ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδίαση και υλοποίηση ενός λογικού προσομοιωτή, καθοδηγούμενου από γεγονότα για ψηφιακά κυκλώματα.

Design and implementation of an event-driven logic simulator for digital circuits.

Διπλωματική Εργασία

Βασίλειος Σ. Λουκάς

Επιβλέποντες Καθηγητές : Σταμούλης Γεώργιος
Καθηγητής Π.Θ.

Ευμορφόπουλος Νέστωρ
Επίκουρος Καθηγητής Π.Θ.

Βόλος, Ιούλιος 2017

UNIVERSITY OF THESSALY
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

Design and implementation of an event-driven logic simulator for digital circuits.

By Vasileios S. Loukas

Graduate Thesis for the degree of
Diploma of Science in Electrical & Computer Engineering

Approved by the two-member inquiry committee at
July 7th 2017

Head Supervisor	Second Supervisor
George Stamoulis	Nestor Eumorfopoulos

Διπλωματική Εργασία για την απόκτηση του Διπλώματος του του Ηλεκτρολόγου Μηχανικού και Μηχανικού Υπολογιστών, του Πανεπιστημίου Θεσσαλίας, στα πλαίσια του Προγράμματος Προπτυχιακών Σπουδών του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πανεπιστημίου Θεσσαλίας.

Βασίλειος Σ. Λουκάς
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών
Πανεπιστημίου Θεσσαλίας

To my family and friends
Στην οικογένεια μου και στους φίλους μου

This page is intentionally left blank.

Ευχαριστίες

Με την περάτωση της παρούσας διπλωματικής εργασίας ολοκληρώνεται ο προπτυχιακός κύκλος σπουδών μου. Θ α ήθ ελα λοιπόν, να ευχαριστήσω θερμά τους επιβλέποντες μου κ. Γεώργιο Σταμούλη και κ. Νέστωρα Ευμορφόπουλο, για την ε μπιστοσύνη που επέδειξαν στο πρόσωπό μου με την ανάθεση του συγκεκριμένου θέματος, την άριστη συνεργασία και την συνεχή καθ οδήγηση, η οποία διευκόλυνε την εκπόνηση της διπλωματικής εργασίας μου.

Επίσης, θα ήθελα να ευχαριστήσω τους φίλους και συνεργάτες μου του Εργαστηρίου Ε 5 για την υποστήριξη και την δημιουργία ενός ευχάριστου και δημιουργικού κλίματος και ιδιαίτερα τον Δημήτριο Γαρυφάλλου για τις εύστοχες υποδείξεις και την συνεχή στήριξή του.

Τέλος, οφείλω ένα μεγάλο ευχαριστώ στην οικογένειά μου και στους φίλους μου για την α μέριστη υποστήριξη και την ανεκτίμητη βοήθεια που μου παρείχαν κατά την διάρκεια των σπουδών μου.

Contents

Περίληψη

Καθώς η πολυπλοκότητα των very large-scale integrated (VLSI) κυκλωμάτων έχει αυξηθεί, προσομοιωτές έχουνε φτιαχτεί για να επιβεβαιώσουνε το σχεδιασμό. Διάφοροι τύποι και επίπεδα προσομοιωτών έχουν παραχθεί, με σκοπό να αντιμετωπιστούν διαφορετικές φάσεις της διαδικασίας της σχεδίασης. Σκοπός αυτής της εργασίας είναι, όπως δηλώθηκε, ο σχεδιασμός και η υλοποίηση ενός λογικού προσομοιωτή καθοδηγούμενου από γεγονότα για ψηφιακά κυκλώματα, σε C++. Πρωταρχικά, υπάρχουν μερικές αναφορές στη διαδικασία ελέγχου των VLSI, διαφορετικές φάσεις αυτού, με επίκεντρο τον έλεγχο του σχεδιασμού και ιδιαίτερα τον λειτουργικό έλεγχο, μέρος του οποίου αποτελούν οι προσομοιωτές. Παρατηρούμε τη διαφορετικότητα μεταξύ διαφόρων ειδών προσομ οιωτών, με βάση τη λειτουργία τους και το μοντέλο καθυστέρησης που υποστηρίζουν, εξηγώντας την σημαντικότητα των κύριων χαρακτηριστικών τους. Επιπλέον, υπάρχει μία ενδελεχής ανάλυση του θεωρητικού υπόβαθρου που χρειάζεται για να χτιστεί ένα τέτοιο εργαλείο καθώς και όλες οι πρακτικές πλη ροφορίες που επέτρεψαν την υλοποίηση να συμβεί. Παρουσιάζονται τα κύρια χαρακτηριστικά που αφορούν τη συντακτική ανάλυση αρχείων, την αποθήκευση σε κατάλληλες δομές δεδομένων και την έκφραση της ολότητας του κυκλώματος με ορθό τρόπο. Στη συνέχεια, ακριβώς πριν υλοποιήσουμε τον πραγματικό μηχανισμό πίσω από τον καθοδηγούμενο από τα γεγονότα προσομ οιωτή μας, αναλύουμε το βασικό μας μοντέλο για να επεξεργαζόμαστε γεγονότα σύμφωνα με το μοντέλο χρονισμού μοναδιαίας καθυστέρησης, το γνωστό και ως Timewheel, από μία θεωρητική πλευρά, και τέλος προχωρούμε σε περισσότερες αλγοριθμικές πληροφορίες όσον αφορά την υλοποίηση μας.

Abstract

As the complexity of very large-scale integrated (VLSI) circuits has increased, simulators have been built to verify design. Various types and levels of simulators have produced in order to cope with different phases of the design process. Aim of this thesis is, as stated, to design and implement an event-driven logic simulator, of unit delay, for digital circuits, in C++. Primarily, there are a few words about VLSI testing, different stages of it, focusing on design testing and specifically on functional testing, part of which are simulators. We observe the differentiation between various kinds of simulators, based on their functionality and delay model they support, explaining the importance of their major characteristics. Furthermore, one can find a thorough analysis of the theoretical background needed to build such a tool, as well as all the practical information that allowed the implementation to happen. Main features are presented concerning parsing from files, storing in appropriate data structures and expressing the entity of the circuit in a right way. Afterwards, just before implementing the actual mechanism behind the event-driven simulator, we take a close look to our basic model for processing events with respect to unit delay timing model, the commonly known as Timewheel, from a theoretical perspective, and then we proceed on more algorithmic information concerning our implementation.

Introduction

Ever since the early days of the electronic age, design verification has been an important part of the design process of digital circuits. The reason is simple. It is much more cost effective to verify accuracy of a design before manufacturing than to repair or rebuild thousands of erroneous circuits.

It seems like it was not too long ago, verification was carried out by constructing an actual prototype of the circuit from discrete components interconnected by external wires, although, it is absolutely not. The prototype was then used to evaluate the logical correctness and the timing characteristics of a design. This method was rendered infeasible by the explosive growth of the size of the digital devices. The number of components in a very large-scale integrated (VLSI) circuit can reach hundreds of thousands. The complexity of circuitry has also increased at the same time. It has become too costly and too time-consuming to build prototypes for VLSI circuits. These factors along with the rapid improvements in speed and size of computers and the rapid decrease in the cost of computing have ushered in the computer aided design (CAD) tools.

A CAD tool which has become a viable replacement for physical prototyping as a design verification tool is the simulator. A simulator allows a designer to simulate how a circuit under design would behave in reality, thus verifying design against the customer specifications. It allows the detection and measurement of events that may be very difficult or impossible to detect in the actual system. A simulator also enables a circuit designer to play "what if" during the design process to test different ideas and optimize the design.

The complexity of electronic devices has reached such a level that even in the field of simulation, no single simulator can handle all aspects of simulation for a complex circuit. As a result, different types of simulators have emerged to tackle different areas of simulation.

Testing in VLSI circuits

VLSI chip testing is done in several different places by several different types of people. When a new chip is designed and fabricated for the first time, testing should verify correctness of design and the test procedure. This often requires the involvement of the design engineer and the testing may even take place in the design laboratory rather than in a factory. Based on the result, both the design and the test procedure may be changed. This is called **verification testing.**
Successful verification testing usually results in some good chips. These are the earliest chips and are normally used by the designers of systems that will use this design. A successful verification also signals the beginning of production. Production means large scale manufacturing. Fabricated chips are tested in the factory.
This is called manufacturing testing. Finally, when the manufactured chips are received by a customer, they may be again tested to ensure quality. This testing, known as incoming inspection (or acceptance testing), is conducted either by the user or for the user by some independent testing house.
Testing typically consists of applying a set of test stimuli to the inputs of the circuit under test (CUT) while analysing the output responses. Circuits that produce the correct output responses for all input stimuli pass the test and are considered to be fault-free. Those circuits that fail to produce a correct response at any point during the test sequence are assumed to be faulty. Testing is performed at various stages in the lifecycle of a VLSI device, including during the VLSI development process, the electronic system manufacturing process, and, in some cases, system-level operation.

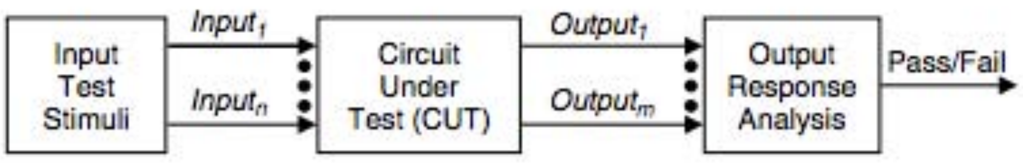


Figure 1 Illustrating Circuit Under Test

Stages of testing in VLSI lifecycle

The VLSI development process is illustrated where it can be seen that some form of testing is involved at each stage of the process. Based on a customer or project need, a VLSI device requirement is determined and formulated as a design specification. Designers are then responsible for synthesizing a circuit that satisfies the design specification and for verifying the design. **Design verification** is a predictive analysis that ensures that the synthesized design will perform the required functions when manufactured. When a design error is found, modifications to the design are necessary and design verification must be repeated. As a result, design verification can be considered as a form of testing.

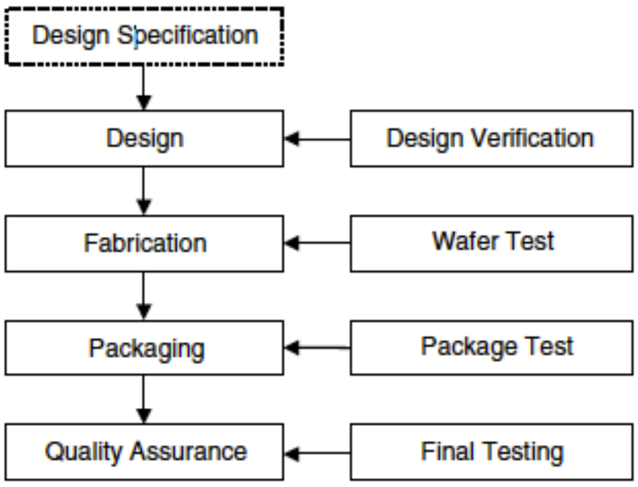


Figure 2 Different stages of testing

Functional Verification

In Electronic Design Automation, **functional verification** is the task of verifying that the logic design conforms to specification. In everyday terms, functional verification attempts to answer the question "Does this proposed design do what is intended?" This is a complex task, and takes the majority of time and effort in most large electronic system design projects. Functional verification is a part of more encompassing *design verification*, which, besides functional verification, considers non-functional aspects like timing, layout and power. Functional verification is very difficult because of the sheer volume of possible test cases that exist in even a simple design. Frequently there are more than 10^80 possible tests to comprehensively verify a design – a number that is impossible to achieve in a lifetime. This effort is equivalent to program verification, and is NP-hard or even worse – and no solution has been found that works well in all cases. However, it can be attacked by many methods. None of them are perfect, but each can be helpful in certain circumstances.

Simulators

Simulation is a powerful set of techniques that are used heavily in digital circuit verification, test development, design debug, and diagnosis.
A simulator is a collection of hardware and software systems which are used to mimic the behaviour of some entity or phenomenon. Simulators may be used to analyse and verify theoretical models which may be too difficult to grasp from a purely conceptual level. Such phenomenon range from examination of black holes to the study of highly abstract models of computation. As such, simulators provide a crucial role in both industry and academia.

