under Test) in a completely automatic way.

Even tough the ATEs are extensively used in the electronics industry, they still are a very expensive solution as they are complex and can operate at really high frequencies. This can be a problem to companies or universities that have the need to use this kind of equipment to develop their integrated circuits, but don't need this level of complexity.

So it's necessary to find a way of creating a simpler solution that can implement a smaller range of automated tests, allowing the user to have all the benefits of the ADE with a decreased final cost.

3 Solution

3.1 Overview

In order to solve the presented problems, a low-cost platform capable of testing integrated circuits was developed. It is able to apply a given set of test vectors to the circuit under test, to receive its responses and to compare them to the expected results. A general overview of the solution is shown in figure 1.

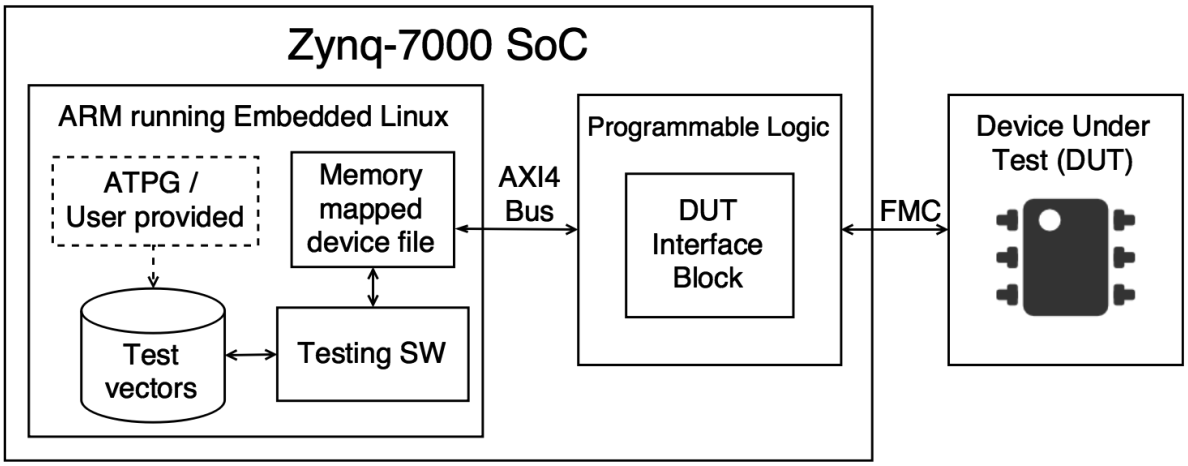


Figure 1: General overview of the solution

The whole solution is contained within a Zynq-7000 system on chip, which is developed by Xilinx. It was chosen for its flexibility; since it contains an ARM processor and programmable logic in the same chip, it allows the development of both software and hardware.

The ARM processor runs an embedded Linux, which is booted from a SD card in which the input files of the solution may also be stored. These input files describe the test information in a format which is specified further in this document. They contain both the inputs to be provided to the device and the expected outputs. They may be hereafter provided in a format supported by an Automatic Test Pattern Generation (ATPG) tool.

The testing software is the piece of code in charge of parsing the input file, sending the test vectors to the DUT interface peripheral (which will apply them to the circuit) and comparing the received responses to the expected results. It controls and communicates with the DUT interface component by mapping its device file in its virtual space address.

The test vectors received by the DUT interface component through the AXI bus are stored in a memory block. This peripheral then retrieves each of these vectors, applies them to the device under test and saves the response in the same memory, which will subsequently be read by the testing software.

The circuit is interfaced with the testing system through a daughterboard which is connected to the FPGA's Mezzanine Connectors (FMC)[1]. This board was developed for the Zynq-7000's ZC706 demonstration kit; however, since the aforementioned interface is standardized, it may be reutilised in another system containing the same connectors.

[1] `https://en.wikipedia.org/wiki/FPGA_Mezzanine_Card`

3.2 Features

- Easy description of the DUT's pinout, of the vectors to be applied to it and of the expected responses through a JSON input file provided to the testing software;
- Supports integrated circuits with up to 32 pins, each of which may be configured as input or output;
- Supports burst transfers between the testing software and the DUT interface block of up to 256 test vectors/results;
- The device's pins may be described as signals, each of which has a name and receives a value separately in the input file for clearer representation;
- Input file supports multi-bit signals for easy handling of numerical values (or for a more compact representation);
- For each test vector, an expected value may or may not be provided for each output signal, hence allowing don't-care terms and preparation vectors;
- Robust parser which detects and indicates possible inconsistencies and problems in the provided input files;
- Triggering of an external signal in the beginning of each test sequence to facilitate its visualization with an oscilloscope;
- Well-documented and reproducible project, which may be easily extended and adapted to any other board containing a Zynq-7000 system on chip.

4 Development

4.1 Architecture

The global architecture of the final system is present in the following image, in which both the DUT interface block (or `DUT_INTERFACE_BURST_CONFIG`) and the testing software are shown in detail.

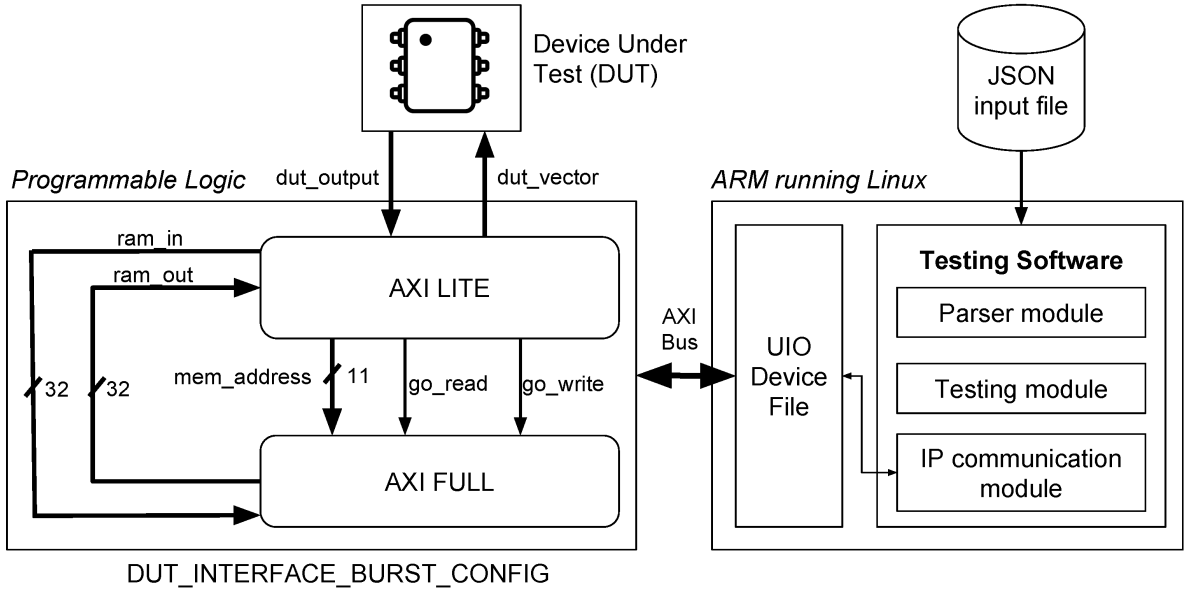


Figure 2: Global architecture of the system

The `DUT_INTERFACE_BURST_CONFIG` is an AXI4 Slave IP divided in two main interfaces: the **LITE**, responsible for controlling the FSM and the transmission of data to the DUT, and the **FULL**, responsible for managing the memory used by the processor (writes the data to be sent to the circuit and reads the output values) and by the lite block (writes the responses of the circuit).

The `tICo` (Tester for Integrated Circuits Operation) testing software communicates with the DUT interface peripheral by mapping its corresponding UIO device (which interfaces with the processor's AXI master component) to its memory space. It obtains the device's pinout, the test vectors to apply and the outputs to expect by parsing a JSON input file provided by the system's user which contains the the test information.

4.2 Implementation

4.2.1 Hardware

4.2.1.1 Hard coded version The first version of the solution was a "hard coded" one, where the user had to define previously all the input and output ports, the vectors to be sent and the expected results. The version made in this project was hard coded to a simple BCD decoder with 4 inputs and 7 outputs, all of them connected to a pinout defined by an `.xdc` file, and working in a specific frequency (if the software part is ignored). The logic of the IP is represented by the following FSM.

