

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCOLA POLITECNICA SUPERIOR DE GANDIA

GRADO EN ING. SIST. DE TELECOM., SONIDO E IMAGEN



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



ESCOLA POLITÈCNICA
SUPERIOR DE GANDIA

**“Systematic testing of digital hardware systems by
means of test automaton and Test Description
Language.”**

TRABAJO FINAL DE GRADO

Autor/a:

Sergio Santos Casal

Tutor/a:

Carl Georg Hartung

Tobias Krawutschke

Trinidad María Sansaloni Balaguer

GANDIA, 2018

Declaration

I, Sergio Santos Casal, resident at Graf-Adolf-Str. 77,51065 Cologne, declare that this bachelor thesis, apart from the support of the Laboratory for Telecommunications Engineering and the supervising professor Prof. Dr. -Ing Georg Hartung, was completed completely independently and only with the given sources and aids. Furthermore, I declare that this thesis has not been published elsewhere or presented in another subject as an examination.

Signature:

Date:

Acknowledgements

In first place, I would like to thank my parents for their love and support; without them I would not have been able to get here.

I would like to express my gratitude to my tutor, Professor Georg Hartung and to Professor Tobias Krawutschke, for giving me the excellent opportunity to do this project, and for their support in finishing it successfully.

I would like to thank Professor Trinidad Maria Sansaloni for being willing to be my co-tutor in this thesis and for her help when I needed it.

Finally, I would like to thank Professor Jürgen Schneider for his cordial treatment and his willingness to help me during my study here.

Abstract

When an obsolete electronic system needs to be replaced or upgraded and there is no documentation explaining how the system works, reverse engineering is needed to model system behavior. Often, FPGAs are used to replace these systems due to their flexibility.

To test the reverse engineered device, it is advisable to use a test description language (TDL). TDL has the following advantages: superior quality testing through better design, easier to verify by non-testing experts, faster and better test development and perfect integration of methodology and tools.

A TDL is being developed by the Technische Hochschule Köln in a research project. From a TDL file, the associated compiler generates test files that are usable within a VHDL toolchain: a testbench file, a stimulus file, an assertion file and a waveform generator file.

To verify that the TDL generator generates testbench files according to the test description, a model has been created in VHDL. With this model, two test cases have been created: in the first case, the model is working correctly, and in the second case, a malfunction has been introduced. With these two test cases, it is possible to verify if the testbench generated by the TDL generator is capable of detecting malfunctions or not.

Index of contents

Acknowledgements.....	1
Abstract.....	2
1 Introduction.....	9
1.1 Purpose.....	9
1.2 Structure of this work.....	10
1.3 Planned Workflow.....	11
2 Fundamentals.....	13
2.1 Concept of the test systems.....	13
2.1.1 General structure of the test system.....	13
2.1.1.1 Requirements for the test environment.....	13
2.1.1.2 Concrete concept of the test environment.....	13
2.1.1.3 File Interfaces and Interface Definition.....	14
2.1.1.4 Evaluation of the DUT signals.....	15
2.1.1.5 Concept of the workflow.....	16
2.1.2 Description of the Value Change Dump File.....	17
2.1.2.1 Header.....	17
2.1.2.2 Timescale.....	17
2.1.2.3 Signal definitions.....	18
2.1.2.4 Initial value of the signals.....	18
2.1.2.5 Signal changes and timestamps.....	19
2.1.2.6 End of the file.....	20
2.2 Test Description Language.....	20
2.2.1 What is a Test Description Language.....	20
2.2.2 Why use a Test Description Language.....	21
2.2.3 Standards for TDLs.....	22
2.2.4 Design Considerations.....	23
2.3 Used prototyping board.....	25

Systematic testing of digital hardware systems by means of test automaton

3	Development of the VHDL model.....	26
3.1	Specifications of the VHDL model.....	26
3.1.1	How TicTacToe works.....	26
3.1.2	Game board.....	28
3.2	Development of the model.....	35
4	Verification.....	51
4.1	Simulation of the VHDL code.....	51
4.1.1	Simulation description.....	51
4.1.2	Simulation results.....	52
4.2	Testbench using Test Description Language.....	54
4.2.1	Testbench description.....	54
4.2.2	Testbench results.....	61
4.2.2.1	Model working right.....	61
4.2.2.2	Model modified to introduce a malfunction.....	63
4.3	Verification in hardware.....	65
4.3.1	Testbench description.....	65
4.3.2	Testbench results.....	67
4.3.3	Verification of the assertions.....	69
4.3.3.1	Original asserts.vcd.....	69
4.3.3.2	Altered asserts.vcd.....	70
5	Conclusions.....	72
6	Appendices.....	75
6.1	Appendix A: TicTacToe game board design.....	75
6.2	Appendix B: VHDL model code.....	77
6.2.1	VHDL code sketch of the model for one player.....	77
6.2.2	VHDL code of the model for two players.....	85
6.2.2.1	Clock generator.....	85
6.2.2.2	Game.....	86
6.2.2.3	TicTacToe.....	92
6.3	Appendix C: Simulation code.....	93

Systematic testing of digital hardware systems by means of test automaton

6.4 Appendix D: Test Description Language code.....	95
6.5 Appendix E: Errors reported by the TDL testbench with malfunctions in the model.....	114
6.6 Appendix F: Terrasic DE0-Nano board.....	121
6.6.1 Features.....	122
6.6.2 DE0-Nano Board Architecture.....	123
6.7 Appendix G: 74LS595 datasheet.....	125
6.8 Appendix H: Content of the CD.....	135
References.....	136

List of figures

Figure 1: Software and hardware architecture of the test automaton.....	10
Figure 2: Workflow planned for this work.....	12
Figure 3: Concept of the test environment.....	14
Figure 4: Processing the files into test vectors and generating the test response.....	15
Figure 5: Test with modelsim vs. signal evaluation with blackbox model.....	16
Figure 6: VCD file: Header.....	17
Figure 7: VCD file: Timescale.....	17
Figure 8: VCD file: signal definition.....	18
Figure 9: VCD file: Initial value.....	18
Figure 10: VCD file: Signal changes and timestamps.....	19
Figure 11: VCD file: end of the file.....	20
Figure 12: DE0-Nano board.....	25
Figure 13: Example of how to play TicTacToe in which X wins.....	26
Figure 14: Optimal strategy for player X.....	27
Figure 15: Optimal strategy for player O.....	27
Figure 16: Sketch of the interface to the user.....	29
Figure 17: Keyboard control circuit.....	30
Figure 18: KiCad main window.....	30
Figure 19: eeschema interface.....	31
Figure 20: Pcbnew interface.....	32
Figure 21: Rendering of the TicTacToe game board: Top side.....	32
Figure 22: Rendering of the TicTacToe game board: Bottom side.....	33
Figure 23: Photo of the game board.....	34
Figure 24: Photo of the game board connected to the prototyping board.....	35
Figure 25: Finite state machine of the model developed for one player (I).....	36
Figure 26: Finite state machine of the model developed for one player (II).....	37
Figure 27: Finite state machine of the model developed for one player (III).....	37

Systematic testing of digital hardware systems by means of test automaton

Figure 28: Finite state machine of the model developed for one player (IV).....	38
Figure 29: Finite state machine of the model developed for one player (V).....	38
Figure 30: Finite state machine of the model developed for one player (VI).....	39
Figure 31: Finite state machine of the model developed for one player (VII).....	39
Figure 32: Finite state machine of the model developed for one player (VIII).....	40
Figure 33: Finite state machine of the model developed for one player (IX).....	40
Figure 34: Finite state machine of the model developed for one player (X).....	41
Figure 35: Connections between the VHDL prototyping board and the game board.....	42
Figure 36: Modul clock generator.....	43
Figure 37: Modul game.....	49
Figure 38: Modul TicTacToe.....	50
Figure 39: Complete waveform obtain in the VHDL simulation.....	52
Figure 40: Detail of the waveform obtain in the VHDL simulation.....	52
Figure 41: Sketch of the game table after the first player wins.....	53
Figure 42: Result of the VHDL simulation (command console).....	53
Figure 43: Connections between the model and the test automaton.....	56
Figure 44: Sketch of the game table after the TDL testbench.....	58
Figure 45: Error reported by Modelsim. Command console. (TDL testbench).....	61
Figure 46: Waveform result of the simulation, where an error was reported. (TDL testbench).....	61
Figure 47: Complete waveform result of the simulation. (TDL testbench).....	62
Figure 48: Information sent by the serial output when the player 1 wins (TDL testbench).....	63
Figure 49: Waveform obtain in the simulation with malfunctions in the model (I) (TDL testbench)	63
Figure 50: Waveform obtain the simulation with malfunctions in the model (II) (TDL testbench).	64
Figure 51: Configuration of the timing simulation (I).....	65
Figure 52: Configuration of the timing simulation (II).....	66
Figure 53: Configuration of the timing simulation (III).....	66
Figure 54: Delay between CLKgenerator and column_i.....	67
Figure 55: Delay between CLKgenerator and clk.....	68
Figure 56: Delay between CLKgenerator and output_s.....	68

Systematic testing of digital hardware systems by means of test automaton

Figure 57: Delay between CLKgenerator and parallelize.....	69
Figure 58: Result of the verification of vcdplayer.....	70
Figure 59: Result of the verification of vcdplayer.vhd when an error in asserts.vcd was introduced	70
Figure 60: Result of the verification of vcdplayer for 200 ms when an error in asserts.vcd was introduced.....	71
Figure 61: Workflow carried out in this work.....	73
Figure 62: TicTacToe board schematic.....	75
Figure 63: TicTacToe game board PCB.....	76
Figure 64: DE0-Nano board.....	121
Figure 65: The DE0-Nano Board PCB and component diagram (top view).....	123
Figure 66: The DE0-Nano Board PCB and component diagram (bottom view).....	124
Figure 67: Block diagram of DE0-Nano Board.....	124

1 Introduction

1.1 Purpose

The purpose of this bachelor thesis is to verify the proper functionality of the test automaton tool and the Test Description Language (TDL), that are being developed by the Technische Hochschule Köln.

The game TicTacToe has been chosen as the design to be implemented and tested. It is a quite interesting example, because it is easy to understand. In addition to that, its VHDL model is straightforward to implement, thus the needed model tests can easily be conducted/performed. Furthermore, the chosen example is a good one to test the behaviour of the examined tools (test automaton and TDL). By using the TicTacToe example it shall be tested whether the tools work as expected/specified or not. If the tools do not work as specified this thesis tries to point out what modifications can be made in order to achieve the specified functionality.

The test automaton is a hardware device. It consists of several signal processing modules (SPM) to perform measurements or generate stimuli under the control of a PC where the files that describe the test and store the measurements are archived. Each SPM contains an interface for transferring measurements or stimulus data from/to the PC and a FPGA/MEMORY combination for data processing. SPMs are synchronized by a common clock and a trigger distribution unit. [1] .

The TDL developed is a concrete syntax of the specific domain language of the European Telecommunications Standard Institute (ETSI). This TDL allows us to write test cases which are processed within a TDL processing system using the Eclipse IDE. The TDL syntax is written in Language Workbench Xtext and the code generator is written in Xtend.

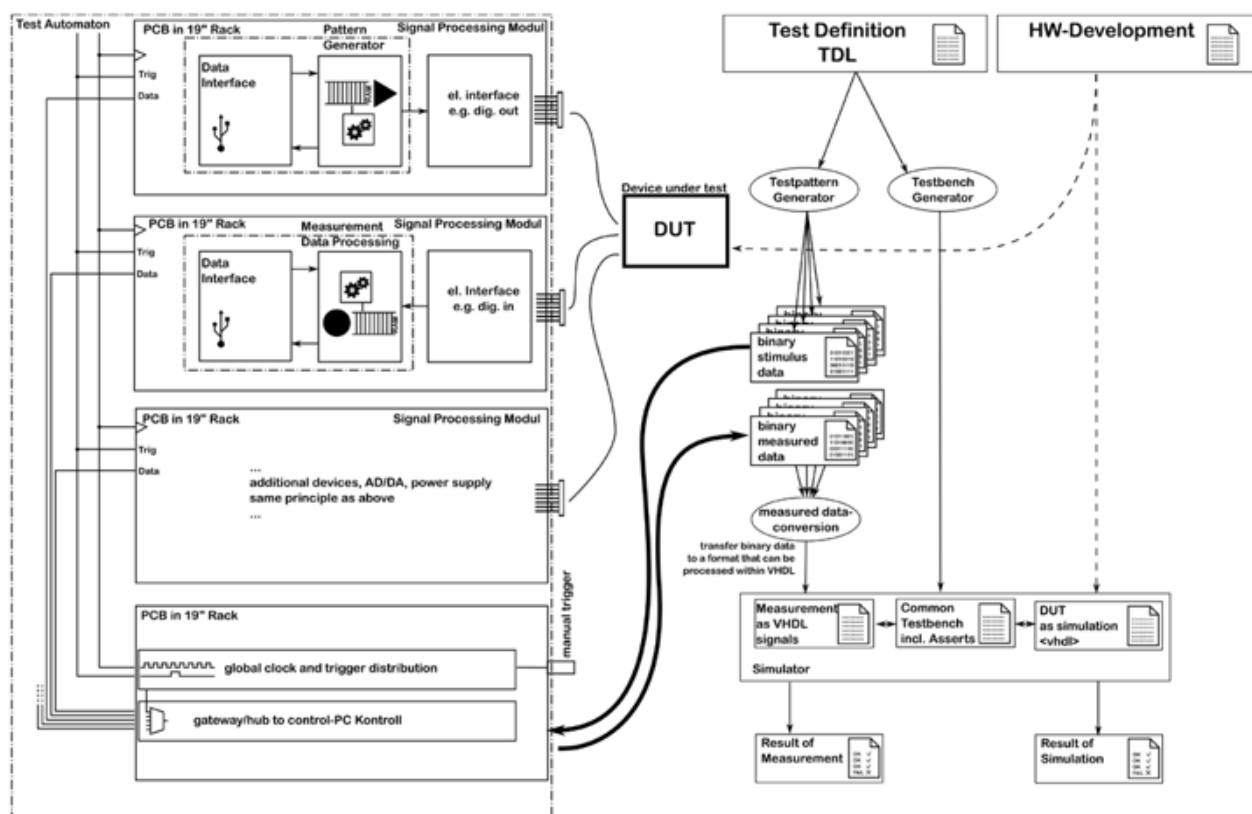


Figure 1: Software and hardware architecture of the test automaton

Obtained from [1]

1.2 Structure of this work

This work has been divided into 6 chapters:

- Chapter 1: Introduction. This chapter explains the tasks that are planned to be carried out in this thesis.
- Chapter 2: Fundamentals. This chapter explains the concepts of test systems, what a TDL is, the characteristics of the TDL that will be used and which prototyping board will be used.
- Chapter 3: Development of the VHDL model. This chapter explains how the TicTacToe game works and how the game board and VHDL model has been developed.
- Chapter 4: Verification. This chapter explains the tests cases, the test descriptions and the simulations that have been carried out and the results obtained.
- Chapter 5: Conclusions. This chapter explains what has been done in the course of the work, what problems have been encountered, what conclusions have been obtained from the work and what future work could be done.

- Chapter 6: Appendices. This chapter shows the game board design, the VHDL model code and TDL test description codes, explains the technical characteristics of the prototyping board and the shift registers used, and the contents of the CD attached to this document, etc.

1.3 Planned Workflow

There are some different stages in the development of this work:

- **Development of a VHDL model.** The game TicTactoe was chosen because it is easy to implement in VHDL, the behavior is predictable and easy to understand, and it is possible to verify the right behavior just playing the game.
- **Creation of a testbench to simulate the right behavior of the model designed.** A testbench written in VHDL is made, and simulated using the software Modelsim in this stage.
- **Creation of a testbench using the TDL.** From a test case description written in TDL, the toolchain written in the laboratory of digital engineering generates the stimuli file to be used in the test automaton, the assertions file to be compared with the measurement file generated by the test automaton and a testbench file used to simulate the model in Modelsim.
- **Testing in the test automaton using the stimuli file generated by the TDL generator.** A measurement file will be generated by the test automaton after the test.
- **Verification of the measurements.** The measurements will be verified using the testbench and the vcdplayer generated by the TDL generator. If no errors are reported, the model works as expected.
- **Creation of a PCB to demonstrate the right behavior of the hardware design in a practical and visual way.** This board is used as interface between the FPGA board and the players. This stage was not part of the work and it is not necessary to test the TDL and the test automaton, also it is not included in the workflow diagram.

The next figure shows the flow diagram of the planned workflow.

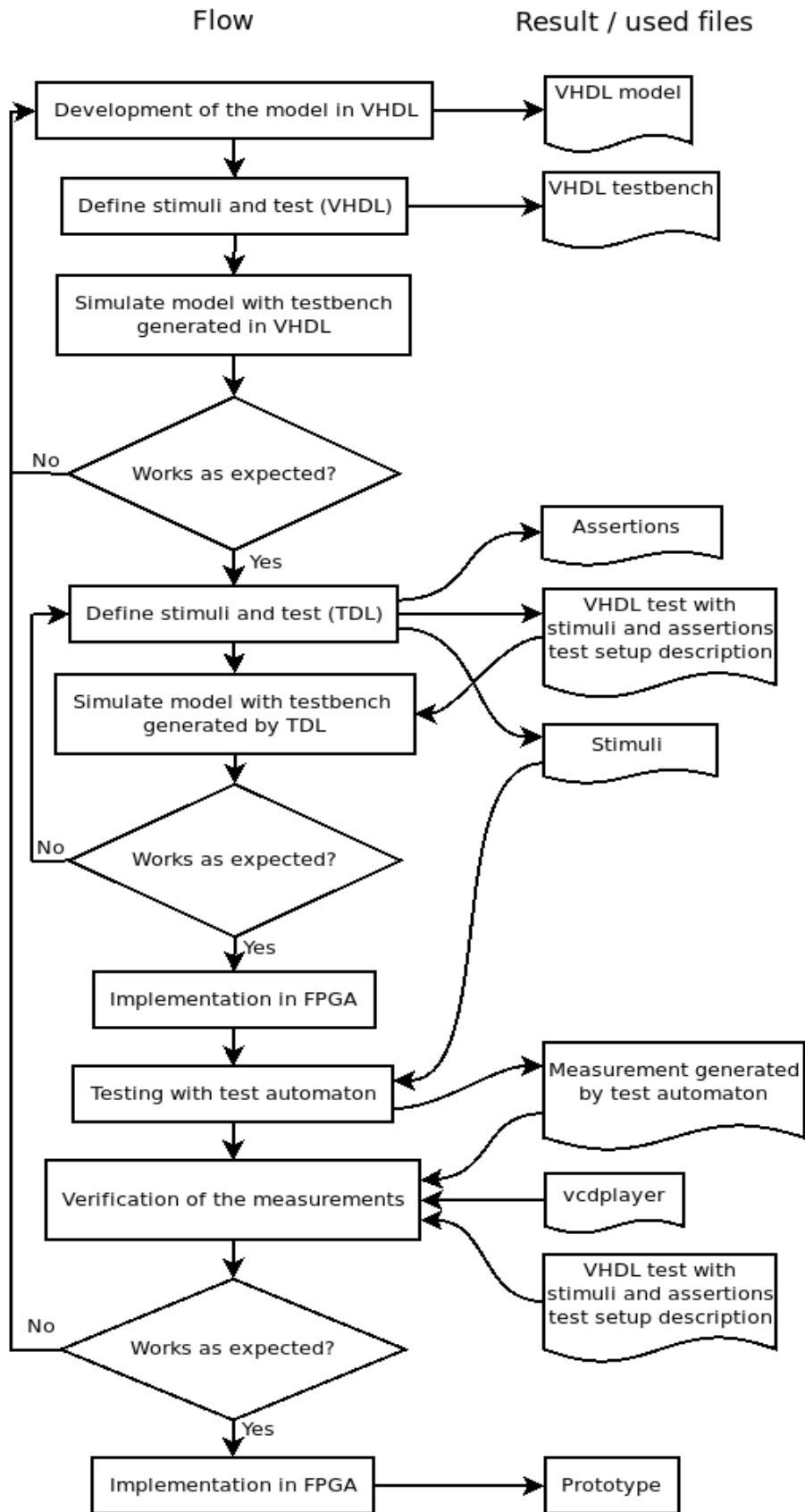


Figure 2: Workflow planned for this work

2 Fundamentals

2.1 Concept of the test systems

2.1.1 General structure of the test system

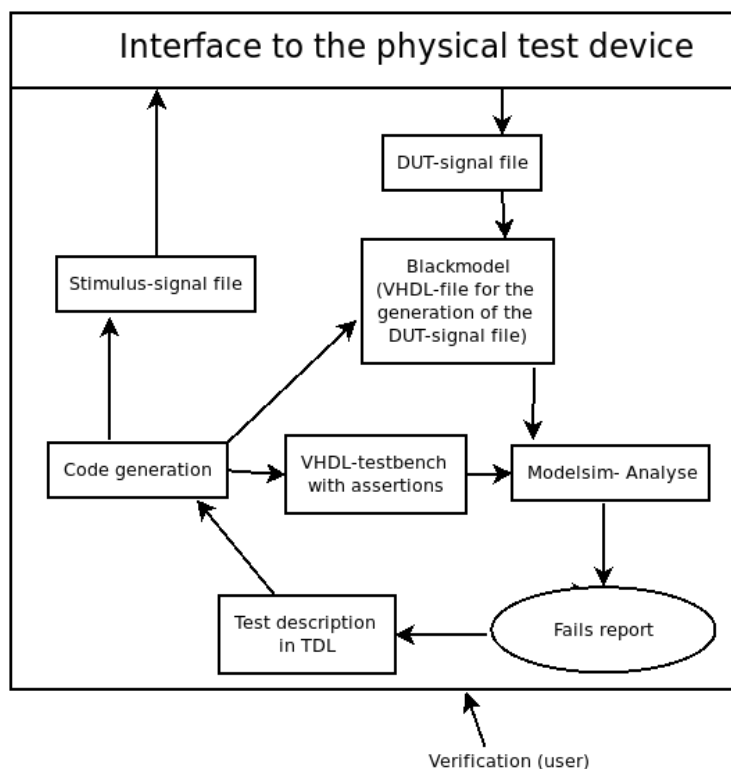
2.1.1.1 *Requirements for the test environment*

To create a good concept, one must first define the requirements for the test environment. The following focal points were laid during the creation of the concept:

- **Ability to write test vectors and evaluate signals from the Device Under Test (DUT).**
The test environment should be able to generate test vectors. At the same time, it must be able to evaluate signals from the DUT.
- **Automating the evaluation of the DUT signals.**
The evaluation of the DUT signals shall happen automatically.
- **Clear Interface Definition.**
There must be a clear interface definition that limits the scope of the project to be managed. This definition must be clearly and intelligibly defined in order to facilitate the work on the project.
- **User interface.**
Writing test vectors and running tests should be as simple as possible. For this reason, the user should only have to deal with a concept that combines test vectors and test description. Only with a uniform user interface, a simplification of the test process can be made.
- **Use of standards.**
To create the most efficient and extensible environment, as many standards as possible should be used to implement this test environment.
- **Use of proven tools.**
Whenever possible, proven tools should be used to evaluate the signals. This facilitates implementation and the creation of a coherent workflow.

2.1.1.2 *Concrete concept of the test environment*

The following concept has been developed based on the requirements defined above, Figure 3 shows an overview of the whole concept. The test principle of the functional test was defined as the test type, since it is inexpensive and can be carried out with comparatively little periphery. This is important later in the development of the physical test device. The test environment only perceives the DUT as a black box defined by its inputs and outputs. The following subchapters describe the partial concepts shown graphically in Figure 3.



*Figure 3: Concept of the test environment
Obtained from [2]*

2.1.1.3 File Interfaces and Interface Definition

The previous figure shows that two files interact with the interface to the physical test device. The stimulus file and the DUT signal file. The stimulus file describes the test vector needed to stimulate the DUT. A test vector always describes one or more states of signals that change over time. These can be binary or real signals. Both the time behavior and the signal value are important here. The other file carries the information of the test vector response and is output from the DUT via the tester and sent to the test system. The block diagram in the next figure shows how both files act as an interface to the Automatic Test Equipment (ATE).

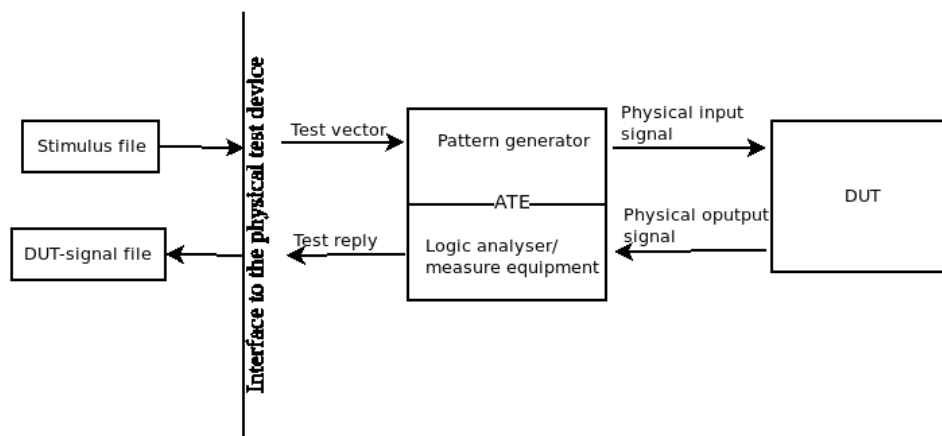


Figure 4: Processing the files into test vectors and generating the test response
Obtained from [2]

Since both files provide information about the time course of signals, it is advisable to save them in a format that supports this. In professional software, waveform viewers are mostly used to store the waveforms in dedicated file formats. There are a number of proprietary and self-developed formats. Since standards are preferred in this work, the choice fell on the Value Change Dump (VCD) file. The VCD standard is described in the Verilog standard (IEEE 1364-2001) and is supported by a variety of tools. Due to its easy-to-understand syntax and wide distribution, this file format is particularly suitable as an interface.

2.1.1.4 Evaluation of the DUT signals

Once the test vectors have been generated and the output signals from the DUT have been recorded and stored, the question arises as on how the signals can be evaluated. It was decided to take advantage of them in the hardware description workflow tested with testbenches. It uses testbenches to automatic test the VHDL logic. These testbenches are standardized and can test the written VHDL code for accuracy with tools such as MentorGraphics ModelSim. Evaluating signals with ModelSim has the advantage of being able to access the capabilities of the test description with testbenches without having to think about their own concepts. ModelSim is also scriptable, and allows the automatic integration into the workflow of the test description. Using the concept of Modelsim raises the question of how it is feasible to check the DUT signals with the help of a testbench for correctness. It was considered, in addition to the testbench, to generate a so-called black box model. This black box model reads out the VCD file with the `std.textio.all` library in VHDL and generates the outputs according to this scheme. This allows you to use the same workflow as traditional VHDL block testing. With this procedure, it is also possible to look at the waveforms with the SignalViewer available in ModelSim and thus intervene manually in the test evaluation. A further advantage results from the fact that such an evaluation also allows conventional testing of VHDL code with the test description language. This then generates testbenches, if desired, that can be used to verify a VHDL file [2].

2.1.1.5 Concept of the workflow

The workflow for creating test descriptions should be centralized. This means that writing test vectors and checking for correctness of the outputs should all take place in one user interface. The concept is to accomplish this from within the Eclipse environment. Since Eclipse can be extended with plugins, it is conceivable to write a plugin which loads the generated test vector file into the ATE after the successful description of the test with the intended test description and after generating the required files with the push of a button on the surface. The ATE then processes this test vector file and generates the test vectors as described in Figure 5 and records the test response. The ATE then sends the test response back to the test system, where it then arrives as a readable VCD file. When the answer arrives, a ModelSim script generates the ModelSim workspace from the generated testbench, generated blackbox model, and VCD file. The script then tests for correctness using the testbench information. The result is then sent back to the Eclipse environment and the test engineer gets the answer whether the test failed or passed. If the test fails, ModelSim's error log can be evaluated and corrective actions can be taken.

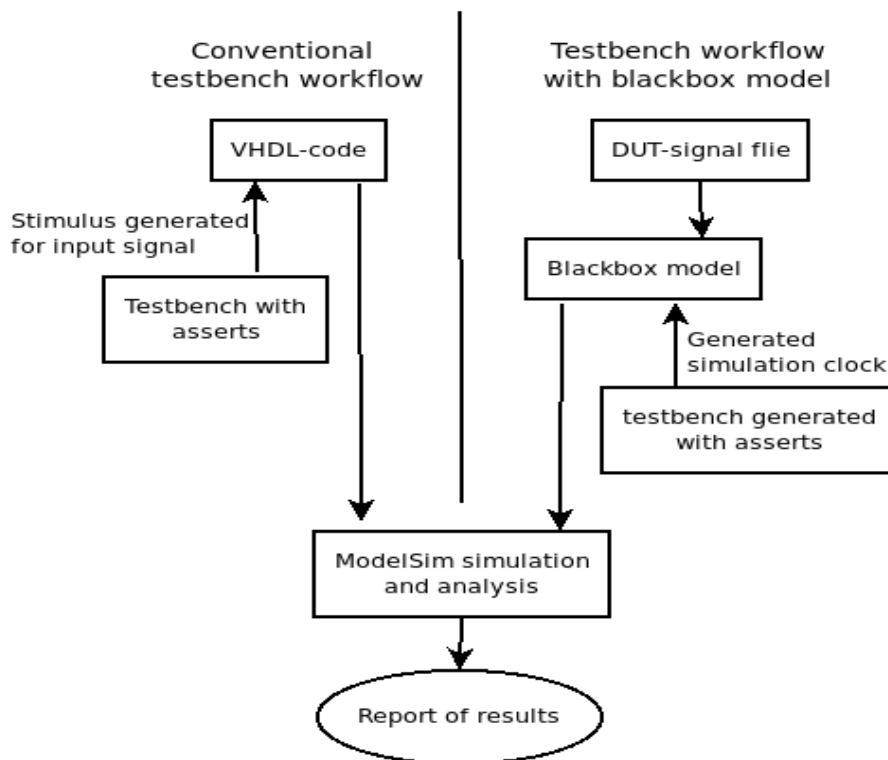


Figure 5: Test with modelsim vs. signal evaluation with blackbox model
Obtained from [2]

2.1.2 Description of the Value Change Dump File

As described in Chapter 2.1.1.3, VCD was taken as an interface to the ATE. This subchapter describes the structure of these files and how they should look to be readable by the black box model.

2.1.2.1 Header

```
$comment
    Test File No. 1
$end
$date
    Wed Nov 26 16:23:28 2014
$end
$version
    dumpports ModelSim Version 10.1e
$end
$scope module ModelSim $end
```

Figure 6: VCD file: Header
Obtained from [2]

The VCD file starts with a header which usually begins with a description of the file. So this file is Test File No. 1. Then follows the date on which the file was created. The version states which program has created the file. In our example, the dumpports is the Modelsim software. Then the scope has to be described, which happens here with the module ModelSim. The header is only for file information and is not necessary for the black box model. The state machine ignores these lines.

2.1.2.2 Timescale

```
$timescale
    1ns
$end
```

Figure 7: VCD file: Timescale
Obtained from [2]

Each VCD file requires a time scale, which provides the tool which provides the tool with the unit in which the signal changes are being made. In this example, the time scale is 1 ns. The state machine of the black box model also ignores this information because in the file is given this information by the code generator and is billed in the generics of the black box model.

2.1.2.3 Signal definitions

```
$var wire 1 ! a $end  
$var wire 8 - b $end  
$enddefinitions $end
```

*Figure 8: VCD file: signal definition
Obtained from [2]*

The signals are defined according to the time scale. These definitions correspond to the Verilog standard and declare which signals or bus signals are used. In this example, two signals have been defined: a 1-bit signal with the flag **!** and the description **a** and an 8-bit signal bus with the flag **-** and the description **b**. The descriptions are not relevant for the evaluation of the signals and are also ignored by the state machine. However, the flags are important for later reading in the files. The current status of the state machine responds to the flags: **! - +% & / () =?** These flags are automatically generated by the code generator in the order of the defined signals. The ports in the black box model are also written in this order. Since the code generator does the work here too, this information is ignored by the black box model.

2.1.2.4 Initial value of the signals

```
$dumpvars  
0 !  
b00000000 -  
$end
```

*Figure 9: VCD file: Initial value
Obtained from [2]*

The keyword `$dumpvars` describes the initial values of the previously defined signal values. This keyword is the first thing the state machine responds to. Here, the signals are described before the state 0.