Advantages and Purposes of using simulators

One of the primary advantages of simulators is that they are able to provide users with practical feedback when designing real world systems. This allows the designer to determine the correctness and efficiency of a design before the system is actually constructed. Consequently, the user may explore the merits of alternative designs without actually physically building the systems. By investigating the effects of specific design decisions during the design phase rather than the construction phase, the overall cost of building the system diminishes significantly. As an example, consider the design and fabrication of integrated circuits. During the design phase, the designer is presented with a myriad of decisions regarding such things as the placement of components and the routing of the connecting wires. It would be very costly to actually fabricate all of the potential designs as a means of evaluating their respective performance.
Through the use of a simulator, however, the user may investigate the relative superiority of each design without actually fabricating the circuits themselves. By imitating the behaviour of the designs, the circuit simulator is able to provide the designer with information pertaining to the correctness and efficiency of alternate designs. After

carefully weighing the ramifications of each design, the best circuit may then be fabricated.
Another benefit of simulators is that they permit system designers to study a problem at several different levels of abstraction. By approaching a system at a higher level of abstraction, the designer is better able to understand the behaviours and interactions of all the high-level components within the system and is therefore better equipped to counteract the complexity of the overall system. This complexity may simply overwhelm the designer if the problem had been approached from a lower level. As the designer better understands the operation of the higher-level components through the use of the simulator, the lower level components may then be designed and subsequently simulated for verification and performance evaluation. The entire system may be built based upon this ``top-down'' technique. This approach is often referred to as *hierarchical decomposition* and is essential in any design tool and simulator which deals with the construction of complex systems. For example, with respect to circuits, it is often useful to think of a microprocessor in terms of its registers, arithmetic logic units, multiplexors and control units. A simulator which permits the construction, interconnection and subsequent simulation of these higher-level entities is much more useful than a simulator which only lets the designer build and connect simple logic gates. Working at a higher-level abstraction also facilitates *rapid prototyping* in which preliminary systems are designed quickly for the purpose of studying the feasibility and practicality of the high-level design.
Thirdly, simulators can be used as an effective means for teaching or demonstrating concepts to students. This is particularly true of simulators that make intelligent use of computer graphics and animation. Such simulators dynamically show the behaviour and relationship of all the simulated system's components, thereby providing the user with a meaningful understanding of the system's nature. Consider again, for example, a circuit simulator. By showing the paths taken by signals as inputs are consumed by components and outputs are produced over their respective fanout, the student can actually see what is happening within the circuit and is therefore left

with a better understanding for the dynamics of the circuit. Such a simulator should also permit students to speed up, slow down, stop or even reverse a simulation as a means of aiding understanding. This is particularly true when simulating circuits which contain feedback loops or other operations which are not immediately intuitive upon an initial investigation.

Simulation Hierarchy

The design process is essentially a process of transforming a higher-level description of a design to a lower level description.

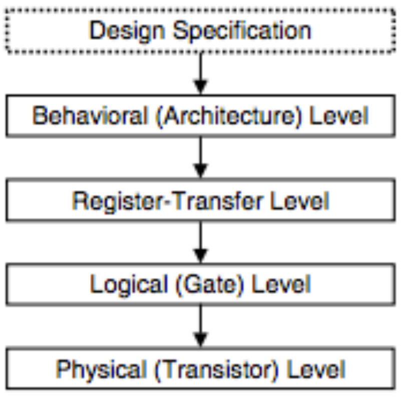


Figure 3 Different levels of simulation

Digital systems can be described at levels of abstraction ranging from behavioural to geometrical. Simulation capability exists at all of these levels.

The **behavioural description** is the highest level of abstraction. At this level, a system is described in terms of the algorithms that it performs, rather than how it is constructed. The development of a large system may begin by characterizing its behaviour at the behavioural level, particularly if it is a "first of a kind". A goal of behavioural simulations is to reveal conceptual flaws. When simulating behaviourally, the user is interested in determining things like optimum instruction set mix.

Once the system has been specified, a **register transfer level (RTL) model**, sometimes referred to as a functional model, can be used to describe the flow of data and control signals within and between functional units. The circuit is described in terms of flip-flops, registers, multiplexers, counters, arithmetic logic units (ALUs), encoders, decoders, and elements of similar level of complexity. Starting from a design specification, a behavioural (architecture) level description is developed in Verilog or as a C program and simulated to determine if it is functionally equivalent to the specification. The design is then described at the register-transfer level (RTL), which contains more structural information in terms of the sequential and combinational logic functions to be performed in the data paths and control circuits. The RTL description must be verified with respect to the functionality of the behavioural description before proceeding with synthesis to the logical level. A logical-level implementation is automatically synthesized from the RTL description to produce the gate-level design of the circuit.
A **logic model** describes a system by means of switching elements or gates. At this level, the designer is interested in correctness of designs intended to implement functional building blocks and units. Performance or timing of the design is a concern at this level. The logical-level implementation should be verified in as much detail as possible to guarantee the correct functionality of the final design.
In the final step, the logical-level description must be transformed to a physical-level description in order to obtain the physical placement and interconnection of the transistors in the VLSI device prior to fabrication.
A **circuit level model** is used on individual gate and functional level devices to verify their behaviour. It describes a circuit in terms of devices such as resistors, capacitors, and current sources. The simulation user is interested in knowing what kind of switching speeds, voltages, and noise margins to expect.
Finally, the **geometrical level model** describes a circuit in terms of physical shapes.

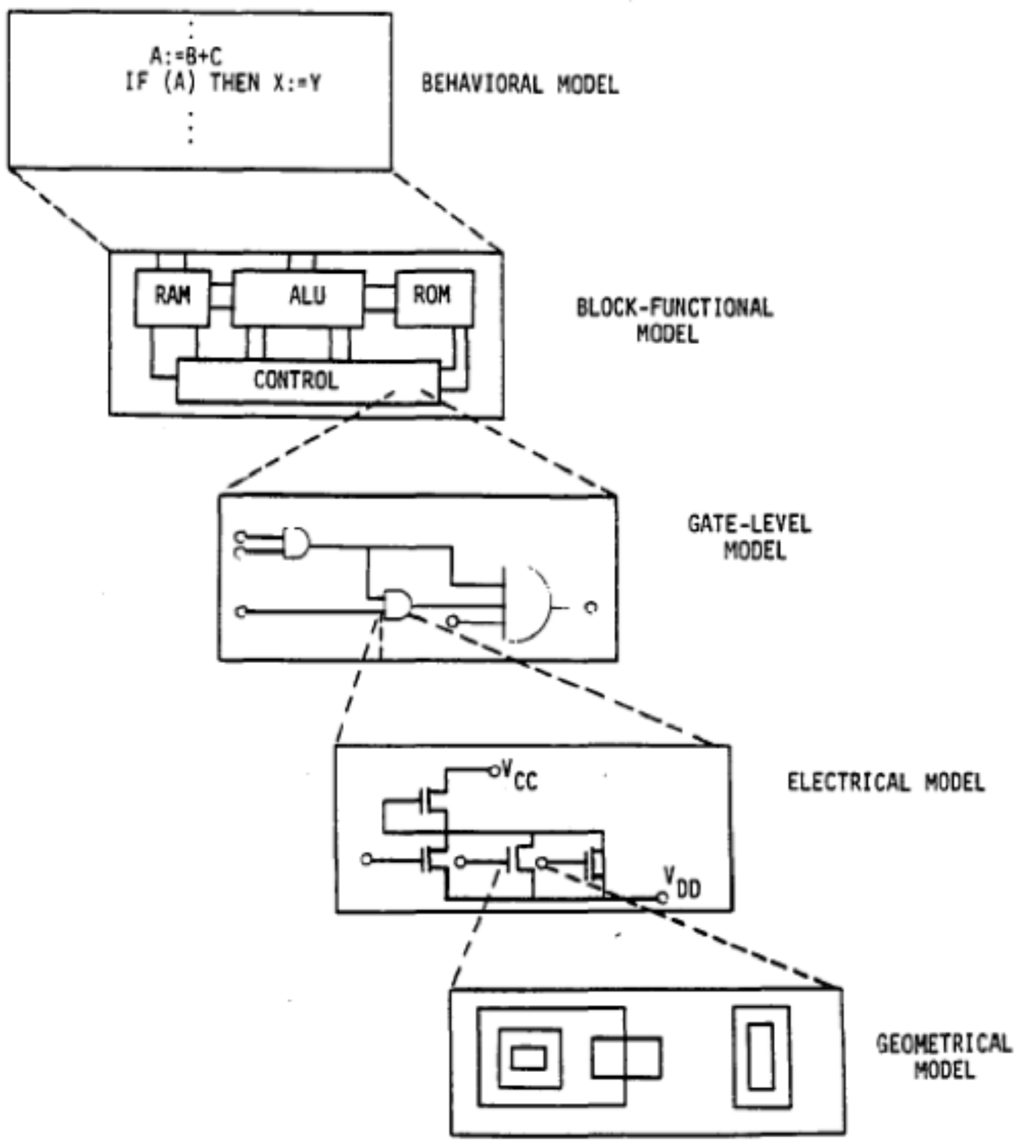


Figure 4 Hierarchy of Digital Circuit Models

In a top down design, the process starts with the system specification at the behavioral level and goes through a series of decomposition until reaching the physical layout at the geometrical level. Simulators are built throughout this process of decomposition.

Logic Simulation

In this section, we will discuss two commonly used gate-level logic simulation methodologies. We can classify simulators according to the type of internal model they process. With this classification, simulators can be divided into two major types; compiled simulators and event-driven simulators.
Compilers for programming languages can be characterized as compiled or interpreted. Simulators are similarly characterized as compiled or event-driven. The compiled simulator is created by converting a netlist directly into a series of machine language instructions that reflect the functions and interconnections of the individual elements of the circuit. For each logic element, there exists a series of one or more machine language instructions and a corresponding entry in a circuit value table that holds the current value for that element. The event-driven simulator, sometimes called table-driven, operates on a circuit description contained in a set of tables, without first converting the network into a machine language image.

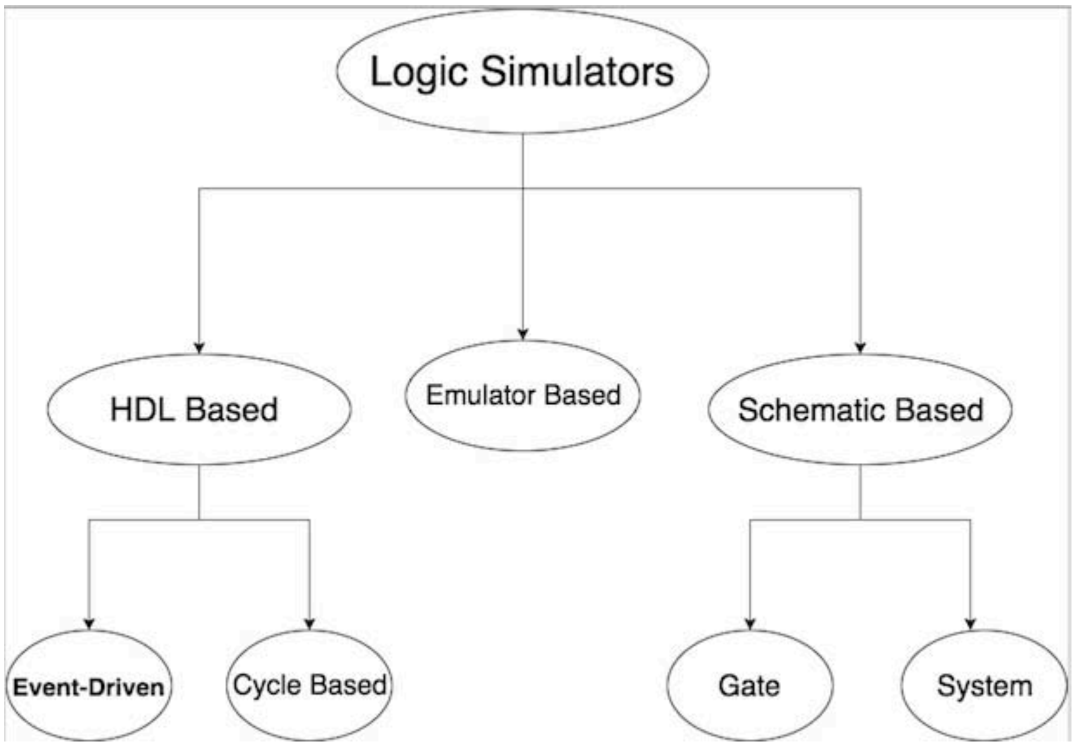


Figure 5 Classification of Logic Simulators

Compiled Code Simulators

Compiled-code simulators are very effective where two-state (0,1) simulation suffices. Otherwise, the larger number of signal states makes the code complex. The two-state simulation is useful for combinational logic and for sequential logic which is already initialized. Another scenario is the high-level design verification where the circuit is described using functional modules and stimuli are generated by testbenches. Because the code execution can be very fast on a computer, such simulators are capable of high speed. Large gate-level circuits present several problems. First, timing problems glitches, race conditions, etc. – are not modelled in a compiled-code simulator. The simulator does detect oscillations when the iterations do not converge, but it is difficult to deal with the situation unless the unknown (X) state is available. The second problem is that of the inefficiency incurred by the evaluation of the entire code when only a few signals may be changing. In digital circuits, generally only 1-10% of signals are found to change at any time. For this reason, event-driven simulators run much faster at the gate-level. Finally, any detailed timing (such as multiple-delay or minmax-delay) is almost impossible to simulate in the compiled-code. For these reasons, the use of compiled-code simulators is usually limited to high-level design verification.