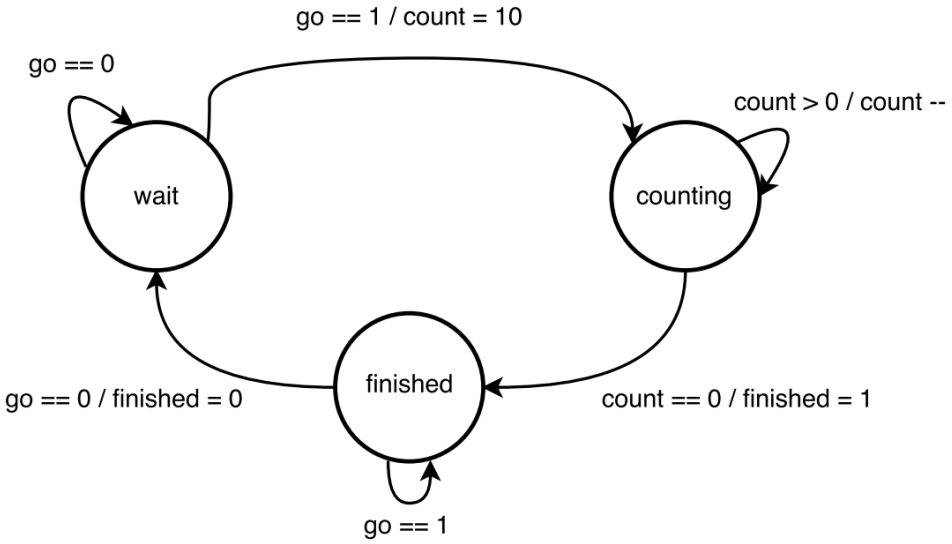


Figure 3: Finite state machine of the solution without burst transmission

Firstly the processor writes the vector to be sent in a particular register of the IP, followed by the sending of a signal, also in a register, indicating the beginning of the operation (called `go` in the figure 3). The FSM then proceed to a state called "counting", responsible for simulating a frequency divider, allowing the user to test circuits with different values of frequency by setting a counter that will divide the original AXI frequency (100MHz).

When this counting process is over, and the FSM knows that the output signal of the DUT is already stable, the response is saved in another register and a signal indicating the end of the operation is written in a register. The processor waits (with a polling process) for this value and then reads the response in order to compare with what was expected.

4.2.1.2 Configurable pins After the implementation of the embedded Linux in the ARM processor, the input and output vectors started to be decided outside, at software level (more details in the followings sections). The definition of the pins, however, was still defined at hardware level, what was very irksome to the final user.

With that in mind, the next iteration of our interface IP was the addition of configurable pins, allowing the user to notify the tester circuit which pins are input and output in the same configuration file were the vectors are described.

In order to implement this, *Xilinx's IOBUF* (Input-Output Buffer or Bi-directional buffer) was used to transform the inputs and outputs of our circuit in one big *inout* bus, using one for each of the 32 possible pins.

As the image 4 shows, the control signal is the responsible for indicating if the bus signal should work as an external output port (sending the vectors for the DUT through the input signal) or an external input port (receiving the response through the output signal). The processor can control the whole structure by sending a 32 bit value, where '1' represents an input (from the tester point of view) or a not connected pin (as its value will be ignored in the comparison made in the software) and '0' represents an output.

The FSM stays the same apart from the "waiting" state, that now waits the `go` signal of the processor and the end of the configuration of the *IOBUFs*, represented by a signal called `io_config`.

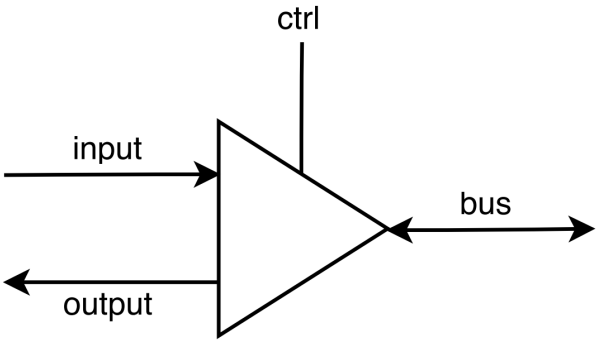


Figure 4: IOBUF structure and signals

4.2.1.3 Burst transmission Even though the configurable solution works and can give the user total control of how the test is made, the performance was not good for a great amount of vectors. That comes from the fact that in our two previous solutions, the IP waits for the processor to send, read and compare each vector. This operation is very slow and stall the hardware, which is able to work way faster. To solve this a new IP capable of working with the AXI burst transfer was created.

In both old solutions, the IP used is what Xilinx call a **LITE IP**, a structure capable of doing single operations of write and read, using a specific set of 32 bits registers. The **FULL IP**, however, is capable of doing burst transfers and store up to 1024 bytes by using a BRAM of same size generated by Xilinx.

The main problem, however, was the fact that both interfaces contained structures useful for the application. Even though the AXI FULL was what was needed for the gain in the performance, the registers used in the LITE structure (and not present in the AXI FULL block) were also useful for the control of the FSM, since the access of these elements is simpler and quicker. The solution was to create a single IP, called `DUT_Interface_Burst_Config` with two different interfaces (one of each kind) generating, in the final design, two independent blocks capable of communicating with each other as shown in the figure 2. The LITE interface contains the FSM and the connection to the DUT, while the FULL contains the memory responsible for storing the testing vectors and outputs.

As the addition of the burst mode changed the main operation of our IP, the FSM was modified to control the different steps of the new execution. Firstly the processor will write, using the burst transfer, all the 32 bit vectors to be sent into the memory (limited to an amount of 256) and the number indicating their amount in a register (using the LITE interface). As in the other two solutions, the FSM then will wait until the configuration of the pins is over and the `go` signal is set to '1' to start the execution. As the machine go from the "waiting" to the "counting" state a signal `go_read` is emitted to the FULL block to indicate that a read will be done. After practical tests using an oscilloscope, it was observed that the counting step in fact should be reduced, since the process of reading and writing in the memory already takes some cycles (three in this case). This, allied with the fact that the frequency desired now is 10MHz, resulted in a variable `count` with the value 1.

After the counting process the FSM proceeds to a state where the number of entries is tested to check if all the vectors of this round were already sent. If it's not the case, the value is written (using the `go_write` signal, in the same address where it was read (overwriting the entry vector in the memory with its response) and the address (that will

be read next) is incremented. Even tough the test is limited to the memory size (1024 bytes) the user can do a test with more than 256 entries, by simple dividing it in many different "rounds".

This will follow until the end of the round is reached, when we indicate (through the same register of the last two IP iterations) that the execution has finished. The processor will then read the memory, again with a burst transmission, and compare the outputs of the DUT with the expected values.

The **timing results** for the hardware with burst mode proved that the initial assumption (that this type of transfer would greatly increase performance) was true. If before the average time of execution for a set of 16k test vectors was $39.3ms$, now the same execution takes $20.2ms$, almost twice as fast.

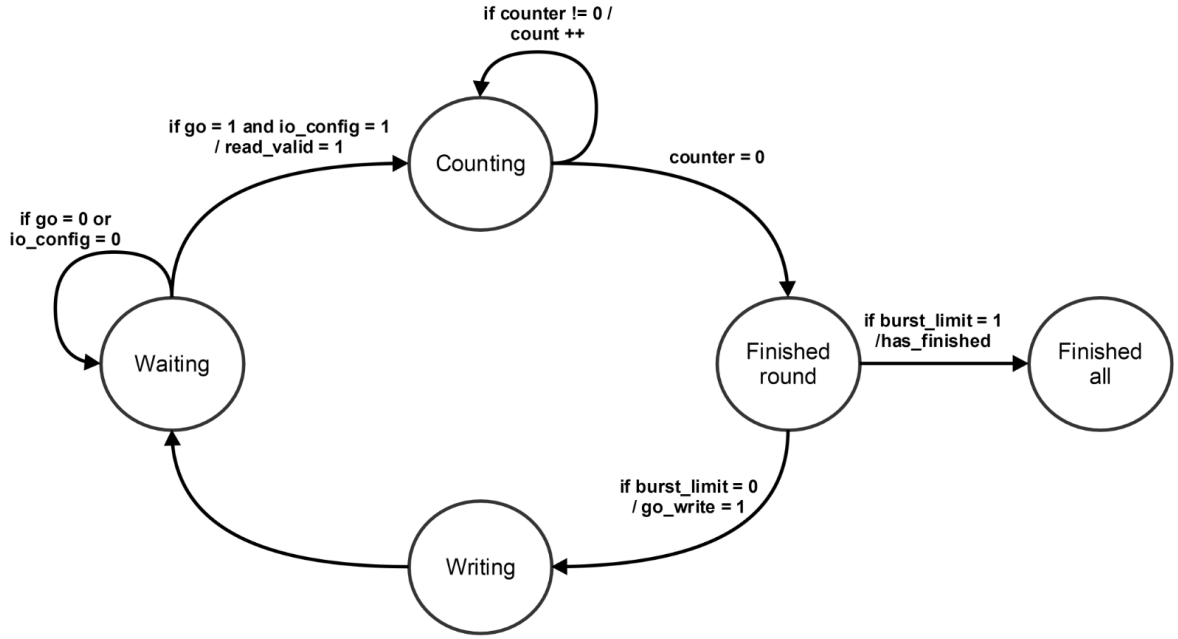


Figure 5: FSM of the final solution

4.2.2 Software

4.2.2.1 Input file parsing module The file taken as input by the software is a JSON[2] file which describes the device's pinout and the test data (inputs to be applied and expected outputs). In order to parse it, we used the `jsmn`[3] JSON parser library, which is meant to be used in resource-limited systems such as an embedded one. This means, however, that it is pretty minimalistic (almost only a lexer); therefore, most of the actual parsing was written by us, such as:

- checking that the input file follows the expected schema;
- verifying that there are no inconsistencies in the pins' declarations and/or test vectors definitions;
- building the structures representing the test information (signals, vectors, etc).

[2] `http://www.json.org/`
[3] `http://zserge.com/jsmn.html`

Through the use of helper functions which take the structures constructed by the parser as argument, the program is able to obtain the necessary values to configure and control the DUT interface peripheral, such as for defining which pins are inputs and which ones are outputs and for writing the test vectors to the IP's memory. Moreover, these structures also facilitate the comparison of the results outputted by the device to the expected values.

4.2.2.2 IP communication module The communication of the testing software with the block that interfaces with the DUT is achieved through the use of Linux's Userspace I/O (UIO) system[4]. Since all we need to control it is to access its memory space, there is no need to write a full kernel driver. The UIO system serves to address this kind of situation. Regarding this project, a small kernel module provided by Xilinx is only in charge of controlling the interface with the AXI bus and providing access to the IP's memory space through a device file. Therefore, the main part of the driver (that is, controlling the DUT interface block) runs in userpace.

The IP communication module of the software simply maps the device file corresponding to the DUT interface peripheral to its virtual space address, allowing the program to configure and control the block's FSM, to write the test vectors to its memory and to read the outputted results from it. It also defines the constants required to correctly control the IP, such as registers' offsets, expected value to define if a pin is an input or an output, and so on.

4.2.2.3 Testing module The main module of the program, it makes use of the other two modules to actually carry out the testing of the DUT. In the main execution flow of the testing software, it:

1. handles the command line options with the `optarg()` function, receiving the path of the input file to be parsed and other optional arguments (such as silent mode and measurement of time, see section 7);
2. calls the parser for the provided input file, obtaining a structure containing the test information if everything went well;
3. maps the DUT interface peripheral to its memory space;
4. configures the IP with the expected value for the device's pinout obtained from the input file;
5. According to the number of input vectors (since the memory may not be big enough to stock them all), repeats as many time as needed:
   (a) writes the test vectors to be applied to the circuit to the IP's memory;
   (b) writes a start value to a register which tells the FSM to begin testing the device;
   (c) waits for the end of the tests by polling another register;
   (d) retrieves the circuit's results from the memory and compares them to the expected values, which are obtained from the test information provided by the parser.

[4] `https://www.kernel.org/doc/htmldocs/uio-howto/`

5 Validation

5.1 Global Validation Plan

Embedded projects put together the challenges of developing hardware and software platforms. Even though they are similar in certain aspects, the process of validating a hardware component is completely different from a validating a software one.

In the hardware world, validation means checking if the `RTL` circuit built attends its purpose. This is done through behavioral, post-synthesis and post-implementation simulation, which allows the designer to compare the results with the user's needs. In the software world, validation means pretty much the same thing: assuring that the program written achieves the requirements. On the other hand, this is done using different tools: formal and static verification, test cases and their code coverage, etc.

Therefore, due to these differences in the validation work flow, its global plan was divided in **three stages**: *hardware, software and system*. The two first stages were done in parallel and the last one incorporated all of the work, thus requiring further collaboration. They will be explained in more details in the next subsections.

5.2 Hardware

5.2.1 Performed Validations

As aforementioned, three peripherals were developed throughout this project. They communicate with the outside world using the AXI4 protocol, and, therefore, it is complex to simulate the IP independently from the rest of the system (too many signals to control and timings to respect). In order to solve this, Xilinx supply the **AXI BFM** (Bus Functional Model) which allows to easily verify the functionality of AXI masters and AXI slaves modules.

In our particular case, we use this resource to simulate AXI slave modules. Thus, the AXI BFM creates an AXI master to control our IP (c.f. Figure 6) and also provides example test benches with `verilog` tasks that manage the communication through the bus. These `verilog` test benches were modified to create custom scenarios capable of validating our IPs.

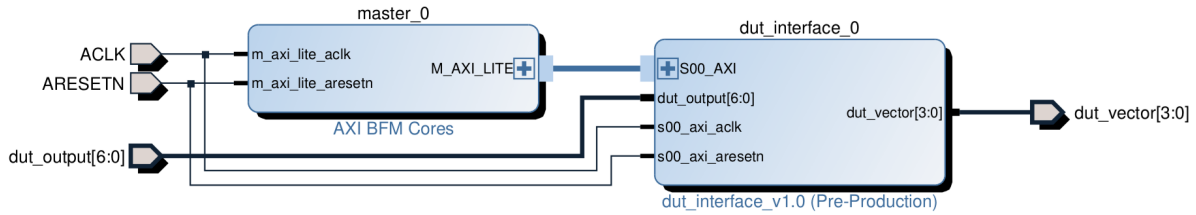


Figure 6: AXI BFM system generated for the IP `dut_interface`

5.2.1.1 dut_interface

Initial IP using the AXI4-Lite interface (register based) and developed specifically for a 4-bit input and 7-bit output BCD decoder circuit. It assumes that all the pins of the

circuit are correctly connected and ordered with the ports `dut_vector` and `dut_output`. The task written in `verilog` to create the module stimuli was `TESTER_AXI_TB`.

Basically it sends an input value to the circuit, 0 for example, and then writes the expected answer in the output port, 0x7E in this case. Then it stimulates the `STATUS` register to initiate the FSM and keeps polling the `FINISHED` register to see if the task has finished. Once it is finished, it reads the register containing the output from the circuit and tells the FSM that this in/out round is over. This process is repeated for all the meaningful input values, i.e. 0 to 9.

5.2.1.2 dut_interface_config

Evolution of the first IP, still using the AXI4-Lite interface, but now offering more flexibility. Instead of providing support just for hard coded circuits, now any circuit with up to 32 pins could be tested by this module. The pins are configured as input or output through the value written in the `DUT_CTRL` register. The task written in `verilog` to create the module stimuli was `TESTER_AXI_CONFIG_TB`.

Before starting the testing sequence, the first thing the task does is to write the appropriate configuration value to `DUT_CTRL`. After that it has to control the signal `tb_dut_bus` which is connected to an input/output port and defined as follows:

```
wire [‘DUT_BUS_WIDTH-1:0] tb_dut_bus;
wire [‘DUT_BUS_WIDTH-1:0] input_bus;
reg [‘DUT_BUS_WIDTH-1:0] output_bus;
reg output_bus_valid;

/*  defines when dut_bus works as an input or output
    if output_bus_valid = 1 then
        tb_dut_bus = output_bus;
    else
        tb_dut_bus = ZZZZZZZZ; (working as an input and read through input_bus)
*/
assign input_bus = tb_dut_bus;
assign tb_dut_bus = (output_bus_valid==1'b1) ? output_bus : 32'hZZZZZZZZ;
```

Therefore, every time the task wanted to read a value it kept `output_bus_valid` set to ‘0‘ and when it wanted to send a data it was set to ‘1‘. The FSM working mode was kept intact, so apart from the control of this signal the rest of the task is pretty much similar to `TESTER_AXI_TB`.

5.2.1.3 dut_interface_burst_config

Evolution from the first and second IPs, now using two interfaces: one AXI4-Lite (register based) and one AXI4-Full (memory based). The lite interface kept the FSM and configuration functionality, while the full interface allowed the user to send the test vectors and read all the outputs from the circuit in burst mode.