2.1.2.5 Signal changes and timestamps

```
#0
1 !
b00000001 -
#10
0 !
b00000010 -
#20
1 !
b00000011 -
#30
0 !
b00000100 -
#40
1 !
b00000101 -
#50
0!
#60
1!
#70
0!
#80
1!
#90
0!
```

*Figure 10: VCD file: Signal changes and timestamps
Obtained from [2]*

This section of a VCD file is the actual waveform. Here, the qualifier `#` defines timestamps that indicate when the signals change. The number behind the qualifier is multiplied by the defined time scale. Thus, in our example at time stamp `# 30`, ie at time 30 ns, the 1-bit signal `a` changes to logical 0 and the 8-bit signal bus `b` to the integer value 4. Here, the state machine responds to each time stamp and prepares the Signal changes in order to then write this time accurate to the output.

2.1.2.6 End of the file

```
$vcdclose  
#100  
$end
```

*Figure 11: VCD file: end of the file
Obtained from [2]*

The keyword **\$vcdclose** marks the end of the recorded signals with the following timestamp. In this example, the file is stopped after 100 ns. The black box model state machine responds to this keyword and terminates both communications with the file and the modelsim simulation.

2.2 Test Description Language

2.2.1 What is a Test Description Language

TDL is a language that supports the design and documentation of formal test descriptions that may be the basis for the implementation of executable tests in a given test framework, such as TTCN-3. Application areas of TDL that will benefit from this homogeneous approach to the test design phase include:

- Manual design of test descriptions from a test purpose specification, user stories in test driven development or other sources.
- Representation of test descriptions derived from other sources such as MBT test generation tools, system simulators, or test execution traces from test runs.

TDL supports the design of black-box tests for distributed, concurrent real-time systems. It is applicable to a wide range of tests including conformance tests, interoperability tests, tests of real-time properties and security tests based on attack traces.

TDL clearly separates the specification of tests from their implementation by providing an abstraction level that lets users of TDL focus on the task of describing tests that cover the given test objectives rather than getting involved in implementing these tests to ensure their fault detection capabilities onto an execution framework.

TDL is designed to support different abstraction levels of test specification. On the one hand, the concrete syntax of the TDL meta-model may hide meta-model elements that are not needed for a declarative (more abstract) style of specifying test descriptions. For example, a declarative test description could work with the time operations wait and quiescence instead of explicit timers and operations on timers (see clause 9).

On the other hand, an imperative (less abstract or refined) style of a test description supported by a dedicated concrete syntax could provide additional means necessary to derive executable test descriptions from declarative test descriptions. For example, an imperative test description could include timers and timer operations necessary to implement the reception of System Under Test (SUT) output at a tester component and further details. It is expected that most details of a refined, imperative test description can be generated automatically from a declarative test description.

Supporting different levels of abstraction by a single TDL meta-model offers the possibility of working within a single language and using the same tools, simplifying the test development process that way.

2.2.2 Why use a Test Description Language

TDL bridges the gap between high-level test purpose specifications and executable test cases. It provides a generic language for the formal specification of test descriptions which can be used as the basis for the implementation of concrete tests on a given test execution platform or simply for the visualization of test scenarios for different stakeholders. TDL is designed to support the black-box test of distributed, concurrent real-time systems.

TDL supports a scenario-based approach using modeling techniques from model-based testing and UML Testing Profile (UTP). Test scenarios are described at a higher abstraction level than what is possible with scripting languages such as TTCN-3. It is indifferent on the basic communication mechanism used between tester and SUT being message-based, procedural or communication-based on shared variables or other types of interfaces. Furthermore, TDL can be used as an intermediate representation of tests generated from other sources, e.g. simulators, test case generators, or logs from previous test runs.

TDL is designed around a meta-model approach based on the OMG's meta-object facility MOF (OMG MOF, 2013) to describe its abstract syntax. This way, it is able to support different concrete syntaxes, also with a different feature set according to the needs of different application domains.

While the TDL meta-model is based on a well defined underlying formal semantics, it is possible to provide supportive tools for correctness analysis of (manually) specified test descriptions, the construction of test cases according to a chosen fault model, the visualization of test run results, or the exchange of test descriptions between different tools. The formal semantics prevents misinterpretation of the artifact specifications between different tools. The approach is driven by industry to foster the benefits of model-based software engineering in the test process. ETSI has set up Special Task Force (STF) to standardize TDL.

The trend towards a higher degree of system integration such as in case of cyber-physical systems or service-oriented architectures leads to a growing importance of integration testing of such distributed, concurrent, and real-time systems. Integration testing, which is a black-box testing approach, encompasses also conformance testing of a system against a standard and interoperability testing of two or more systems of different vendors.

Test automation is required for many phases of the quality assurance process such as regression tests, smoke tests, or acceptance tests. Automating tests is a software development activity that

involves the production of test code/scripts. Moving towards a model-based approach in testing, there are some obstacles to overcome for the wide-scale introduction of model-based testing. One of these obstacles is the existing divergence between manually created testing artifacts (which must be understood and managed by humans) and the need for defining them formally to allow automation. As a consequence, there has been a methodology gap between the simple expression of a test purpose described frequently in prose and the complex coding of executable tests scripts. TDL (ETSI ES 203 119, 2013) covers that gap.

Dedicated test descriptions will have a positive impact on the quality of the tests through better design and by making them easier to review by non-testing experts. This will improve the general productivity of test development. Moreover, it is also important to provide a fault-free transfer of specifications between tools participating in the development of tool-chains where manual interaction by a test engineer is often needed.

The language design of TDL centers on the meta-modeling approach for the abstract syntax. A number of concrete syntaxes can be defined that all map to the same meta-model to provide dedicated support for different application domains. Given that the elements of the meta-model are formally defined, TDL specifications can be analysed beforehand for consistency and internal correctness to ensure a high quality of the test descriptions. Being an abstract test specification language, different test implementations can be derived to reflect the particularities of concrete test environments, e.g. a distributed tester could be derived supporting asynchronous message-passing communication between tester and system under test (SUT) or a sequential tester that puts emphasis on validating real-time constraints between tester/SUT interactions.

2.2.3 Standards for TDLs

ETSI ES 203 119-1: Abstract Syntax and Associated Semantics

“This document specifies the abstract syntax of the Test Description Language (TDL) in the form of a meta-model based on the OMG ® Meta Object Facility™ (MOF). It also specifies the semantics of the individual elements of the TDL meta-model. The intended use of the present document is to serve as the basis for the development of TDL concrete syntaxes aimed at TDL users and to enable TDL tools such as documentation generators, specification analyzers and code generators.

The specification of concrete syntaxes for TDL is outside the scope of the present document. However, for illustrative purposes, an example of a possible textual syntax together with its application on some existing ETSI test descriptions are provided.” [3]

ETSI ES 203 119-2: Graphical Syntax

“This document specifies the concrete graphical syntax of the Test Description Language (TDL). The intended use of the present document is to serve as the basis for the development of graphical

TDL tools and TDL specifications. The meta-model of TDL and the meanings of the meta-classes are described in ETSI ES 203 119-1.” [4]

ETSI ES 203 119-3: Exchange Format

“This document specifies the exchange format of the Test Description Language (TDL) in the form of an XML Schema derived from the TDL meta-model [1]. The intended use of the present document is to serve as the specification of the format used for exchange of model instances and tool interoperability between TDL-compliant tools.” [5]

ETSI ES 203 119-4: Structured Test Objective Specification

“This document specifies an extension of the Test Description Language (TDL) enabling the specification of structured test objectives. The extension covers the necessary additional constructs in the abstract syntax, their semantics, as well as the concrete graphical syntactic notation for the added constructs. In addition, textual syntax examples of the TDL Structured Test Objectives extensions as well as BNF rules for a textual syntax for TDL with the Structured Test Objectives extensions are provided. The intended use of the present document is to serve both as a foundation for TDL tools implementing support for the specification of structured test objectives, as well as a reference for end users applying the standardized syntax for the specification of structured test objectives with TDL.” [6]

2.2.4 Design Considerations

TDL makes a clear distinction between concrete syntax that is adjustable to different application domains and a common abstract syntax, which a concrete syntax is mapped to.

The definition of the abstract syntax for a TDL specification plays the key role in offering interchangeability and unambiguous semantics of test descriptions. It is defined in the present document in terms of a MOF meta-model.

A TDL specification consists of the following major parts that are also reflected in the meta-model:

- A test configuration consisting of at least one tester and at least one SUT component and connections among them reflecting the test environment.
- A set of test descriptions, each of them describing one test scenario based on interactions between the components of a given test configuration and actions of components or actors. The control flow of a test description is expressed in terms of sequential, alternative, parallel, iterative, etc. behavior.
- A set of data definitions that are used in interactions and as parameters of test description invocations.
- Behavioral elements used in test descriptions that operate on time.

Using these major ingredients, a TDL specification is abstract in the following sense:

- Interactions between tester and SUT components of a test configuration are considered to be atomic and not detailed further. For example, an interaction can represent a message exchange, a remote function/procedure call, or a shared variable access.
- All behavioral elements within a test description are totally ordered, unless it is specified otherwise. That is, there is an implicit synchronization mechanism assumed to exist between the components of a test configuration.
- The behavior of a test description represents the expected, foreseen behavior of a test scenario assuming an implicit test verdict mechanism, if it is not specified otherwise. If the specified behavior of a test description is executed, the 'pass' test verdict is assumed. Any deviation from this expected behavior is considered to be a failure of the SUT, therefore the 'fail' verdict is assumed.
- An explicit verdict assignment may be used if in a certain case there is a need to override the implicit verdict setting mechanism (e.g. to assign 'inconclusive' or any user-defined verdict values).
- The data exchanged via interactions and used in parameters of test descriptions are represented as values of an abstract data type without further details of their underlying semantics, which is implementation-specific.
- There is no assumption about verdict arbitration, which is implementation-specific. If a deviation from the specified expected behavior is detected, the subsequent behavior becomes undefined. In this case, an implementation might stop executing the TDL specification.

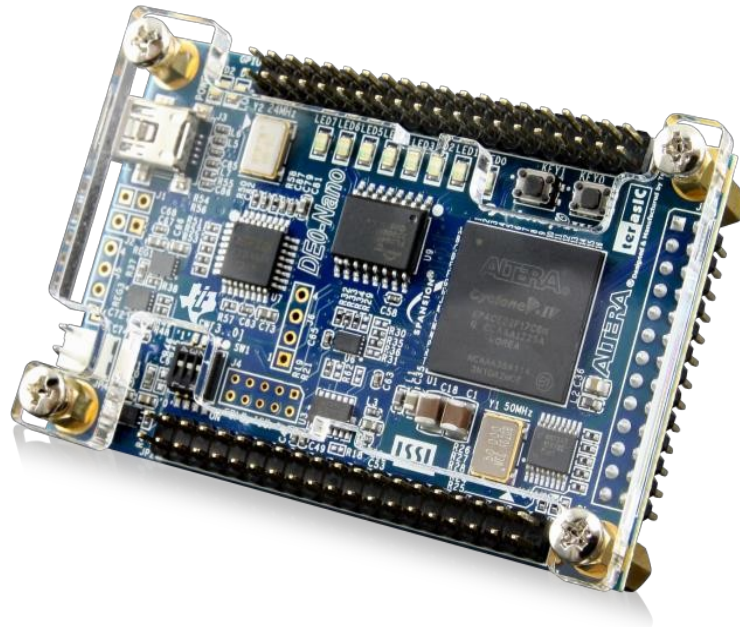
A TDL specification represents a closed system of tester and SUT components. That is, each interaction of a test description refers to one source component and at least one target component that are part of the underlying test configuration a test description runs on. The actions of the actors (entities of the environment of the given test configuration) may be indicated in an informal way.

Time in TDL is considered to be global and progresses in discrete quantities of arbitrary granularity. Progress in time is expressed as a monotonically increasing function. Time starts with the execution of the first ('base') test description being invoked.

The elements in a TDL specification may be extended with tool, application, or framework specific information by means of annotations.

2.3 Used prototyping board

The Terrasic DE0-Nano board has been used because it is a low-cost prototyping board, easy to program and has enough power for the model to be implemented.



*Figure 12: DE0-Nano board
Obtained from [7]*

The technical characteristics of the board are explained in appendix E.

3 Development of the VHDL model

3.1 Specifications of the VHDL model

A simple model in which fail detection is easy was decided to be implemented because the main point of the thesis is not the model, but to ensure that the TDL and the test automaton are working as specified. The TicTacToe game was chosen because it is easy to understand and to test. In addition a simple board can be developed in order to test the correct behavior of the model in an interactive way.

3.1.1 How TicTacToe works

Tick-tock is a pen and paper game for two players, **X** and **O**. Players take turns marking spaces on a 3x3 grid. The player who places three of his marks in a horizontal, vertical or diagonal row wins the game.

The following example shows a game that is won by the first player, **X** [8] :



Figure 13: Example of how to play TicTacToe in which X wins.
Obtained from [8]

Strategy

Optimal strategy for player **X**: In each grid, the shaded red **X** shows the optimal move, and the location of **O**'s next move gives the next subgrid to evaluate. It is important to notice that only two sequences of moves by **O** (both starting with center, top-right, left-mid) lead to a draw, with the remaining sequences leading to wins from **X** [8] .

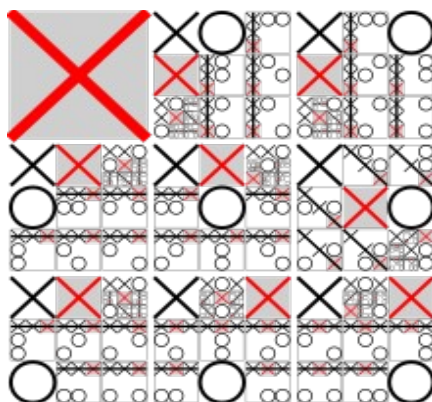


Figure 14: Optimal strategy for player X
Obtained from [8]

Optimal strategy for player O: Player O can always force a win or draw by taking the central space. If it is taken by X, then O must take a corner. [8]

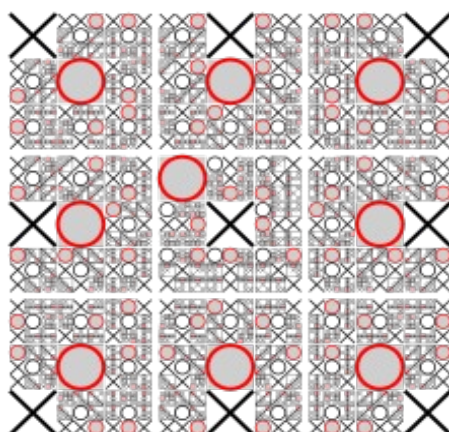


Figure 15: Optimal strategy for player O
Obtained from [8]

A player can play a perfect game of tic-tac-toe (to win or, at least, draw) if they choose the first available move from the following list:

1. **Win:** If the player has two in a row, they can place a third to get three in a row.
2. **Block:** If the opponent has two in a row, the player must play the third to block the opponent.
3. **Fork:** Create an opportunity where the player has two ways to win (two non-blocked lines of 2).
4. **Blocking an opponent's fork:**

- **Option 1:** The player should create two in a row to force the opponent into defending, as long as it doesn't result in them creating a fork. For example, if "X" has two opposite corners and "O" has the center, "O" must not take a corner in order to win. (Taking a corner in this situation creates a fork for "X" to win.)
 - **Option 2:** If there is a situation where the opponent can fork, the player should block that fork.
5. **Center:** A player took the center. (If it is the first move of the game, taking a corner gives the second player more opportunities to make a mistake and may therefore be the better choice; however, it makes no difference between perfect players.)
 6. **Opposite corner:** If the opponent is in the corner, the player plays the opposite corner.
 7. **Empty corner:** The player plays in a corner square.
 8. **Empty side:** The player plays in a middle square on any of the 4 sides.

The first player, **X**, has 3 possible positions to mark during the first turn. Apparently, it might appear that there are 9 possible positions, corresponding to the 9 squares of the grid. However, by turning the board, we will find that in the first turn, each corner mark is equivalent to any other corner mark. The same is true for each border mark (middle side). Therefore, for strategy purposes, there are only three possible first marks: corner, edge or center. Player **X** may win or force a draw from any of these starting marks; however, taking the corner gives the opponent the smallest choice of squares that must be played to avoid losing. This makes the corner the best opening move for **X**, when the opponent is not a perfect player.

The second player, **O**, must respond to **X**'s opening in such a way as to avoid the forced win. Player **O** must always respond to a corner opening with a center, and to a center opening with a corner. An edge opening must be answered either with a center, a corner mark next to the **X**, or an edge mark opposite the **X**. Any other responses will allow **X** to force the win. Once the opening is completed, **O**'s task is to follow the above list of priorities in order to force the draw, or else to gain a win if **X** makes a weak play [8].

3.1.2 Game board

A printed circuit board (PCB) has been designed and built to prove the behavior of the model in a practical way.

The plan was to develop a cheap, easy to design and intuitive board that represents the TicTacToe game board.

The board includes:

A 3x3 matrix of bi-color LEDs (green and red) that represents the game board. Colours are assigned to players: player 1 is red (R) and player 2 is green (G).

A 3x3 matrix of push buttons create a keyboard in which the position of the push button corresponds to the position on the game board.

Systematic testing of digital hardware systems by means of test automaton

Additionally there are another three LEDs indicating the game's result.. They indicate who is the winner or if there is a draw.

The board also includes a push button to reinitialize the game.

The next figure shows the sketch of the interface to the user where the yellow circles represent LEDs, and the gray squares are push buttons. P1 shows that the player 1 has won, P2 shows that the player 2 has won and DR shows that the game has finished with a draw.

The button ST is used to start a new game.

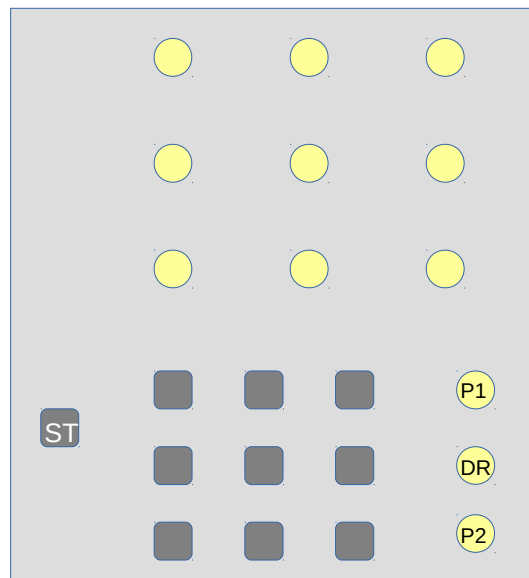


Figure 16: Sketch of the interface to the user

Shift registers with serial input and parallel output have been used to minimize the number of output lines dedicated to controlling the LEDs. The integrated circuit used is the 74LS595N whose datasheet is included in Appendix D. With this design it is possible to control 21 signals with only three lines.

To minimize the number of lines dedicated to controlling the keyboard, the keyboard has been treated like a matrix where the outputs are the columns and the inputs are the rows. In this manner it only takes six lines to control all nine push buttons.

The next figure shows the electrical schematic of the developed game board.

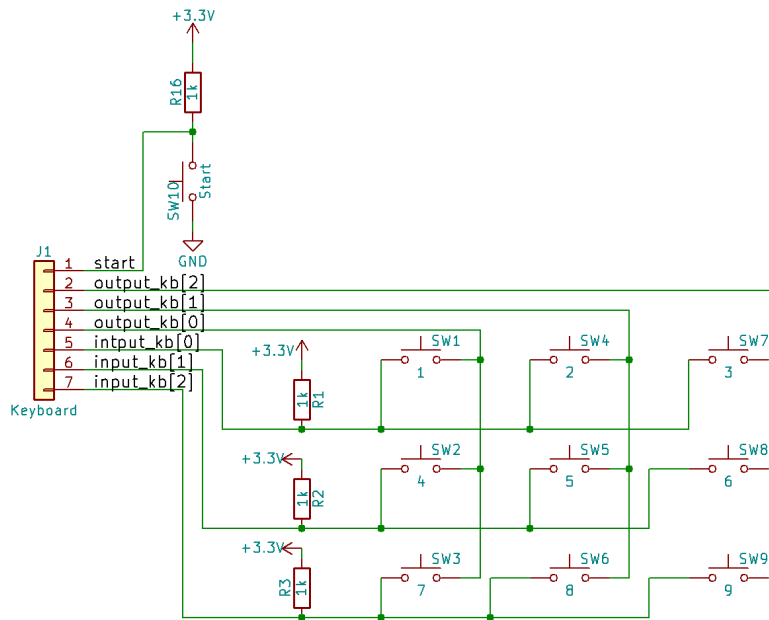


Figure 17: Keyboard control circuit

KiCAD (A cross-platform and open source electronics design automation suite) has been used to develop the board. The main window is shown in the next figure.

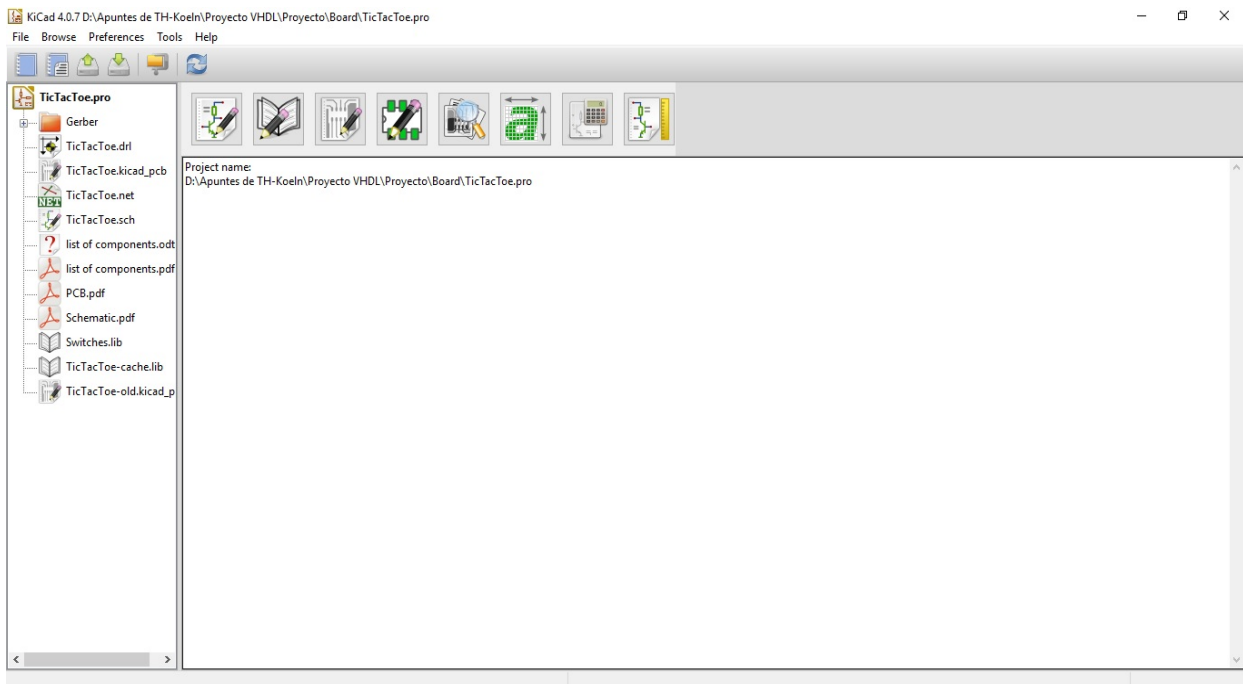


Figure 18: KiCad main window

Systematic testing of digital hardware systems by means of test automaton

Eeschema is the software used to develop the electrical schematic. The next figure shows the interface of Eeschema and the electrical schematic of the game board.

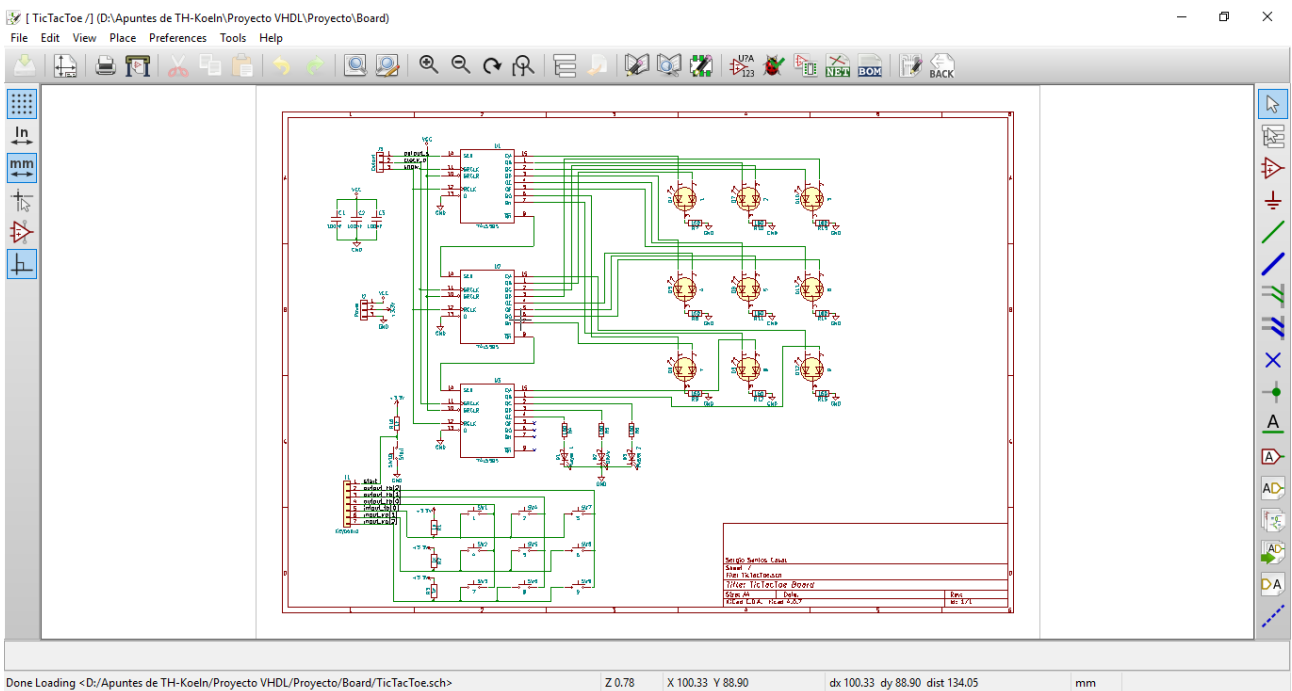


Figure 19: eeschema interface

Pcbnew is the software used to develop the PCD board. The next figure shows the interface of Pcbnew and the conducted board layout of the game board.

Systematic testing of digital hardware systems by means of test automaton

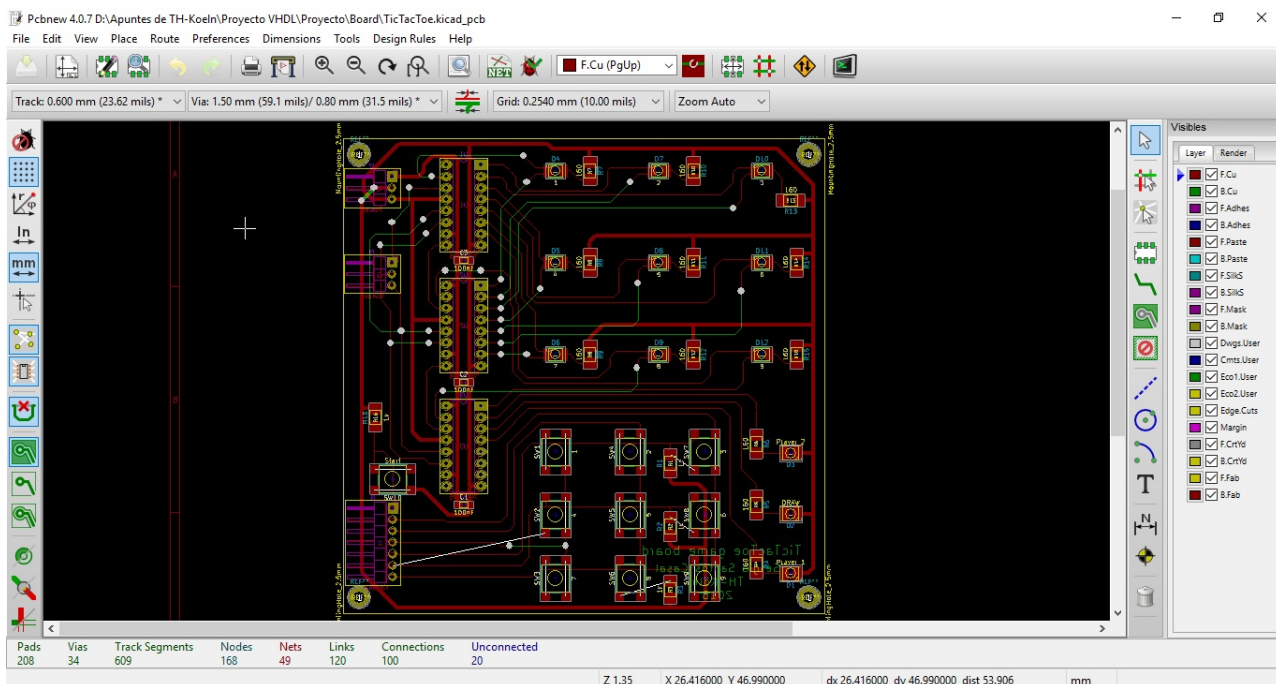


Figure 20: Pcbnew interface

The rendering of the designed board is shown in the two following figures.

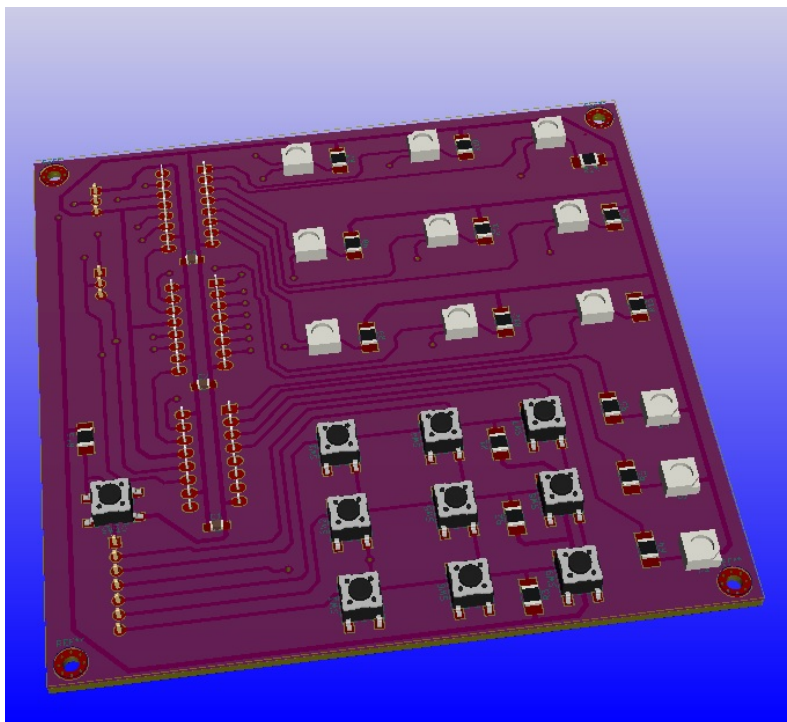


Figure 21: Rendering of the TicTacToe game board: Top side

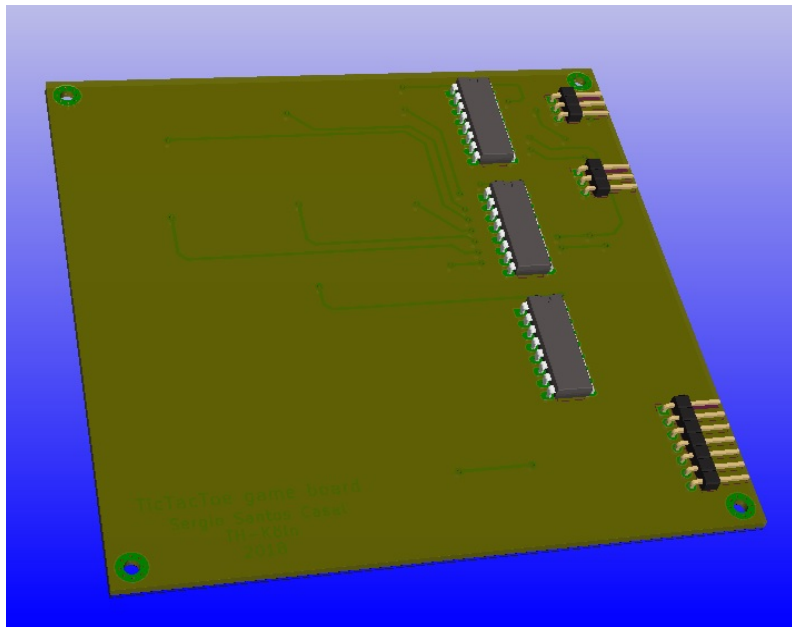


Figure 22: Rendering of the TicTacToe game board: Bottom side

The schematic and PCB designs are included in Appendix A.