Event-Driven Simulators

Event-driven simulation is a very effective procedure for discrete-event simulation. It is based on the recognition that any signal change (event) must have a cause, which is also an event. Thus, an event causes new events, which in turn may cause more events. An event-driven simulator follows the path of events. Consider a circuit at the gate-level. Suppose, all signals are in steady-state when a new vector is applied to primary inputs. Some inputs change, causing events on those input signals. Gates whose inputs now have events are called active and are placed in an activity list. The simulation proceeds by removing a gate from the activity list and evaluating it to determine whether its output has an event. A changing output makes all fanout gates active, which are then added to the activity list. The process of evaluation stops when the activity list becomes empty. An event-driven simulator only does the necessary amount of work. For logic circuits, in which typically very few signals change at a time, this can result in significant savings of computing effort. However, the biggest advantage of this technique is in its ability to simulate any arbitrary delays. This is done by a procedure known as event scheduling. Suppose the evaluation of an active gate generates an event at its output. If the gate has a switching delay of units, then the event should take effect time units later. For correctly considering the effects of delays, the simulator distributes the activity list in time. Event scheduling is the procedure of distributing the activity caused by events over time according to the specified delays.

Suitability

Compiled-code and event-driven simulation each have their advantages and disadvantages. Compiled-code simulation is good for cycle-based simulation, where only the circuit behaviour at the end of each clock cycle is of interest and zero-delay simulation can be used. Compiled-code simulation is also good for hiding the details of a simulation model, such as a processor core. Compiled-code simulation is also good when the circuit activity is high or when bitwise parallel simulation is used. The overhead of compilation restricts compiled-code simulation to applications where a large number of input vectors will be simulated. Event-driven simulation is the best approach for implementing general delay models, and detecting hazards. It is also the best approach for circuits with low activity, such as low-power circuits
that employ clock gating. Event-driven simulation is also the best approach during circuit debug, when frequent edit-simulate-debug cycles occur and simulation start-up time is important.

Delay Model support

Event-driven simulation can be performed in either a **zero** or a **nominal** or **unit delay** environment. A zero-delay simulator ignores delay values within a logic element; it simply calculates the logic function performed by the element. A nominal-delay simulator assigns delay values to logic elements based on manufacturer's recommendations or measurements with precision instruments. Some simulators, trying to strike a balance between the two, perform a unit-delay simulation in which each logic element is assigned a fixed delay, and since the elements are all assigned the same delay, the value 1 (unit delay) is as good as any other. The nominal delay simulator can give precise results but at a cost in CPU time. The zero-delay simulator usually runs faster but does not indicate when events occur, so races and hazards can present problems. The unit-delay simulator lies between the other two in range of performance. It records time units during simulation, so it requires more computations than zero-delay simulation, but the mechanism for scheduling events is simpler than for time based simulation. However, regarding all element delays as being equal can produce inaccurate results in timing sensitive circuits and may give the user a false sense of security. Unit delay simulation in sequential circuits does, however, have the advantage that time advances; so, if oscillations occur, they will eventually reach the end of the clock period and be detected without a need for additional code dedicated to oscillation detection.

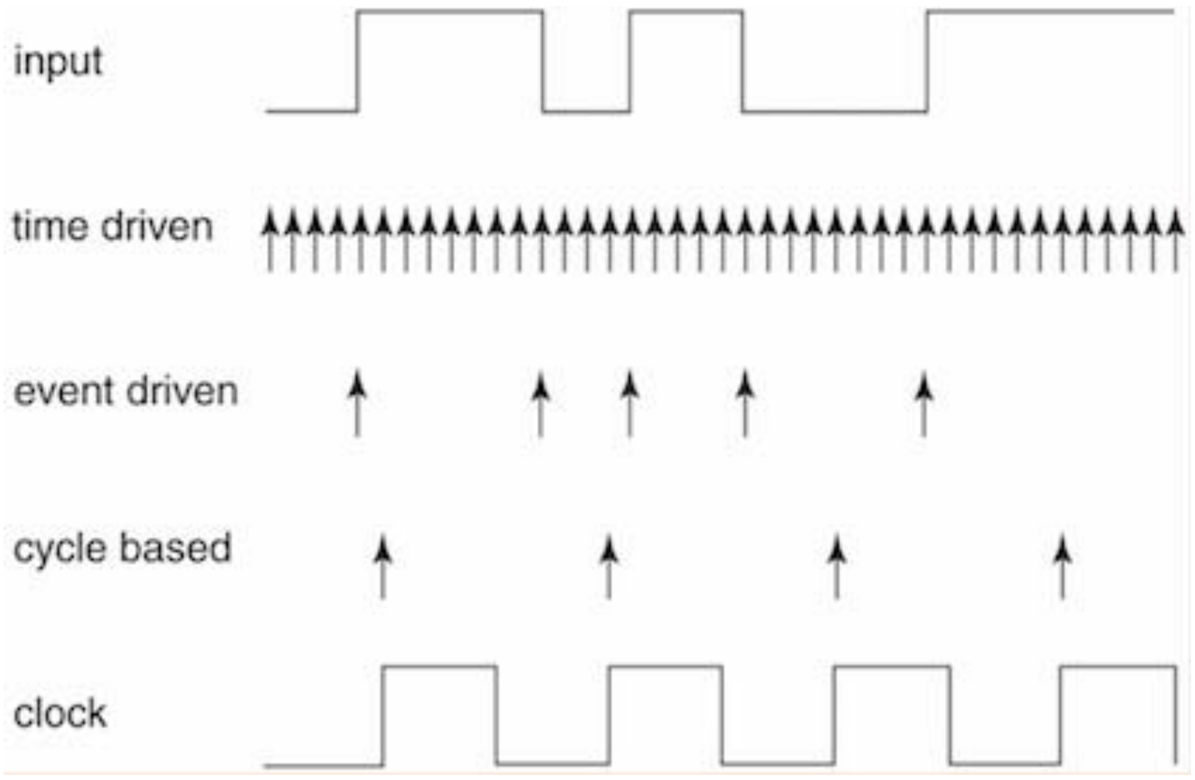


Figure 6 Spectrum of events based on the delay model applied

Software Architecture

Design background

The inputs of the simulator consist of the design's external topology or cell connectivity information (.v file), the cell's internal information such as internal connectivity and logic function (.lib file), and value changes for specified variables or all variables (.vcd file). Basic software architecture and data structures are based on CCSOpt, which is a stand-alone cell resizing tool.

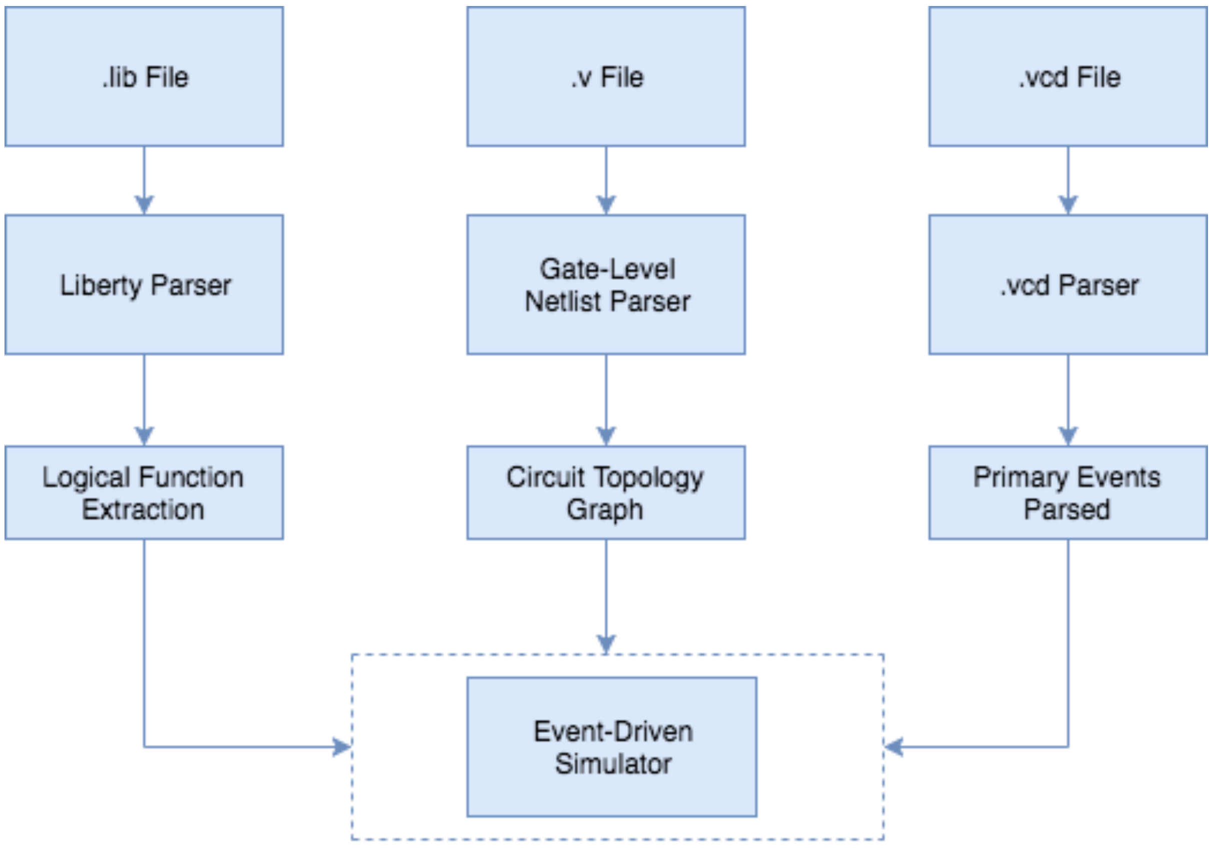


Figure 7 The Software Architecture

Files parsed

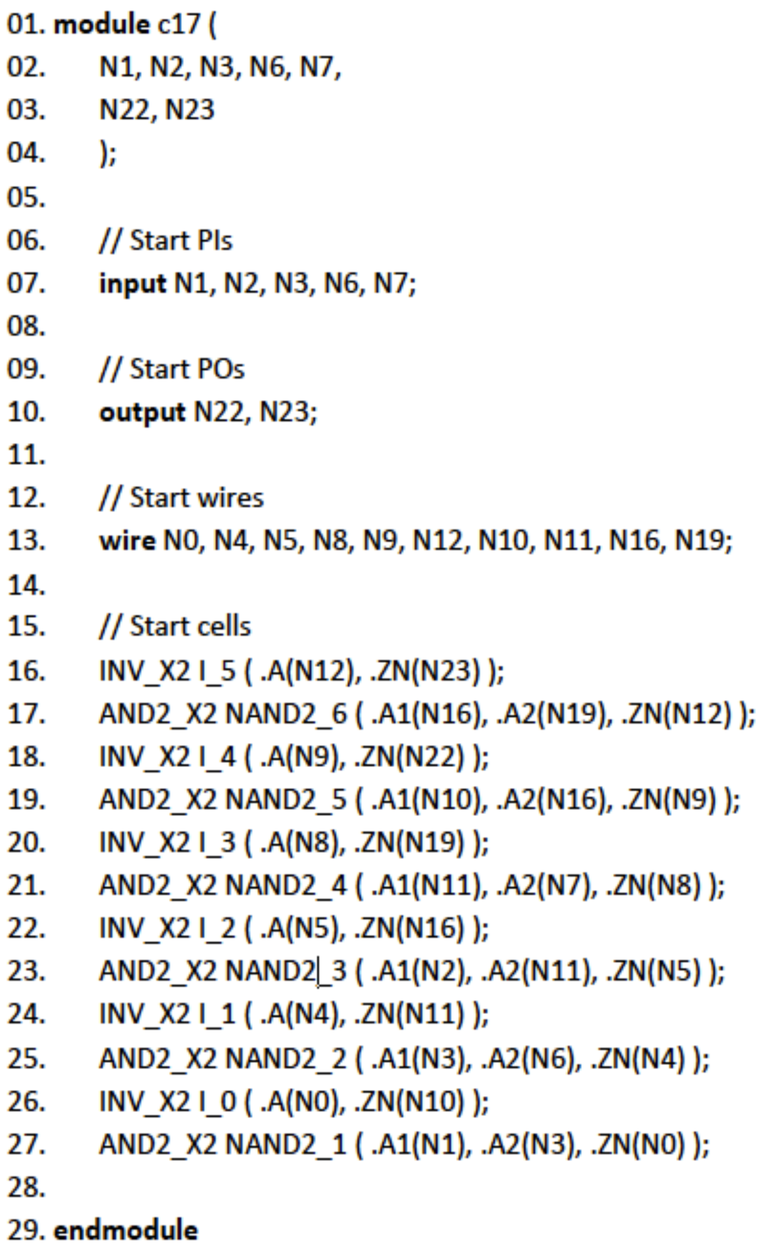
Verilog (.v) File

The Verilog file specifies the top-level hierarchy of the design. For this thesis, we will be using a small set of keywords with the Verilog language. Our Verilog parser supports the set of keywords found within the simple.v file (reproduced below for clarity). It also supports comments that start with '//'. The expected syntax is shown below.

```
module <circuit name> (
        <input 1>,
        …,
        <input n>,
        <output 1>,
        …,
        <output m> );

        input <input 1>;
        …
        input <input n>;
        output <output 1>;
        …
        output <output m>;

        // begin wire definitions
        wire <wire 1>;
        // end wire definitions

        // begin cell definitions
        <cell type> <cell instance name> (.<pin name> (<net name) );
        // end cell definitions
endmodule
```