The task written in `verilog` to create the module stimuli was `TESTER_AXI_BURST_CONFIG_TB`. It is similar to the previous one and the main differences rely in how the data is exchanged and that the task has no more control over each independent in/out round.

Firstly, all the vector values are sent using the burst mode. Then the task writes in two registers: the pinout in `DUT_CTRL` and the number of vectors sent in `BURST_SIZE`. The `STATUS` register is set to '1' and the simulated output values of the circuit are sent each 100 ps while the FSM is working. When the `FINISHED` register is set to '1' the task reads all the outputs values stored in the memory and compare them with initial vectors using the function `COMPARE_DATA`. If they are equal it writes '0' to the `STATUS` register finishing the task.

Two different test benches were created for the validation of this module: one sending the default test with 10 vectors and another one sending 256 vectors so the corner cases of the memory and burst functionality could be explored.

5.2.2 Results

5.2.2.1 dut_interface

To validate this IP the read value in `S00_AXI_rd_data_lite` register should be stable and correspond to the expected from the input in `dut_vector` when the FSM is in the state 'finished'. As we can see in the figure 7 the blue markers precise this exact moment, with all the correct values, thus validating the system.

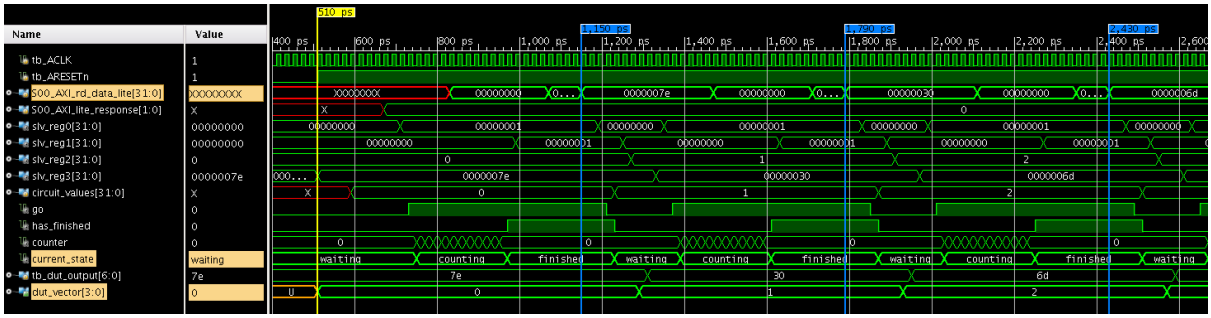


Figure 7: BFM simulation of the IP `dut_interface`

5.2.2.2 dut_interface_config

Even though the IP is now configurable, its external behavior should be the same as before. Consequently, to validate it the read value in `S00_AXI_rd_data_lite` register should be stable and correspond to the expected from the input in `dut_vector_ip` when the FSM is in the state 'finished'.

In this simulation the read values have changed due to the pinout configuration present in `slv_reg2` (0 is input, 1 is output, from the circuit's perspective). Nevertheless, the old values can still be seen in `dut_output_ip` as the bits have been reordered.

As we can see in the figure 8 the blue markers precise the aforementioned moment, and all the signal exhibit the correct values, thus validating the system.

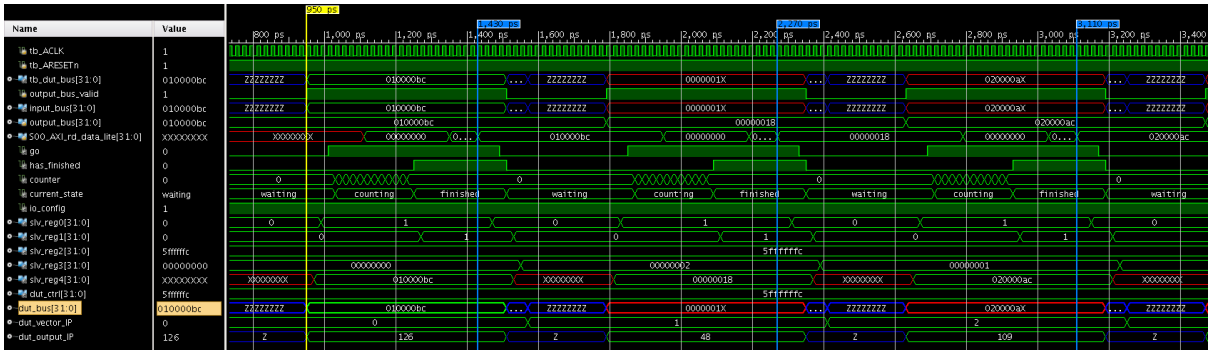


Figure 8: BFM simulation of the IP `dut_interface_config`

5.2.2.3 dut_interface_burst_config Despite the fact the IP has now a memory and a burst interface, its general functionality should remain intact. There are two validation points for this module:

- Each vector value (input for the circuit) should be read from the memory and available when the FSM is in the `'counting'` state.
- At the end of the task, when the FSM is in the state `'finished_all'`, the values read from the memory (output from the circuit) should be in accordance with the input values sent at the beginning.

As already stated two test benches were made, but the images below were extracted from the default test with 10 vectors due to readability factors. The simulation shows that the component does pass in the two validations points, having the proper vector value in the right time (figure 9) and the expected outputs when reading from the memory (figure 10).

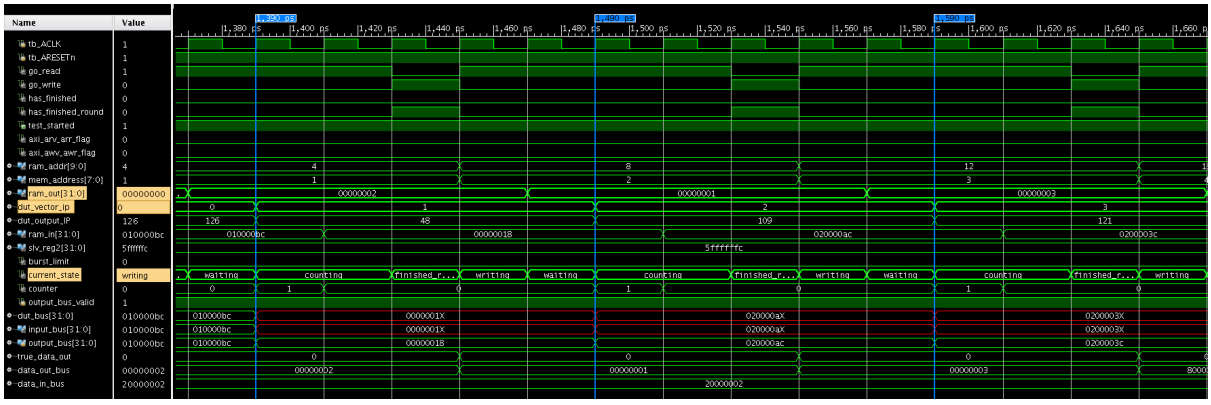


Figure 9: BFM simulation of the IP `dut_interface_burst_config` (input vector)

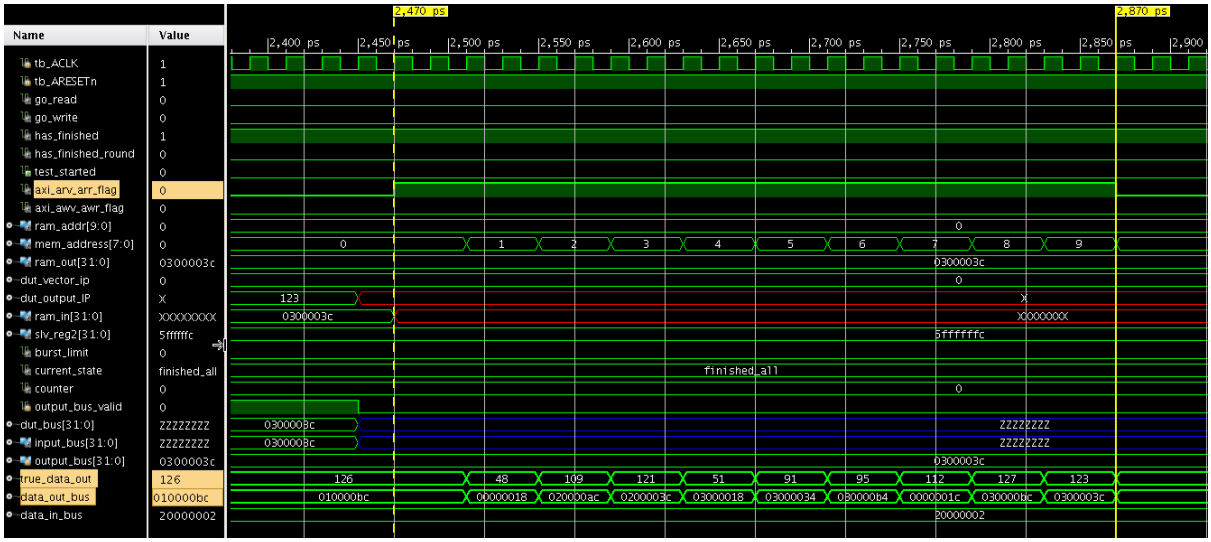


Figure 10: BFM simulation of the IP `dut_interface_burst_config` (mem read)

5.3 Software

In order to validate the correct behavior of the testing software, an automated test script was developed. It verifies that the program gives the correct result for a set of input files, each of which is representative of a specific functionality or possibly problematic case. This is achieved through the comparison of the outputs provided by the software for each test-case file with the expected ones. For each input, the expected outputs are provided in a separate file.