Next pictures show the first prototype, in which the start button was not included on the board itself, due to a unforeseen design error. Therefore the start button has been attached to the DE00 nano via jumping wires, as can be seen in figure 24.

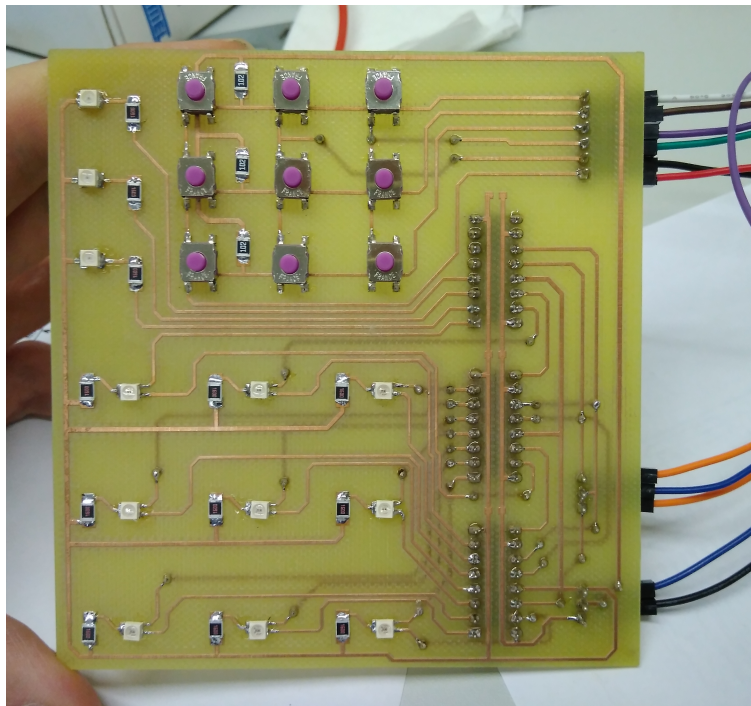


Figure 23: Photo of the game board

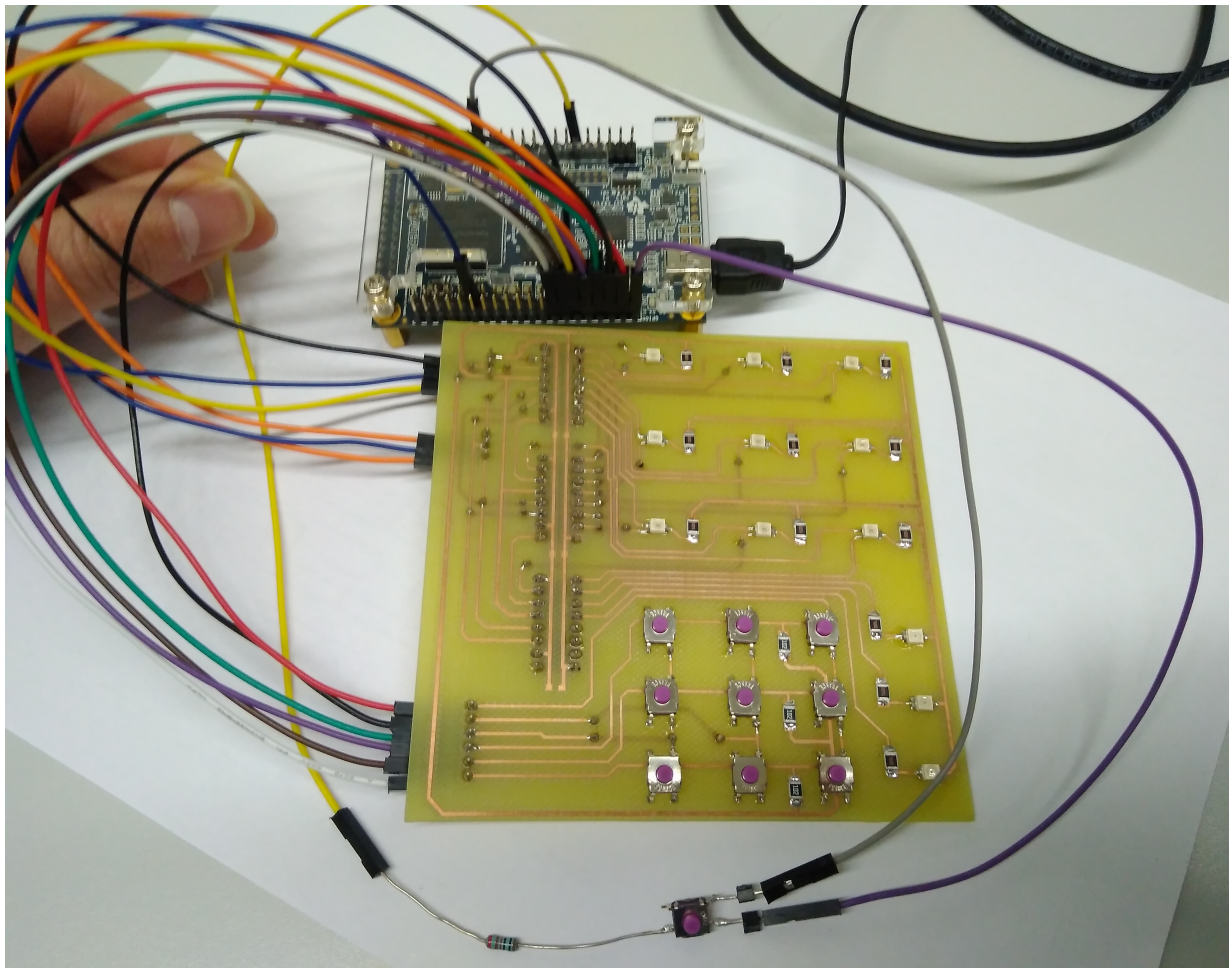


Figure 24: Photo of the game board connected to the prototyping board

3.2 Development of the model

Different models have been made along the project depending on the objectives, improvements or found problems.

The **first model "Game against the machine"** is based on this situation. In this case, the machine always starts the game in the central position and in consequence, it is not possible to win the game.

In this model, a keyboard with 8 keys (the central position is always used by the machine, so does not need a key), one normal LED (in the central position) and eight bi-color LEDs would be used.

This model was developed by using a Finite State Machine (FSM), as it is described in the following flow diagrams, where $P1[x]$ represents the vector of positions used by the machine, $P2[x]$ represents the vector of positions used by the player and $in[x]$ represents the vector of the inputs in the shown flow diagram. The states are represented by circles.

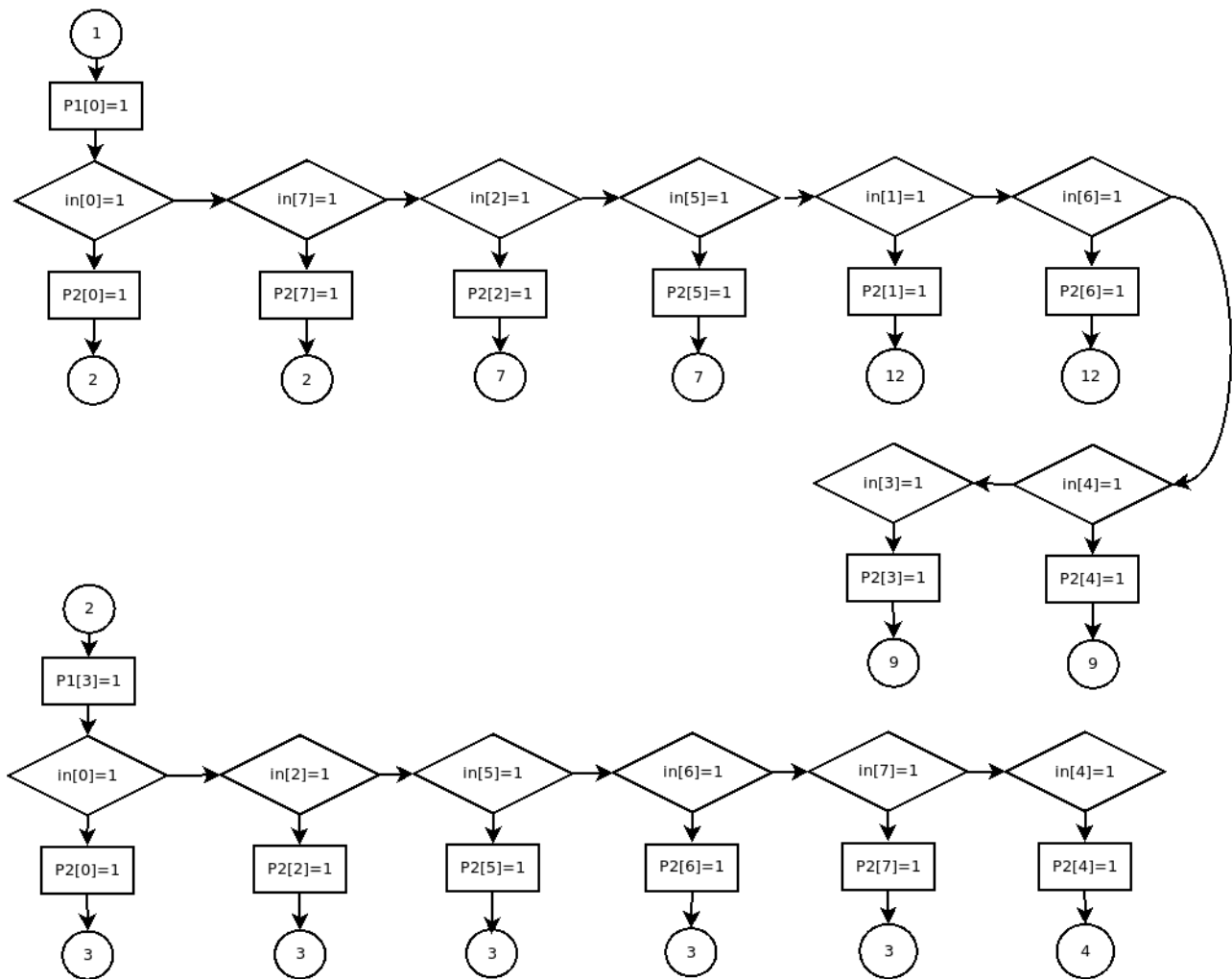


Figure 25: Finite state machine of the model developed for one player (I)

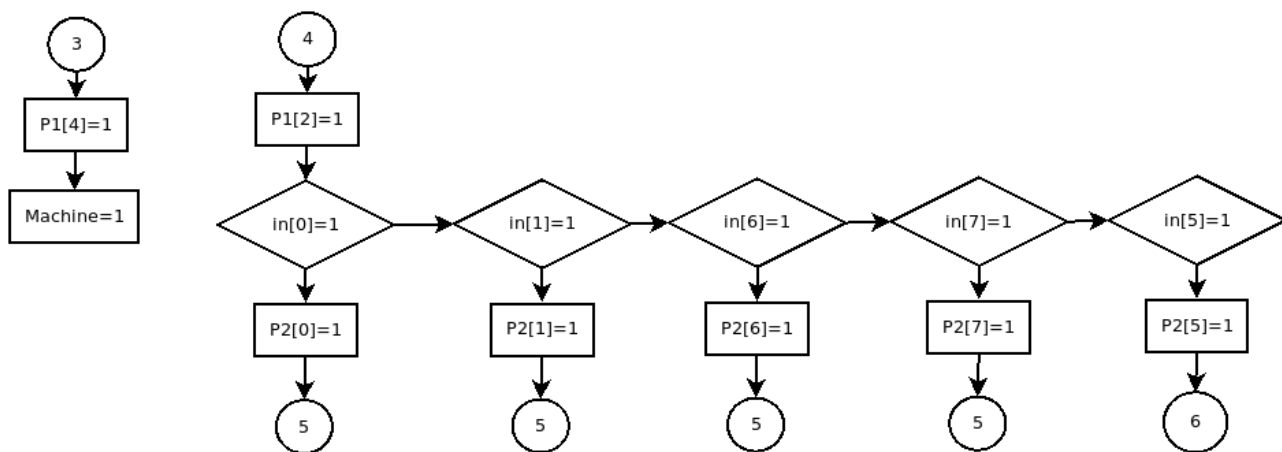


Figure 26: Finite state machine of the model developed for one player (II)

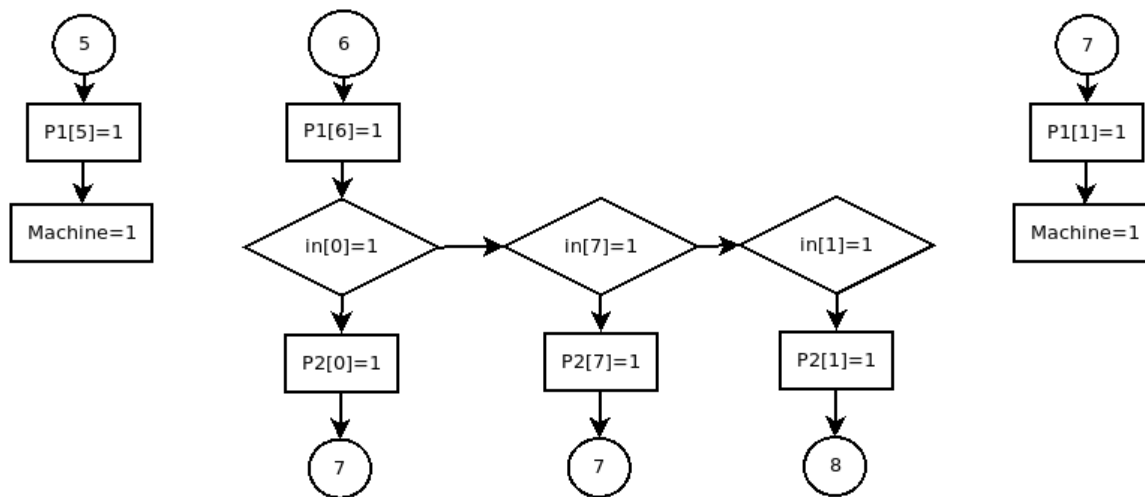


Figure 27: Finite state machine of the model developed for one player (III)

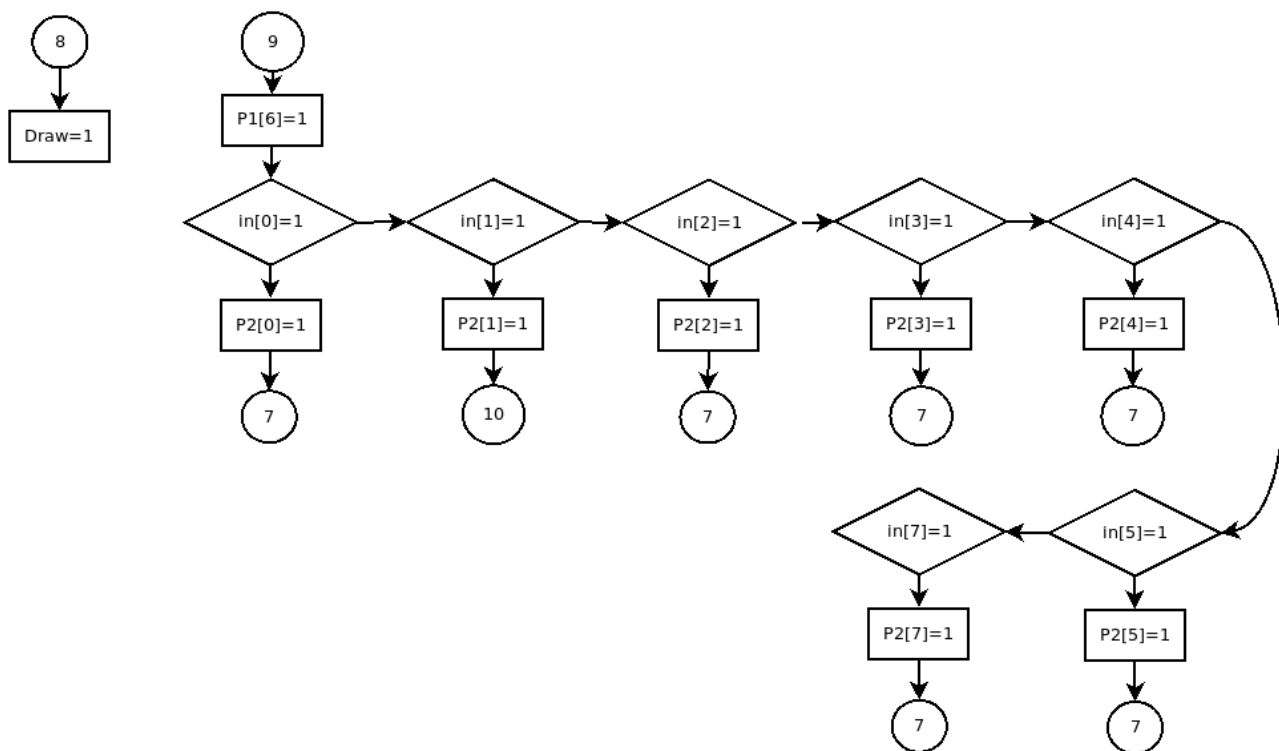


Figure 28: Finite state machine of the model developed for one player (IV)

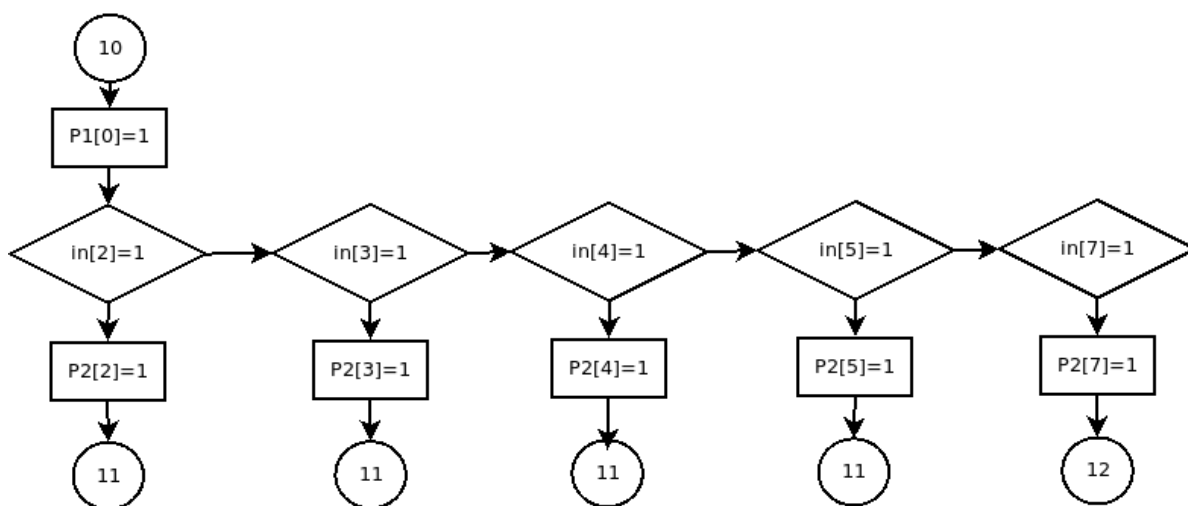


Figure 29: Finite state machine of the model developed for one player (V)

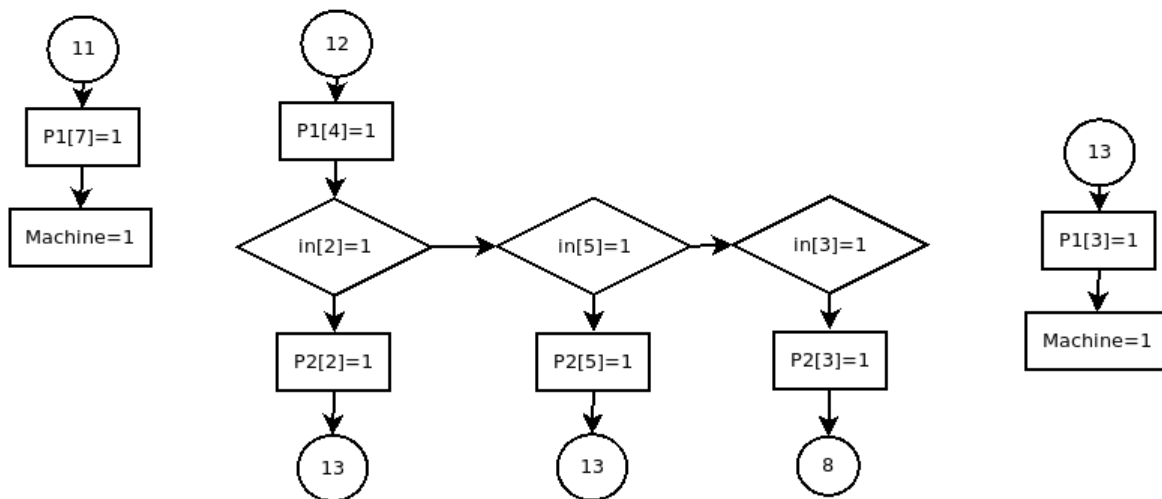


Figure 30: Finite state machine of the model developed for one player (VI)

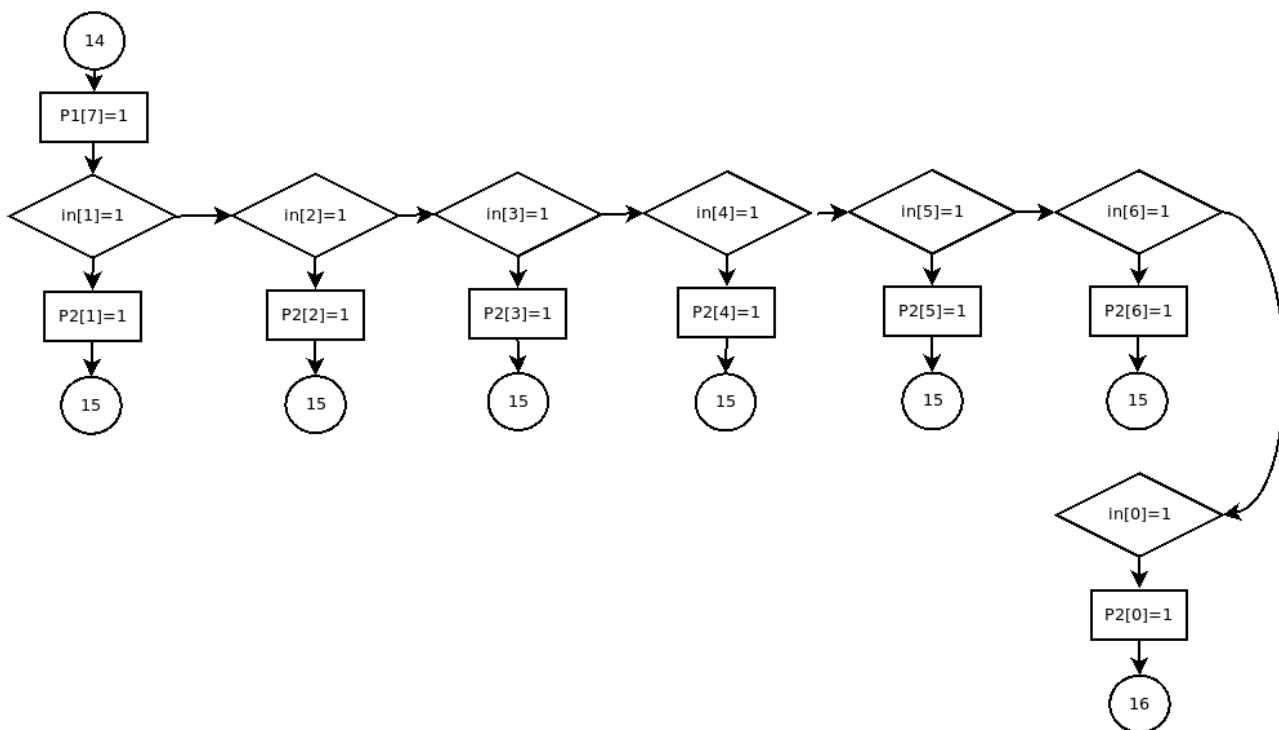


Figure 31: Finite state machine of the model developed for one player (VII)

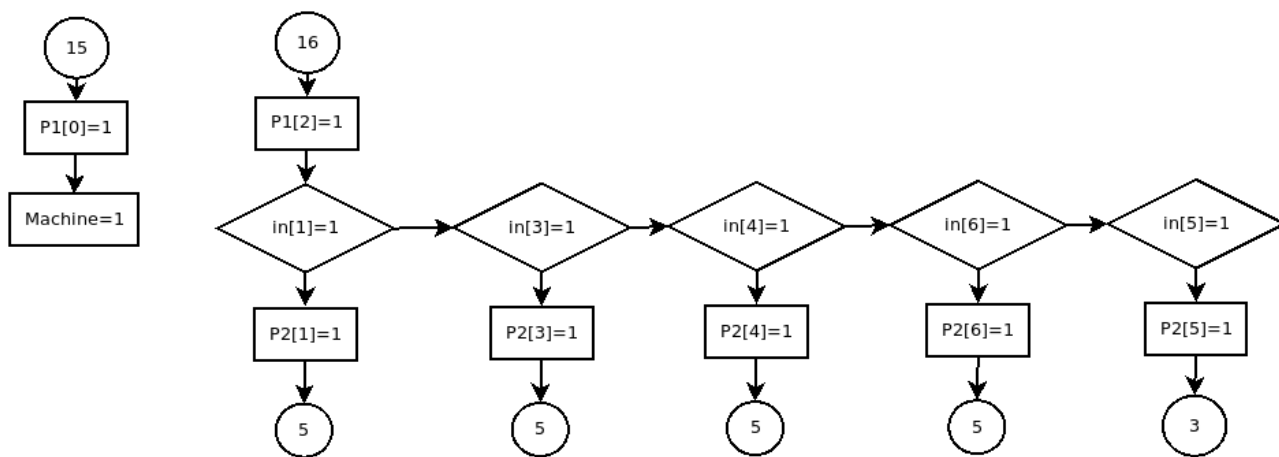


Figure 32: Finite state machine of the model developed for one player (VIII)

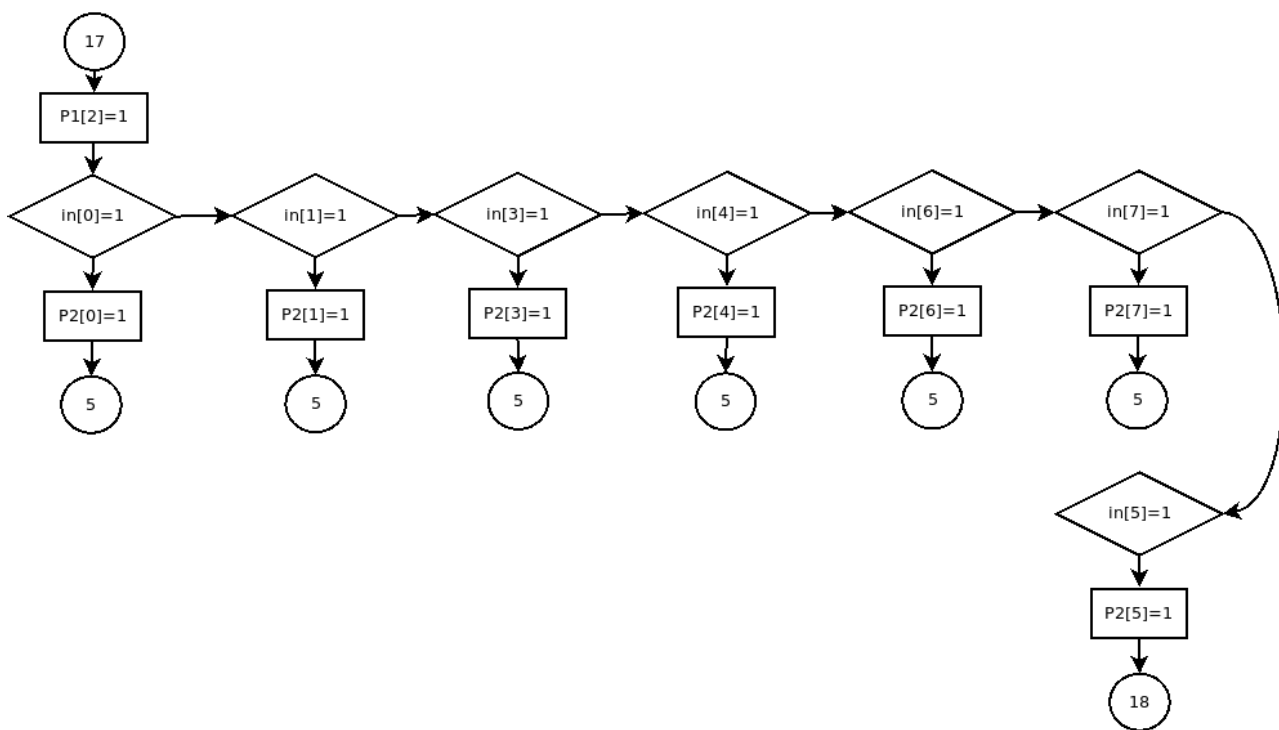


Figure 33: Finite state machine of the model developed for one player (IX)

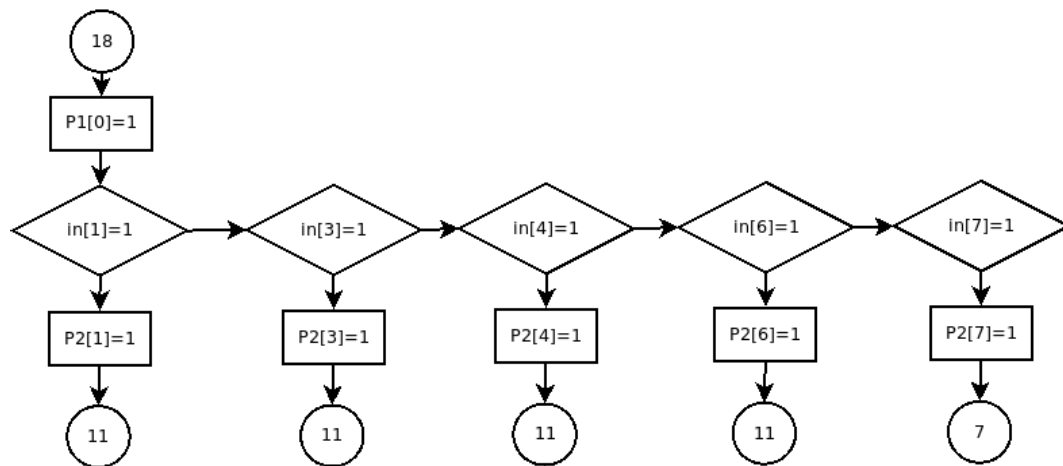


Figure 34: Finite state machine of the model developed for one player (X)

This model was not finished because meanwhile it was decided that a game in which can play two players is more intuitive and the testbench and simulation have more possibilities. Then, the model was changed and the development of a new model allowing two players play was started. The development of the game board was started in parallel.

A new design of inputs and outputs was made in order to reduce the number of lines necessary to connect the VHDL prototyping board and the game board. This point has been explained in the previous chapter during the presentation of the game board design.

The connections between both boards are shown in the next figure.