Figure 8 Expected Syntax for .v files

The expected structure of the Verilog file is to start with the module declaration, defining the interface of the module with name <circuit name>. The inputs and output pins are explicitly declared; the initial wires are optionally declared with the keyword wire. For each cell definition, every <cell type> (.<pin name>) should be a specified cell type (pin) in the library file, and every <cell instance name> and <net name> should be found in the design specification. Each field is considered a string.



```
module c17 (
    N1, N2, N3, N6, N7,
    N22, N23
    );

    // Start PIs
    input N1, N2, N3, N6, N7;

    // Start POs
    output N22, N23;

    // Start wires
    wire N0, N4, N5, N8, N9, N12, N10, N11, N16, N19;

    // Start cells
    INV_X2 I_5 ( .A(N12), .ZN(N23) );
    AND2_X2 NAND2_6 ( .A1(N16), .A2(N19), .ZN(N12) );
    INV_X2 I_4 ( .A(N9), .ZN(N22) );
    AND2_X2 NAND2_5 ( .A1(N10), .A2(N16), .ZN(N9) );
    INV_X2 I_3 ( .A(N8), .ZN(N19) );
    AND2_X2 NAND2_4 ( .A1(N11), .A2(N7), .ZN(N8) );
    INV_X2 I_2 ( .A(N5), .ZN(N16) );
    AND2_X2 NAND2_3 ( .A1(N2), .A2(N11), .ZN(N5) );
    INV_X2 I_1 ( .A(N4), .ZN(N11) );
    AND2_X2 NAND2_2 ( .A1(N3), .A2(N6), .ZN(N4) );
    INV_X2 I_0 ( .A(N0), .ZN(N10) );
    AND2_X2 NAND2_1 ( .A1(N1), .A2(N3), .ZN(N0) );

endmodule
```

Figure 9 More specific example of a .v file

Consider the example given above.
Lines 01 and 29 define the start and end of the specified design with the keywords module and endmodule. Lines 01-04 specify the input and output connection names of the module (note that the direction is not specified here). Line 07 specifies the primary inputs (PIs) of the module with the keyword input. These names must match the ones started with module (lines 01-04). Line 10 specifies the primary output (PO) of the module with the keyword output. This name must match the one stated with the module (lines 01-04). Line 13 specifies the connections or nets within the module with the keyword wire. These connections specify both the external PIs and POs as well as the internal connections between gates (explained further after lines 16-27). Lines 17-27 specify the cells used in the design, as well as how the cells are connected. For example, on line 16, an INV_X2-type cell instance of I_5 is specified, it's A pin is fed by primary input N12, and its ZN pin feeds the primary output N23. On line 27, N1 feeds the A1 pin of the AND2_X2-type cell instance NAND2_1. Line 29 terminates the module definition.

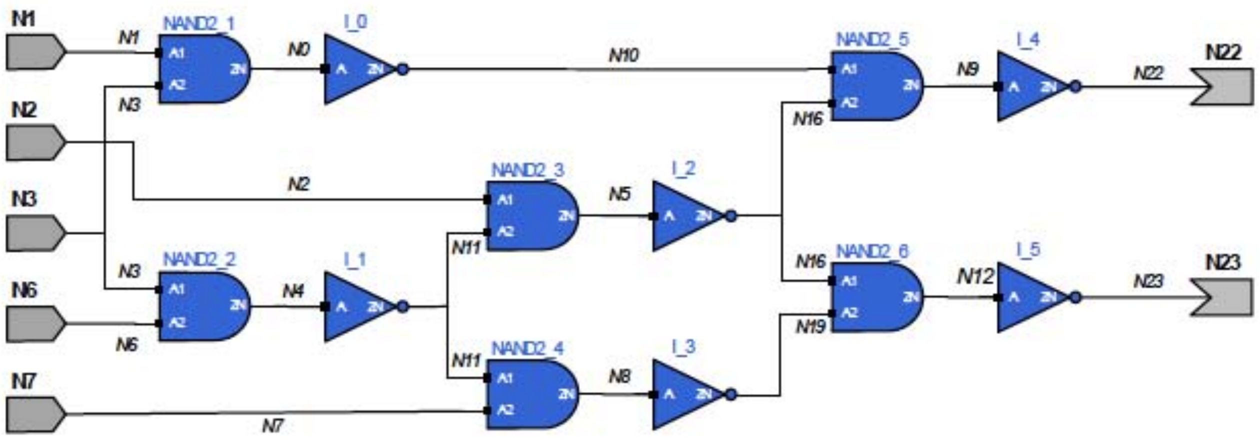
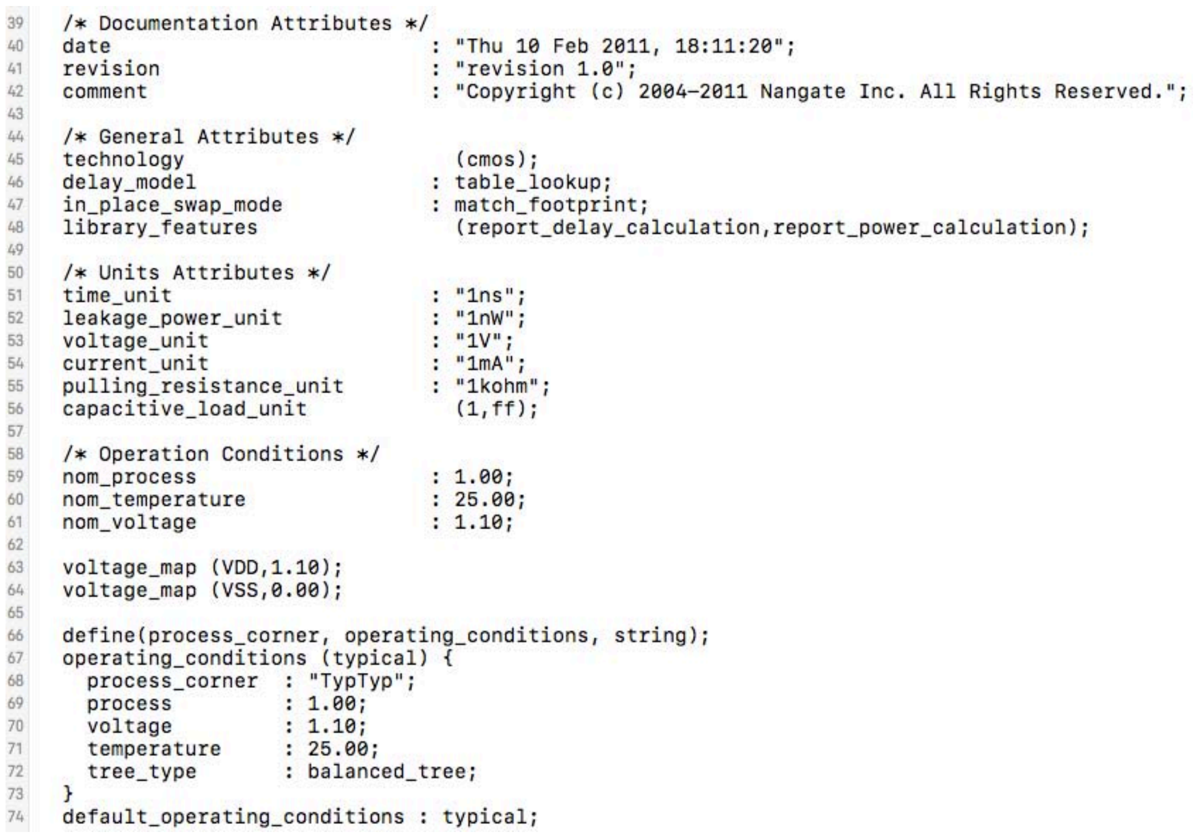


Figure 10 Internal Representation

Library (.lib) File

This file contains the set of all cells or gates that are available to the design. All cell instances found in the .v file will have corresponding cell type that is located in this file. Gate-level delay and output slew calculations will use the relevant timing information found for the appropriate cell type. For this thesis, we will be using the NanGate 45nm Open Cell Library and the Open Source Liberty parser. The parser supports the full logical (.lib) set of constructs including Composite Current Source (CCS) Modelling Technology, and noise, plus syntax, and common semantic checks.
The relevant portions of the .lib file are explained below. The library consists of (i) a header, (ii) a set of lookup-table definitions, and (iii) a set of cell definitions, where a cell will be a combinational element (e.g., NAND2) or a sequential element (e.g., flip-flop DFF). While there are many keywords available, this thesis will only use the following set.

HEADER. The header sets the general information about the library, and is defined in the NanGate 45nm Open Cell Library with the following format:

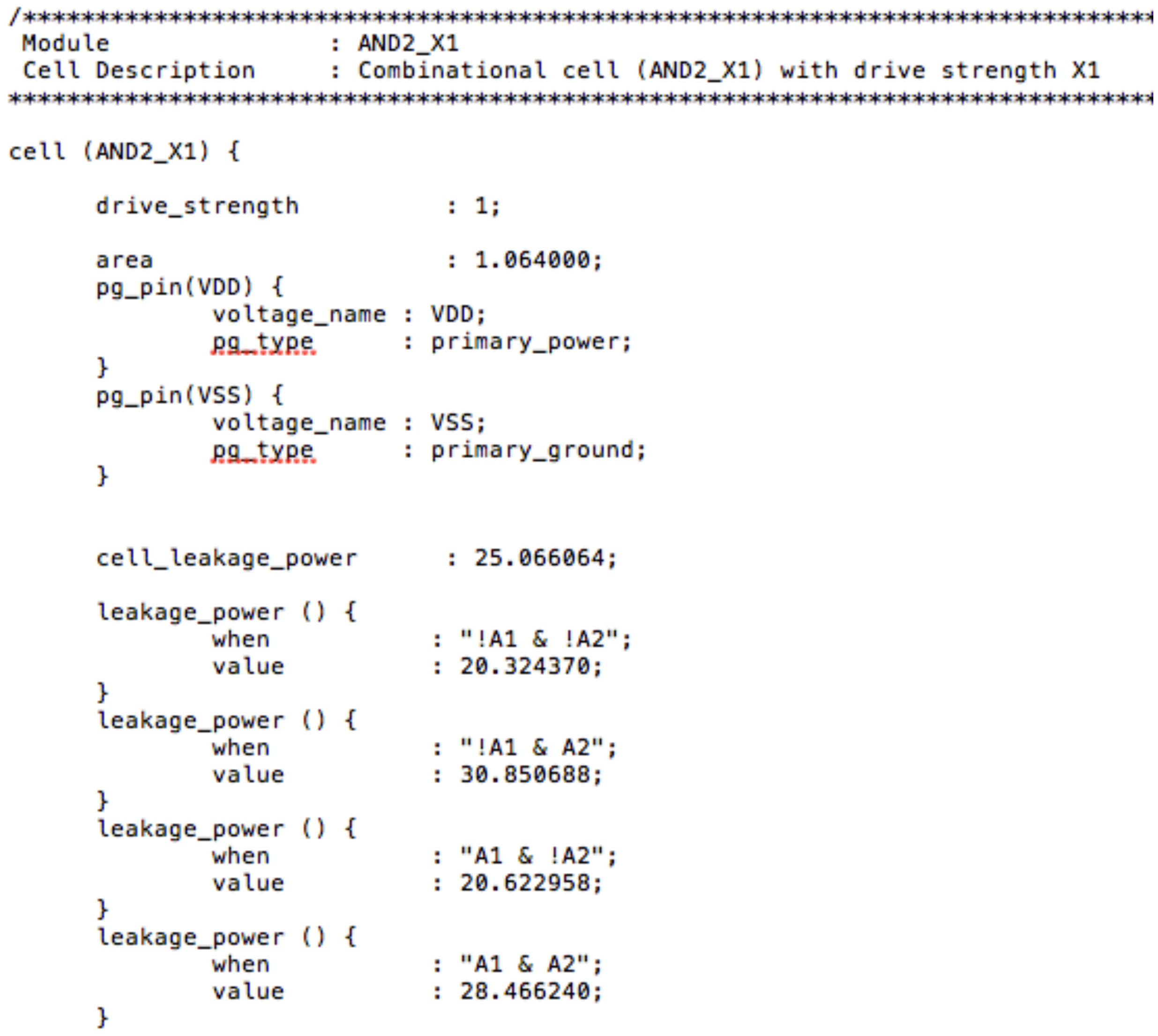


```
/* Documentation Attributes */
date                          : "Thu 10 Feb 2011, 18:11:20";
revision                      : "revision 1.0";
comment                       : "Copyright (c) 2004-2011 Nangate Inc. All Rights Reserved.";

/* General Attributes */
technology                      (cmos);
delay_model                   : table_lookup;
in_place_swap_mode            : match_footprint;
library_features                (report_delay_calculation,report_power_calculation);

/* Units Attributes */
time_unit                     : "1ns";
leakage_power_unit            : "1nW";
voltage_unit                  : "1V";
current_unit                  : "1mA";
pulling_resistance_unit       : "1kohm";
capacitive_load_unit            (1,ff);

/* Operation Conditions */
nom_process                   : 1.00;
nom_temperature               : 25.00;
nom_voltage                   : 1.10;

voltage_map (VDD,1.10);
voltage_map (VSS,0.00);

define(process_corner, operating_conditions, string);
operating_conditions (typical) {
  process_corner  : "TypTyp";
  process         : 1.00;
  voltage         : 1.10;
  temperature     : 25.00;
  tree_type       : balanced_tree;
}
default_operating_conditions : typical;
```