For each of the software responsibilities, a set of input files was written in order to confirm it works as intended:

- **Parsing of the input file**[5]**:** a set of invalid files was written to verify that the software correctly detects possible input problems (repeated pin declaration, invalid value assigned to a signal, undefined input pin value, etc). Valid files are also present to test that the different input features (hexadecimal vs decimal representation, multi-bit signals, etc) are accurately parsed.

- **Configuration of the peripheral:** for a set of valid input files, we check if the software correctly produces the value to be written to a register of the DUT interface peripheral which defines whether each of the DUT's pins is an input or an output.

- **Expected output comparison:** for a set of valid input files, we check if the software correctly compares the value provided by the DUT interface peripheral corresponding to the circuit's outputs to the expected values described in the input file. Since the actual circuit and hardware interface are not present, a separate file serves as a mockup of these components and contains the values which could be output by the device.

[5]The actual JSON parsing is handled by the `jsmn` library, which already includes a significant amount of tests. Therefore, we focused on testing that the input file correctly matches the schema we expect and that the test information is properly obtained from it.

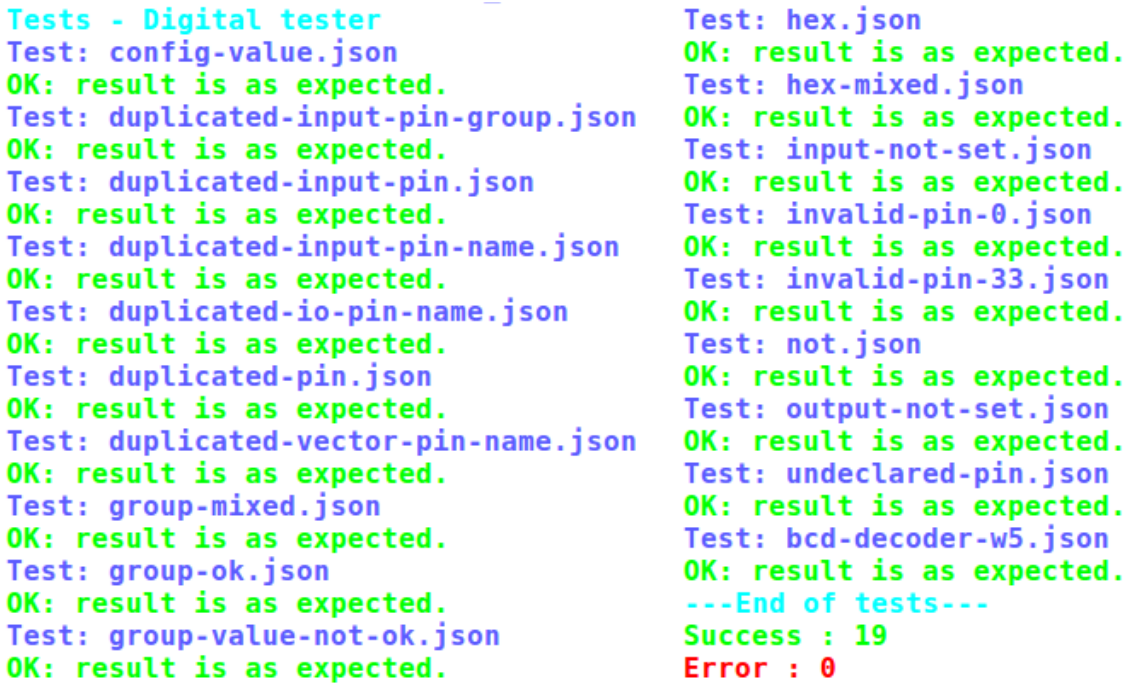


Figure 11: Software test script results

The validation of the software's behavior in the complete system, including its communication with the actual DUT interface peripheral, is described in the following subsection.

5.4 System

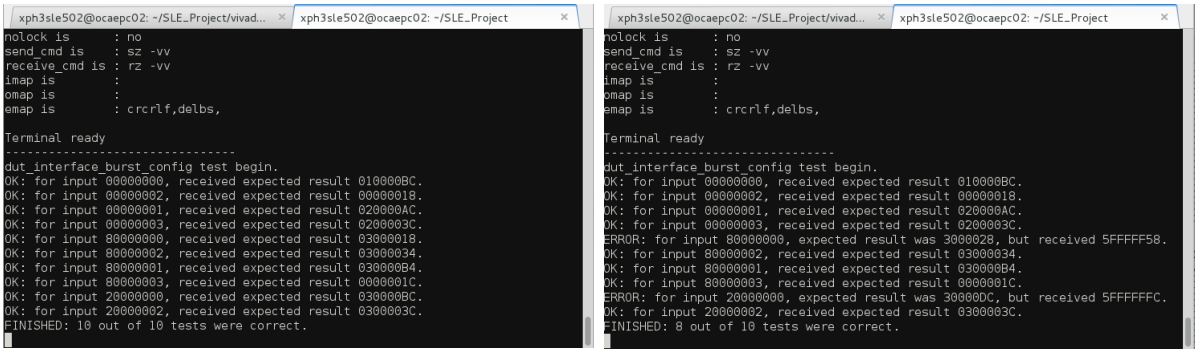
The system validation involves integrating the hardware and software subsystems while still being able to exploit and evaluate their functionality. This was done in two different ways in our project, which will be explained in the next sections.

5.4.1 Bare Metal

After the synthesis and implementation of the `digital_tester` system (which includes the processing system and the developed programmable logic) the *Vivado* suite offers the possibility to export the hardware and develop the embedded software.

This is done through the *SDK* (Software Development Kit) where an application project was written for each developed IP. The software was written in `C` and was basically a translation of the `verilog` test benches used in the simulation. The *SDK* offers also the possibility to program the FPGA with the previously generated bistream and then load the software directly, without any OS.

To validate the system the program does comparisons between the expected outputs and the actual outputs (considering just the specified bits in the `DUT_CTRL` register) coming from the circuit. The results of these value checks are informed to the user through serial debug messages. In order to visualize them the terminal application `picocom` is required. The next images show a regular and an erroneous execution of the system.



(a) Perfect DUT (b) DUT with errors

Figure 12: Bare metal software execution

5.4.2 OS Support

After the synthesis and implementation the generated bitstream can be incorporated into the bootloader of an operating system. In our project this can be done because the ZC706 has an ARM processor, and the OS chosen was *PetaLinux*, as it is recommended by *Xilinx* and supplies various tools to facilitate the work of embedding the linux.

A SD card containing the OS and eventual software to be run is put into the board. The communication is still done through the serial port, thus requiring `picocom`. Once the OS has finished its booting process, a standard linux environment is available.

The embedded program developed for the linux is more complex than the bare metal version. It allows to charge different configuration files and test different circuits. Although its complexity, the validating functionality is the same: it does comparisons between the expected outputs and the actual outputs (masked with the `DUT_CTRL` register) coming from the circuit. The files (c.f. A.1 and A.2) included in the appendix A show a regular and an erroneous execution of the software being run in the linux environment.

A final verification of the signals was done using the oscilloscope. This allowed a real time feedback from the circuit, similar to the simulation point of view, that helped to confirm the timing and stabilization of all the values.

Figure 13: OS sytem version being monitored with an oscilloscope

6 Project schedule and division of work

6.1 Initial planning

Since developing the different parts of the solution independently and integrating them at the end wouldn't work well for a project as complex as this one, it was developed in an incremental way. The functionality of the system was sliced into increments, which were constructed sequentially. After an initial phase during which the base behavior of the system was developed, the further work was divided into two parts: the testing software and the communication with the circuit. By defining increments separately for each of them, their development was carried out independently.