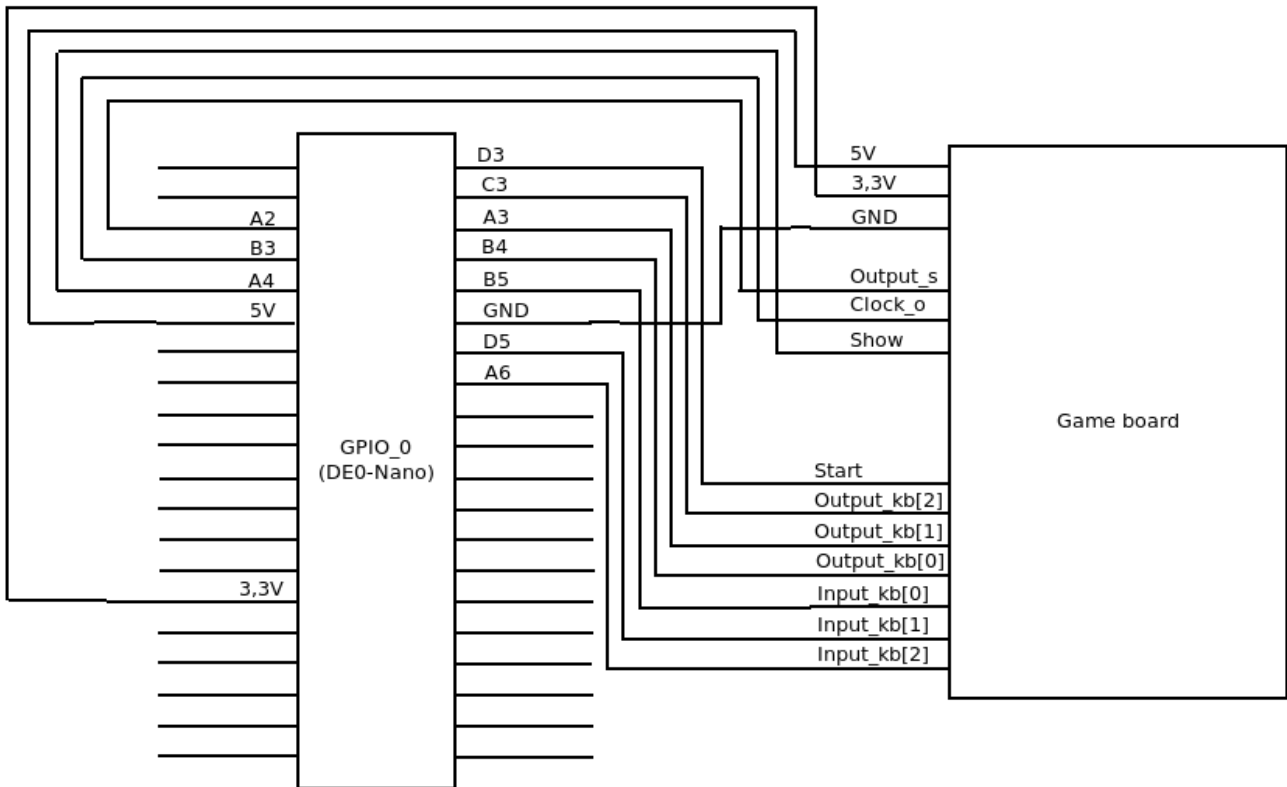


Figure 35: Connections between the VHDL prototyping board and the game board

The prototyping board works with a 50 MHz clock and the model works with a 100 Hz clock, so it was necessary to create a clock divider.

A process with a counter from 0 to 250.000 was created. When the counter reaches 250.000 it inverts the value of the output and starts again.

```

ARCHITECTURE synth OF clock_generator IS
    CONSTANT max_count : INTEGER := 250000;
    SIGNAL clock : std_logic := '0';
    SIGNAL count : INTEGER range 0 to max_count;
BEGIN
    PROCESS (clock_i) --period of 10 ms

    BEGIN
        IF clock_i'event and clock_i='1' THEN
            IF count < max_count THEN
                count <= count+1;
            ELSE
                clock <= not clock;
                count <= 0;
            END IF;
        END IF;
    
```



```

        END IF;
    END PROCESS;
    clock_o <= clock;

END synth;

```



Figure 36: Modul clock generator

The complete code of the frequency divider is in chapter 6.2.2.1 *Clock generator*.

The next step was the development of a process which describes the behavior of the game.

It is important to initialize the signals and outputs at the system startup or when the user restart the game. This is the code written to do so:

```

IF start = '0' or beginning = '1' THEN
    count2 <= 0;
    count3 <= 1;
    count6 <= 0;
    launch <= '0';
    show_i <= '0';
    input_reg <= "000000000";
    reg <= "000000000";
    reg_2 <= "000000000000000000000000";
    reg_3 <= "000000000000000000000000";
    input_reg <= "000000000";
    reg <= "000000000";
    player <= '1';
    serialize <= '1';
    beginning <= '0';

```

A process has been implemented to control the keyboard. All the columns have a value of 1 but one of them, that will have a value of 0. This 0 will be shifting cyclically. For example:

	Column 1	Column 2	Column 3
Instant 1	1	1	0
Instant 2	1	0	1

Instant 3	0	1	1
Instant 4	1	1	0
Instant 5	1	0	1
Instant 6	0	1	1

All the inputs of the keyboard have a value of 1 by default. When a keystroke is produced, the input of that row will have a value of 0 when the value of the column in which this push button is placed is 0 too. In this way is possible to know which push button has been pressed. A variable called `input_i` will take a value determined depending on the row and column of the push button pressed.

There are two counters; one of them, controlled by the variable `count4`, counts from 0 to 9 to produce a frequency of 100 Hz. The other counter, controlled by the variable `count5`, counts from 0 to 2 to choose the column depending on the value of this variable.

```
output_kb_i <= "110" WHEN count5 = 0 ELSE
    "101" WHEN count5 = 1 ELSE
    "011" WHEN count5 = 2 ELSE
    "111";
```

This is the implemented process that controls the keyboard:

```
keyboard: PROCESS (clock_i) --period of 100 ms
    BEGIN
        IF clock_i'event and clock_i='1' THEN
            IF start = '0' or beginning = '1' THEN
                count4 <= 0;
                count5 <= 0;
            ELSIF count4 < 9 THEN
                count4 <= count4+1;
            ELSIF pushed = '1' THEN
                count4 <= 0;
            ELSE
                IF count5 < 2 THEN
                    count5 <= count5+1;
                ELSE
                    count5 <= 0;
                END IF;
                count4 <= 0;
            END IF;
        END IF;
    END IF;
END PROCESS;
```

The keystroke is saved in the variable `input_i`:

```
input_i <= "000000001" WHEN input_kb ="110" AND output_kb_i = "110" else
"000000010" WHEN input_kb ="110" AND output_kb_i = "101" else
"000000100" WHEN input_kb ="110" AND output_kb_i = "011" else
"000001000" WHEN input_kb ="101" AND output_kb_i = "110" else
"000010000" WHEN input_kb ="101" AND output_kb_i = "101" else
"000100000" WHEN input_kb ="101" AND output_kb_i = "011" else
"001000000" WHEN input_kb ="011" AND output_kb_i = "110" else
"010000000" WHEN input_kb ="011" AND output_kb_i = "101" else
"100000000" WHEN input_kb ="011" AND output_kb_i = "011" else
"000000000";
```

`reg_3` contains a vector with the keys pushed by each player. Through the reading of this vector, it is possible to detect if the game has finished because any of the players has won. The variable ***play*** will take the value 0 when any played has won and the value 1 in the other case. When a draw happened, although ***play*** does not take value 0, no more keyboard inputs will be processed.

```
Play<='0' when reg_3(9)='1' and reg_3(10)='1' and reg_3(11)='1' else
'0' when reg_3(9)='1' and reg_3(12)='1' and reg_3(15)='1' else
'0' when reg_3(9)='1' and reg_3(13)='1' and reg_3(17)='1' else
'0' when reg_3(10)='1' and reg_3(13)='1' and reg_3(16)='1' else
'0' when reg_3(11)='1' and reg_3(14)='1' and reg_3(17)='1' else
'0' when reg_3(11)='1' and reg_3(13)='1' and reg_3(15)='1' else
'0' when reg_3(12)='1' and reg_3(13)='1' and reg_3(14)='1' else
'0' when reg_3(15)='1' and reg_3(16)='1' and reg_3(17)='1' else
'0' when reg_3(0)='1' and reg_3(1)='1' and reg_3(2)='1' else
'0' when reg_3(0)='1' and reg_3(3)='1' and reg_3(6)='1' else
'0' when reg_3(0)='1' and reg_3(4)='1' and reg_3(8)='1' else
'0' when reg_3(1)='1' and reg_3(4)='1' and reg_3(7)='1' else
'0' when reg_3(2)='1' and reg_3(5)='1' and reg_3(8)='1' else
'0' when reg_3(2)='1' and reg_3(4)='1' and reg_3(6)='1' else
'0' when reg_3(3)='1' and reg_3(4)='1' and reg_3(5)='1' else
'0' when reg_3(6)='1' and reg_3(7)='1' and reg_3(8)='1' else
'1';
```

In order to better understand this piece of code, the position corresponding to each bit used by the players is shown below.

Player 1			Player 2		
9	10	11	0	1	2
12	13	14	3	4	5
15	16	17	6	7	8

When a keystroke has been detected, it is important to ensure that it is not a false keystroke. In order to ensure it, a counter controlled by the variable count2 that counts from 0 to 10 has been implemented. After 10 clock cycles (100 ms) in which the input signal has not changed, it is possible to ensure that a keystroke was made.

The variable requirement will take the value "000000000" if the current pushed key was not pushed before, and then the keystroke will be processed.

```
requirement <= input_i and reg;
```

The variable reg is a register in which all the pressed keys are stored. This variable is used to know if the current key was pushed before, or if all the keys has been already pushed and a draw happened.

The logical function 'or' is made with input_i and reg to add the current pushed key to reg. After that, the value of input_i is stored in input_reg, the other player selected for the next turn, and the sending of the information to the LEDs is activated.

```
IF pushed = '1' THEN
  IF count2 < 10 THEN
    count2 <= count2+1;
  ELSE
    IF requirement = "000000000" THEN
      reg <= input_i or reg;
      input_reg <= input_i;
      player <= not player;
      serialize <= '1';
    END IF;
    count2 <= 0;
  END IF;
ELSE
  count2 <= 0;
END IF;
```



```
    reg_2(15)='1' else
reg_2 or "00100000000000000000" when reg_2(9)='1' and reg_2(13)='1' and
    reg_2(17)='1' else
reg_2 or "00100000000000000000" when reg_2(10)='1' and reg_2(13)='1' and
    reg_2(16)='1' else
reg_2 or "00100000000000000000" when reg_2(11)='1' and reg_2(14)='1' and
    reg_2(17)='1' else
reg_2 or "00100000000000000000" when reg_2(11)='1' and reg_2(13)='1' and
    reg_2(15)='1' else
reg_2 or "00100000000000000000" when reg_2(12)='1' and reg_2(13)='1' and
    reg_2(14)='1' else
reg_2 or "00100000000000000000" when reg_2(15)='1' and reg_2(16)='1' and
    reg_2(17)='1' else
reg_2 or "10000000000000000000" when reg_2(0)='1' and reg_2(1)='1' and
    reg_2(2)='1' else
reg_2 or "10000000000000000000" when reg_2(0)='1' and reg_2(3)='1' and
    reg_2(6)='1' else
reg_2 or "10000000000000000000" when reg_2(0)='1' and reg_2(4)='1' and
    reg_2(8)='1' else
reg_2 or "10000000000000000000" when reg_2(1)='1' and reg_2(4)='1' and
    reg_2(7)='1' else
reg_2 or "10000000000000000000" when reg_2(2)='1' and reg_2(5)='1' and
    reg_2(8)='1' else
reg_2 or "10000000000000000000" when reg_2(2)='1' and reg_2(4)='1' and
    reg_2(6)='1' else
reg_2 or "10000000000000000000" when reg_2(3)='1' and reg_2(4)='1' and
    reg_2(5)='1' else
reg_2 or "10000000000000000000" when reg_2(6)='1' and reg_2(7)='1' and
    reg_2(8)='1' else

reg_2 or "01000000000000000000" when reg="11111111" else
reg_2;
```

To send the information to the LEDs, the system waits for 3 clock cycles in order to ensure that `output_i` is updated. After that, the bits contained in `output_i` are sent to the shift registers in serial, bit to bit from the most significant bit to the least significant bit. When all the bits have been sent, the signal `show_i` is activated to send the LEDs the bits stored in the shift registers in parallel.

```
IF serialize='1' THEN
    IF count6 < 2 THEN
        count6 <= count6+1;
    ELSE
        count3 <= 21;
        serialize <='0';
        count6 <= 0;
    END IF;
```

```
ELSIF count3 > 0 THEN
    count3 <= count3-1;
    output_s <= output_i(count3-1);
    launch <= '1';
ELSIF launch = '1' THEN
    launch <= '0';
    show_i <= '1';
ELSE
    show_i <='0';
END IF;
```

At the same time, the signal `clock_o` is sent to control the shift registers.

```
clock_o <= not clock_i WHEN launch = '1' else
    '0';
```

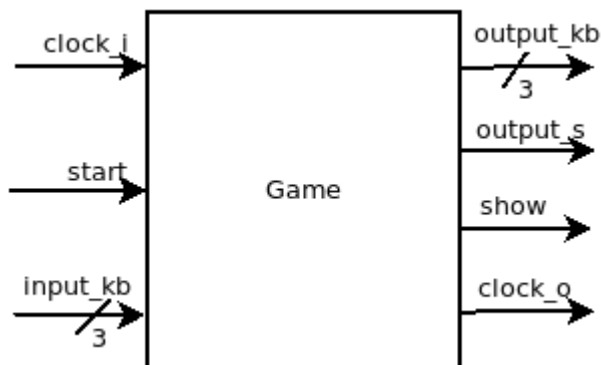


Figure 37: Modul game

When the development of the TDL code was started, it has been discovered that it is not possible to generate so many clock signals. Testing the behavior of the model for 14 seconds was necessary and, because the system clock signal has a frequency of 50 MHz, was necessary generating of 1.4 billion of signals.

The solution found to this problem was splitting the design. The design was split in a component called `clock_generator` and a component called `game`. The component `clock_generator` describes the frequency divider, and the component `game` describes the behavior of the game. In this case, only testing the component `game` was necessary to test the behavior of the model, which needs only 1,400 clock cycles.

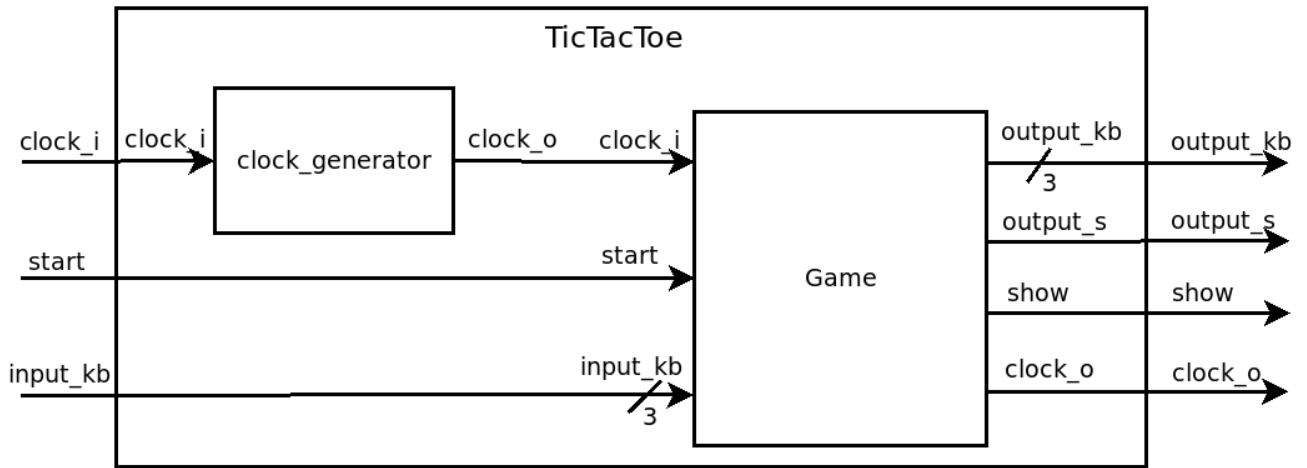


Figure 38: Modul TicTacToe

4 Verification

4.1 Simulation of the VHDL code

4.1.1 Simulation description

A functional simulation has been made in order to ensure the right working of the model. To carry out this simulation the software Modelsim, which is part of the suite provided by intel, has been used.

The simulation used the following files: TicTacToe.vhd, game.vhd, clock_generator.vhd and TicTacToe_tb.vhd.

The file TicTacToe_tb.vhd contains the testbench.

The simulation of three different game scenarios was carried out: player 1 wins, player 2 wins, nobody wins - draw. A simulation in which all the possibilities are simulated would take too much time to develop, to simulate and to verify. The simulation takes 83,025 ms

The testbench was developed trying to represent a real situation, so there are big waiting periods between input signals. This was made in order to use this test in the test automaton with the possibility of connecting the game board in parallel in order to see the outputs while the test is running.

In the following lines, some instructions used in the testbench are explained in order to make the code more comprehensible.

```
input_i <= "111"; Start_i <= '1';
wait for 1020 ms;
wait until output_kb_i = "110";
input_i <= "110"; wait for 300 ms;
input_i <= "111"; wait for 2020 ms;
```

The signals input_i and start are assigned with the value that they should have by default.

The program waits until output_kb_i has the value 110 to take the value 110 in the input_i because is desired to simulate the effect of pressing the key 1. When is pressed the key 1, input_kb (input_i in the testbench code) will changes only when the value of output_kb (output_kb_1 in the testbench code) is 110, and the value of input_kb will be 110 because of the structure of the circuit and the possible values of output_kb. After that, input_i takes again the value by default (111) to simulate the releasing of this key.

```
input_i <= "011"; wait for 175 ms; --player 1 wins
assert output_s_i='1' report "Expected 1" severity failure;
report "Player 1 wins. The model works as expected";
```

Systematic testing of digital hardware systems by means of test automaton

With the input_i = 011, player 1 has won. To verify the right working of the model, it is tested if has been activated the bit that shows that player 1 won. The activation of this bit means that the model works as expected, and is reported a message saying that player 1 wins and the model works as expected. If this bit is not activated, an error will be reported saying that value 1 was expected.

4.1.2 Simulation results

The model works as expected. There are not many conclusions to obtain here because the objective of this simulation is to ensure the right working of the model.

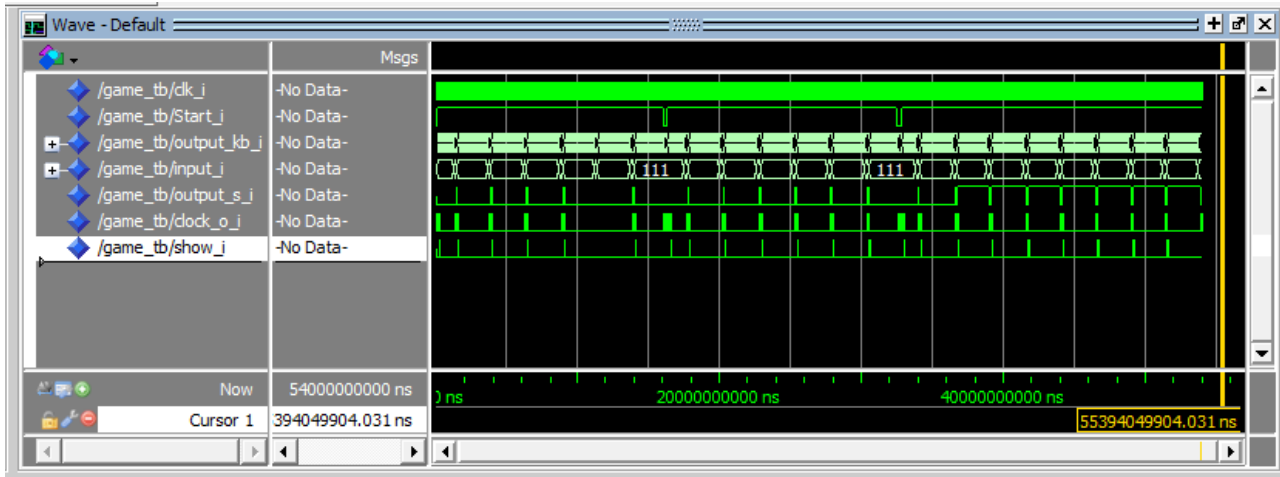


Figure 39: Complete waveform obtain in the VHDL simulation

The next figure shows all the signals generated in the simulation.

In the next figure, the last sent vector of the signal output_s when player 1 wins is represented.

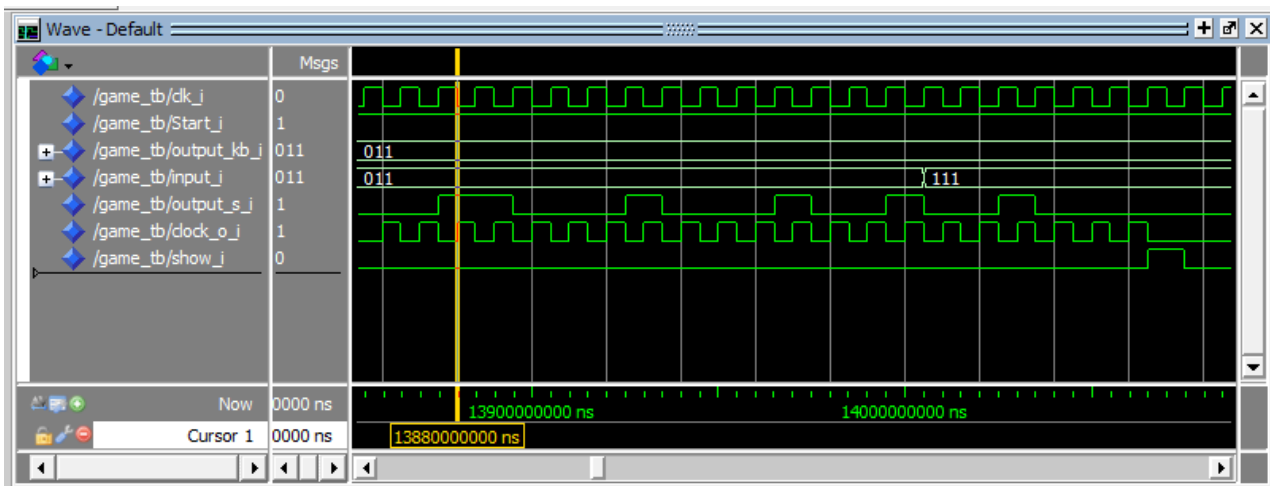


Figure 40: Detail of the waveform obtain in the VHDL simulation

Systematic testing of digital hardware systems by means of test automaton

In the previous picture it is possible to see that the information sent through *output_s_1* is:

P2	DR	P1	Player 1									Player 2								
0	0	1	1	0	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	0

This information correspond with the game table shown below where the red color is player 1 and the green color is player 2. The LED which shows that player 1 has won has been turned on.

Red	White	White
Green	Red	White
Green	White	Red

Figure 41: Sketch of the game table after the first player wins

These are the messages launched by Modelsim.

```
#  
# ** Note: Player 1 wins. The model works as expected  
# Time: 13880 ms Iteration: 0 Instance: /game_tb  
# ** Note: Player 2 wins. The model works as expected  
# Time: 30370 ms Iteration: 0 Instance: /game_tb  
# ** Note: Draw. The model works as expected  
# Time: 53890 ms Iteration: 0 Instance: /game_tb
```

Figure 42: Result of the VHDL simulation (command console)

It is verified that the model works as expected.

4.2 Testbench using Test Description Language

4.2.1 Testbench description

Because the right working of the model has already been verified through the simulation and the board developed, the next step is verifying the code generated by the TDL generator. To carry out that, it is enough to simulate one game, so the same simulation used in the functional simulation when the player 1 wins has been implemented here. As it was said before, it was not possible to simulate all the model because of the amount of necessary clock signals. Therefore it was only possible to simulate the component game.

Defining the components used in the testbench and the types of inputs and outputs that they have is the first step. This testbench has the following two components:

- **Type hardware.** This component corresponds to the model to test.
- **Type TB.** This component corresponds to the test automaton.

The types of inputs used are *std logic* and *std logic vector*.

```
Data Set logic {
    instance std_logic;
    instance std_logic_vector;
}

Gate Type input accepts logic;
Gate Type output accepts logic;

Component Type hardware {
    gate types: input,output;
}

Component Type TB {
    gate types: input,output;
}
```

Now, the time unit is defined:

```
Time Unit milliseconds;
```

The next step is defining the configuration of the test:

- Instantiation of the components

```
instantiate game as DUT of type hardware having {
  gate Start of type input with length of 1;
  gate clock_i of type input with length of 1;
  gate input_kb of type input with length of 3;
  gate output_kb of type output with length of 3;
  gate output_s of type output with length of 1;
  gate show of type output with length of 1;
  gate clock_o of type output with length of 1;
}

instantiate TB_a as Tester of type TB having {
  gate Starter of type output with length of 1;
  gate CLKgenerator of type output with length of 1;
  gate row_o of type output with length of 3;
  gate column_i of type input with length of 3;
  gate parallelize of type input with length of 1;
  gate clk of type input with length of 1;
  gate input_s of type input with length of 1;
}
```

- Description of the connection between the components:

```
connect gate Starter to gate Start;
connect gate CLKgenerator to gate clock_i;
connect gate row_o to gate input_kb;

connect gate column_i to gate output_kb;
connect gate parallelize to gate show;
connect gate clk to gate clock_o;
connect gate input_s to gate output_s;
```

- The maximum assert deviation:

```
Assert deviation is (1 milliseconds);
```

Systematic testing of digital hardware systems by means of test automaton

The schematic of the connections between the DUT (game) and the test automaton (TB_a) is shown below.

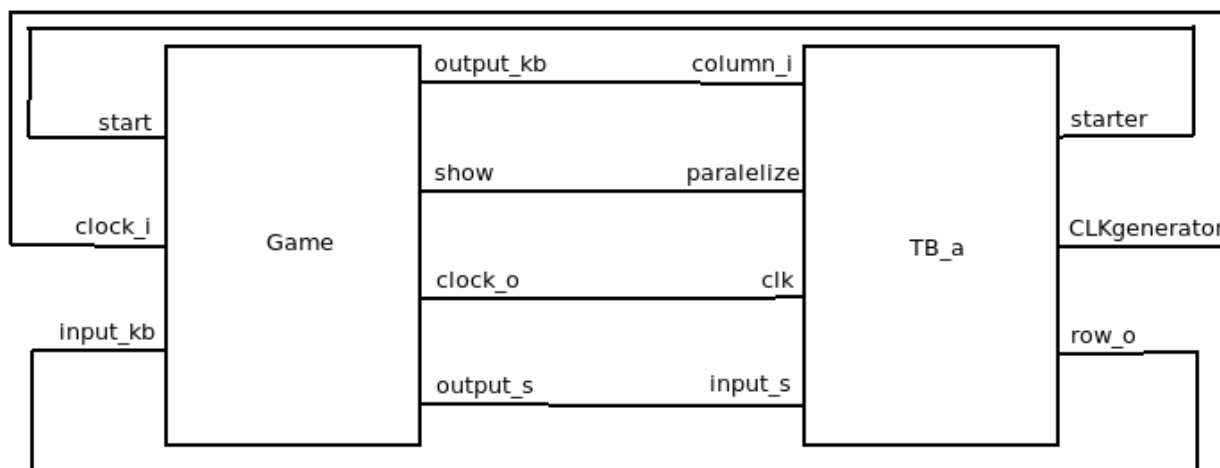


Figure 43: Connections between the model and the test automaton

After that, the physical characteristics of the components are defined:

```
SignalAdapter Configuration de0_nano_output {
    signaladapter output_adapter1 of type output having{
        attach Start 0 downto 0 to position 0 downto 0;
        attach clock_i 0 downto 0 to position 1 downto 1;
        attach input_kb 2 downto 0 to position 4 downto 2;
        logiclevel LVTTTL;
        type de0_nano_pappkisteOut;
        hardware_revision "0.1";
        software_revision "0.1";
        serial number "001";
        connection JTAG;
        address "USB-Blaster [1-6.1]";
    }

    signaladapter input_adapter1 of type input having{
        attach output_kb 2 downto 0 to position 2 downto 0;
        attach show 0 downto 0 to position 3 downto 3;
        attach clock_o 0 downto 0 to position 4 downto 4;
        attach output_s 0 downto 0 to position 5 downto 5 ;
        logiclevel LVTTTL;
        type de0_nano_pappkisteIn;
        hardware_revision "0.1";
        software_revision "0.1";
        serial number "001";
    }
}
```

```
        connection JTAG;
        address "USB-Blaster [1-6.3]";
    }
}
```

The last step is the description of the test. All the signals are outputs in this language. The outputs of the model are described as outputs of game, and the outputs of the test automaton are described as outputs of TB_a.

```
Test Description test_the_game{
    use Test configuration: game_tdl_cf{
        run {
            repeat 2000 times {
                TB_a sends bit value of b0 to gate clock_i;
                gate clock_i waits for (5 milliseconds);
                TB_a sends bit value of b1 to gate clock_i;
                gate clock_i waits for (5 milliseconds);
            }
        }
        in parallel to {
            TB_a sends bus value of 7 to gate input_kb;
            gate input_kb waits for (1205 milliseconds);
            TB_a sends bus value of 6 to gate input_kb;
            gate input_kb waits for (300 milliseconds);
            ...
            ...
            ...
        }
        terminate;
    }
}
```

The complete code is in appendix D. Here, only some commands will be explained in order to make the code understandable. The test has been made trying to emulate a real game, so there are waiting periods between the keystrokes.

The clock is the first signal generated. The period is 10 ms, so there are 5 ms in which the signal has a value of 0 and 5 ms in which the value is 1. In order to simulate 20 seconds, it is necessary to generate 2000 clock cycles.

```
run {
    repeat 2000 times {
        TB_a sends bit value of b0 to gate clock_i;
        gate clock_i waits for (5 milliseconds);
        TB_a sends bit value of b1 to gate clock_i;
        gate clock_i waits for (5 milliseconds);
    }
}
```

Systematic testing of digital hardware systems by means of test automaton

The signals `input_kb` and `Start` are initialized with the value by default (the value that should have if the push buttons are not pushed).

```
TB_a sends bus value of 7 to gate input_kb;  
TB_a sends bit value of b1 to gate Start;
```

In order to emulate the keystrokes of the keyboard, it is necessary to wait until the column of this key is active and then active the input of this row. It is important to remember that both inputs and outputs are low level active. The key is pushed for 300 ms and released again. A waiting of 2020 ms separates the keystrokes. In the piece of code shown below, the keys 1 and 4 are pushed.

```
TB_a sends bus value of 7 to gate input_kb;  
gate input_kb waits for (1205 milliseconds);  
TB_a sends bus value of 6 to gate input_kb;  
gate input_kb waits for (300 milliseconds);  
TB_a sends bus value of 7 to gate input_kb;  
gate input_kb waits for (2100 milliseconds);  
TB_a sends bus value of 5 to gate input_kb;  
gate input_kb waits for (300 milliseconds);  
TB_a sends bus value of 7 to gate input_kb;
```

The sequence of the key pushed in this test is: 1, 4, 5, 7, 5 and 9. It is important to notice that the key 5 is pushed twice, but the second time is not considered by the system because was previously pushed. The player 1 has pushed the keys 1, 5 and 9, and the player 2 has pushed the keys 4 and 7. This is the result in the matrix of the game where the red color is player 1 and the green color is player 2:

Red	White	White
Green	Red	White
Green	White	Red

Figure 44: Sketch of the game table after the TDL testbench

Systematic testing of digital hardware systems by means of test automaton

Each time that a key is pushed, the information to the LEDs is sent through the serial output. As was explained in the previous chapter, the order of the bits is player 2 wins, draw, player 1 wins, positions used by player 2 and positions used by player 1. The code to send the information to the LEDs when player 1 has won is shown below.

```
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b1 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b1 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b1 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b1 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b1 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
gate input_s waits for (10 milliseconds);
game sends bit value of b0 to gate input_s;
```

Systematic testing of digital hardware systems by means of test automaton

This is the information that has been sent:

P2	DR	P1	Player 1									Player 2								
0	0	1	1	0	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	0

The clock to control the shift register has a delay of 5 ms from the bit sent. With this delay, it is ensured that the bit has arrived at the input of the register and the level is stable. One clock cycle is sent with each information bit.

```

game sends bit value of b1 to gate clk;
gate clk waits for (5 milliseconds);
game sends bit value of b0 to gate clk;
gate clk waits for (5 milliseconds);

```

After that, one bit to release the information to the LEDs is sent.

```

game sends bit value of b1 to gate parallelize;
gate parallelize waits for (10 milliseconds);
game sends bit value of b0 to gate parallelize;

```

The TDL generator has generated a file called testbench.vhd. This file contains the same test description in VHDL and can be used to verify the right working of the model through a simulation using Modelsim like it has been done in the previous testbench..

When it was verified that the testbench was correctly written and the files generated are ready to test the DUT (it was necessary correcting some mistakes in this file), some changes were made in the model in order to simulate that the cable connected to the input_kb[1] is broken. The objective is to verify that the testbench generated by the TDL generator is able to detect that the DUT does not work right.

These are the changes made in the model:

```

input_i <= "000000001" WHEN input_kb ="110" AND output_kb_i = "110" else
    "000000010" WHEN input_kb ="110" AND output_kb_i = "101" else
    "000000100" WHEN input_kb ="110" AND output_kb_i = "011" else
--    "000001000" WHEN input_kb ="101" AND output_kb_i = "110" else
--    "000010000" WHEN input_kb ="101" AND output_kb_i = "101" else
--    "000100000" WHEN input_kb ="101" AND output_kb_i = "011" else
    "001000000" WHEN input_kb ="011" AND output_kb_i = "110" else
    "010000000" WHEN input_kb ="011" AND output_kb_i = "101" else
    "100000000" WHEN input_kb ="011" AND output_kb_i = "011" else
    "000000000";

```

Systematic testing of digital hardware systems by means of test automaton

The values in input_kb[1] are not considered with these changes. It is the same effect that when the cable connected to this input is broken.