Figure 11 Header section in .lib file

LOOKUP TABLES. Most of the cell libraries include table models to specify the delays and timing checks for various timing arcs of the cell. The table models are referred to as NLDM (Non-Linear Delay Model) and are used for delay, output slew, or other timing checks. The table models capture the delay through the cell for various combinations of input transition time at the cell input pin and total output capacitance at the cell output.

CELL DEFINITIONS. A cell specifies a gate that could be used as part of a design, e.g., combinational gate NAND2 and flip-flop DFF. Its relevant specified syntax in the .lib format is:

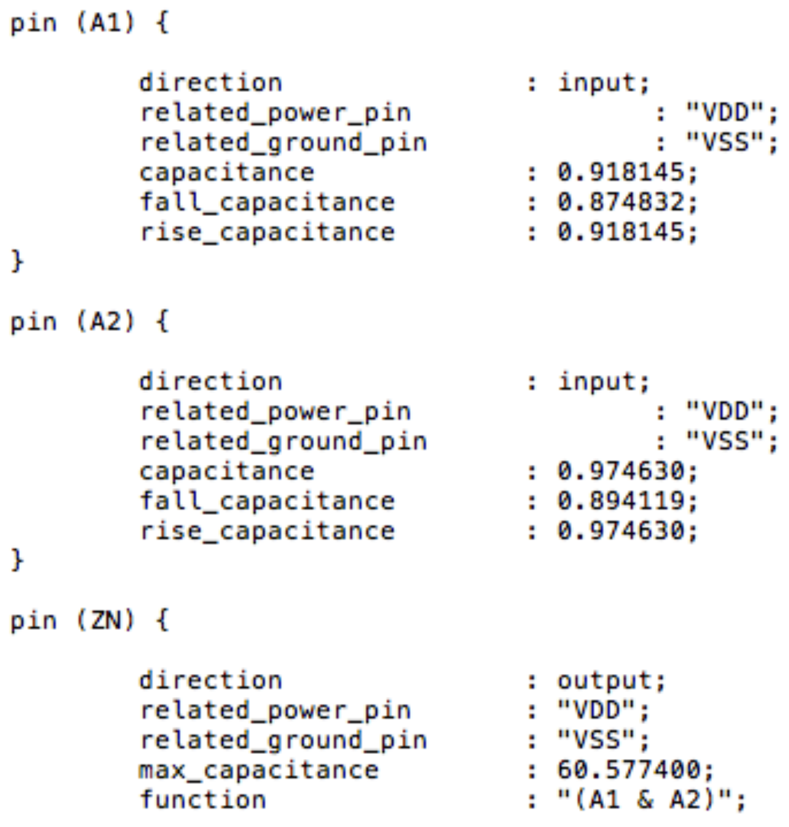


```
/****************************************************************************
 Module                  : AND2_X1
 Cell Description        : Combinational cell (AND2_X1) with drive strength X1
*****************************************************************************

cell (AND2_X1) {

      drive_strength           : 1;

      area                     : 1.064000;
      pg_pin(VDD) {
              voltage_name : VDD;
              pg_type      : primary_power;
      }
      pg_pin(VSS) {
              voltage_name : VSS;
              pg_type      : primary_ground;
      }


      cell_leakage_power       : 25.066064;

      leakage_power () {
              when            : "!A1 & !A2";
              value           : 20.324370;
      }
      leakage_power () {
              when            : "!A1 & A2";
              value           : 30.850688;
      }
      leakage_power () {
              when            : "A1 & !A2";
              value           : 20.622958;
      }
      leakage_power () {
              when            : "A1 & A2";
              value           : 28.466240;
      }
```

Figure 12 Cell Description of .lib file

In a cell, multiple pins can be defined, e.g., a standard NAND2 will have 3 pins – two inputs and one output. For each pin, the direction field indicates the type of pin: (i) input, (ii) output, or (iii) internal. The capacitance, max capacitance, and min capacitance fields specify the respective pin capacitance, maximum and minimum expected pin

loads. A timing( ) definition creates a timing arc (directed pin-to-pin) inside a cell.
Combinational timing arcs. Combinational arcs propagate delay and output slew from a source pin to a sink pin. They are found in common combinational logic gates, e.g., NAND2 or as a clock-trigger segment in flip-flops.
Below, we can observe how the cell definition part looks like, giving emphasis on the right illustration, as it contains the logic function that characterises each gate. This very own function that we extract for each gate, parse it in order to obtain a literal logic function, and route successfully every suitable net/edge as input, so as we can calculate the right output every time, and finally create a new event.



```
pin (A1) {

        direction               : input;
        related_power_pin               : "VDD";
        related_ground_pin              : "VSS";
        capacitance             : 0.918145;
        fall_capacitance        : 0.874832;
        rise_capacitance        : 0.918145;
}

pin (A2) {

        direction               : input;
        related_power_pin               : "VDD";
        related_ground_pin              : "VSS";
        capacitance             : 0.974630;
        fall_capacitance        : 0.894119;
        rise_capacitance        : 0.974630;
}

pin (ZN) {

        direction               : output;
        related_power_pin       : "VDD";
        related_ground_pin      : "VSS";
        max_capacitance         : 60.577400;
        function                : "(A1 & A2)";
```

Figure 13 Function Section of each gate in .lib file

Value Change Dump (.vcd) File

Value change dump (also known less commonly as "Variable Change Dump") is an ASCII-based format for dump files generated by EDA logic simulation tools. The standard, four-value VCD format was defined along with the Verilog hardware description language by the IEEE Standard 1364-1995 in 1995. An Extended VCD format defined six years later in the IEEE Standard 1364-2001 supports the logging of signal strength and directionality. The simple and yet compact structure of the VCD format has allowed its use to become ubiquitous and to spread into non-Verilog tools such as the VHDL simulator GHDL and various kernel tracers. A limitation of the format is that it is unable to record the values in memories.
The VCD file comprises a header section with date, simulator, and timescale information; a variable definition section; and a value change section, in that order. The sections are not explicitly delineated within the file, but are identified by the inclusion of keywords belonging to each respective section.

VCD keywords are marked by a leading $ (but variable identifiers can also start with a $). In general, every keyword starts a section which is terminated by an $end keyword.
All VCD tokens are delineated by whitespace. Data in the VCD file is case sensitive.

HEADER

The header section of the VCD file includes a timestamp, a simulator version number, and a timescale, which maps the time increments listed in the value change section to simulation time units.

VARIABLE DEFINITION

The variable definition section of the VCD file contains scope information as well as lists of signals instantiated in a given scope. Each variable is assigned an arbitrary, compact ASCII identifier for use in the value change section. The identifier is composed of printable ASCII characters from ! to ~ (decimal 33 to 126). Several variables can share an identifier if the simulator determines that they will always have the same value. The scope type definitions closely follow Verilog concepts, and include the types module, task, function, and fork.

$dumpvars

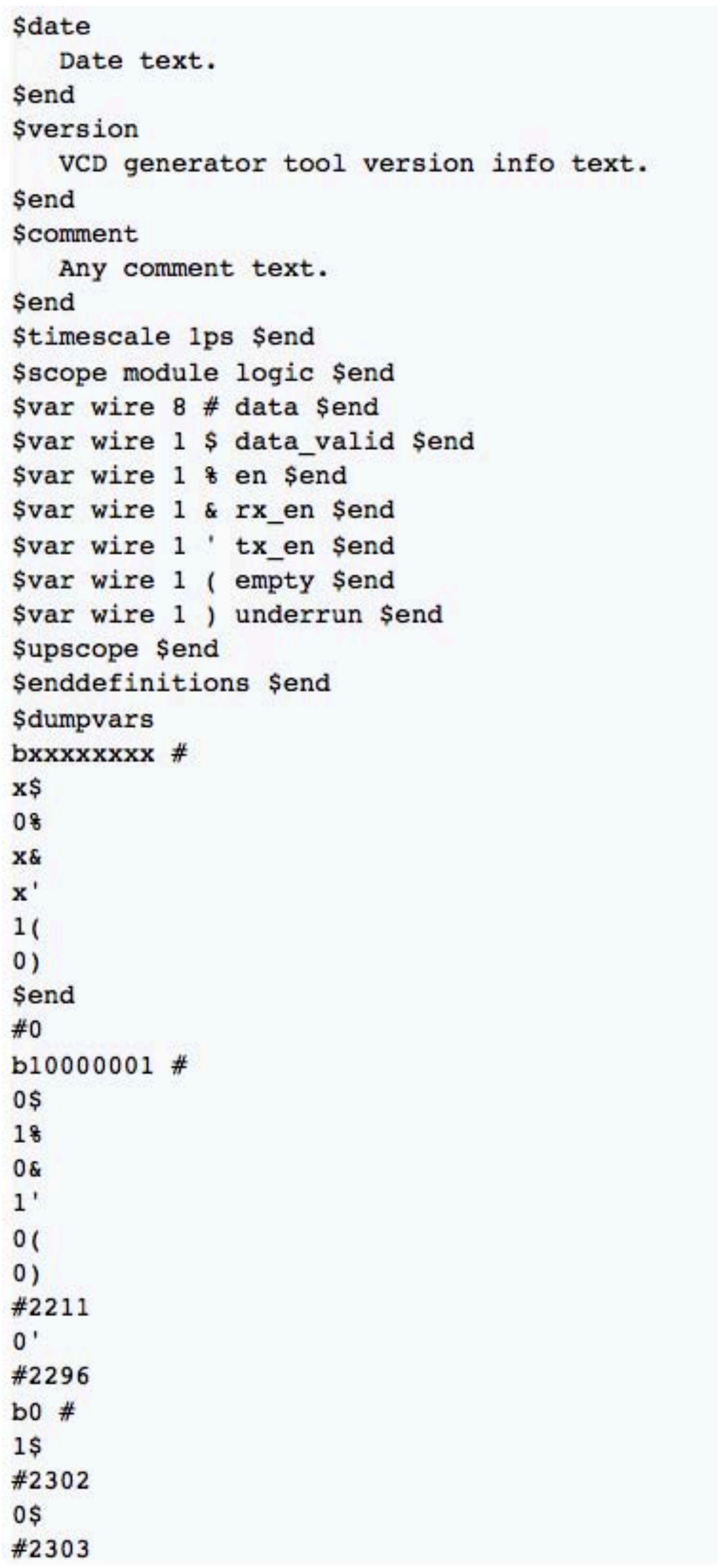
The section beginning with $dumpvars keyword contains initial values of all variables dumped.