The following Gantt diagram illustrates the initial planning:

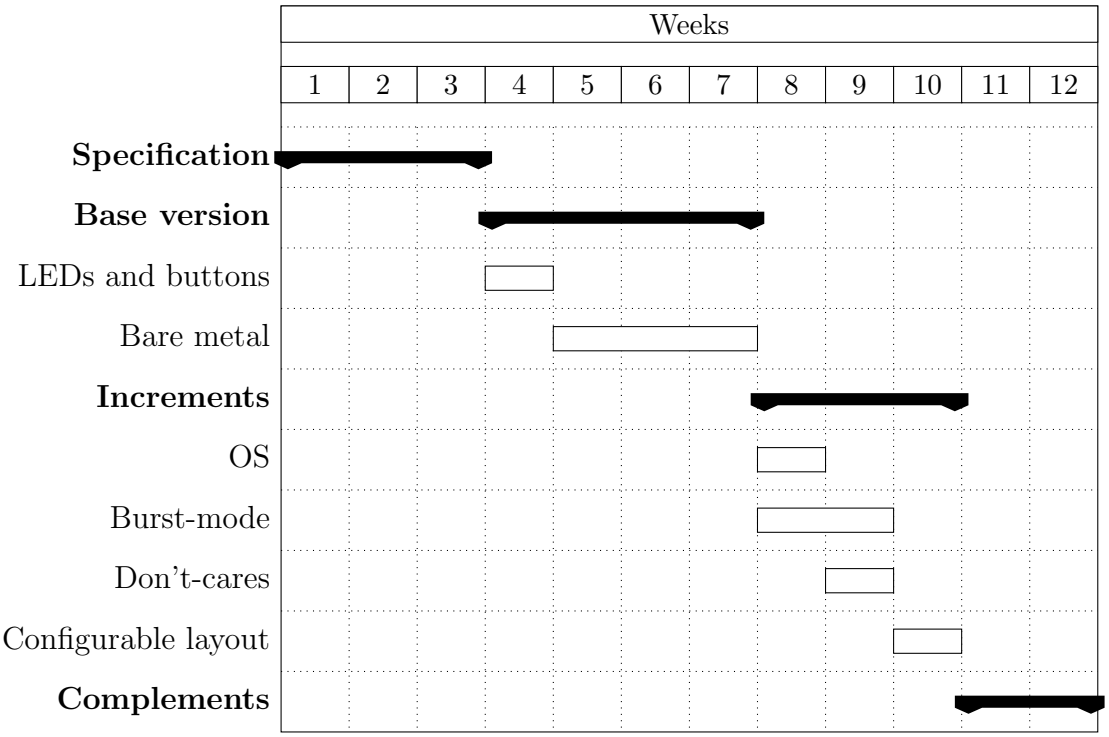


Figure 14: Gantt diagram containing the provisional planning of the project

6.2 Final planning

This planning, however, was not followed strictly due to different problems found through the development of the project (most of them related to the lack of previous experience with Xilinx's tools), causing a general delay in the project. Nonetheless this delay was compensated with the addition of three weeks in the end of the project.

The division of work followed the one defined in the planning, where two different paths were taken after the definition and implementation of the base version of the system. Two members of the group focused in the hardware development (being responsible for the configurable layout and the burst-mode) and the last one was in charge of the software side of the application (embedding the Linux in the processor and creating the software solution). This division can be better seen in the following Grantt diagram, where white

represents a task without division, green a task made by the software team and blue one made by the hardware team.

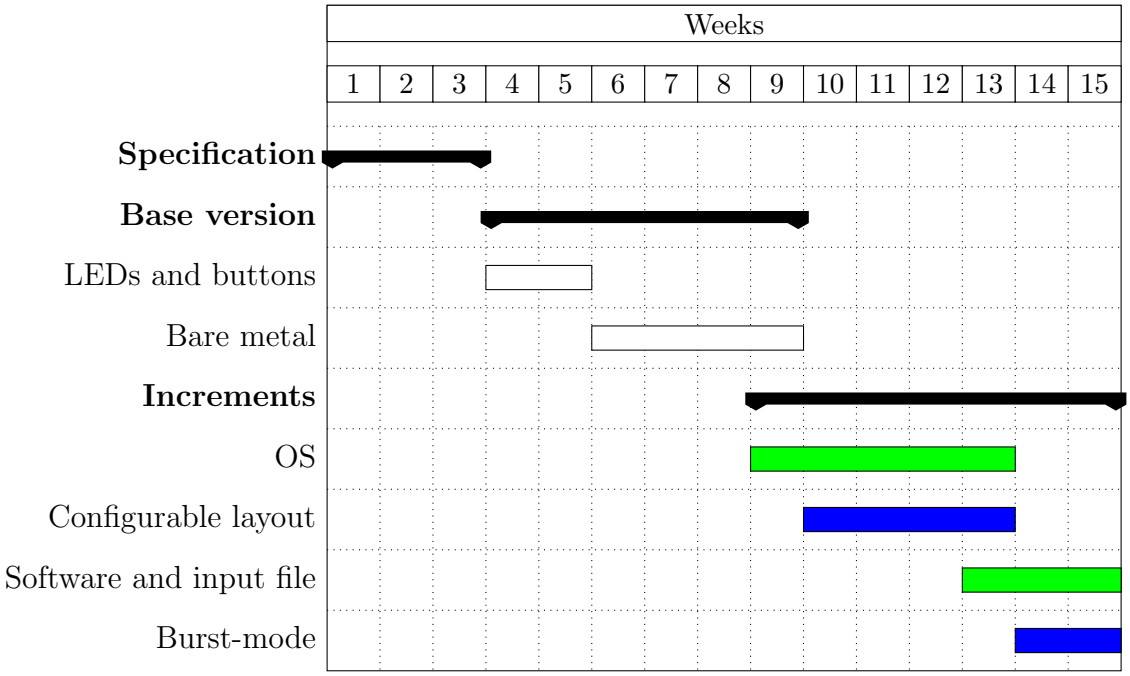


Figure 15: Grantt diagram of the final work distribution

7 User Guide

7.1 Writing the input file

The file taken as input by the testing software describes the device's pinout and the test data (inputs to be applied and expected outputs). It is a JSON file[6] which should match the following schema:

```
{
    "circuit-name": "4-bit-adder",
    "freq-divider": 1,
    "inputs": {"a": [4, 3, 2, 1], "b": [8, 7, 6, 5], "cin": 9},
    "outputs": {"s": [13, 12, 11, 10], "cout": 14},
    "vectors": [
        {"in": {"a": 2, "b": 3, "cin": 1}, "out": {"s": 6, "cout": 0}},
        {"in": {"a": 8, "b": 10, "cin": 0}, "out": {"s": 3, "cout": 1}},
        {"in": {"a": 0xA, "b": 0x5, "cin": 0x0}, "out": {"s": 0xF}}
        ]
}
```

Listing 1: Example JSON input file

The root object contains 5 properties (name/value pair):

- **circuit-name:** the name of the device under test.
- **freq-divider:** value for which the base frequency of the test system should be divided. This functionality has not yet been implemented.
- **inputs:** the input signals of the device. Each property represents a signal, where the property's name corresponds to the signal's name and the property's value to the tester's pins to which the device's signal pins are connected. The pins numbers should match the ones from the diagram shown below in figure 16. A signal may correspond to one or more bits. If more than one bit are used, the corresponding pins should be between square brackets, with the most significant bit on the left. Repeated names and pins are not allowed.
- **outputs:** the output signals of the device. They should follow the same rules described above for the inputs. An output and input signal may have the same name, but this should be avoided for clarity.
- **vectors:** the test data, which corresponds to an array containing a variable numbered of ordered vectors. Each vector is an object which describes the inputs' values which should be applied to the circuit (`"in"` property) and the expected outputs' values for this application (`"out"` property). For each vector, a value must be given for each of the previously declared inputs, but only the outputs which we want to compare need to receive a value (the other ones are considered *don't-care*, such

[6] Our testing software uses a less strict version of the JSON standard, since hexadecimal numbers (which are not supported by the official standard) are accepted.

as for the last test vector in the example). You can use this if you need to apply preparation vectors to the circuit before obtaining the value you which to verify. A value may be given either in decimal or in hexadecimal, but if the latter is used, the value should be prefixed by `0x`. Signals which correspond to more than one bit are considered to be unsigned (for example, a 3-bit signal may receive a value from 0 to 7).