4.2.2 Testbench results

4.2.2.1 Model working right

Some mistakes were found in the file testbench.vhd.

When the simulation was executed, Modelsim reported an error. The next figure shows the error reported by Modelsim:

```
##
##
## ** Warning: Assert failed at 104 ms
##   Time: 2104 ms  Iteration: 0  Instance: /game_tdl_tb
;VSIM 22> ]
```

Figure 45: Error reported by Modelsim. Command console. (TDL testbench)

The first expected value of the signal column_i was wrong because the code expected “011” and the right value was “110”. The value generated in the simulation is right, but the value expected is wrong. In the next figure is shown the generated waveform for that instant of time:

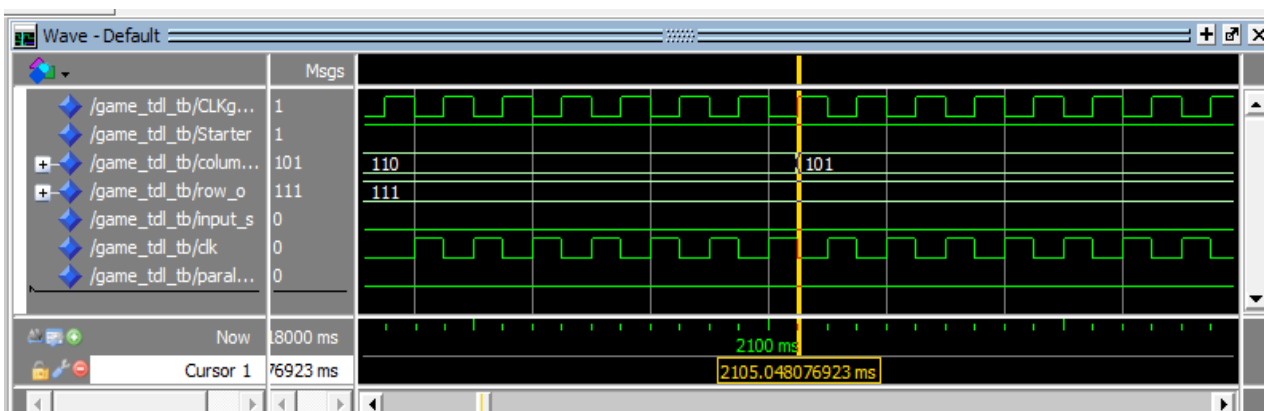


Figure 46: Waveform result of the simulation, where an error was reported. (TDL testbench)

Systematic testing of digital hardware systems by means of test automaton

It is also possible to see that the message sent by some reports does not meet with the current instant of time. For the instant of time 104 ms, the message says “assert failed at 106 ms” for the signals `clk` and `input_s`.

```
--#104
    assert clk = '1'
    report "Assert failed at 106 ms" severity warning;
    assert input_s = '0'
    report "Assert failed at 106 ms" severity warning;
    assert column_i = "011"
    report "Assert failed at 104 ms" severity warning;
    wait for 2 ms;
```

The next figure shows the complete waveform of the simulation.

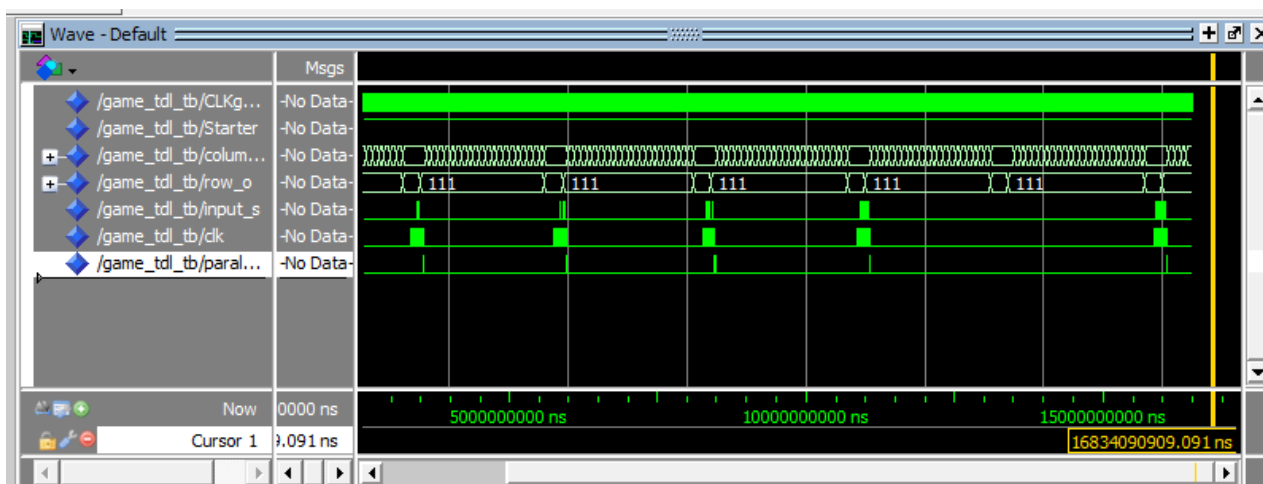


Figure 47: Complete waveform result of the simulation. (TDL testbench)

This following figure shows the last sent `output_s` vector.

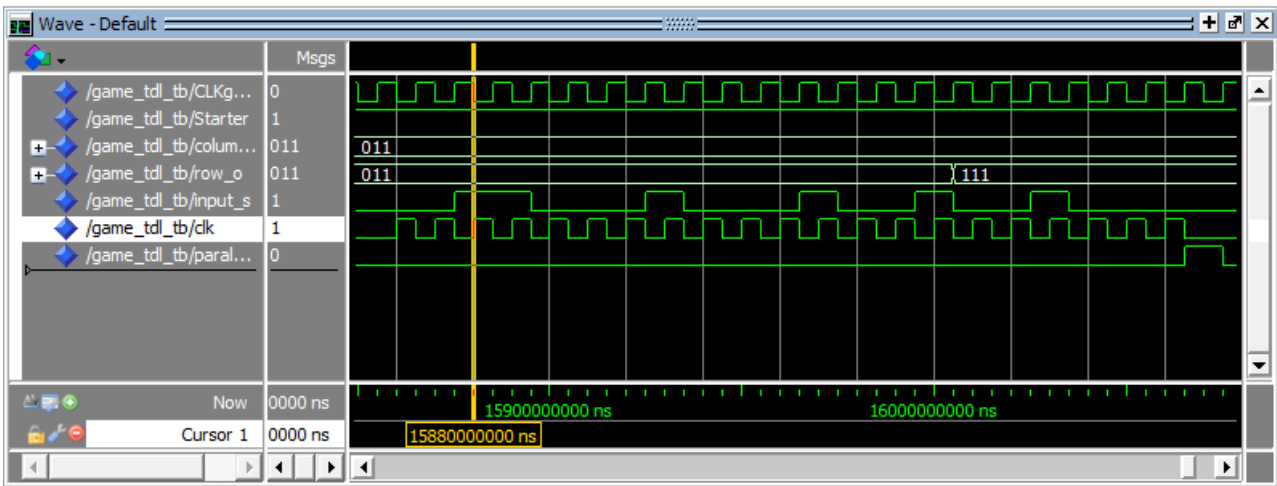


Figure 48: Information sent by the serial output when the player 1 wins (TDL testbench)

It is verified that all the generated signals are right. The signals are the same than the signals obtained in the previous simulation, and the only errors are the errors commented previously.

4.2.2.2 Model modified to introduce a malfunction

Many errors appear because the timing of the output column_i depends on the input row_o. Due to some values in row_o are ignored, the values of the output column_i are different than expected in each instant of time. Because of the key detected depends on column_i and row_o, a different value in column_i originates a different key detected for a same row_o. Besides, because of some inputs are ignored, some outputs in input_s, clk and parallelize are not originated.

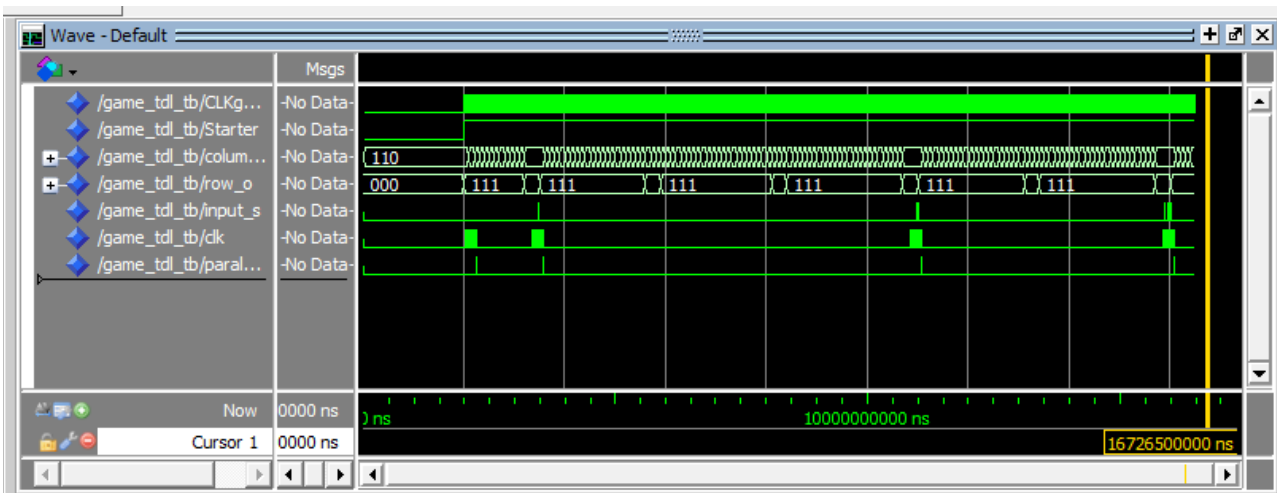


Figure 49: Waveform obtain in the simulation with malfunctions in the model (I) (TDL testbench)

Systematic testing of digital hardware systems by means of test automaton

In the figure shown below, it is possible to see that the signals generated are very different from the signals generated in previous simulations.

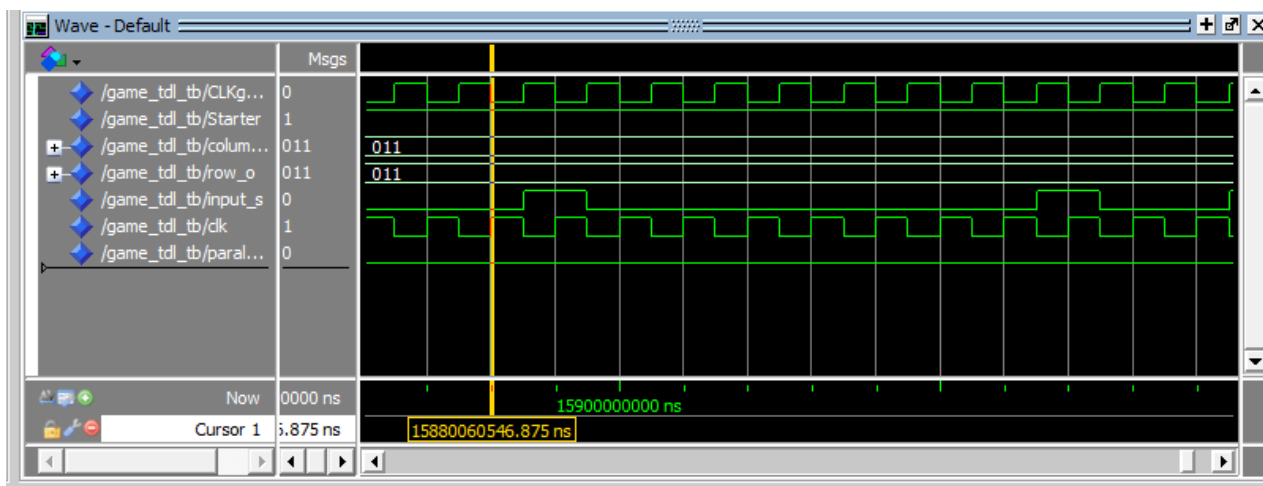


Figure 50: Waveform obtain the simulation with malfunctions in the model (II) (TDL testbench)

Some of the errors reported by Modelsim are shown here:

```
# ** Warning: Assert failed at 3761 ms
#   Time: 5761 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3766 ms
#   Time: 5764 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3771 ms
#   Time: 5771 ms  Iteration: 0  Instance: /game_tdl_tb
```

All the reported errors are shown in Appendix E.

With this test is verified that the testbench generated by the TDL compiler is able to detect malfunctions in the model, so after correction of the errors found in the simulation of the right model, the file can be used to test the models' reliably.

4.3 Verification in hardware

4.3.1 Testbench description

Due to the test automaton was not ready yet, the only possibility to simulate the model in a more realistic way and to verify that the delays created inside the FPGA do not exceed the maximum delay allowed by the testbench was conducting a timing simulation. This simulation is called timing simulation and requires the file testbench.vhd which contains the testbench profile, the file game_6_1200mv_85c_slow.vho which contains the netlist of the circuit to be implemented inside the FPGA and the file game_6_1200mv_85c_vhd_slow.sdo which contains the delays in the circuit to be implemented inside the FPGA.

To carry out this simulation it is necessary to do some configurations in the simulation profile. To carry it out, it is necessary to go to **Simulate** in the menu and choose **Start Simulation**. This windows will be opened:

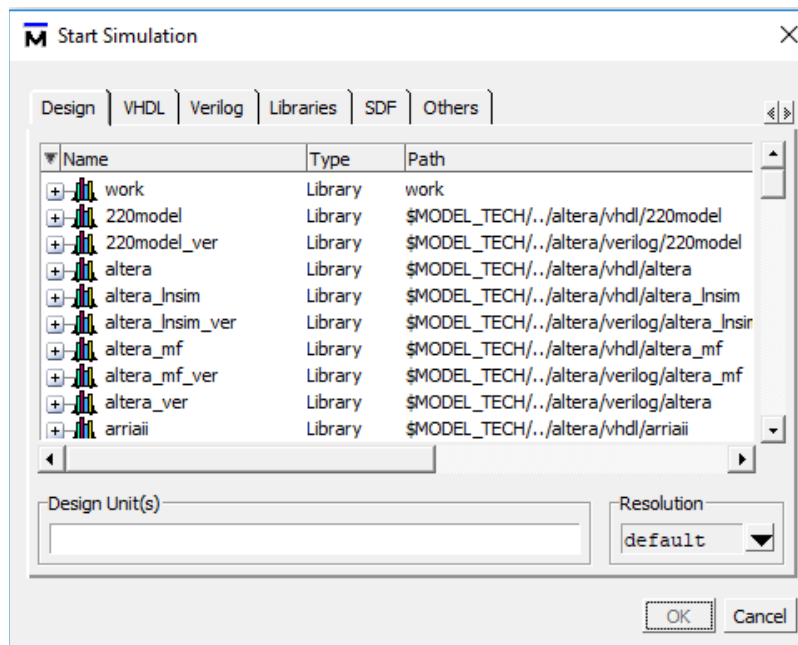


Figure 51: Configuration of the timing simulation (I)

In the tab SDF the file game_6_1200mv_85c_vhd_slow.sdo is added and the region applied. The region corresponds to the component instantiate in the file testbench.vhd. In this case, the component instantiate is called UUT.

```

UUT: game
port map(
clock_o => clk,
show => parallelize,

```

Systematic testing of digital hardware systems by means of test automaton

```
output_kb => column_i,  
input_kb => row_o,  
clock_i => CLKgenerator,  
Start => Starter,  
output_s => input_s);
```

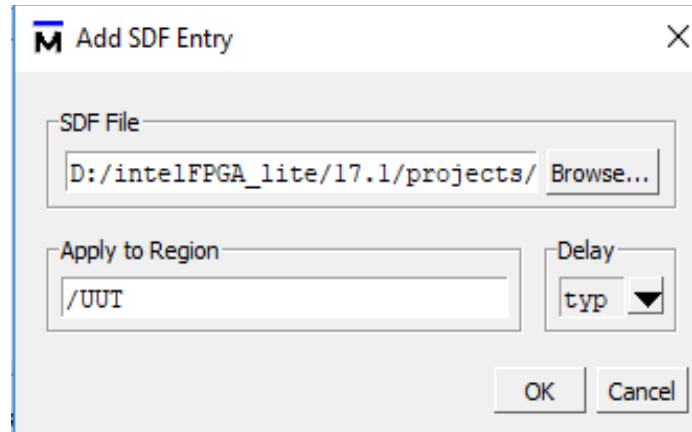


Figure 52: Configuration of the timing simulation (II)

Now, in the tap design, the testbench profile is chosen.

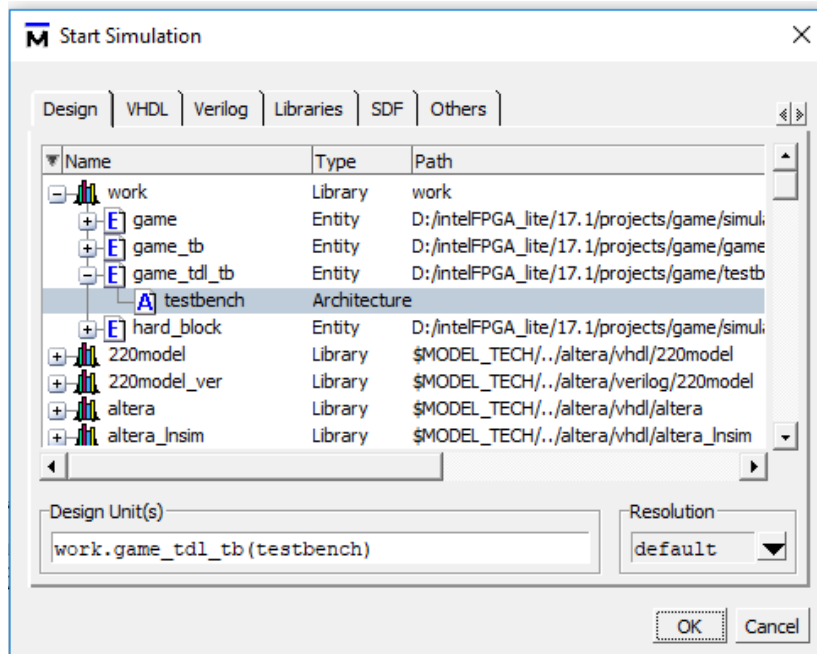


Figure 53: Configuration of the timing simulation (III)

Systematic testing of digital hardware systems by means of test automaton

The simulation configuration is already finished and it is ready to make the simulation in the normal way.

The last step is the verification of the signals defined in the file asserts.vhd and the right working of the file vcdplayer.vhd. This verification is carried out doing a simulation with the testbench generated by the TDL generator where the entity to simulate is the entity described in the file vcdplayer.vhd and the outputs are the signals contained the file asserts.vcd.

In order to ensure that the testbench is able to detect wrong values in the measurements, a wrong value was introduced in the file asserts.vcd. The value of the signal clk was changed from 1 to 0 for 200 ms.

Original value	Changed value
#200	#200
b1 %	b0 %

4.3.2 Testbench results

The delay between CLKgenerator (clock signal) and column_i (output_kb in the model) is shown here:

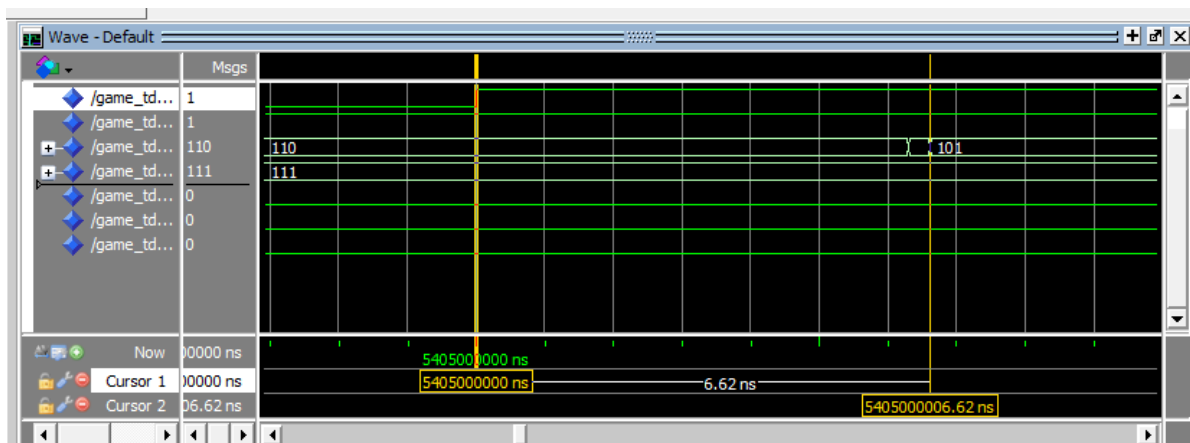


Figure 54: Delay between CLKgenerator and column_i

This delay has a value of 6.62 ns.

Systematic testing of digital hardware systems by means of test automaton

The delay between CLKgenerator (clock signal) and clk (clock_o in the model) is shown here:

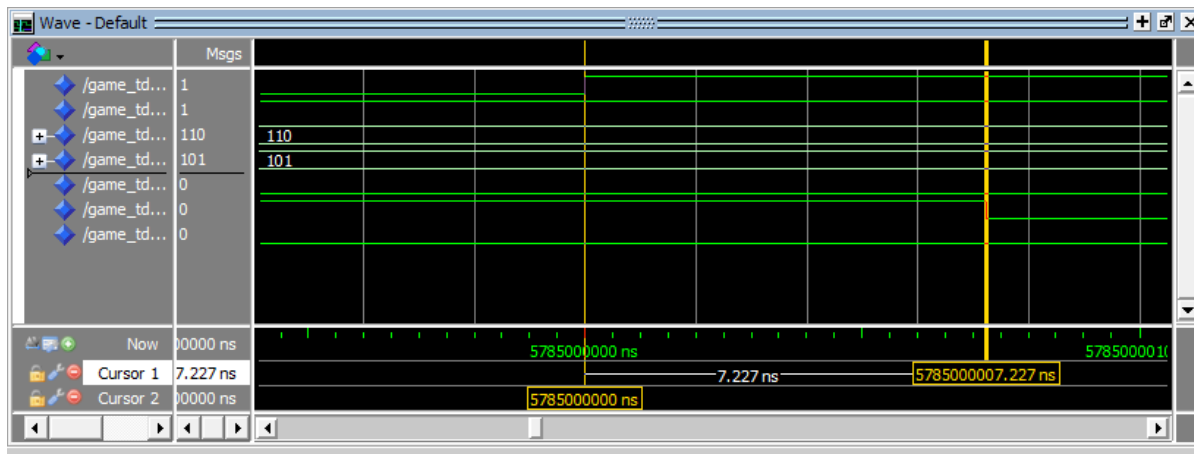


Figure 55: Delay between CLKgenerator and clk

This delay has a value of 7.227 ns.

The delay between CLKgenerator (clock signal) and input_s (output_s in the model) is shown here:

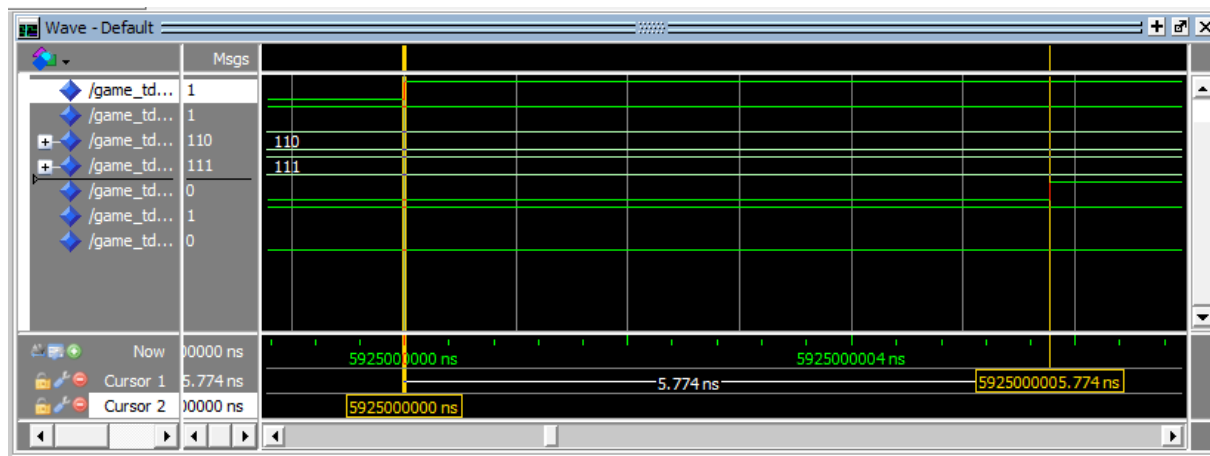


Figure 56: Delay between CLKgenerator and output_s

This delay has a value of 5.774 ns.

The delay between CLKgenerator (clock signal) and parallelize (show in the model) is shown here:

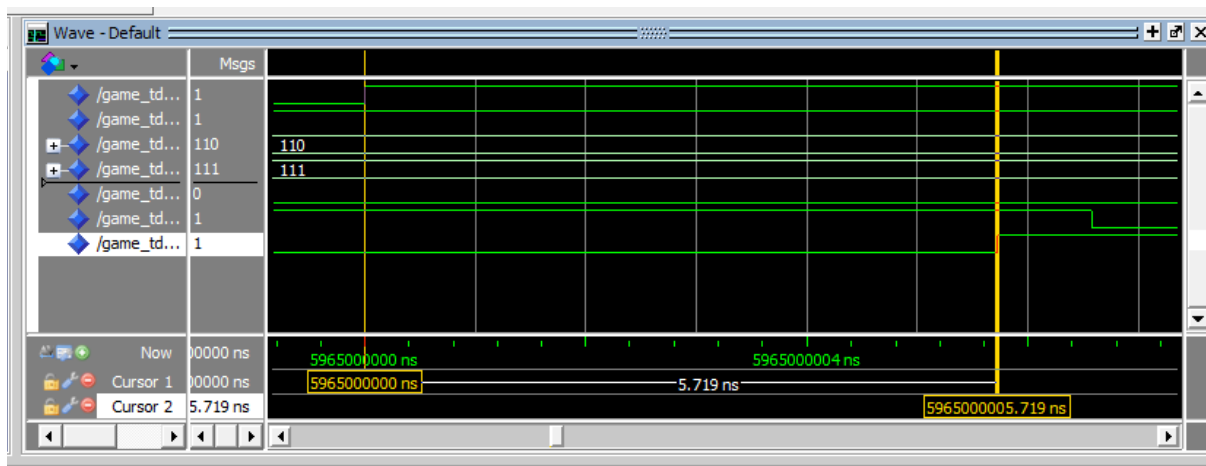


Figure 57: Delay between CLKgenerator and parallelize

This delay has a value of 5.719 ns.

The maximum delay in the circuit detected in the simulation is 7.227 ns. This value is very small compared to the maximum delay allowed by the testbench, which is 1 ms. No errors were reported.

Malfunctions in the behavior of the designed model are not expected.

4.3.3 Verification of the assertions

4.3.3.1 *Original asserts.vcd*

A mistake has been found in the file `asserts.vcd`. There is no definition of the signals described in the file. So when the file is used directly in Modelsim, an error is launched. The file `vcdplayer.vhd` does not need this definitions because this file contains the necessary information to be able to recognize the signals and gives them names when they are represented in the wave window.

This mistake is created by the TDL generator, and will be solved in the future.

Code generated:

```
$scope module TDLGenerator $end
$upscope $end
$enddefinitions $end
```

The right generated code to be opened directly in Modelsim should be:

```
$scope module TDLGenerator $end
```

Systematic testing of digital hardware systems by means of test automaton

```
$var wire 3 ! column_i $end
$var wire 1 % clk $end
$var wire 1 - input_s $end
$var wire 1 + parallelize $end
$upscope $end
$enddefinitions $end
```

The simulation was right and no errors were reported.

The following figure shows the output signals generated for 200 ms.

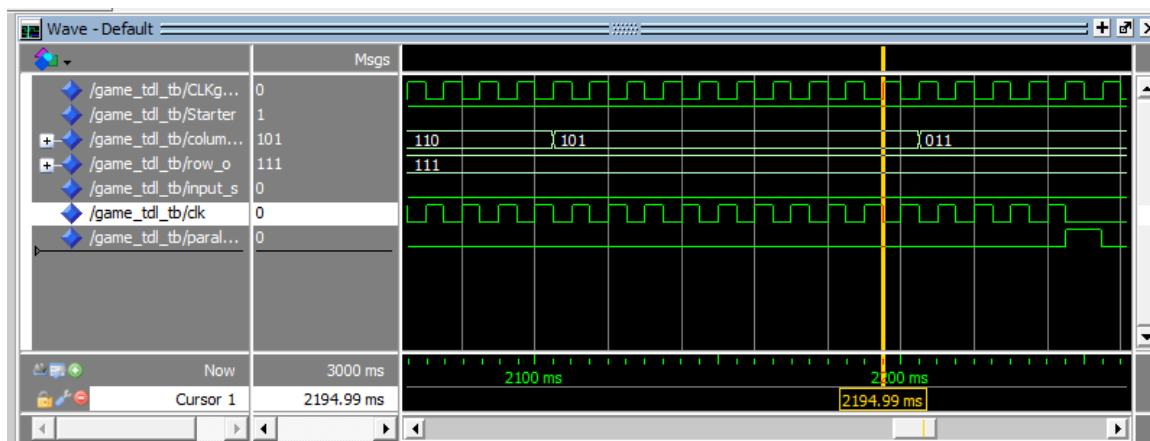


Figure 58: Result of the verification of vcdplayer

4.3.3.2 Altered asserts.vcd

Some errors were reported:

```
#
# ** Warning: Assert failed at 201 ms
#   Time: 2201 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 206 ms
#   Time: 2204 ms  Iteration: 0  Instance: /game_tdl_tb
VSIM 48>
```

Figure 59: Result of the verification of vcdplayer.vhd when an error in asserts.vcd was introduced

Systematic testing of digital hardware systems by means of test automaton

In the following figure is shown the output signal generated when the wrong value was introduced.

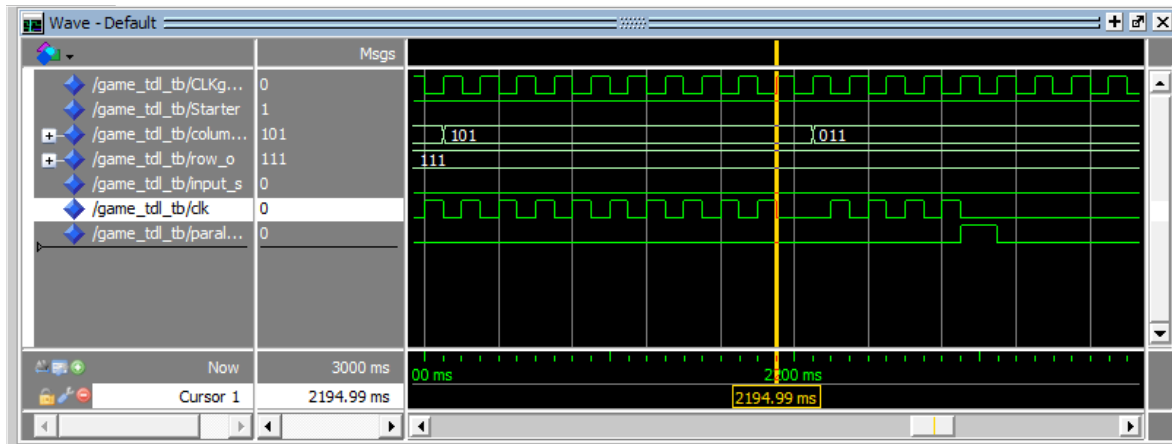


Figure 60: Result of the verification of vcdplayer for 200 ms when an error in asserts.vcd was introduced.

It is verified that the file testbench.vhd is able to detect wrong values in the measurements file.

5 Conclusions

As initially planned, the TicTacToe game was developed as a VHDL model. In a next step a testbench to check the models specified behavior has been developed.

In this testbench three test descriptions have been described: player 1 wins, player 2 wins and there is a draw. When it was verified that the model works right, the test description in which player 1 wins was described using TDL.

A simulation using the testbench generated by TDL was carried out and it was seen that an assert generated an error that did not exist. The testbench file was examined and it was found that one signal described in an assert was wrong. After manually correcting this error and verifying that the testbench does not report any error, a new test case was made in which a malfunction was introduced to the model to check whether the testbench was capable of detecting this malfunction. It has been verified that the testbench was well generated and detected errors.