VALUE CHANGE

The value change section contains a series of time-ordered value changes for the signals in a given simulation model. For scalar (single bit) signal the format is signal value denoted by 0 or 1 followed immediately by the signal identifier with no space between the value and the signal identifier. For vector (multi-bit) signals the format is signal value denoted by letter 'b' or 'B' followed by the value in binary format followed by space and then the signal identifier. Value for real variables is denoted by letter 'r' or 'R' followed by the data using %.16g printf( ) format followed by space and then the variable identifier.

This last section, is the section where we parse the primary inputs.

Below we can observe a typical Value Change Dump file.



```
$date
   Date text.
$end
$version
   VCD generator tool version info text.
$end
$comment
   Any comment text.
$end
$timescale 1ps $end
$scope module logic $end
$var wire 8 # data $end
$var wire 1 $ data_valid $end
$var wire 1 % en $end
$var wire 1 & rx_en $end
$var wire 1 ' tx_en $end
$var wire 1 ( empty $end
$var wire 1 ) underrun $end
$upscope $end
$enddefinitions $end
$dumpvars
bxxxxxxxx #
x$
0%
x&
x'
1(
0)
$end
#0
b10000001 #
0$
1%
0&
1'
0(
0)
#2211
0'
#2296
b0 #
1$
#2302
0$
#2303
```

Figure 14 A typical .vcd file

Classes Implemented

In order to achieve both an implementation and a visualisation in a later stage, of a digital circuit, given as inputs it's Verilog file, a Liberty library and a Value Change Dump file we must resort to using some classes to express our objects.

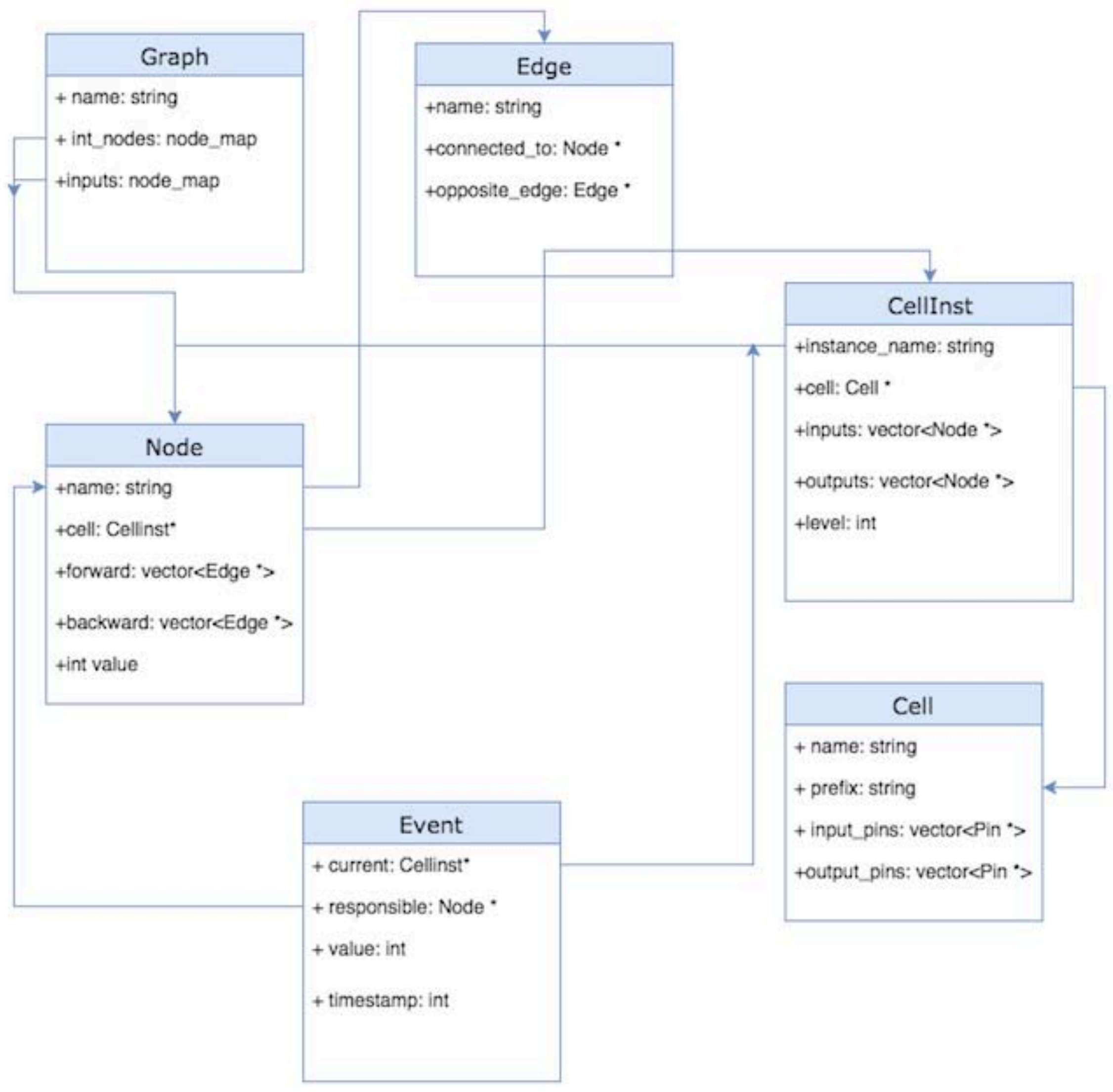


Figure 15 Classes Implemented

Below, one can find some explanatory information about the object-oriented construction of classes.

Graph

We created a class called Graph which refers to the implementation of a circuit as graph. Noteworthy information should be the following fields: **node-map inputs** as to represent the Primary Inputs of the circuit.

CellInst

This class refers to gates as a unity, the exact instance of every gate this time though. To put it in a simple way, each gate of the circuit is represented in the program or the graph as an object of type CellInst. The main fields we need to focus right now are the ones concerning the interaction of the particular nodes inside the CellInst, inputs and outputs. Thus, we can observe: **vector < Node * > inputs**, **vector < Node * > outputs**, **vector < Node * >& getInputs( )**, **vector < Node * >& getOutputs( )** are the ones necessary for the time being.

Node

Moving on to the Node class representing the inputs and outputs of each gate. This class allows us to further analyze the connectivity between inner parts of a gate, such as inputs, outputs and edges that exist. Some parts of this class that are important: **vector < Edge * > forward; vector < Edge * > getForwardEdges( )** which refer to forward Edges from a gate's output to another gate's input, and the field called int value and it's suitable set and get method, referring to the current value that a Node has. This value is set to 0 ***? Finally there exist two fields **cellInst * getCellInst(**) and **string getName( )** whose role is to make us able to know which CellInst contains the Node currently being examined.

Event
The main "Event" for this thesis. We constructed a class named exactly after the object representing it. This class should fit exactly ones' purposes, thus for our current purposes we chose to implement the following fields: **CellInst* current**, **Node* responsible** , **int value** , **int timestamp** as for the current CellInstance, the responsible for the event that happened, the value this time and the timestamp.

Internal Representation

The Verilog files describe the circuit connectivity, in the abstract level of cells and nets. In order to apply the simulation algorithm, that connectivity information must be interpreted and mapped into a graph, which is performed in the parsing module. For example, visualizing the circuit c17.v, in the abstract model of cells and nets will yield the following connectivity map.

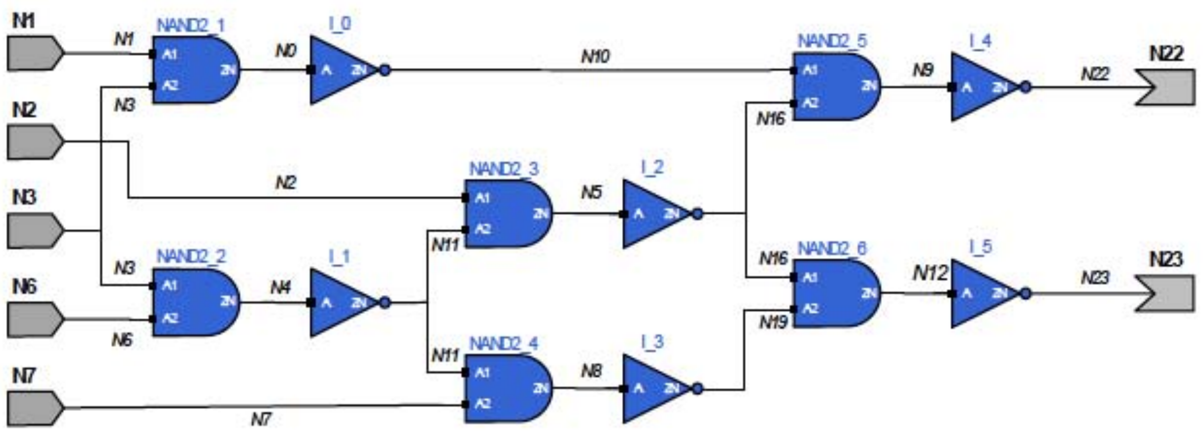


Figure 16 Visualisation of connectivity map of circuit c17.v

Many of the algorithms used in the industry, consists of graph explorations or different graph operations, which means that the connectivity map must be transformed into a simple directed graph representation (nodes/edges).

The nodes of the graph will be the pins of each cell, and the ports (Primary inputs/Primary outputs). The names of each node must be unique, which means that the name of the pins cannot uniquely define a node. In that case, an enhanced name must be used, which uses as a prefix the instance name of the cell, which is already unique. The ports are already unique, but for simplicity reasons, have a prefix of input/output.

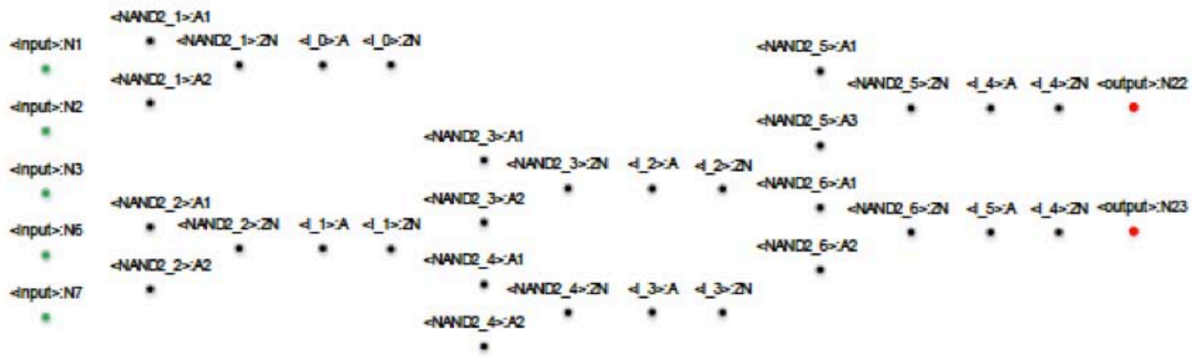


Figure 17 The nodes of a pin based graph

The edges of the graph, that describe the connectivity between cells, will be the nets of the circuit. There is no need for naming the edges, but for simplification the actual name of the nets is used. These edges are described as net edges.

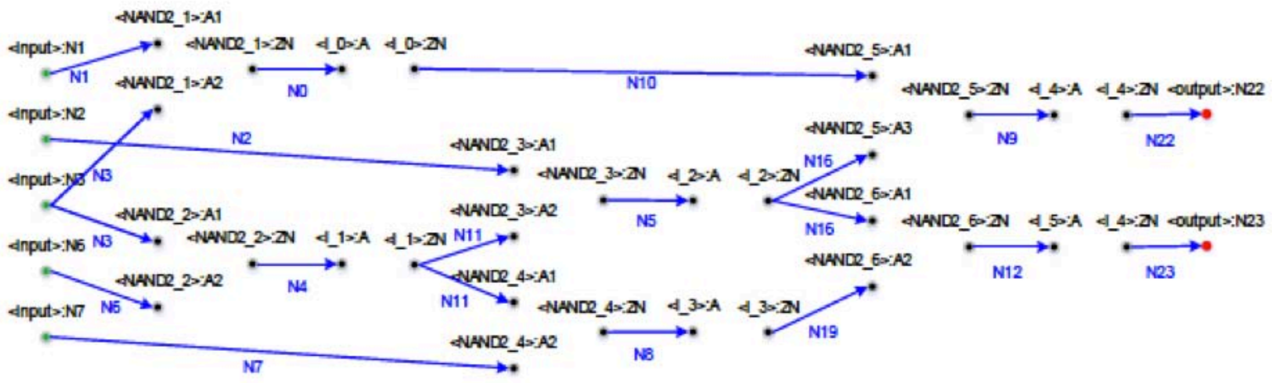


Figure 18 The graph including the cell connectivity

The tool cannot extract the total graph of the circuit by only using the Verilog files. The connectivity information inside its cell, is described in the .lib file. Each output of a cell has a set of input related pins, for which the timing information is specified. Those relations specify the internal connectivity of a cell and the edges are described as cell edges.