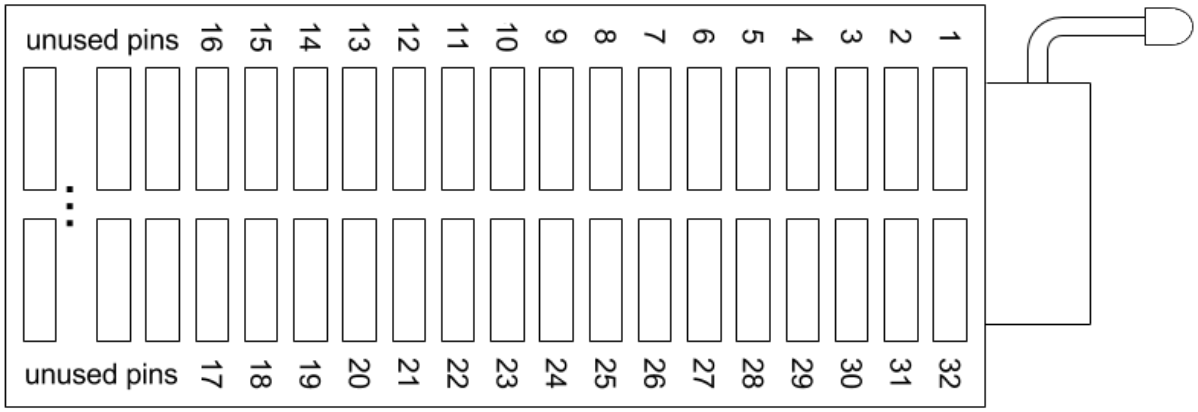


Figure 16: Tester pinout numbers to be used in pins configuration

The circuit's power-supply pins should not be declared in the `json` file and should be connected to an external power source. In order to test a sequential circuit, you should declare an input signal corresponding to the clock and oscillate it between 0 and 1 in each test vector.

7.2 Circuit setup

Once the input file is written, the circuit should be placed accordingly onto the 40 pin DIP socket. In order to have a feedback from the ZC706 regarding when the testing process has started and when it has finished, an external pin was setup.

An oscilloscope should be branched to the GPIO Header J58, pin 1, to read this signal. The board also supplies a ground signal in the same header, in the pins 9 and 10. See[7] for further information.

7.3 Booting Linux

In order to boot Linux with the ZC706 evaluation board to run the testing software and to carry out the tests, the subsequent steps should be followed:

1. Connect the circuit to be tested to the daughterboard according to the pinout described in the written input file(s);

2. In the first partition of a FAT32 formatted SD card, save the following files: `BOOT.bin` (bootloader), `image.ub` (kernel image), `tico` (testing software) and the desired `json` input files;

[7] `http://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf`, page 60-62

3. Connect board's serial port (UART) to a computer with a mini USB cable;
4. In a terminal, open your serial communication program (such as `picocom`) with a baudrate of 115200 (you may require special user rights in order to access the serials ports);
    - Example: `picocom -b 115200 /dev/ttyUSBx` (`x` is normally 0 or 1);
5. Configure the boot mode to SD boot by setting the board's switch 11 (SW11) to 00110 (1, 2 and 5 down, 3 and 4 up);
6. Plug the SD card into the board and turn it on (if nothing appears in the serial console, you may need to press the board's reset processor switch (SW3));
7. When asked for login information, type user name `root` and password `root`;
8. You now have access to a Linux headless terminal through your serial console.
    (a) To access the SD card's files, you need to mount it: create a directory with `mkdir sd` and mount its first partition with `mount /dev/mmcblk0p1 sd`;
    (b) To execute the testing software, change to the directory in which you mounted the SD card (`cd sd`) and execute it with `./tico input-file.json` (for further options, check subsection 7.4);
    (c) To view the content of a input file, you can use `more file.json` or `cat file.json`;
    (d) To turn off the system, unmount the SD card with `umount sd` (do not forget to go back to the parent directory with `cd ..` if you are inside the `sd` folder) and run the `halt` command.

7.4 Testing software command-line options

The `tICo` testing software supports the following arguments:

```
./tico [-s -t] file ..
```

- **file ..**: a list of `json` input files (the tests for each of them are carried out consecutively).
- **-s (silent mode)**: disables most of the program's outputs to the console, therefore allowing a faster execution of the tests.
- **-t (measure time)**: enables measurement of the time taken by the program to parse and execute the tests for each of the input files.

8 Maintenance Guide

The following subsections refer to the source tree as present in the project's GitHub[8].

8.1 Compiling the software

The `tICo` testing software may be recompiled through the Makefile present in the `software/tico` folder. It uses the ARM compiler (`arm-xilinx-linux-gnueabi-gcc`) provided with Xilinx's tools, do not forget to `source` the adequate settings script before running `make`. The resulting `tico` file will be in the same folder. To execute its tests, simply execute `make test`.

8.2 Generating the bitstream

The Vivado project of the system can be created by running the `create-project.tcl` script present in the `hardware` folder. To execute it, open Vivado, place yourself in the aforementioned folder in the `Tcl` console and run `source create-project.tcl`. The project will be created in `hardware/digital-tester`.

This project makes use of the other source files contained in the `hardware` folder. When adding new source files to the project, do not include them directly in the project's folder, but rather put them in the adequate subfolder of `hardware` and include them as external source files. The `Tcl` project creation script may then be easily regenerated in Vivado with `File > Write Project Tcl`.

After editing the project as wished and generating the bitstream, export it with `File > Export hardware` (check the include bitstream option). The exported folder will be used by `PetaLinux` to configure the build process of the embedded Linux adequately.

8.3 Rebuilding the Linux image

The `PetaLinux` project is contained in `software/linux/linux-burst`. To reconfigure the `PetaLinux` project with the new hardware's description, place yourself in the exported hardware directory and run (do not forget to source `PetaLinux`'s settings script):

```
petalinux-config --get-hw-description -p path-to-project/software/linux/linux-burst
```

You may need to further adapt other configuration options of the project according to the changes you made (please refer to `PetaLinux` documentation). The system image may then be built by running `petalinux-build`. After this process is finished, you must generate the bootloader file. Place yourself in the `images/linux` folder inside the `PetaLinux` project, source `Vivado`'s settings script (required by `petalinux-package`) and run:

```
petalinux-package --boot --fsbl zynq_fsbl.elf --fpga exported-bitstream-file --uboot
```

The resulting `BOOT.bin` and `image.ub` files, which should be stocked in the SD card to boot Linux, will be present in this same folder.

[8] `https://github.com/fdmusse/zynq-digital-tester`

9 Conclusion

By having the opportunity of developing both the hardware and the software solutions of a system, this project allowed us to consolidate great part of the knowledge acquired at the SLE speciality. Apart from that, many new things were learned such as the functioning and usage of some components as the IOBUF and the use of embedded Linux.

Despite the initial delay in the project, thanks to the three weeks added to initial planning, all the features previewed in our specification were fulfilled and the final solution is simple to use, has a good perfomance (thanks to the addition of the burst mode) and can be easily exported to another Zynq-7000 system on chip or improved in the future. This system was already used (in the bare metal version) in a class of the CSI (Integrated Systems Conception) specialisation at Phelma, and the plan is to continue using it in futures classes and projects.

Figure 17: Phelma students using our system to test circuits in a practical course

A Example system output

A.1 Perfect DUT execution example

```
root@linux-burst:~/sd# ./tico w5/10.json
Parsing input file w5/10.json...
Successfully parsed.
Started testing circuit bcd-decoder-w5-high.
Input/Output signals:
---Input signals:
------y: 4 bits, pins = 30 32 1 2
---Output signals:
------s: 7 bits, pins = 3 4 5 6 8 25 26
Sending test vectors...
---Vector 1...
------Sent inputs:
---------y: 0x0
------Expected outputs:
---------s: 0x7E
------OK: all signals as expected.
---Vector 2...
------Sent inputs:
---------y: 0x1
------Expected outputs:
---------s: 0x30
------OK: all signals as expected.
---Vector 3...
------Sent inputs:
---------y: 0x2
------Expected outputs:
---------s: 0x6D
------OK: all signals as expected.
---Vector 4...
------Sent inputs:
---------y: 0x3
------Expected outputs:
---------s: 0x79
------OK: all signals as expected.
---Vector 5...
------Sent inputs:
---------y: 0x4
------Expected outputs:
---------s: 0x33
------OK: all signals as expected.
---Vector 6...
------Sent inputs:
---------y: 0x5
------Expected outputs:
```

```
---------s: 0x5B
------OK: all signals as expected.
---Vector 7...
------Sent inputs:
---------y: 0x6
------Expected outputs:
---------s: 0x5F
------OK: all signals as expected.
---Vector 8...
------Sent inputs:
---------y: 0x7
------Expected outputs:
---------s: 0x70
------OK: all signals as expected.
---Vector 9...
------Sent inputs:
---------y: 0x8
------Expected outputs:
---------s: 0x7F
------OK: all signals as expected.
---Vector 10...
------Sent inputs:
---------y: 0x9
------Expected outputs:
---------s: 0x7B
------OK: all signals as expected.
Finished testing bcd-decoder-w5-high: 10 out of 10 test vectors were correct.
```