The next step was checking the correct operation of the test automaton and the other files generated by the TDL, but the test automaton was not ready yet. This made it necessary to think of another way to continue checking the files. A timing simulation (post fitting simulation) was then decided to do in order to test if the testbench was able to detect if any timing deviation bigger than expected occurs. It was found that the timing deviations were much smaller than the maximum allowed deviation and that the testbench did not report errors.

vcdplayer.vhd, which is used to verify the file assert.vcd, was tested for proper functioning and it was found to be working correctly. It was discovered that it was not possible to open the file asserts.vcd directly in Modelsim. Examining the assert.vcd file, it was seen that the signals used were not declared in the header, and after declaring them Modelsim was able to open the file and represent the waveforms.

The last step was the implementation of the model in the prototyping board and the testing of the game board. The behavior of the system was right and it was possible to play TicTacToe.

The next flow diagram shows the work process followed along this thesis.

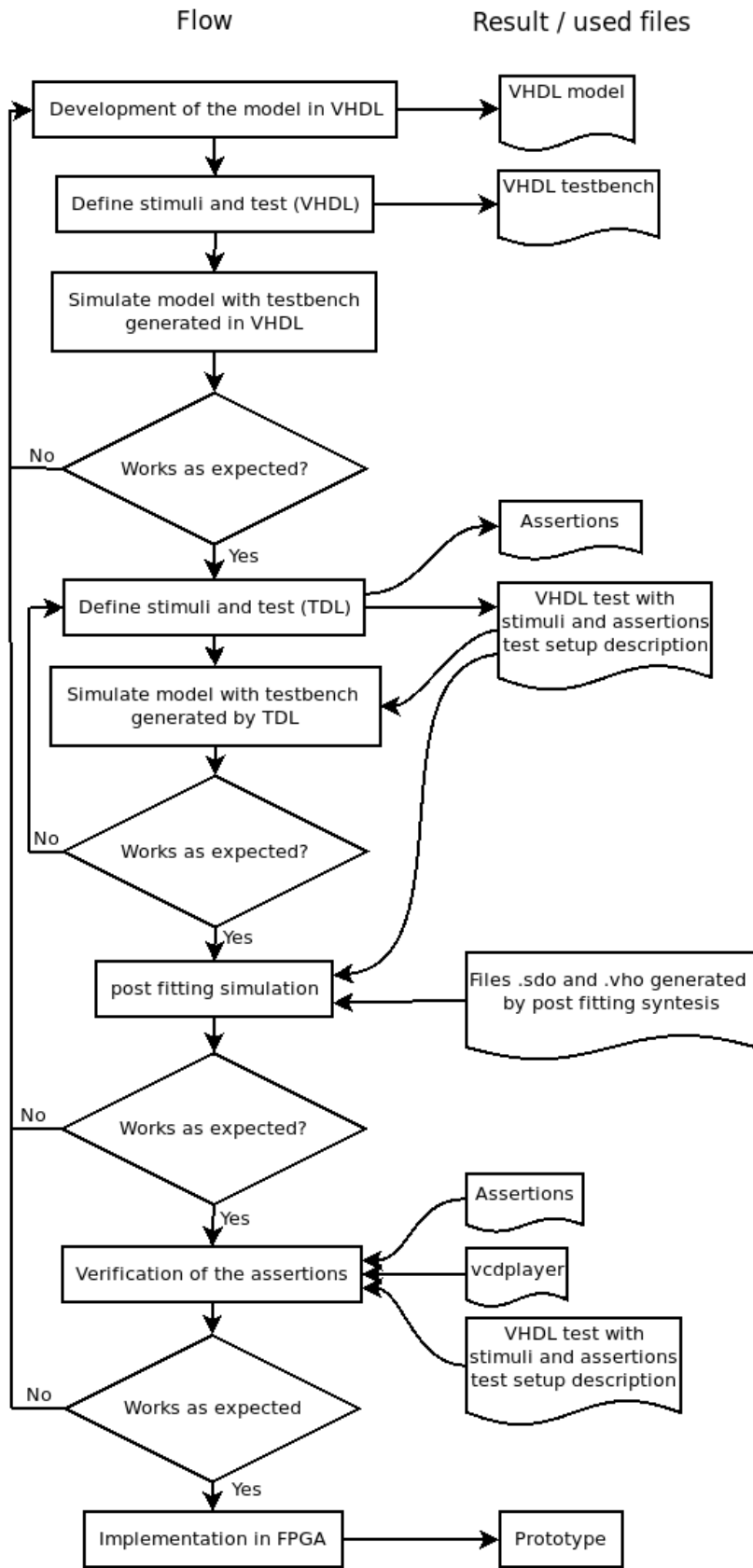


Figure 61: Workflow carried out in this work

Systematic testing of digital hardware systems by means of test automaton

This test description language is a powerful tool because only by describing the inputs, outputs and how the test automaton is connected to the device under test, a VHDL testbench file, a stimulus file for the test automaton, and an asserts file to verify the result obtain in the test performed with the test automaton are automatically generated as the connection, stimuli and assertions are described. It is easy to make changes to the test descriptions because it only requires modifying one file to modify all the test files generated.

The syntax is very simple, intuitive and quick to learn. Commands such as "*<device> sends bit value of <value> to gate <input>*" or "*gate <input> waits for (<delay>)*" clearly show what they do even for people who do not know this TDL.

Waits between value changes are described by delays, which makes them easy to describe and follow. In addition, all the value changes of a signal are grouped in a block, so the signal are separated from each other making it easy to do modifications.

This TDL is a reliable way to test digital systems because there is no human intervention in the measures that may put the results into question. The electrical stimulus signals are automatically generated by the test automaton and the measurements are also performed by it and saved in a file.

This TDL is being developed for testing reverse engineered systems based on FPGA, but it is equally usable for testing other digital systems even if they are not based on FPGAs.

Now, the most important work is to fix the errors found in the files generated by the TDL. When the TDL generator works correctly, the next step is to finish the development of the test automaton.

For future work, it would be advisable to create a faster model to find the speed limitations of the test automaton, and after that, it is important to certify the test automaton.

In order to automate the entire testing process, a software tool might be created. This software would only require the stimuli file, the testbench file and the vcdplayer file created by the TDL generator. This software would send the stimuli file to the test automaton, receive the measurements obtained by the test automaton and carry out the verification of the measurements. Then the software would report to the user if any error was found or if the DUT works as expected/specified.

6 Appendices

6.1 Appendix A: TicTacToe game board design

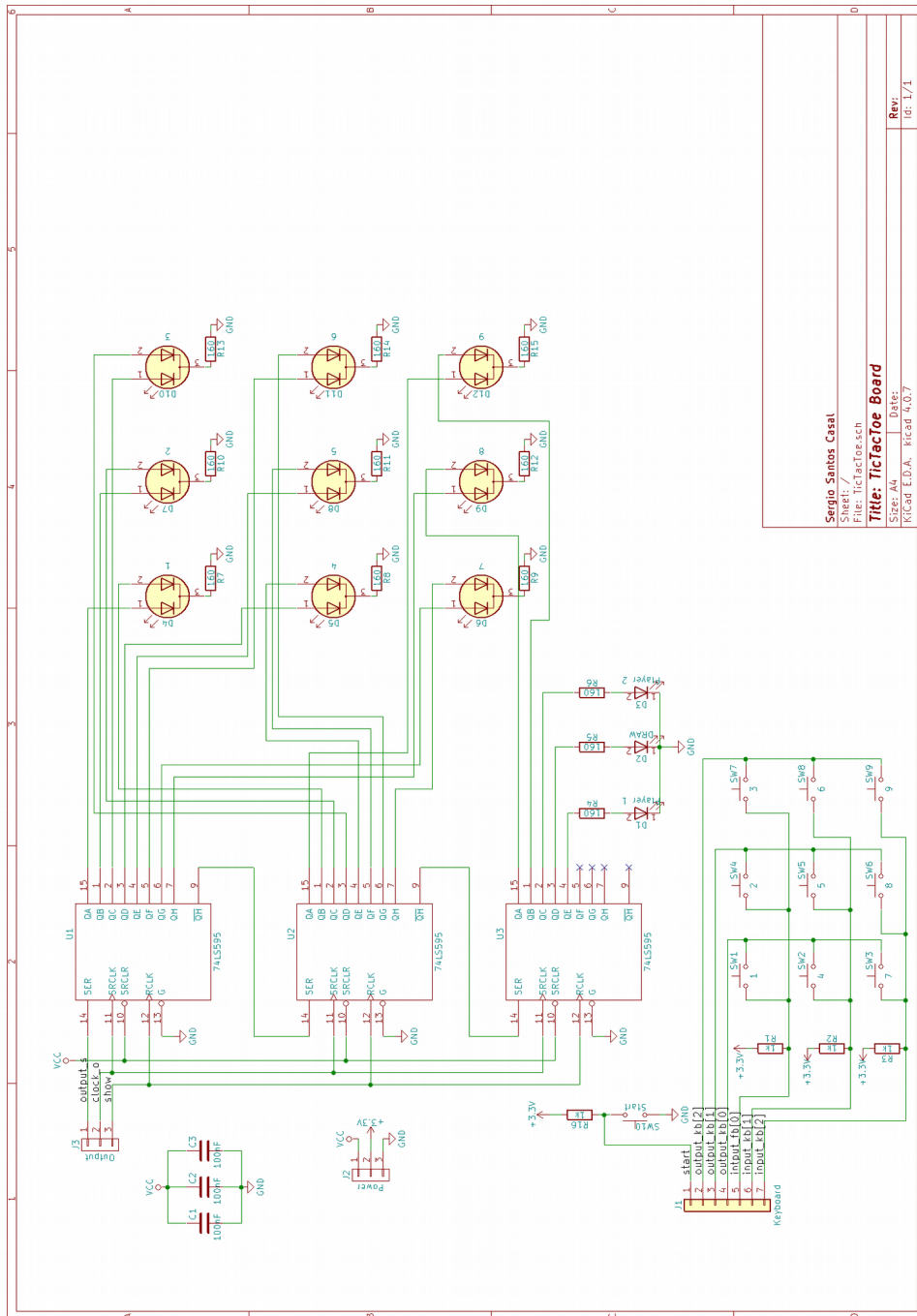


Figure 62: TicTacToe board schematic

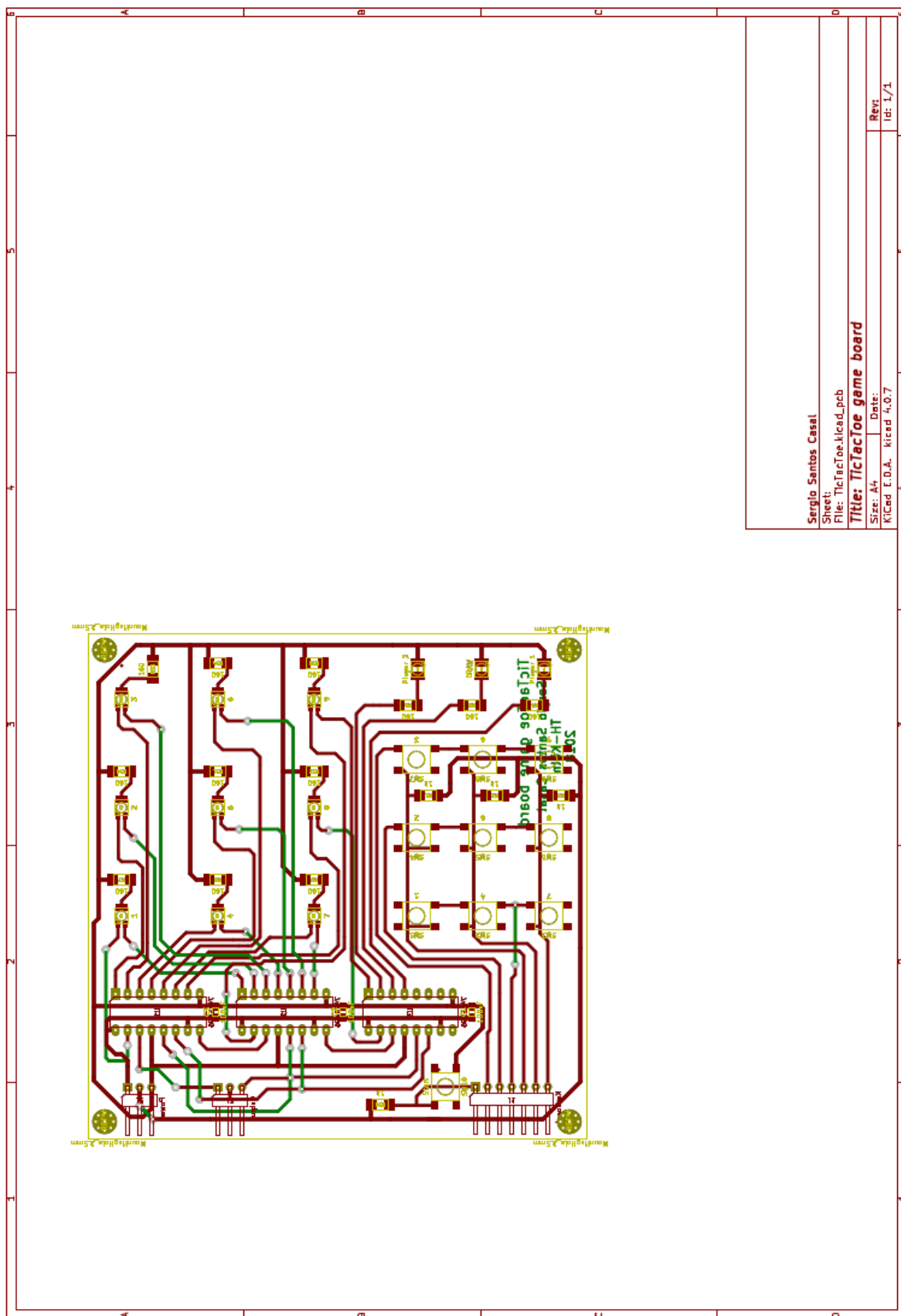


Figure 63: TicTacToe game board PCB

6.2 Appendix B: VHDL model code

6.2.1 VHDL code sketch of the model for one player

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY TicTacToe IS
  PORT (Start, clk : IN std_logic;
        input : IN std_logic_vector (7 downto 0);
        PC, human, draw : OUT std_logic;
        pstate_o : OUT std_logic_vector(4 downto 0));

END TicTacToe;

ARCHITECTURE synth OF TicTacToe IS
  SIGNAL pstate, n_state : std_logic_vector(4 downto 0) := "00001";
  CONSTANT max_count : INTEGER := 125; --must be 500000 for 100 hz
  SIGNAL count : INTEGER range 0 to max_count;
  SIGNAL count2 : integer range 0 to 10;
  SIGNAL clock : std_logic := '0';
  SIGNAL input_old, input_reg : std_logic_vector (7 downto 0) :=
"00000000";

BEGIN

  gen_clock: PROCESS (clk, count)
  BEGIN
    IF clk'event and clk='1' THEN
      IF count < max_count THEN
        count <= count+1;
      ELSE
        clock <= not clock;
        count <= 0;
      END IF;
    END IF;
  END PROCESS;

  read_input: PROCESS (clock, input)
  BEGIN
    IF clock'event and clock='1' THEN
      IF input /= input_reg THEN
        IF count2 < 10 THEN
          count2 <= count2+1;
        ELSE
          input_reg <= input;
        END IF;
      END IF;
    END IF;
  END PROCESS;


```

Systematic testing of digital hardware systems by means of test automaton

```
                count2 <= 0;
            END IF;
        END IF;
    END IF;
END PROCESS;

state_machine: PROCESS (clock, start, input, input_old)

BEGIN

    IF clock'event and clock='1' THEN
        IF start = '1' THEN
            n_state <= "00001";
            input_old <="00000000";
        ELSIF input_reg /= input_old THEN
            input_old <= input_reg;

            IF pstate = "00001" THEN                                -- STATE 1
                --PLACE A CROSS IN THE CENTRE
                IF input_reg="00000001" THEN
                    --PLACE 0 IN the SPACE 0
                    n_state <= "00010";
                ELSIF input_reg="10000000" THEN
                    --PLACE 0 IN THE SPACE 7
                    n_state <= "00010";
                ELSIF input_reg="00000100" THEN
                    --PLACE 0 IN the SPACE 2
                    n_state <="01001";
                ELSIF input_reg="00100000" THEN
                    --PLACE 0 IN THE SPACE 5
                    n_state <="01001";
                ELSIF input_reg="00000010" THEN
                    --PLACE 0 IN the SPACE 1
                    n_state <="01110";
                ELSIF input_reg="01000000" THEN
                    --PLACE 0 IN THE SPACE 6
                    n_state <="01110";
                ELSIF input_reg="00001000" THEN
                    --PLACE 0 IN the SPACE 3
                    n_state <="10001";
                ELSIF input_reg="00010000" THEN
                    --PLACE 0 IN THE SPACE 4
                    n_state <="10001";
                END IF;

            ELSIF pstate = "00010" THEN                                --STATE 2
                --PLACE A CROSS IN THE SPACE 3
                IF input_reg="00000001" THEN
```

```
        --PLACE 0 IN THE SPACE 0
        n_state <= "00011";
    ELSIF input_reg="00000010" THEN
        --PLACE 0 IN THE SPACE 1
        n_state <= "00011";
    ELSIF input_reg="00000100" THEN
        --PLACE 0 IN THE SPACE 2
        n_state <= "00011";
    ELSIF input_reg="00100000" THEN
        --PLACE 0 IN THE SPACE 5
        n_state <= "00011";
    ELSIF input_reg="01000000" THEN
        --PLACE 0 IN THE SPACE 6
        n_state <= "00011";
    ELSIF input_reg="10000000" THEN
        --PLACE 0 IN THE SPACE 7
        n_state <= "00011";
    ELSIF input_reg="00010000" THEN
        --PLACE 0 IN THE SPACE 4
        n_state <= "00100";
    END IF;

    ELSIF pstate = "00011" THEN                                --STATE 3
        --PLACE A CROSS IN THE SPACE 4
        --ACTIVATE SIGNAL 'MACHINE'

    ELSIF pstate = "00100" THEN                                --STATE 4
        --PLACE A CROSS IN THE SPACE 2
        IF input_reg="00000001" THEN
            --PLACE 0 IN the SPACE 0
            n_state <= "00101";
        ELSIF input_reg="00000010" THEN
            --PLACE 0 IN THE SPACE 1
            n_state <= "00101";
        ELSIF input_reg="01000000" THEN
            --PLACE 0 IN the SPACE 6
            n_state <= "00101";
        ELSIF input_reg="10000000" THEN
            --PLACE 0 IN THE SPACE 7
            n_state <= "00101";
        ELSIF input_reg="00100000" THEN
            --PLACE 0 IN THE SPACE 5
            n_state <= "00110";
        END IF;

    ELSIF pstate = "00101" THEN                                --STATE 5
        --PLACE A CROSS IN THE SPACE 5
        --ACTIVATE SIGNAL 'MACHINE'
```

```
ELSIF pstate = "00110" THEN --STATE 6
  --PLACE A CROSS IN THE SPACE 6
  IF input_reg="00000001" THEN
    --PLACE 0 IN the SPACE 0
    n_state <= "00111";
  ELSIF input_reg="10000000" THEN
    --PLACE 0 IN THE SPACE 7
    n_state <= "00111";
  ELSIF input_reg="00000010" THEN
    --PLACE 0 IN the SPACE 1
    n_state <= "01000";
  END IF;

ELSIF pstate = "00111" THEN --STATE 7
  --PLACE A CROSS IN THE SPACE 1
  --ACTIVATE SIGNAL 'MACHINE'

ELSIF pstate = "01000" THEN --STATE 8
  --ACTIVATE SIGNAL 'DRAW'

ELSIF pstate = "01001" THEN --STATE 9
  --PLACE A CROSS IN THE SPACE 6

  IF input_reg="00000001" THEN
    --PLACE 0 IN THE SPACE 0
    n_state <= "00111";
  ELSIF input_reg="00000010" THEN
    --PLACE 0 IN THE SPACE 1
    n_state <= "01010";
  ELSIF input_reg="00000100" THEN
    --PLACE 0 IN THE SPACE 2
    n_state <= "00111";
  ELSIF input_reg="00001000" THEN
    --PLACE 0 IN THE SPACE 3
    n_state <= "00111";
  ELSIF input_reg="00010000" THEN
    --PLACE 0 IN THE SPACE 4
    n_state <= "00111";
  ELSIF input_reg="00100000" THEN
    --PLACE 0 IN THE SPACE 5
    n_state <= "00111";
  ELSIF input_reg="10000000" THEN
    --PLACE 0 IN THE SPACE 7
    n_state <= "00111";
  END IF;

ELSIF pstate = "01010" THEN --STATE 10
```

```
--PLACE A CROSS IN THE SPACE 0
IF input_reg="00000100" THEN
  --PLACE 0 IN THE SPACE 2
  n_state <= "01011";
ELSIF input_reg="00001000" THEN
  --PLACE 0 IN THE SPACE 3
  n_state <= "01011";
ELSIF input_reg="00010000" THEN
  --PLACE 0 IN THE SPACE 4
  n_state <= "01011";
ELSIF input_reg="00100000" THEN
  --PLACE 0 IN THE SPACE 5
  n_state <= "01011";
ELSIF input_reg="10000000" THEN
  --PLACE 0 IN THE SPACE 7
  n_state <= "01100";
END IF;

ELSIF pstate = "01011" THEN --STATE 11
  --PLACE A CROSS IN THE SPACE 7
  --ACTIVATE SIGNAL 'MACHINE'

ELSIF pstate = "01100" THEN --STATE 12
  --PLACE A CROSS IN THE SPACE 4
  IF input_reg="00000100" THEN
    --PLACE 0 IN THE SPACE 2
    n_state <= "01101";
  ELSIF input_reg="00100000" THEN
    --PLACE 0 IN THE SPACE 5
    n_state <= "01101";
  ELSIF input_reg="00001000" THEN
    --PLACE 0 IN THE SPACE 3
    n_state <= "01000";
  END IF;

ELSIF pstate = "01101" THEN --STATE 13
  --PLACE A CROSS IN THE SPACE 3
  --ACTIVATE SIGNAL 'MACHINE'

ELSIF pstate = "01110" THEN --STATE 14
  --PLACE A CROSS IN THE SPACE 7
  IF input_reg="00000010" THEN
    --PLACE 0 IN THE SPACE 1
    n_state <= "01111";
  ELSIF input_reg="00000100" THEN
    --PLACE 0 IN THE SPACE 2
    n_state <= "01111";
  ELSIF input_reg="00001000" THEN
```

```
        --PLACE 0 IN THE SPACE 3
        n_state <= "01111";
    ELSIF input_reg="00010000" THEN
        --PLACE 0 IN THE SPACE 4
        n_state <= "01111";
    ELSIF input_reg="00100000" THEN
        --PLACE 0 IN THE SPACE 5
        n_state <= "01111";
    ELSIF input_reg="01000000" THEN
        --PLACE 0 IN THE SPACE 6
        n_state <= "01111";
    ELSIF input_reg="00000001" THEN
        --PLACE 0 IN THE SPACE 0
        n_state <= "10000";
    END IF;

    ELSIF pstate = "01111" THEN                                --STATE 15
        --PLACE A CROSS IN THE SPACE 0
        --ACTIVATE SIGNAL 'MACHINE'

    ELSIF pstate = "10000" THEN                                --STATE 16
        --PLACE A CROSS IN THE SPACE 2
        IF input_reg="00000010" THEN
            --PLACE 0 IN THE SPACE 1
            n_state <= "00101";
        ELSIF input_reg="00001000" THEN
            --PLACE 0 IN THE SPACE 3
            n_state <= "00101";
        ELSIF input_reg="00010000" THEN
            --PLACE 0 IN THE SPACE 4
            n_state <= "00101";
        ELSIF input_reg="01000000" THEN
            --PLACE 0 IN THE SPACE 6
            n_state <= "00101";
        ELSIF input_reg="00100000" THEN
            --PLACE 0 IN THE SPACE 5
            n_state <= "00011";
        END IF;

    ELSIF pstate = "10001" THEN                                --STATE 17
        --PLACE A CROSS IN THE SPACE 2
        IF input_reg="00000001" THEN
            --PLACE 0 IN THE SPACE 0
            n_state <= "00101";
        ELSIF input_reg="00000010" THEN
            --PLACE 0 IN THE SPACE 1
            n_state <= "00101";
        ELSIF input_reg="00001000" THEN
```



```

        --PLACE 0 IN THE SPACE 3
        n_state <= "00101";
    ELSIF input_reg="00010000" THEN
        --PLACE 0 IN THE SPACE 4
        n_state <= "00101";
    ELSIF input_reg="01000000" THEN
        --PLACE 0 IN THE SPACE 6
        n_state <= "00101";
    ELSIF input_reg="10000000" THEN
        --PLACE 0 IN THE SPACE 7
        n_state <= "00101";
    ELSIF input_reg="00100000" THEN
        --PLACE 0 IN THE SPACE 5
        n_state <= "10010";
    END IF;

    ELSIF pstate = "10010" THEN                                --STATE 18
        --PLACE A CROSS IN THE SPACE 0
        IF input_reg="00000010" THEN
            --PLACE 0 IN THE SPACE 1
            n_state <= "01011";
        ELSIF input_reg="00001000" THEN
            --PLACE 0 IN THE SPACE 3
            n_state <= "01011";
        ELSIF input_reg="00010000" THEN
            --PLACE 0 IN THE SPACE 4
            n_state <= "01011";
        ELSIF input_reg="01000000" THEN
            --PLACE 0 IN THE SPACE 6
            n_state <= "01011";
        ELSIF input_reg="10000000" THEN
            --PLACE 0 IN THE SPACE 7
            n_state <= "00111";
        END IF;
    END IF;
END IF;
END IF;
END IF;
END PROCESS;
pstate_o <= pstate;
pstate <= n_state;

pc <= '1' WHEN (pstate = "00011") ELSE
      '1' WHEN (pstate = "00101") ELSE
      '1' WHEN (pstate = "00111") ELSE
      '1' WHEN (pstate = "01011") ELSE
      '1' WHEN (pstate = "01101") ELSE
      '1' WHEN (pstate = "01111") ELSE
      '0';

```

```
    draw <= '1' WHEN (PSTATE = "01000") ELSE  
        '0';  
END synth;
```

6.2.2 VHDL code of the model for two players

6.2.2.1 *Clock generator*

```
LIBRARY IEEE;
  USE IEEE.STD_LOGIC_1164.ALL;

  ENTITY clock_generator IS
    PORT (clock_i : IN std_logic;
          clock_o : OUT std_logic);
  END clock_generator;

  ARCHITECTURE synth OF clock_generator IS
    CONSTANT max_count : INTEGER := 250000; --must be 250000 for 100 Hz
    SIGNAL clock : std_logic := '0';
    SIGNAL count : INTEGER range 0 to max_count;
  BEGIN
    PROCESS (clock_i) --period of 10 ms

      BEGIN
        IF clock_i'event and clock_i='1' THEN
          IF count < max_count THEN
            count <= count+1;
          ELSE
            clock <= not clock;
            count <= 0;
          END IF;
        END IF;
      END PROCESS;
    clock_o <= clock;

  END synth;
```

6.2.2.2 Game

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY game IS
    PORT (start, clock_i : IN std_logic;
          input_kb : IN std_logic_vector (2 downto 0);
          output_kb : OUT std_logic_vector (2 downto 0);
          output_s, show, clock_o : OUT std_logic);
END game;

ARCHITECTURE synth OF game IS
    SIGNAL count2 : integer range 0 to 10 := 0; --delay of 100ms to
    ensure the keystroke
    SIGNAL count3 : integer range 0 to 21 := 1; --selector of bit in
the output vector
    SIGNAL count4 : integer range 0 to 9; --counter to selec the column
of the keyboard
    SIGNAL count5 : integer range 0 to 2; --selector of column
    SIGNAL count6 : integer range 0 to 2; --delay before starting the
output
    SIGNAL play, launch, show_i : std_logic := '0';
    SIGNAL player, serialize : std_logic := '1';
    SIGNAL output_kb_i : std_logic_vector (2 downto 0) := "111";
    SIGNAL input_i : std_logic_vector (8 downto 0) := "000000000";
    --vector with the pushed key
    SIGNAL input_reg : : std_logic_vector (8 downto 0) := "000000000";
    --registered pushed key
    SIGNAL reg : std_logic_vector (8 downto 0) := "000000000";
--register of the pushed keys during the game
    SIGNAL requirement : std_logic_vector (8 downto 0) := "000000000";
    --has been already pushed this key?
    SIGNAL output_i : std_logic_vector (20 downto 0) :=
"11111111111111111111"; --output vector
    SIGNAL mask : std_logic_vector (20 downto 0) :=
"00000000000000000000"; --mask to update reg_2 with the current
pushed key
    SIGNAL reg_2 : std_logic_vector (20 downto 0) :=
"00000000000000000000"; --output register without the state flags
    SIGNAL reg_3 : std_logic_vector (20 downto 0) :=
"00000000000000000000"; --reg_2 delayed one clock cycle. Used to
control 'play'
    SIGNAL beginning : std_logic := '1';

BEGIN
    keyboard: PROCESS (clock_i) --period of 100 ms
    BEGIN
```

```
IF clock_i'event and clock_i='1' THEN
  IF start = '0' or beginning = '1' THEN
    count4 <= 0;
    count5 <= 0;
  ELSIF count4 < 9 THEN
    count4 <= count4+1;
  ELSIF input_i /= "000000000" THEN
    count4 <= 0;
  ELSE
    IF count5 < 2 THEN
      count5 <= count5+1;
    ELSE
      count5 <= 0;
    END IF;
    count4 <= 0;
  END IF;
END IF;
END PROCESS;

read_input: PROCESS (clock_i)
BEGIN

  IF clock_i'event and clock_i='1' THEN

    IF start = '0' or beginning = '1' THEN
      count2 <= 0;
      count3 <= 1;
      count6 <= 0;
      launch <= '0';
      show_i <= '0';
      input_reg <="000000000";
      reg <="000000000";
      reg_2 <="000000000000000000000000";
      reg_3 <="000000000000000000000000";
      input_reg <= "000000000"; --registered input
      reg <= "000000000";
      player <= '1';
      serialize <= '1';
      beginning <= '0';
    ELSIF play = '1' THEN

      IF input_i /= "000000000" THEN
        IF count2 < 10 THEN
          count2 <= count2+1;
        ELSE
          IF requirement = "000000000" THEN
            reg <= input_i or reg;
          END IF;
        END IF;
      END IF;
    END IF;
  END IF;
END PROCESS;
```

```

        input_reg <= input_i;
        player <= not player;
        serialize <= '1';
    END IF;
    count2 <= 0;
END IF;
ELSE
    count2 <= 0;
END IF;
reg_2 <= reg_2 or mask;
reg_3 <= reg_2;

END IF;

IF serialize='1' THEN
    IF count6 < 2 THEN
        count6 <= count6+1;
    ELSE
        count3 <= 21;

        serialize <='0';
        count6 <= 0;
    END IF;
ELSIF count3 > 0 THEN
    count3 <= count3-1;
    output_s <= output_i(count3-1);
    launch <= '1';
ELSIF launch = '1' THEN
    launch <= '0';
    show_i <= '1';

ELSE
    show_i <='0';
END IF;

END IF;
END PROCESS;

output_i <= reg_2 or "00100000000000000000" when reg_2(9)='1' and
reg_2(10)='1' and reg_2(11)='1' else
reg_2 or "001000000000000000000000" when reg_2(9)='1' and
reg_2(12)='1' and reg_2(15)='1' else
reg_2 or "001000000000000000000000" when reg_2(9)='1' and
reg_2(13)='1' and reg_2(17) = '1' else
reg_2 or "001000000000000000000000" when reg_2(10)='1' and
reg_2(13)='1' and reg_2(16)='1' else
reg_2 or "001000000000000000000000" when reg_2(11)='1' and
reg_2(14)='1' and reg_2(17)='1' else
```

```
reg_2 or "00100000000000000000" when reg_2(11)='1' and
reg_2(13)='1' and reg_2(15)='1' else
reg_2 or "00100000000000000000" when reg_2(12)='1' and
reg_2(13)='1' and reg_2(14)='1' else
reg_2 or "00100000000000000000" when reg_2(15)='1' and
reg_2(16)='1' and reg_2(17)='1' else
reg_2 or "10000000000000000000" when reg_2(0)='1' and
reg_2(1)='1' and reg_2(2)='1' else
reg_2 or "10000000000000000000" when reg_2(0)='1' and
reg_2(3)='1' and reg_2(6)='1' else
reg_2 or "10000000000000000000" when reg_2(0)='1' and
reg_2(4)='1' and reg_2(8)='1' else
reg_2 or "10000000000000000000" when reg_2(1)='1' and
reg_2(4)='1' and reg_2(7)='1' else
reg_2 or "10000000000000000000" when reg_2(2)='1' and
reg_2(5)='1' and reg_2(8)='1' else
reg_2 or "10000000000000000000" when reg_2(2)='1' and
reg_2(4)='1' and reg_2(6)='1' else
reg_2 or "10000000000000000000" when reg_2(3)='1' and
reg_2(4)='1' and reg_2(5)='1' else
reg_2 or "10000000000000000000" when reg_2(6)='1' and
reg_2(7)='1' and reg_2(8)='1' else
reg_2 or "01000000000000000000" when reg="111111111" else
reg_2;
```

```
play<='0' when reg_3(9)='1' and reg_3(10)='1' and reg_3(11)='1'
else
    '0' when reg_3(9)='1' and reg_3(12)='1' and reg_3(15)='1'
else
    '0' when reg_3(9)='1' and reg_3(13)='1' and reg_3(17)='1'
else
    '0' when reg_3(10)='1' and reg_3(13)='1' and reg_3(16)='1'
else
    '0' when reg_3(11)='1' and reg_3(14)='1' and reg_3(17)='1'
else
    '0' when reg_3(11)='1' and reg_3(13)='1' and reg_3(15)='1'
else
    '0' when reg_3(12)='1' and reg_3(13)='1' and reg_3(14)='1'
else
    '0' when reg_3(15)='1' and reg_3(16)='1' and reg_3(17)='1'
else
    '0' when reg_3(0)='1' and reg_3(1)='1' and reg_3(2)='1' else
    '0' when reg_3(0)='1' and reg_3(3)='1' and reg_3(6)='1' else
    '0' when reg_3(0)='1' and reg_3(4)='1' and reg_3(8)='1' else
    '0' when reg_3(1)='1' and reg_3(4)='1' and reg_3(7)='1' else
    '0' when reg_3(2)='1' and reg_3(5)='1' and reg_3(8)='1' else
    '0' when reg_3(2)='1' and reg_3(4)='1' and reg_3(6)='1' else
    '0' when reg_3(3)='1' and reg_3(4)='1' and reg_3(5)='1' else
```