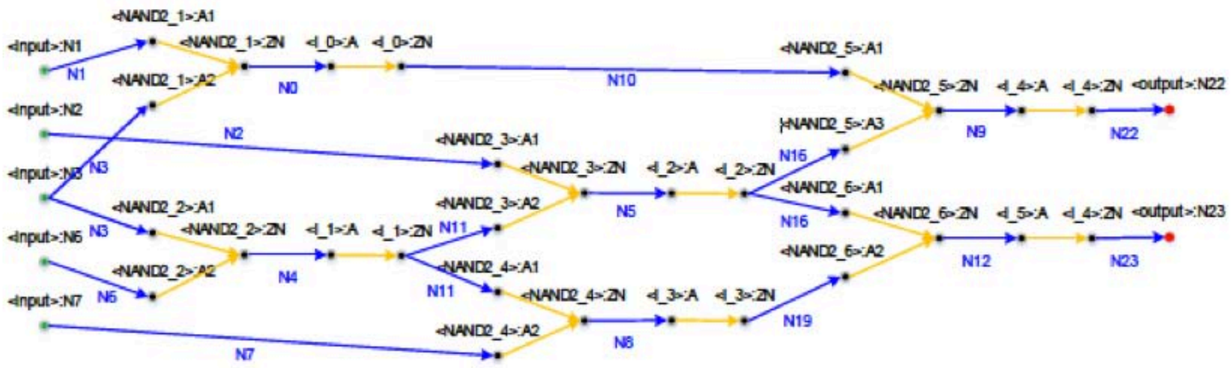


Figure 19 The graph including the cell internal connectivity

The internal representation is further enhanced by grouping all the associated pins in the same data structure, which is the cell instance, Figure 20. The pins are also categorized by its type, input or output.

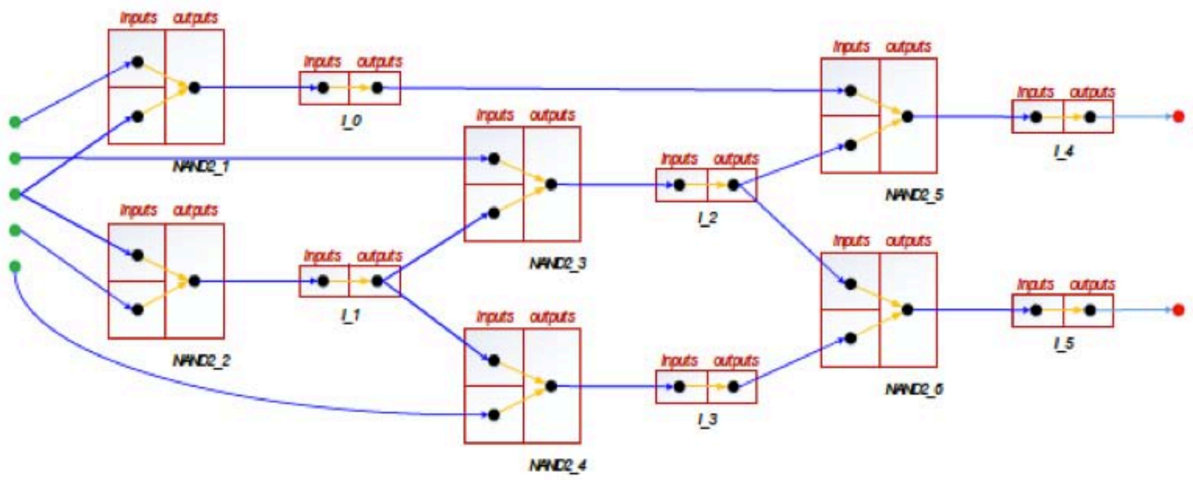


Figure 20 The final internal representation including the CellInstances

The nodes of the circuit store the transition and arrival times, as well as its logical value. The net edges store the wire capacitance, the resistance, and the calculated delay. The cell edges store the timing tables and calculated delay.

Theoretical Approach to our simulator

Unit Delay event-driven simulator design

The scheduler is an important component of an event-drive simulator. It keeps track of event occurrences and schedules the necessary gate evaluations. For zero-delay simulation, the event queue is a good enough scheduler because timing is not considered. For unit-delay simulation, however, a more sophisticated scheduler is required to determine not only which gates to evaluate but also when to evaluate them. Because events must be evaluated in chronological order, the scheduler is implemented as a priority queue. Figure 20 depicts one possible priority queue implementation for a unit delay event-driven simulator. In the priority queue, the vertical list is an ordered list that stores the time stamps when events occur. Attached to each time stamp t_i is a horizontal list of events that occur at time t_i. During simulation, a new event that will occur at time ti is appended to the event list of time stamp t_i. For example, in Figure 20, the value of signal w will switch to v+w at t_i. If t_i is not in the time stamp list yet, the scheduler will first place it in the list according to the chronological order.

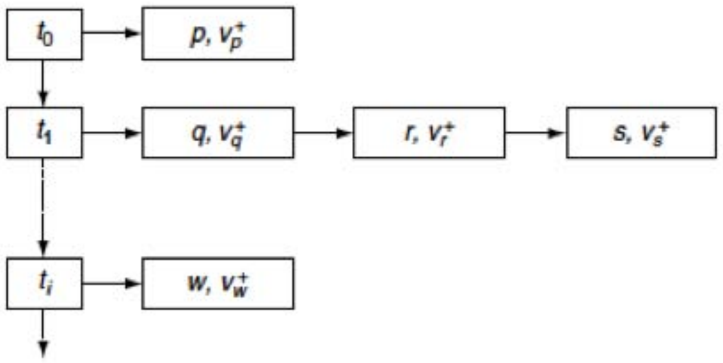


Figure 21 Priority queue event scheduler

The TimeWheel Concept

For the priority queue scheduler in Figure 21, the time needed to locate a time stamp to insert an event grows with the circuit size. To improve the event scheduler efficiency, one may use, instead of a linked list, an array of evenly spaced time stamps. Although some entries in the array may have empty event lists, the overall search time is reduced because the target time stamp can be indexed by its value. Further enhancement is possible with the concept of timing wheel as shown in Figure 22.

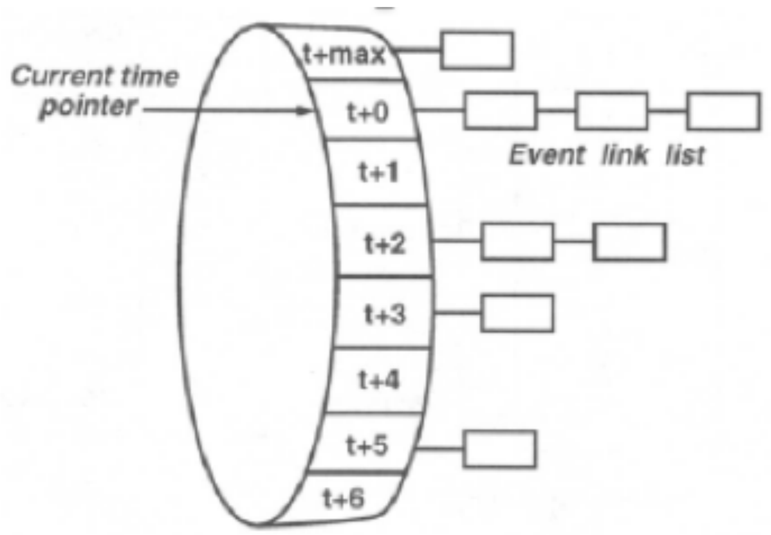


Figure 22 A circular stack - TimeWheel - implementation of the priority queue of our event-driven simulator

Let the time resolution be one time unit and the array size M. A time stamp that is d time units ahead of current simulation time (with array index i) is stored in the array and indexed by (i+d) mod M if d is less than M; otherwise, it is stored in an overflow remote event list similar to that is shown in Figure 21. Remote event lists are brought into the timing wheel once their time stamps are within M−1 time units from current simulation time. A two-pass strategy for unit delay event-driven simulation is depicted in Figure 23. When there are still future time stamps to process, the event list L_E of next time stamp t is retrieved. L_E is processed in a two-pass manner. In pass one (the left shaded box), the simulator determines the set of gates to be evaluated. The notation (g, v+g) indicates that the output of gate g is to become v+g. For each event (g, v+g), if v+g is the same as g's current value vg, this event is false and is discarded. On the other hand, if v+g = vg (i.e., (g, v+g) is a valid event), then vg is updated to v+g, and the fanout gates of g are appended to the activity list L_A. In the second pass (the right shaded box), gates are evaluated and new events are scheduled. While the activity list L_A is non-empty, a gate g is retrieved and evaluated. Let the evaluation result be v+g. The scheduler will schedule the new event (g, v+g) at time stamp t+delay(g), where delay(g) denotes the nominal delay of gate g. The two-pass strategy avoids repeated evaluation of gates with events on multiple inputs.

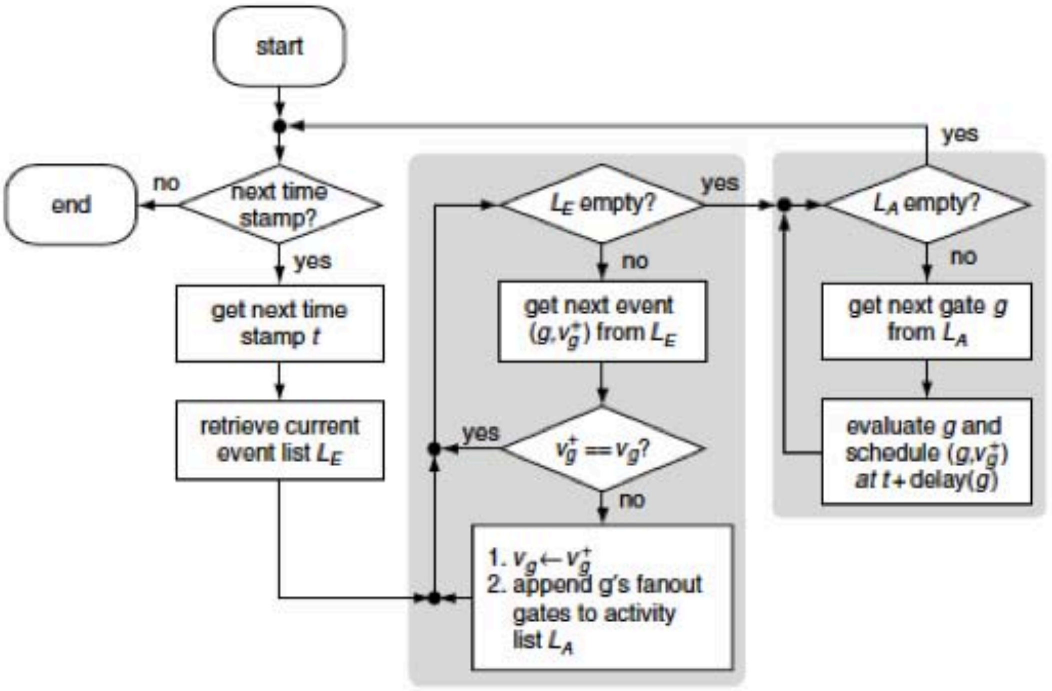

start
end
no
next time stamp?
yes
get next time stamp t
retrieve current event list L_E
L_E empty?
yes
no
get next event (g,v_g^+) from L_E
yes
$v_g^+ == v_g$?
no
1. $v_g \leftarrow v_g^+$
2. append g's fanout gates to activity list L_A
yes
L_A empty?
no
get next gate g from L_A
evaluate g and schedule (g,v_g^+) at t+delay(g)

Figure 23 Two-pass simulation strategy flowchart

Implementation of the TimeWheel

After having successfully parsed the three aforementioned files (.v, .lib, .vcd), and having constructed the suitable objects, classes to categorise and identify our circuits' elements, we should have obtained numerous information helping us have an algorithmic approach to our problem. By now, from the .v file, we should have constructed and "visualised" our circuit as a graph, upon which our simulator is based. This graph should contain, as we have seen both at the UML and the Internal Representation above, CellInst(ances) representing each time the respective gate, which should contain input and output pins represented as Nodes and lists of inputs and outputs. Nets that are outputs or inputs from gate to gate, that should translate to CellInst, are characterised as Edge-type.