A.2 DUT execution with pin problem

```
root@linux-burst:~/sd# ./tico bcd-decoder-y6.json
Parsing input file bcd-decoder-y6.json...
Successfully parsed.
Started testing circuit bcd-decoder-y6-low.
Input/Output signals:
---Input signals:
------input_value: 4 bits, pins = 7 8 25 26
---Output signals:
------sa: 1 bit, pins = 27
------sb: 1 bit, pins = 28
------sc: 1 bit, pins = 31
------sd: 1 bit, pins = 32
------se: 1 bit, pins = 1
------sf: 1 bit, pins = 2
------sg: 1 bit, pins = 4
Sending test vectors...
---Vector 1...
------Sent inputs:
```

```
---------input_value: 0x0
------Expected outputs:
---------sa: 0x0
---------sb: 0x0
---------sc: 0x0
---------sd: 0x0
---------se: 0x0
---------sf: 0x0
---------sg: 0x1
------Error for signal sg: expected 0x1, received 0x0.
---Vector 2...
------Sent inputs:
---------input_value: 0x1
------Expected outputs:
---------sa: 0x1
---------sb: 0x0
---------sc: 0x0
---------sd: 0x1
---------se: 0x1
---------sf: 0x1
---------sg: 0x1
------Error for signal sg: expected 0x1, received 0x0.
---Vector 3...
------Sent inputs:
---------input_value: 0x2
------Expected outputs:
---------sa: 0x0
---------sb: 0x0
---------sc: 0x1
---------sd: 0x0
---------se: 0x0
---------sf: 0x1
---------sg: 0x0
------OK: all signals as expected.
---Vector 4...
------Sent inputs:
---------input_value: 0x3
------Expected outputs:
---------sa: 0x0
---------sb: 0x0
---------sc: 0x0
---------sd: 0x0
---------se: 0x1
---------sf: 0x1
---------sg: 0x0
------OK: all signals as expected.
---Vector 5...
------Sent inputs:
```

```
---------input_value: 0x4
------Expected outputs:
---------sa: 0x1
---------sb: 0x0
---------sc: 0x0
---------sd: 0x1
---------se: 0x1
---------sf: 0x0
---------sg: 0x0
------OK: all signals as expected.
---Vector 6...
------Sent inputs:
---------input_value: 0x5
------Expected outputs:
---------sa: 0x0
---------sb: 0x1
---------sc: 0x0
---------sd: 0x0
---------se: 0x1
---------sf: 0x0
---------sg: 0x0
------OK: all signals as expected.
---Vector 7...
------Sent inputs:
---------input_value: 0x6
------Expected outputs:
---------sa: 0x0
---------sb: 0x1
---------sc: 0x0
---------sd: 0x0
---------se: 0x0
---------sf: 0x0
---------sg: 0x0
------OK: all signals as expected.
---Vector 8...
------Sent inputs:
---------input_value: 0x7
------Expected outputs:
---------sa: 0x0
---------sb: 0x0
---------sc: 0x0
---------sd: 0x1
---------se: 0x1
---------sf: 0x1
---------sg: 0x1
------Error for signal sg: expected 0x1, received 0x0.
---Vector 9...
------Sent inputs:
```

```
---------input_value: 0x8
------Expected outputs:
---------sa: 0x0
---------sb: 0x0
---------sc: 0x0
---------sd: 0x0
---------se: 0x0
---------sf: 0x0
---------sg: 0x0
------OK: all signals as expected.
---Vector 10...
------Sent inputs:
---------input_value: 0x9
------Expected outputs:
---------sa: 0x0
---------sb: 0x0
---------sc: 0x0
---------sd: 0x0
---------se: 0x1
---------sf: 0x0
---------sg: 0x0
------OK: all signals as expected.
Finished testing bcd-decoder-y6-low: 7 out of 10 test vectors were correct.
```

B Post-Implementation Reports

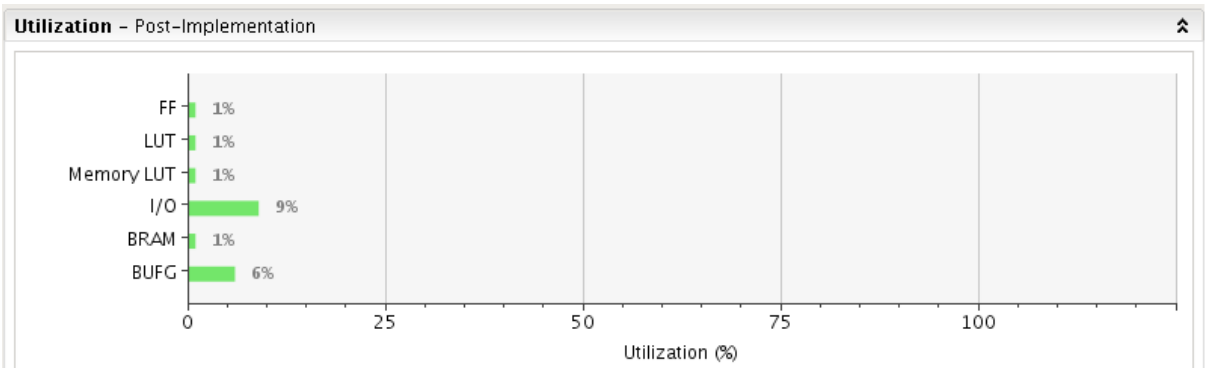


Figure 18: Utilization implementation report

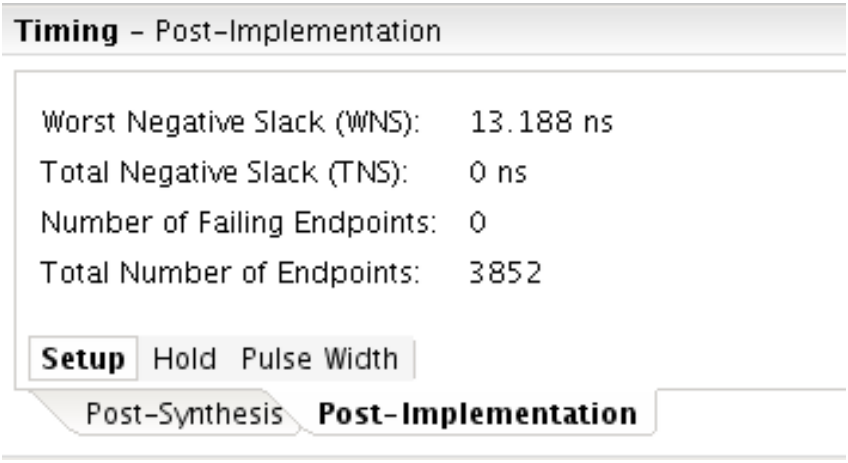


Figure 19: Timing implementation report

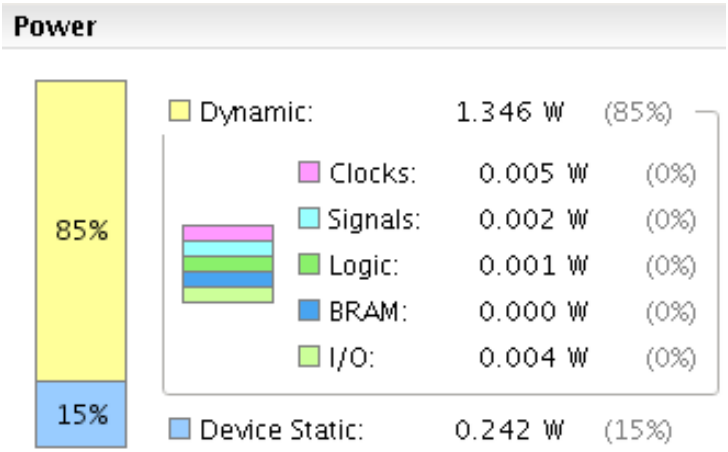


Figure 20: Power implementation report