Systematic testing of digital hardware systems by means of test automaton

```
'0' when reg_3(6)='1' and reg_3(7)='1' and reg_3(8)='1' else
'1';

mask <= "000000000001000000000" WHEN input_reg="000000001" AND
player='0' else
"000000000010000000000" WHEN input_reg="000000010" AND
player='0' else
"000000000100000000000" WHEN input_reg="000000100" AND
player='0' else
"000000001000000000000" WHEN input_reg="000001000" AND
player='0' else
"000000010000000000000" WHEN input_reg="000010000" AND
player='0' else
"000000100000000000000" WHEN input_reg="000100000" AND
player='0' else
"000001000000000000000" WHEN input_reg="001000000" AND
player='0' else
"000010000000000000000" WHEN input_reg="010000000" AND
player='0' else
"000100000000000000000" WHEN input_reg="100000000" AND
player='0' else
"0000000000000000000001" WHEN input_reg="000000001" AND
player='1' else
"0000000000000000000010" WHEN input_reg="000000010" AND
player='1' else
"00000000000000000000100" WHEN input_reg="000000100" AND
player='1' else
"000000000000000000001000" WHEN input_reg="000001000" AND
player='1' else
"0000000000000000000010000" WHEN input_reg="000010000" AND
player='1' else
"00000000000000000000100000" WHEN input_reg="000100000" AND
player='1' else
"000000000000000000001000000" WHEN input_reg="001000000" AND
player='1' else
"0000000000000000000010000000" WHEN input_reg="010000000" AND
player='1' else
"00000000000000000000100000000" WHEN input_reg="100000000" AND
player='1' else
"000000000000000000000000" WHEN start='1' ELSE
"000000000000000000000000";

input_i <= "000000001" WHEN input_kb="110" AND output_kb_i="110"
else
"0000000010" WHEN input_kb="110" AND output_kb_i="101" else
"0000000100" WHEN input_kb="110" AND output_kb_i="011" else
"0000001000" WHEN input_kb="101" AND output_kb_i="110" else
"000010000" WHEN input_kb="101" AND output_kb_i="101" else
```


Systematic testing of digital hardware systems by means of test automaton

```
"000100000" WHEN input_kb="101" AND output_kb_i="011" else  
"001000000" WHEN input_kb="011" AND output_kb_i="110" else  
"010000000" WHEN input_kb="011" AND output_kb_i="101" else  
"100000000" WHEN input_kb="011" AND output_kb_i="011" else  
"000000000";
```

```
output_kb_i <= "110" WHEN count5=0 ELSE  
    "101" WHEN count5=1 ELSE  
    "011" WHEN count5=2 ELSE  
    "111";
```

```
clock_o <= not clock_i WHEN launch='1' else  
    '0';
```

```
requirement <= input_i and reg;  
output_kb <= output_kb_i;  
show <= show_i;
```

```
END synth;
```

6.2.2.3 TicTacToe

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY TicTacToe IS
  PORT (Start, clk : IN std_logic;
        input_kb : IN std_logic_vector (2 downto 0);
        output_kb : OUT std_logic_vector (2 downto 0);
        output_s, show, clock_o : OUT std_logic);

END TicTacToe;

ARCHITECTURE synth OF TicTacToe IS

  COMPONENT clock_generator IS
    PORT (clock_i : IN std_logic;
          clock_o : OUT std_logic);
  END COMPONENT;

  COMPONENT game IS
    PORT (start, clock_i : IN std_logic;
          input_kb : IN std_logic_vector (2 downto 0);
          output_kb : OUT std_logic_vector (2 downto 0);
          output_s, show, clock_o : OUT std_logic);
  END COMPONENT;

  SIGNAL clk_ii, output_si, show_i, clock_oi : std_logic;
  SIGNAL output_kbi : std_logic_vector (2 downto 0);

BEGIN

  mod_clock: clock_generator    PORT MAP (clk, clk_ii);

  mod_game: game    PORT MAP (start, clk_ii, input_kb,
    output_kbi, output_si, show_i, clock_oi);

  output_kb <= output_kbi;
  output_s <= output_si;
  show <= show_i;
  clock_o <= clock_oi;

END synth;
```

6.3 Appendix C: Simulation code

```

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY TicTacToe_tb IS
END TicTacToe_tb ;

ARCHITECTURE DUT OF TicTacToe_tb IS

COMPONENT TicTacToe IS
    PORT (Start, clk : IN std_logic;
          input_kb : IN std_logic_vector (2 downto 0);
          output_kb : OUT std_logic_vector (2 downto 0);
          output_s, show, clock_o : OUT std_logic);
END component ;

signal Start_i, clk_i : std_logic := '0';
signal input_i : std_logic_vector(2 downto 0);
signal output_kb_i : std_logic_vector(2 downto 0);
signal output_s_i, show_i, clock_o_i : std_logic;

BEGIN
    TicTacToe_i: TicTacToe port map (Start_i, clk_i, input_i, output_kb_i,
    output_s_i, show_i, clock_o_i);
    clk_i <= NOT clk_i after 10 ns;
    process

    begin

        input_i <= "111"; Start_i <= '1';
        wait for 1020 ms;
        wait until output_kb_i = "110";
        input_i <= "110"; wait for 300 ms;
        input_i <= "111"; wait for 2020 ms;
        wait until output_kb_i = "110";
        input_i <= "101"; wait for 300 ms;
        input_i <= "111"; wait for 2020 ms;
        wait until output_kb_i = "101";
        input_i <= "101"; wait for 300 ms;
        input_i <= "111"; wait for 2020 ms;
        wait until output_kb_i = "110";
        input_i <= "011"; wait for 300 ms;
        input_i <= "111"; wait for 2020 ms;
        wait until output_kb_i = "110";
        input_i <= "101"; wait for 300 ms;
        input_i <= "111"; wait for 2020 ms;
        wait until output_kb_i = "011";
        input_i <= "011"; wait for 175 ms; --player 1 wins
        assert output_s_i='1' report "Expected 1" severity failure;
        report "Player 1 wins. The model works as expected";
        wait for 125 ms;
        input_i <= "111"; wait for 2000 ms;

        --start again
        Start_i <= '0'; wait for 310 ms;
        Start_i <= '1'; wait for 1000 ms;

        wait until output_kb_i = "110";
        input_i <= "110"; wait for 300 ms;
        input_i <= "111"; wait for 2020 ms;
        wait until output_kb_i = "101";
        input_i <= "101"; wait for 300 ms;
        input_i <= "111"; wait for 2020 ms;
        wait until output_kb_i = "110";
        input_i <= "101"; wait for 300 ms;
    end process;
end

```

Systematic testing of digital hardware systems by means of test automaton

```
input_i <= "111"; wait for 2020 ms;
wait until output_kb_i = "101";
input_i <= "011"; wait for 300 ms;
input_i <= "111"; wait for 2020 ms;
wait until output_kb_i = "011";
input_i <= "011"; wait for 300 ms;
input_i <= "111"; wait for 2020 ms;
wait until output_kb_i = "101";
input_i <= "110"; wait for 155 ms; -- player 2 wins
assert output_s_i='1' report "Expected 1" severity failure;
report "Player 2 wins. The model works as expected";
wait for 145 ms;
input_i <= "111"; wait for 2000 ms;

--start again
Start_i <= '0'; wait for 310 ms;
Start_i <= '1'; wait for 1000 ms;

wait until output_kb_i = "101";
input_i <= "101"; wait for 300 ms;
input_i <= "111"; wait for 2020 ms;
wait until output_kb_i = "110";
input_i <= "110"; wait for 300 ms;
input_i <= "111"; wait for 2020 ms;
wait until output_kb_i = "110";
input_i <= "101"; wait for 300 ms;
input_i <= "111"; wait for 2020 ms;
wait until output_kb_i = "011";
input_i <= "101"; wait for 300 ms;
input_i <= "111"; wait for 2020 ms;
wait until output_kb_i = "011";
input_i <= "110"; wait for 300 ms;
input_i <= "111"; wait for 2020 ms;
wait until output_kb_i = "110";
input_i <= "011"; wait for 300 ms;
input_i <= "111"; wait for 2020 ms;
wait until output_kb_i = "101";
input_i <= "011"; wait for 300 ms;
input_i <= "111"; wait for 2020 ms;
wait until output_kb_i = "101";
input_i <= "110"; wait for 300 ms;
input_i <= "111"; wait for 2020 ms;
wait until output_kb_i = "011";
input_i <= "011"; wait for 165 ms; -- Nobody win
assert output_s_i='1' report "Expected 1" severity failure;
report "Draw. The model works as expected";
wait for 135 ms;
input_i <= "111"; wait for 1000 ms;
report "End of the test. The model works right";

--      assert f_i='0' report "Expected 0" severity failure;
--      report "The model works as expected";
--      wait;
--      end process;
END DUT ;
```

6.4 Appendix D: Test Description Language code

```

TDLan Specification game_tdl {
  Data Set logic {
    instance std_logic;
    instance std_logic_vector;
  }

  Gate Type input accepts logic;
  Gate Type output accepts logic;

  Component Type hardware {
    gate types: input,output;
  }

  Component Type simulation {
    gate types: input,output;
  }

  Component Type TB {
    gate types: input,output;
  }

  Time Unit milliseconds;

  Test Configuration game_tdl_cf {
    instantiate game as DUT of type hardware having {
      gate Start of type input with length of 1;
      gate clock_i of type input with length of 1;
      gate input_kb of type input with length of 3;
      gate output_kb of type output with length of 3;
      gate output_s of type output with length of 1;
      gate show of type output with length of 1;
      gate clock_o of type output with length of 1;
    }

    instantiate TB_a as Tester of type TB having {
      gate Starter of type output with length of 1;
      gate CLKgenerator of type output with length of 1;
      gate row_o of type output with length of 3;
      gate column_i of type input with length of 3;
      gate parallelize of type input with length of 1;
      gate clk of type input with length of 1;
      gate input_s of type input with length of 1;
    }

    connect gate Starter to gate Start;
    connect gate CLKgenerator to gate clock_i;
    connect gate row_o to gate input_kb;

    connect gate column_i to gate output_kb;
    connect gate parallelize to gate show;
    connect gate clk to gate clock_o;
    connect gate input_s to gate output_s;

    Assert deviation is (1 milliseconds);
  }

  SignalAdapter Configuration de0_nano_output {
    signaladapter output_adapter1 of type output having{
      attach Start 0 downto 0 to position 0 downto 0;
      attach clock_i 0 downto 0 to position 1 downto 1;
      attach input_kb 2 downto 0 to position 4 downto 2;
      logiclevel LVTTTL;
      type de0_nano_pappkisteOut;
      hardware_revision "0.1";
    }
  }
}

```

Systematic testing of digital hardware systems by means of test automaton

```
        software_revision "0.1";
        serial number "001";
        connection JTAG;
        address "USB-Blaster [1-6.1]";
    }

    signaladapter input_adapter1 of type input having{
        attach output_kb 2 downto 0 to position 2 downto 0;
        attach show 0 downto 0 to position 3 downto 3;
        attach clock_o 0 downto 0 to position 4 downto 4;
        attach output_s 0 downto 0 to position 5 downto 5 ;
        logiclevel LVTTTL;
        type de0_nano_pappkisteIn;
        hardware_revision "0.1";
        software_revision "0.1";
        serial number "001";
        connection JTAG;
        address "USB-Blaster [1-6.3]";
    }
}

Test Description test_the_game{
    use Test configuration: game_tdl_cf{
        run {
            repeat 2000 times {
                TB_a sends bit value of b0 to gate clock_i;
                gate clock_i waits for (5 milliseconds);
                TB_a sends bit value of b1 to gate clock_i;
                gate clock_i waits for (5 milliseconds);
            }
        }
        in parallel to {
            TB_a sends bus value of 7 to gate input_kb;
            gate input_kb waits for (1205 milliseconds);
            TB_a sends bus value of 6 to gate input_kb;
            gate input_kb waits for (300 milliseconds);
            TB_a sends bus value of 7 to gate input_kb;
            gate input_kb waits for (2100 milliseconds);
            TB_a sends bus value of 5 to gate input_kb;
            gate input_kb waits for (300 milliseconds);
            TB_a sends bus value of 7 to gate input_kb;
            gate input_kb waits for (2200 milliseconds);
            TB_a sends bus value of 5 to gate input_kb;
            gate input_kb waits for (300 milliseconds);
            TB_a sends bus value of 7 to gate input_kb;
            gate input_kb waits for (2300 milliseconds);
            TB_a sends bus value of 3 to gate input_kb;
            gate input_kb waits for (300 milliseconds);
            TB_a sends bus value of 7 to gate input_kb;
            gate input_kb waits for (2100 milliseconds);
            TB_a sends bus value of 5 to gate input_kb;
            gate input_kb waits for (300 milliseconds);
            TB_a sends bus value of 7 to gate input_kb;
            gate input_kb waits for (2300 milliseconds);
            TB_a sends bus value of 3 to gate input_kb;
            gate input_kb waits for (300 milliseconds);
            TB_a sends bus value of 7 to gate input_kb;
            gate input_kb waits for (1540 milliseconds);
            TB_a sends bus value of 6 to gate input_kb;
            gate input_kb waits for (300 milliseconds);
            TB_a sends bus value of 7 to gate input_kb;
            gate input_kb waits for (2200 milliseconds);
            TB_a sends bus value of 5 to gate input_kb;
            gate input_kb waits for (300 milliseconds);
            TB_a sends bus value of 7 to gate input_kb;
            gate input_kb waits for (2300 milliseconds);
            TB_a sends bus value of 5 to gate input_kb;
            gate input_kb waits for (300 milliseconds);
        }
    }
}
```

```

TB_a sends bus value of 7 to gate input_kb;
gate input_kb waits for (2200 milliseconds);
TB_a sends bus value of 3 to gate input_kb;
gate input_kb waits for (300 milliseconds);
TB_a sends bus value of 7 to gate input_kb;
gate input_kb waits for (2200 milliseconds);
TB_a sends bus value of 3 to gate input_kb;
gate input_kb waits for (300 milliseconds);
TB_a sends bus value of 7 to gate input_kb;
gate input_kb waits for (2300 milliseconds);
TB_a sends bus value of 6 to gate input_kb;
gate input_kb waits for (300 milliseconds);
TB_a sends bus value of 7 to gate input_kb;
gate input_kb waits for (15500 milliseconds);
TB_a sends bus value of 5 to gate input_kb;
gate input_kb waits for (300 milliseconds);
TB_a sends bus value of 7 to gate input_kb;
gate input_kb waits for (2300 milliseconds);
TB_a sends bus value of 6 to gate input_kb;
gate input_kb waits for (300 milliseconds);
TB_a sends bus value of 7 to gate input_kb;
gate input_kb waits for (2100 milliseconds);
TB_a sends bus value of 5 to gate input_kb;
gate input_kb waits for (300 milliseconds);
TB_a sends bus value of 7 to gate input_kb;
gate input_kb waits for (2300 milliseconds);
TB_a sends bus value of 5 to gate input_kb;
gate input_kb waits for (300 milliseconds);
TB_a sends bus value of 7 to gate input_kb;
gate input_kb waits for (2100 milliseconds);
TB_a sends bus value of 6 to gate input_kb;
gate input_kb waits for (300 milliseconds);
TB_a sends bus value of 7 to gate input_kb;
gate input_kb waits for (2100 milliseconds);
TB_a sends bus value of 3 to gate input_kb;
gate input_kb waits for (300 milliseconds);
TB_a sends bus value of 7 to gate input_kb;
gate input_kb waits for (2200 milliseconds);
TB_a sends bus value of 3 to gate input_kb;
gate input_kb waits for (300 milliseconds);
TB_a sends bus value of 7 to gate input_kb;
}
in parallel to {
TB_a sends bit value of b1 to gate Start;
gate Start waits for (16105 milliseconds);
TB_a sends bit value of b0 to gate Start;
gate Start waits for (310 milliseconds);
TB_a sends bit value of b1 to gate Start;
gate Start waits for (15690 milliseconds);
TB_a sends bit value of b0 to gate Start;
gate Start waits for (310 milliseconds);
TB_a sends bit value of b1 to gate Start;
}
in parallel to {
game sends bit value of b0 to gate clk;
gate clk waits for (40 milliseconds);
game sends bit value of b1 to gate clk;
gate clk waits for (5 milliseconds);
game sends bit value of b0 to gate clk;
gate clk waits for (5 milliseconds);
game sends bit value of b1 to gate clk;
}

```


Systematic testing of digital hardware systems by means of test automaton

```
        gate column_i waits for (100 milliseconds);
        game sends bus value of 5 to gate column_i;
        gate column_i waits for (100 milliseconds);
        game sends bus value of 3 to gate column_i;
        gate column_i waits for (100 milliseconds);
        game sends bus value of 6 to gate column_i;
        gate column_i waits for (100 milliseconds);
        game sends bus value of 5 to gate column_i;
        gate column_i waits for (100 milliseconds);
        game sends bus value of 3 to gate column_i;
        gate column_i waits for (100 milliseconds);
        game sends bus value of 6 to gate column_i;
        gate column_i waits for (100 milliseconds);
        game sends bus value of 5 to gate column_i;
        gate column_i waits for (100 milliseconds);
        game sends bus value of 3 to gate column_i;
        gate column_i waits for (100 milliseconds);
        game sends bus value of 6 to gate column_i;
        gate column_i waits for (100 milliseconds);
        game sends bus value of 5 to gate column_i;
        gate column_i waits for (100 milliseconds);
        game sends bus value of 3 to gate column_i;
        gate column_i waits for (400 milliseconds);
    }
    terminate;
}
}
```

6.5 Appendix E: Errors reported by the TDL testbench with malfunctions in the model

```
# ** Warning: Assert failed at 3761 ms
#   Time: 5761 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3766 ms
#   Time: 5764 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3771 ms
#   Time: 5771 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3776 ms
#   Time: 5774 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3781 ms
#   Time: 5781 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3786 ms
#   Time: 5784 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3791 ms
#   Time: 5791 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3796 ms
#   Time: 5794 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3801 ms
#   Time: 5801 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3806 ms
#   Time: 5804 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3811 ms
#   Time: 5811 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3816 ms
#   Time: 5814 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3821 ms
#   Time: 5821 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3826 ms
#   Time: 5824 ms  Iteration: 0  Instance: /game_tdl_tb
```



```
# ** Warning: Assert failed at 3831 ms
#   Time: 5831 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3836 ms
#   Time: 5834 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3841 ms
#   Time: 5841 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3846 ms
#   Time: 5844 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3851 ms
#   Time: 5851 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3856 ms
#   Time: 5854 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3861 ms
#   Time: 5861 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3866 ms
#   Time: 5864 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3866 ms
#   Time: 5866 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3871 ms
#   Time: 5871 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3876 ms
#   Time: 5874 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3876 ms
#   Time: 5874 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3881 ms
#   Time: 5881 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3886 ms
#   Time: 5884 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3891 ms
#   Time: 5891 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3896 ms
#   Time: 5894 ms  Iteration: 0  Instance: /game_tdl_tb
```

```
# ** Warning: Assert failed at 3901 ms
#   Time: 5901 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3906 ms
#   Time: 5904 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3911 ms
#   Time: 5911 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3916 ms
#   Time: 5914 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3921 ms
#   Time: 5921 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3926 ms
#   Time: 5924 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3926 ms
#   Time: 5926 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3931 ms
#   Time: 5931 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3936 ms
#   Time: 5934 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3936 ms
#   Time: 5934 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3941 ms
#   Time: 5941 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3946 ms
#   Time: 5944 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3951 ms
#   Time: 5951 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3956 ms
#   Time: 5954 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3961 ms
#   Time: 5961 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3966 ms
#   Time: 5964 ms  Iteration: 0  Instance: /game_tdl_tb
```

```
# ** Warning: Assert failed at 3966 ms
#   Time: 5966 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 3976 ms
#   Time: 5974 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6261 ms
#   Time: 8261 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6266 ms
#   Time: 8264 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6271 ms
#   Time: 8271 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6276 ms
#   Time: 8274 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6281 ms
#   Time: 8281 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6286 ms
#   Time: 8284 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6291 ms
#   Time: 8291 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6296 ms
#   Time: 8294 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6301 ms
#   Time: 8301 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6306 ms
#   Time: 8304 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6311 ms
#   Time: 8311 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6316 ms
#   Time: 8314 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6321 ms
#   Time: 8321 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6326 ms
#   Time: 8324 ms  Iteration: 0  Instance: /game_tdl_tb
```

```
# ** Warning: Assert failed at 6326 ms
#   Time: 8326 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6331 ms
#   Time: 8331 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6336 ms
#   Time: 8334 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6336 ms
#   Time: 8334 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6341 ms
#   Time: 8341 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6346 ms
#   Time: 8344 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6351 ms
#   Time: 8351 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6356 ms
#   Time: 8354 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6361 ms
#   Time: 8361 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6366 ms
#   Time: 8364 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6366 ms
#   Time: 8366 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6371 ms
#   Time: 8371 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6376 ms
#   Time: 8374 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6376 ms
#   Time: 8374 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6381 ms
#   Time: 8381 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6386 ms
#   Time: 8384 ms  Iteration: 0  Instance: /game_tdl_tb
```

```
# ** Warning: Assert failed at 6391 ms
#   Time: 8391 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6396 ms
#   Time: 8394 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6401 ms
#   Time: 8401 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6406 ms
#   Time: 8404 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6411 ms
#   Time: 8411 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6416 ms
#   Time: 8414 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6421 ms
#   Time: 8421 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6426 ms
#   Time: 8424 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6426 ms
#   Time: 8426 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6431 ms
#   Time: 8431 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6436 ms
#   Time: 8434 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6436 ms
#   Time: 8434 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6441 ms
#   Time: 8441 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6446 ms
#   Time: 8444 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6451 ms
#   Time: 8451 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6456 ms
#   Time: 8454 ms  Iteration: 0  Instance: /game_tdl_tb
```

```
# ** Warning: Assert failed at 6461 ms
#   Time: 8461 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6466 ms
#   Time: 8464 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6466 ms
#   Time: 8466 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 6476 ms
#   Time: 8474 ms  Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 8926 ms
#   Time: 10926 ms Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 8936 ms
#   Time: 10934 ms Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 9026 ms
#   Time: 11026 ms Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 9036 ms
#   Time: 11034 ms Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 13876 ms
#   Time: 15876 ms Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 13886 ms
#   Time: 15884 ms Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 13926 ms
#   Time: 15926 ms Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 13936 ms
#   Time: 15934 ms Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 14026 ms
#   Time: 16026 ms Iteration: 0  Instance: /game_tdl_tb
# ** Warning: Assert failed at 14036 ms
#   Time: 16034 ms Iteration: 0  Instance: /game_tdl_tb
```

6.6 Appendix F: Terasic DE0-Nano board

The DE0-Nano board is a compact-sized FPGA development platform suitable for portable design projects, robots and mobile projects.

DE0-Nano is ideal for use with embedded processors—it includes an Altera Cyclone IV FPGA (with 22,320 logic elements), 32 MB of SDRAM, 2 Kb EEPROM, and a 64 Mb serial configuration memory device. DE0-Nano includes a National Semiconductor 8-channel 12-bit A/D converter for connecting to real-world sensors , and it also includes an Analog Devices 13-bit, 3-axis accelerometer device.

The DE0-Nano board includes a built-in USB Blaster for FPGA programming, and the board can be powered either from this USB port or by an external power source. The board includes expansion headers that can be used to attach various Terasic daughter cards or other devices, such as motors and actuators, 2 pushbuttons, 8 user LEDs and a set of 4 dip-switches [7].

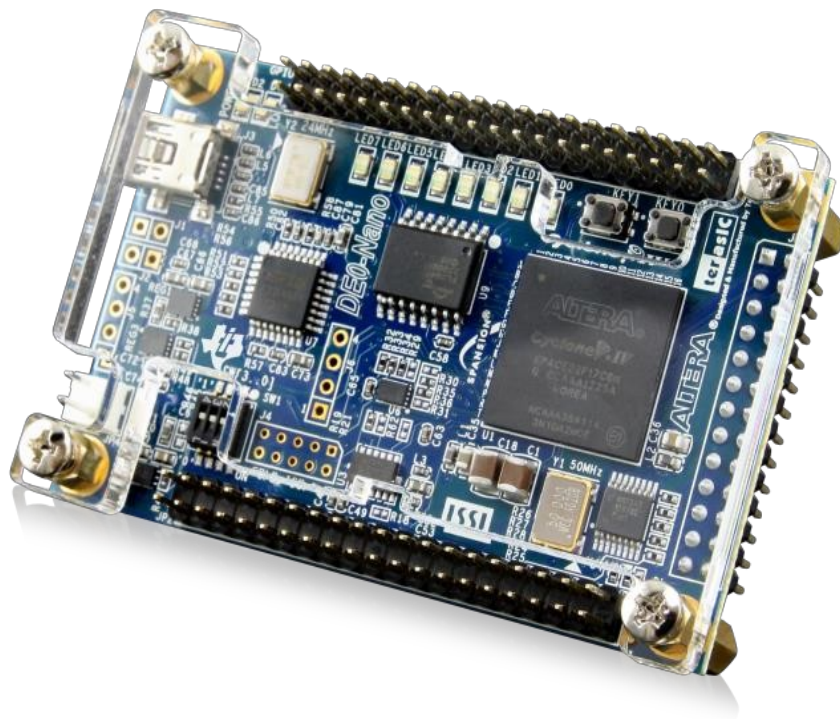


Figure 64: DE0-Nano board
Obtained from [7]

6.6.1 Features

The key features of the board are listed below:

- Featured device
 - Altera Cyclone® IV EP4CE22F17C6N FPGA
 - 153 maximum FPGA I/O pins

- Configuration status and set-up elements
 - On-board USB-Blaster circuit for programming
 - Spansion EPCS64

- Expansion header
 - Two 40-pin Headers (GPIOs) provide 72 I/O pins, 5V power pins, two 3.3V power pins and four ground pins

- Memory devices
 - 32MB SDRAM
 - 2Kb I2C EEPROM

- General user input/output
 - 8 green LEDs
 - 2 debounced pushbuttons
 - 4-position DIP switch

- G-Sensor
 - ADI ADXL345, 3-axis accelerometer with high resolution (13-bit)

- A/D Converter
 - NS ADC128S022, 8-Channel, 12-bit A/D Converter
 - 50 Ksps to 200 Ksps

- Clock system
 - On-board 50MHz clock oscillator
- Power Supply
 - USB Type mini-AB port (5V)
 - DC 5V pin for each GPIO header (2 DC 5V pins)
 - 2-pin external power header (3.6-5.7V)

6.6.2 DE0-Nano Board Architecture

In this chapter, the architecture on the DE0-Nano board is described, including components and block diagram[7].

Layout and components

Figure 64 and Figure 65 show the DE0-Nano board. They represent the layout of the board and indicates the locations of the most relevant components and the connectors [7].

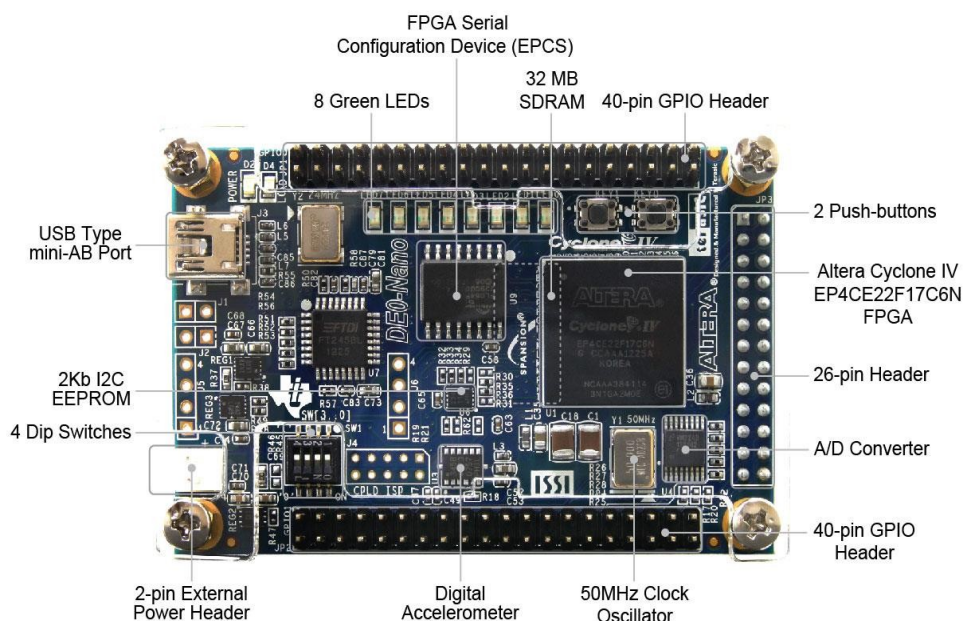


Figure 65: The DE0-Nano Board PCB and component diagram (top view)
Obtained from [7]

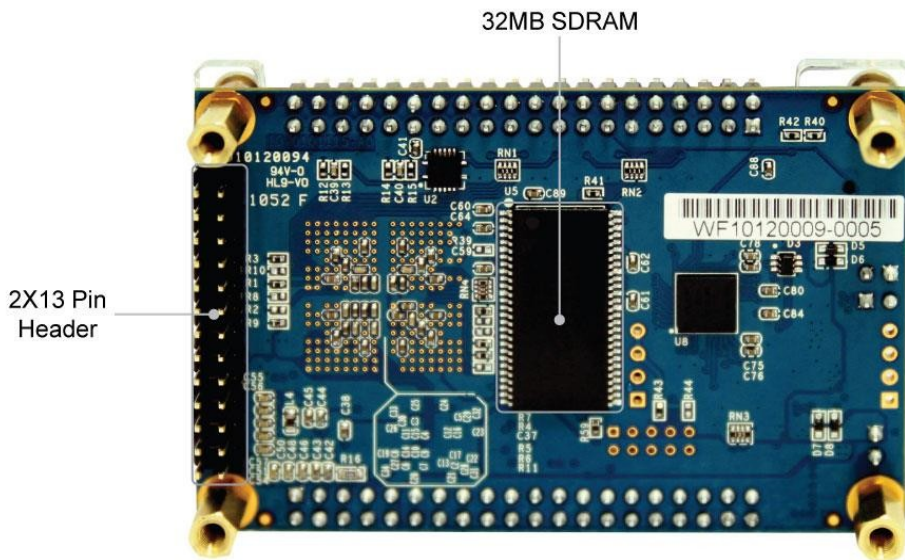


Figure 66: The DE0-Nano Board PCB and component diagram (bottom view)
Obtained from [7]

Block diagram of the DE0-Nano board

Figure 66 shows the block diagram of the DE0-Nano board. All connections are made through the Cyclone IV FPGA device in order to provide maximum flexibility for the user. In this way, the user can configure the FPGA to implement any system design [7].

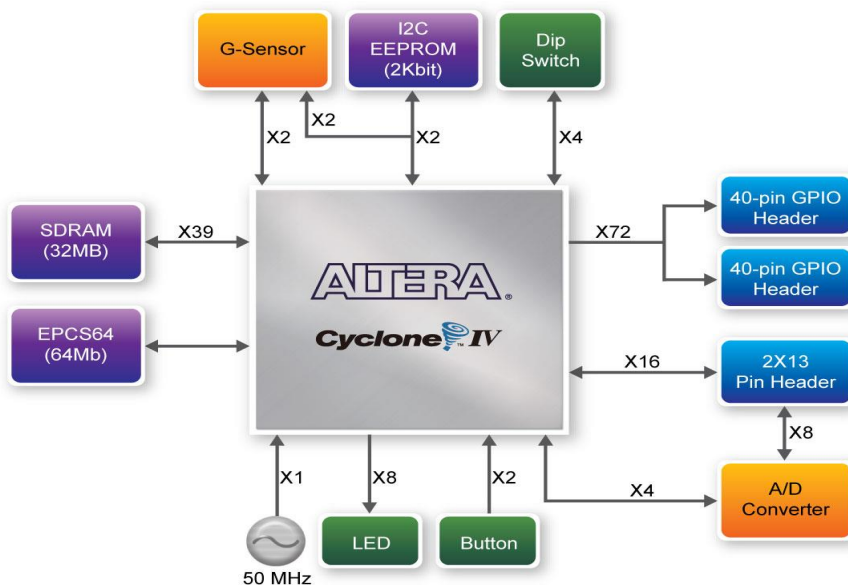


Figure 67: Block diagram of DE0-Nano Board
Obtained from [7]

6.7 Appendix G: 74LS595 datasheet

SN54HC595, SN74HC595 8-BIT SHIFT REGISTERS WITH 3-STATE OUTPUT REGISTERS

SCLS041G – DECEMBER 1982 – REVISED FEBRUARY 2004

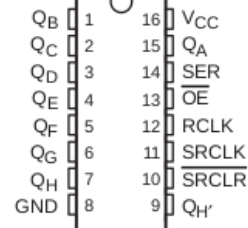
- 8-Bit Serial-In, Parallel-Out Shift
- Wide Operating Voltage Range of 2 V to 6 V
- High-Current 3-State Outputs Can Drive Up To 15 LSTTL Loads
- Low Power Consumption, 80- μ A Max I_{CC}
- Typical $t_{pd} = 13$ ns
- ± 6 -mA Output Drive at 5 V
- Low Input Current of 1 μ A Max
- Shift Register Has Direct Clear

description/ordering information

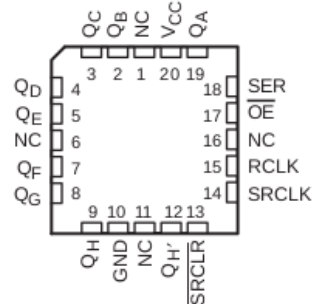
The 'HC595 devices contain an 8-bit serial-in, parallel-out shift register that feeds an 8-bit D-type storage register. The storage register has parallel 3-state outputs. Separate clocks are provided for both the shift and storage register. The shift register has a direct overriding clear (\overline{SRCLR}) input, serial (SER) input, and serial outputs for cascading. When the output-enable (\overline{OE}) input is high, the outputs are in the high-impedance state.

Both the shift register clock (SRCLK) and storage register clock (RCLK) are positive-edge triggered. If both clocks are connected together, the shift register always is one clock pulse ahead of the storage register.

SN54HC595 ... J OR W PACKAGE
SN74HC595 ... D, DB, DW, N, OR NS PACKAGE
(TOP VIEW)



SN54HC595 ... FK PACKAGE
(TOP VIEW)



NC – No internal connection

ORDERING INFORMATION

T_A	PACKAGE†		ORDERABLE PART NUMBER	TOP-SIDE MARKING
-40°C to 85°C	PDIP – N	Tube of 25	SN74HC595N	SN74HC595N
	SOIC – D	Tube of 40	SN74HC595D	HC595
		Reel of 2500	SN74HC595DR	
	SOIC – DW	Tube of 40	SN74HC595DW	HC595
		Reel of 2000	SN74HC595DWR	
	SOP – NS	Reel of 2000	SN74HC595NSR	HC595
SSOP – DB		Reel of 2000	SN74HC595DBR	HC595
-55°C to 125°C	CDIP – J	Tube of 25	SNJ54HC595J	SNJ54HC595J
	CFP – W	Tube of 150	SNJ54HC595W	SNJ54HC595W
	LCCC – FK	Tube of 55	SNJ54HC595FK	SNJ54HC595FK

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.



Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this data sheet.

PRODUCTION DATA information is current as of publication date. Products conform to specifications per the terms of Texas Instruments standard warranty. Production processing does not necessarily include testing of all parameters.



POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Copyright © 2004, Texas Instruments Incorporated
On products compliant to MIL-PRF-38535, all parameters are tested unless otherwise noted. On all other products, production processing does not necessarily include testing of all parameters.

1

SN54HC595, SN74HC595
8-BIT SHIFT REGISTERS
WITH 3-STATE OUTPUT REGISTERS

SCLS041G - DECEMBER 1982 - REVISED FEBRUARY 2004

FUNCTION TABLE

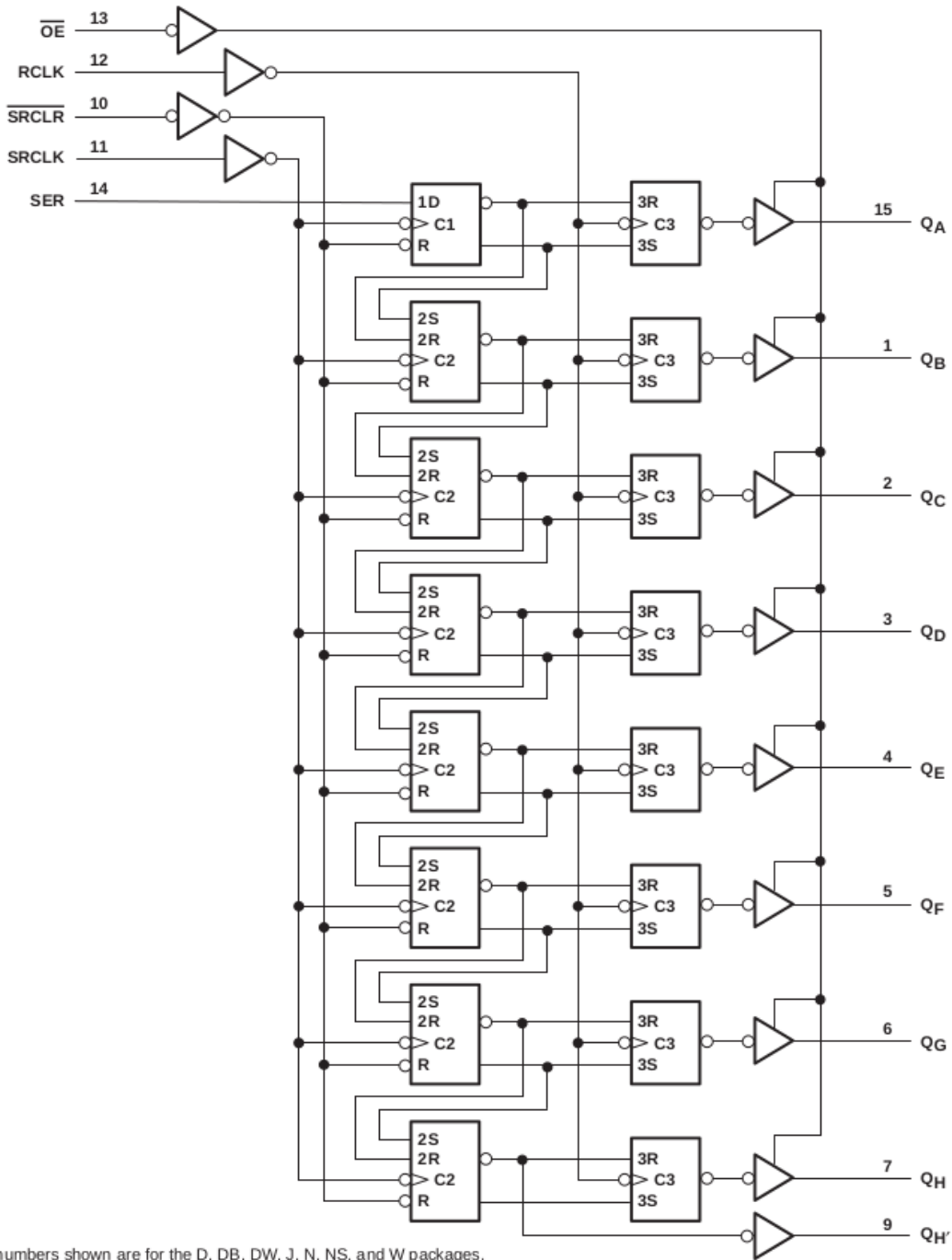
INPUTS					FUNCTION
SER	SRCLK	SRCLR	RCLK	OE	
X	X	X	X	H	Outputs Q _A -Q _H are disabled.
X	X	X	X	L	Outputs Q _A -Q _H are enabled.
X	X	L	X	X	Shift register is cleared.
L	↑	H	X	X	First stage of the shift register goes low. Other stages store the data of previous stage, respectively.
H	↑	H	X	X	First stage of the shift register goes high. Other stages store the data of previous stage, respectively.
X	X	X	↑	X	Shift-register data is stored in the storage register.



POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

SN54HC595, SN74HC595
8-BIT SHIFT REGISTERS
WITH 3-STATE OUTPUT REGISTERS
SCLS041G - DECEMBER 1982 - REVISED FEBRUARY 2004

logic diagram (positive logic)



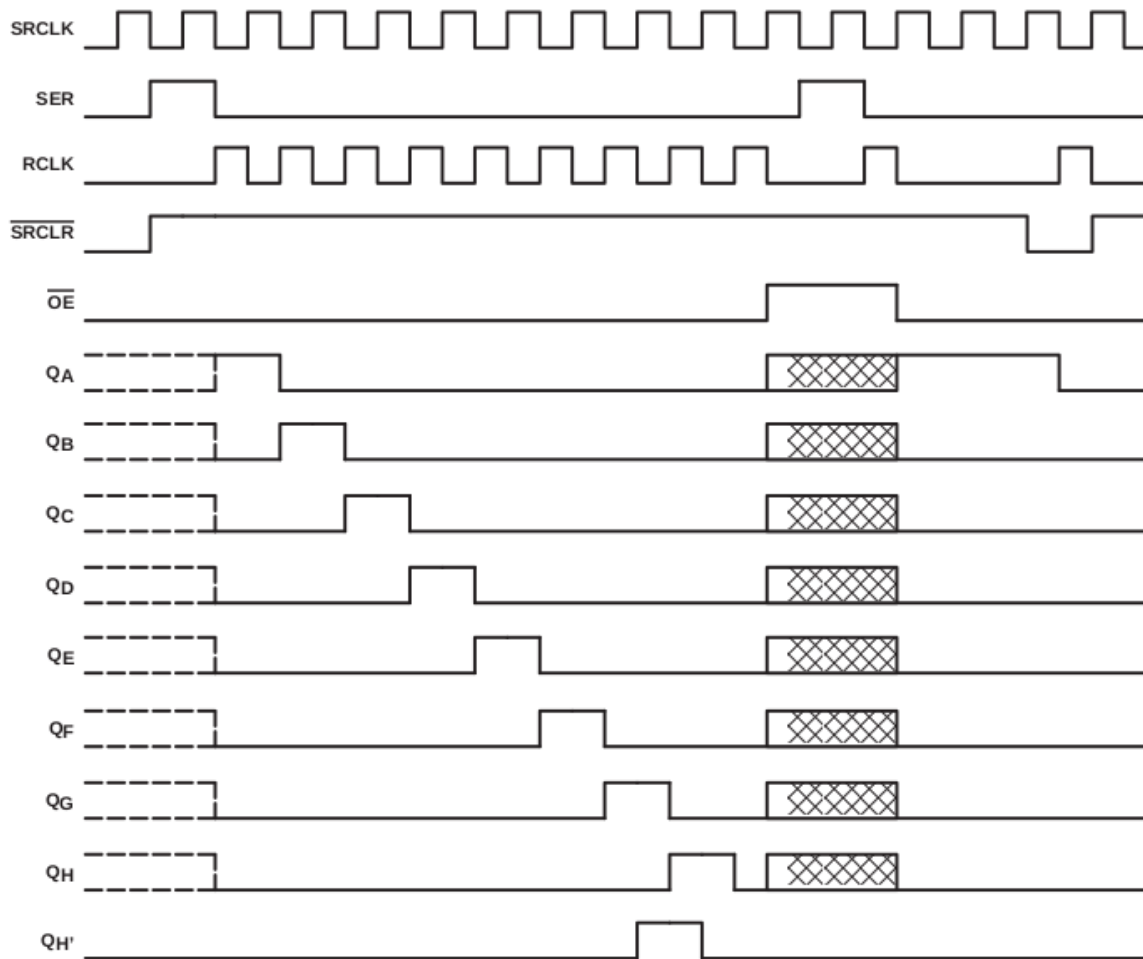
Pin numbers shown are for the D, DB, DW, J, N, NS, and W packages.




POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

SN54HC595, SN74HC595
8-BIT SHIFT REGISTERS
WITH 3-STATE OUTPUT REGISTERS
SCLS041G - DECEMBER 1982 - REVISED FEBRUARY 2004

timing diagram



NOTE:  implies that the output is in 3-State mode.

SN54HC595, SN74HC595
8-BIT SHIFT REGISTERS
WITH 3-STATE OUTPUT REGISTERS
SCLS041G – DECEMBER 1982 – REVISED FEBRUARY 2004

absolute maximum ratings over operating free-air temperature range (unless otherwise noted)[†]

Supply voltage range, V_{CC}	-0.5 V to 7 V
Input clamp current, I_{IK} ($V_I < 0$ or $V_I > V_{CC}$) (see Note 1)	±20 mA
Output clamp current, I_{OK} ($V_O < 0$ or $V_O > V_{CC}$) (see Note 1)	±20 mA
Continuous output current, I_O ($V_O = 0$ to V_{CC})	±35 mA
Continuous current through V_{CC} or GND	±70 mA
Package thermal impedance, θ_{JA} (see Note 2): D package	73°C/W
DB package	82°C/W
DW package	57°C/W
N package	67°C/W
NS package	64°C/W
Storage temperature range, T_{stg}	-65°C to 150°C

[†] Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

- NOTES: 1. The input and output voltage ratings may be exceeded if the input and output current ratings are observed.
 2. The package thermal impedance is calculated in accordance with JESD 51-7.

recommended operating conditions (see Note 3)

		SN54HC595			SN74HC595			UNIT
		MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage	2	5	6	2	5	6	V
V_{IH}	High-level input voltage	$V_{CC} = 2\text{ V}$		1.5	$V_{CC} = 2\text{ V}$		1.5	V
		$V_{CC} = 4.5\text{ V}$		3.15	$V_{CC} = 4.5\text{ V}$		3.15	
		$V_{CC} = 6\text{ V}$		4.2	$V_{CC} = 6\text{ V}$		4.2	
V_{IL}	Low-level input voltage	$V_{CC} = 2\text{ V}$			0.5	$V_{CC} = 2\text{ V}$		V
		$V_{CC} = 4.5\text{ V}$			1.35	$V_{CC} = 4.5\text{ V}$		
		$V_{CC} = 6\text{ V}$			1.8	$V_{CC} = 6\text{ V}$		
V_I	Input voltage	0		V_{CC}	0		V_{CC}	V
V_O	Output voltage	0		V_{CC}	0		V_{CC}	V
$\Delta t/\Delta v^{\ddagger}$	Input transition rise/fall time	$V_{CC} = 2\text{ V}$			1000	$V_{CC} = 2\text{ V}$		ns
		$V_{CC} = 4.5\text{ V}$			500	$V_{CC} = 4.5\text{ V}$		
		$V_{CC} = 6\text{ V}$			400	$V_{CC} = 6\text{ V}$		
T_A	Operating free-air temperature	-55		125	-40		85	°C

NOTE 3: All unused inputs of the device must be held at V_{CC} or GND to ensure proper device operation. Refer to the TI application report, *Implications of Slow or Floating CMOS Inputs*, literature number SCBA004.

[‡] If this device is used in the threshold region (from $V_{ILmax} = 0.5\text{ V}$ to $V_{IHmin} = 1.5\text{ V}$), there is a potential to go into the wrong state from induced grounding, causing double clocking. Operating with the inputs at $t_f = 1000\text{ ns}$ and $V_{CC} = 2\text{ V}$ does not damage the device; however, functionally, the CLK inputs are not ensured while in the shift, count, or toggle operating modes.



SN54HC595, SN74HC595
8-BIT SHIFT REGISTERS
WITH 3-STATE OUTPUT REGISTERS
SCLS041G - DECEMBER 1982 - REVISED FEBRUARY 2004

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS	V _{CC}	T _A = 25°C			SN54HC595		SN74HC595		UNIT
			MIN	TYP	MAX	MIN	MAX	MIN	MAX	
V _{OH}	V _I = V _{IH} or V _{IL}	I _{OH} = -20 μA	2 V	1.9	1.998	1.9	1.9	V		
			4.5 V	4.4	4.499	4.4	4.4			
			6 V	5.9	5.999	5.9	5.9			
		4.5 V	Q _H ', I _{OH} = -4 mA	3.98	4.3	3.7	3.84			
			Q _A -Q _H , I _{OH} = -6 mA	3.98	4.3	3.7	3.84			
			6 V	5.48	5.8	5.2	5.34			
Q _H ', I _{OH} = -5.2 mA	5.48	5.8	5.2	5.34						
Q _A -Q _H , I _{OH} = -7.8 mA	5.48	5.8	5.2	5.34						
V _{OL}	V _I = V _{IH} or V _{IL}	I _{OL} = 20 μA	2 V	0.002	0.1	0.1	0.1	V		
			4.5 V	0.001	0.1	0.1	0.1			
			6 V	0.001	0.1	0.1	0.1			
		4.5 V	Q _H ', I _{OL} = 4 mA	0.17	0.26	0.4	0.33			
			Q _A -Q _H , I _{OL} = 6 mA	0.17	0.26	0.4	0.33			
			6 V	0.15	0.26	0.4	0.33			
Q _H ', I _{OL} = 5.2 mA	0.15	0.26	0.4	0.33						
Q _A -Q _H , I _{OL} = 7.8 mA	0.15	0.26	0.4	0.33						
I _I	V _I = V _{CC} or 0	6 V	±0.1	±100	±1000	±1000	nA			
I _{OZ}	V _O = V _{CC} or 0, Q _A -Q _H	6 V	±0.01	±0.5	±10	±5	μA			
I _{CC}	V _I = V _{CC} or 0, I _O = 0	6 V		8	160	80	μA			
C _i		2 V to 6 V		3 10	10	10	pF			

SN54HC595, SN74HC595
8-BIT SHIFT REGISTERS
WITH 3-STATE OUTPUT REGISTERS
SCLS041G – DECEMBER 1982 – REVISED FEBRUARY 2004

timing requirements over recommended operating free-air temperature range (unless otherwise noted)

		VCC	T _A = 25°C		SN54HC595		SN74HC595		UNIT
			MIN	MAX	MIN	MAX	MIN	MAX	
f _{clock}	Clock frequency	2 V	6		4.2		5		MHz
		4.5 V	31		21		25		
		6 V	36		25		29		
t _w	SRCLK or RCLK high or low	2 V	80		120		100		ns
		4.5 V	16		24		20		
		6 V	14		20		17		
	SRCLR low	2 V	80		120		100		
		4.5 V	16		24		20		
		6 V	14		20		17		
t _{su}	SER before SRCLK↑	2 V	100		150		125		ns
		4.5 V	20		30		25		
		6 V	17		25		21		
	SRCLK↑ before RCLK↑	2 V	75		113		94		
		4.5 V	15		23		19		
		6 V	13		19		16		
	SRCLR low before RCLK↑	2 V	50		75		65		
		4.5 V	10		15		13		
		6 V	9		13		11		
	SRCLR high (inactive) before SRCLK↑	2 V	50		75		60		
		4.5 V	10		15		12		
		6 V	9		13		11		
t _h	Hold time, SER after SRCLK↑	2 V	0		0		0		ns
		4.5 V	0		0		0		
		6 V	0		0		0		

† This setup time allows the storage register to receive stable data from the shift register. The clocks can be tied together, in which case the shift register is one clock pulse ahead of the storage register.



SN54HC595, SN74HC595
8-BIT SHIFT REGISTERS
WITH 3-STATE OUTPUT REGISTERS
SCLS041G – DECEMBER 1982 – REVISED FEBRUARY 2004

switching characteristics over recommended operating free-air temperature range, $C_L = 50 \text{ pF}$ (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	VCC	T _A = 25°C			SN54HC595		SN74HC595		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
f _{max}			2 V	6	26		4.2		5	MHz	
			4.5 V	31	38		21		25		
			6 V	36	42		25		29		
t _{pd}	SRCLK	Q _{H'}	2 V		50	160		240		200	ns
			4.5 V		17	32		48		40	
			6 V		14	27		41		34	
	RCLK	Q _A -Q _H	2 V		50	150		225		187	
			4.5 V		17	30		45		37	
			6 V		14	26		38		32	
t _{PHL}	$\overline{\text{SRCLR}}$	Q _{H'}	2 V		51	175		261		219	ns
			4.5 V		18	35		52		44	
			6 V		15	30		44		37	
t _{en}	$\overline{\text{OE}}$	Q _A -Q _H	2 V		40	150		225		187	ns
			4.5 V		15	30		45		37	
			6 V		13	26		38		32	
t _{dis}	$\overline{\text{OE}}$	Q _A -Q _H	2 V		42	200		300		250	ns
			4.5 V		23	40		60		50	
			6 V		20	34		51		43	
t _t		Q _A -Q _H	2 V		28	60		90		75	ns
			4.5 V		8	12		18		15	
			6 V		6	10		15		13	
		Q _{H'}	2 V		28	75		110		95	
			4.5 V		8	15		22		19	
			6 V		6	13		19		16	

switching characteristics over recommended operating free-air temperature range, $C_L = 150 \text{ pF}$ (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	VCC	T _A = 25°C			SN54HC595		SN74HC595		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
t _{pd}	RCLK	Q _A -Q _H	2 V		60	200		300		250	ns
			4.5 V		22	40		60		50	
			6 V		19	34		51		43	
t _{en}	$\overline{\text{OE}}$	Q _A -Q _H	2 V		70	200		298		250	ns
			4.5 V		23	40		60		50	
			6 V		19	34		51		43	
t _t		Q _A -Q _H	2 V		45	210		315		265	ns
			4.5 V		17	42		63		53	
			6 V		13	36		53		45	

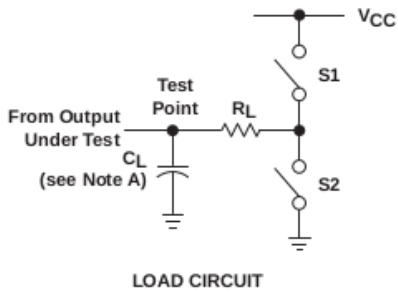
operating characteristics, T_A = 25°C

PARAMETER	TEST CONDITIONS	TYP	UNIT
C _{pd} Power dissipation capacitance	No load	400	pF

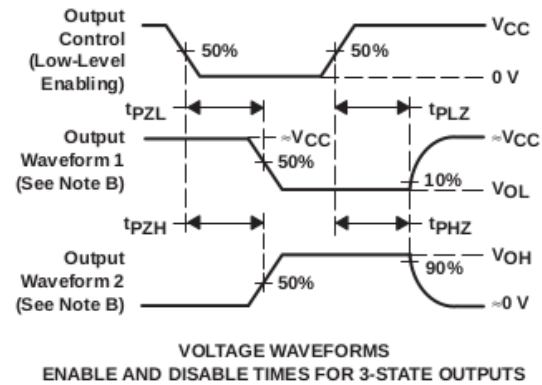
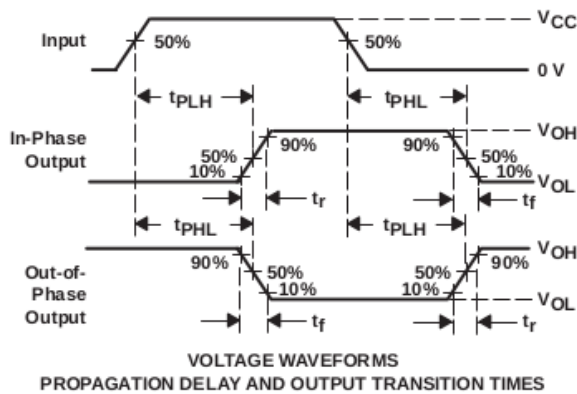
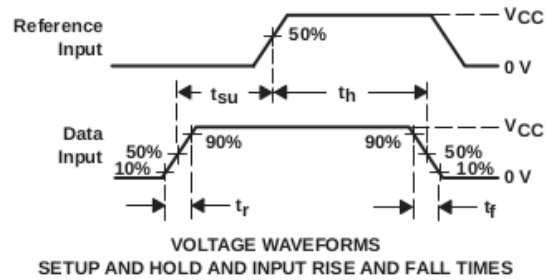
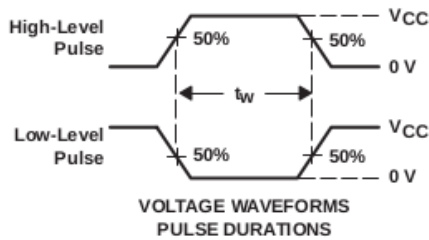


SN54HC595, SN74HC595
8-BIT SHIFT REGISTERS
WITH 3-STATE OUTPUT REGISTERS
SCLS041G - DECEMBER 1982 - REVISED FEBRUARY 2004

PARAMETER MEASUREMENT INFORMATION



PARAMETER	R_L	C_L	S1	S2	
t_{en}	t_{pZH}	1 k Ω	50 pF or 150 pF	Open	Closed
	t_{pZL}			Closed	Open
t_{dis}	t_{pHZ}	1 k Ω	50 pF	Open	Closed
	t_{pLZ}			Closed	Open
t_{pd} or t_t	—	50 pF or 150 pF	Open	Open	



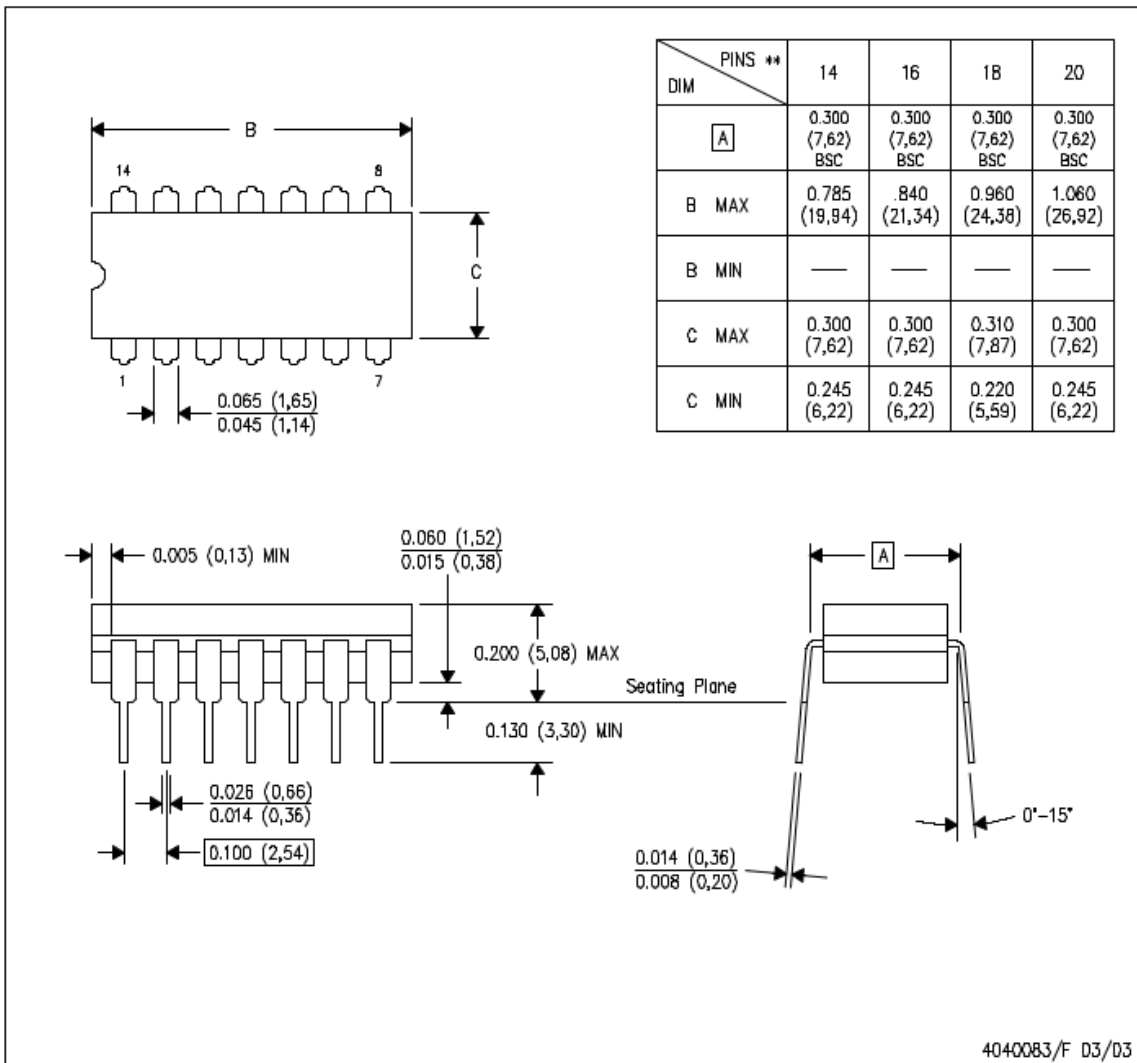
- NOTES:
- A. C_L includes probe and test-fixture capacitance.
 - B. Waveform 1 is for an output with internal conditions such that the output is low, except when disabled by the output control. Waveform 2 is for an output with internal conditions such that the output is high, except when disabled by the output control.
 - C. Phase relationships between waveforms were chosen arbitrarily. All input pulses are supplied by generators having the following characteristics: $PRR \leq 1$ MHz, $Z_O = 50 \Omega$, $t_r = 6$ ns, $t_f = 6$ ns.
 - D. For clock inputs, f_{max} is measured when the input duty cycle is 50%.
 - E. The outputs are measured one at a time, with one input transition per measurement.
 - F. t_{pLZ} and t_{pHZ} are the same as t_{dis} .
 - G. t_{pZL} and t_{pZH} are the same as t_{en} .
 - H. t_{PLH} and t_{PHL} are the same as t_{pd} .

Figure 1. Load Circuit and Voltage Waveforms

Systematic testing of digital hardware systems by means of test automaton

J (R-GDIP-T**)
14 LEADS SHOWN

CERAMIC DUAL IN-LINE PACKAGE



- NOTES:
- A. All linear dimensions are in inches (millimeters).
 - B. This drawing is subject to change without notice.
 - C. This package is hermetically sealed with a ceramic lid using glass frit.
 - D. Index point is provided on cap for terminal identification only on press ceramic glass frit seal only.
 - E. Falls within MIL STD 1835 GDIP1-T14, GDIP1-T16, GDIP1-T18 and GDIP1-T20.

6.8 Appendix H: Content of the CD

- Document of the Bachelor thesis.
- Source code of the TDL testbench
- Source code of the VHDL model
- Source code of the VHDL testbench
- ETSI-TDL standard
- Terasic DE0-Nano user manual
- Design of the game board

References

- [1] T. Krawutschke, G. Hartung, N. Kopshoff, M. Schulze, G. B. Faluwoye, and C. Hoffmann. *Test automation for reengineering modules using test description language and FPGA*. To be published in www.embedded-world.eu
- [2] David Lauber. *Konzept und Entwicklung einer Testbeschreibungssprache und eines Systems zur Generierung von Testbenches und Testvektoren für einen Testautomaten*. Master thesis. 18 June 2015.
- [3] ETSI standard 203 119-1. *Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 1: Abstract Syntax and Associated Semantics*. <http://www.etsi.org/technologies-clusters/technologies/test-description-language>
- [4] ETSI standard 203 119-2. *Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 2: Graphical Syntax*. <http://www.etsi.org/technologies-clusters/technologies/test-description-language>
- [5] ETSI standard 203 119-3. *Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 3: Exchange Format*. <http://www.etsi.org/technologies-clusters/technologies/test-description-language>
- [6] ETSI standard 203 119-4. *Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 4: Structured Test Objective Specification (Extension)*. <http://www.etsi.org/technologies-clusters/technologies/test-description-language>
- [7] Terasic DE0-Nano board user manual. http://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=593&FID=75023fa36c9bf8639384f942e65a46f3
- [8] TicTacToe. <https://en.wikipedia.org/wiki/Tic-tac-toe>