Furthermore, parsing the .lib file, is the task that gives life and meaning to CellInst and Nodes. By browsing specific characteristics about each gate and storing them properly we can simulate each time the function we want to. One of these, is the logic function of each gate, which we find and store as a type of string.

Traversing throughout the graph we can link the right nets to the prototype's function every time. That is obligatory in order to have the right values inserted to the logical function in order to evaluate the result. Linking the right inputs to each gate's nodes, obtaining an equivalent string as result, this time not with the formal logic function but with the real inputs inserted.

As we earlier saw, inside the liberty library one can find the logic function for each available gate. It is not only parsing that is needed in this occasion though, as the function comes as a type of string. Using a stack, we can turn that string to a fully functional logic equation.

There can be seen two main structures for creating the so-called event-driven unit delay logic simulator. Besides the basic class created (Event), one can find the **vector < Event * > activity**. This activity list, withholds any events that are pending to happen. Every event that is executed is removed from this list so as not to be executed another time. Another major data structure used in this thesis is the **multimap < int , Event * > mymap**. This is the history of all events that happened during the simulation and until our circuit's output has stabilised. It is obvious that two variables are stored every time. The first one is the timestamp – which every time increases by one unit – and the second one is a pointer to the specific every time Event-object that contains the information about the event, such as the Node that the event happened, the responsible for the event, the timestamp, and the value that caused the event.
We proceed creating the starting events as parsed from the .vcd file. These events concern the primary inputs at first. The primary inputs can be found stored as type of node_map in the Graph class. These events of course will have a timestamp of 0.
As of now, our simulation has just got started. We continue spinning the timewheel, advancing the time plus one unit each time, exploring the primary events the first time, the events that are caused because of them, evaluating and executing.
Below we can observe two flow charts. The first one is more general, as to the tasks that we must carry out in each cycle, while the second one is more specific, on how we implemented a two pass strategy, similar to the one explained above.

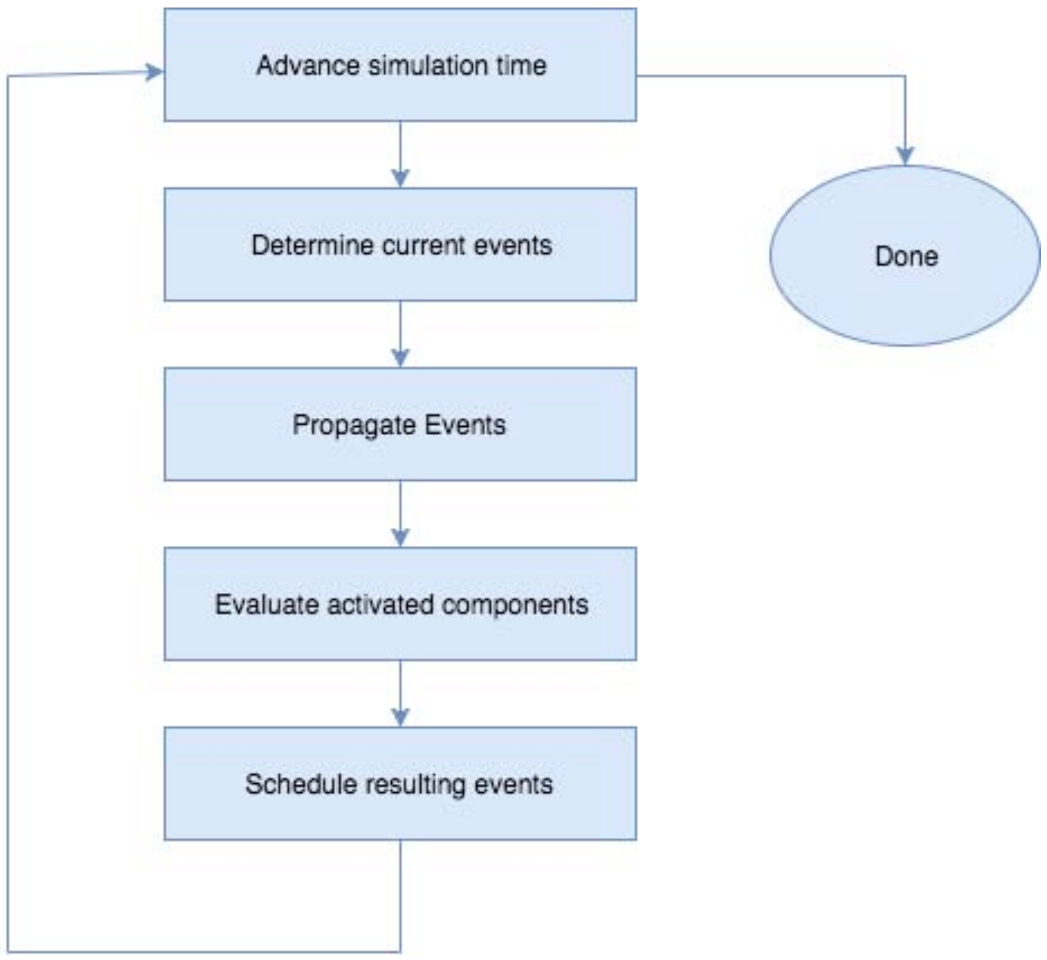


Figure 24 Main Flow of Event-Driven Simulation. During simulation, these tasks are carried out in each cycle until no events are left.

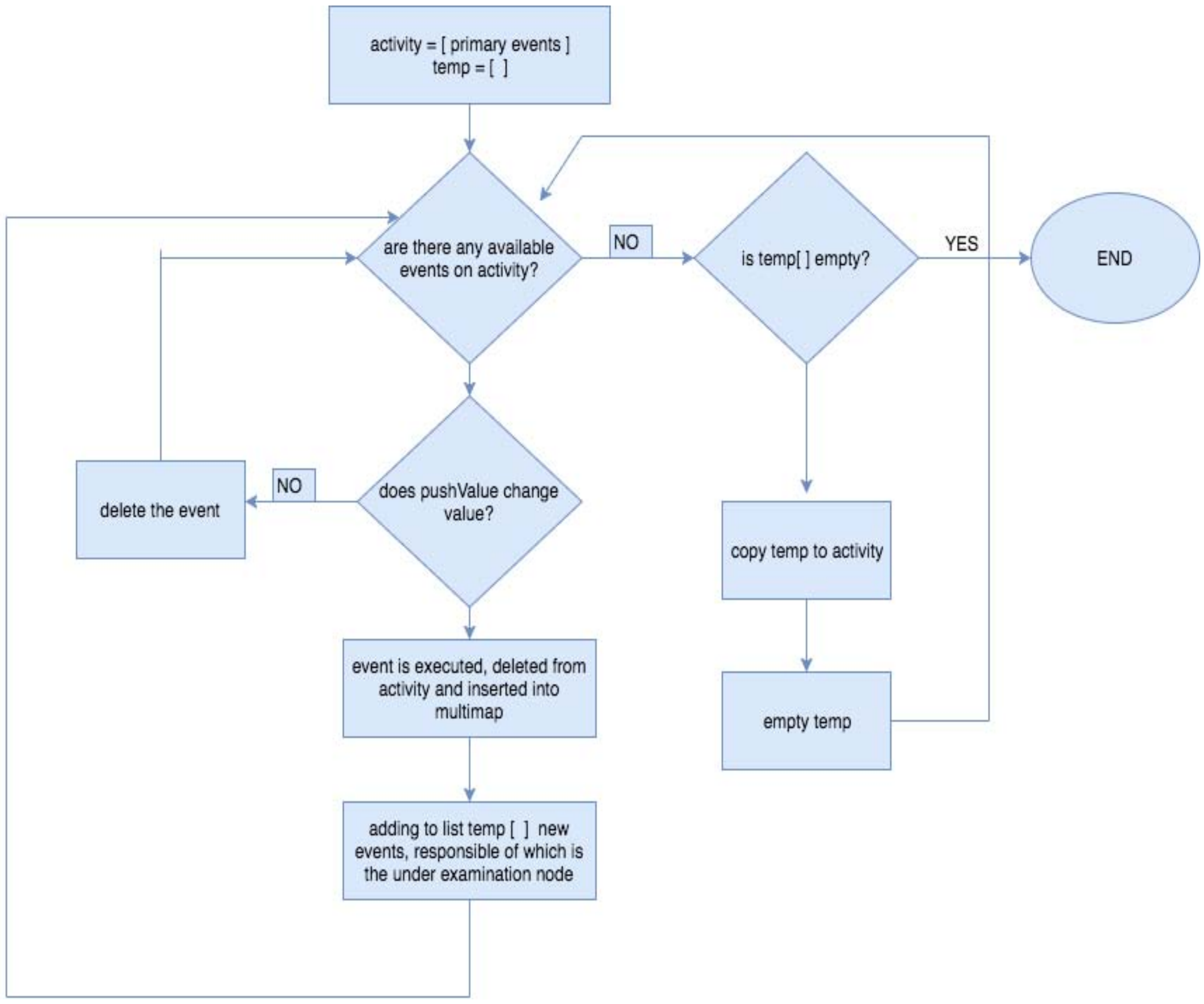

activity = [primary events]
temp = [ ]
are there any available events on activity?
NO
is temp[ ] empty?
YES
END
does pushValue change value?
NO
delete the event
copy temp to activity
event is executed, deleted from activity and inserted into multimap
empty temp
adding to list temp [ ] new events, responsible of which is the under examination node

Figure 25 Two-pass simulation strategy flowchart implementation

Multimap Container

After an Event has been executed, it is deleted from the Activity List and it is inserted into a multimap, as a kind of history, for further verification and testing purposes. Multimap is a Sorted Associative Container that associates objects of type Key with objects of type Data. multimap is a Pair Associative Container, meaning that its value type is **pair<const Key, Data>**. It is also a Multiple Associative Container, meaning that there is no limit on the number of elements with the same key.

Multimap has the important property that inserting a new element into a multimap does not invalidate iterators that point to existing elements. Erasing an element from a multimap also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.

We have chosen to keep history of event that way, having in mind that multimap is usually implemented as sorted binary tree. Such trees can most commonly be a red black known for their good time complexity in big O notation in searching insertion and deletion. In this case, our concern for further debugging reasons is the searching complexity, as we want to investigate events that happened in the past and who was the responsible for the particular event plus the timestamp of the event as well.

Red-Black Trees

A red–black tree is a kind of self-balancing binary search tree. Each node of the binary tree has an extra bit, and that bit is often interpreted as the colour (red or black) of the node. These colour bits are used to ensure the tree remains approximately balanced during insertions and deletions.

Balance is preserved by painting each node of the tree with one of two colours (typically called 'red' or 'black') in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the colouring properties. The properties are designed in such a way that this rearranging and recolouring can be performed efficiently.

The balancing of the tree is not perfect, but it is good enough to allow it to guarantee searching in O (log n) time, where n is the total number of elements in the tree. The insertion and deletion operations, along with the tree rearrangement and recolouring, are also performed in O (log n) time.

Tracking the colour of each node requires only 1 bit of information per node because there are only two colours. The tree does not contain any other data specific to its being a red–black tree so its memory footprint is almost identical to a classic (uncoloured) binary search tree. In many cases, the additional bit of information can be stored at no additional memory cost.

Conclusion

As the era emerges, with the complexity of the circuits augmenting so fast, CAD tools not only for design and verification but also for designing and implementing circuits in general, have become as aforementioned a viable replacement for physical prototyping. Aim of this thesis was to design and implement an event-driven logic simulator for digital circuits. After analysing different types of simulators and the theoretical background we implemented the one fitting our purposes.

Studying in general about digital circuits has been intriguing, having always in mind the scalability involved in VLSI circuits. It has been a pleasure for me to have worked under the supervision of such a great environment at the office E5.

Once again, I would like to attribute huge acknowledgements to all the people in my life that assisted me and shared their wisdom with me, through good and tough times, during my studies at the University of Thessaly.

References

[1] Abramovici, M., Breuer, M.A, and Friedman, AD.Digital Systems Testing and Testable Design. New York: W.H. Freeman & Co., 1990.

[2] d'Abrea, M. "Gate-Level Simulation" IEEE Design and Test, Dec.l985.

[3] Breuer, M.A "A Note on Three-Valued Logic Simulation" IEEE Trans. on Computers, Vol. C-21, No.4, April 1972.

[4] Breuer, M.A., and Parker AC. "Digital System Simulation: Current Status and Future
Trends" Proceedings 18th Design Automation Conference, 1981.

[5] Brown, A VLSI Circuits and Systems in Silicon New York: McGraw-Hill, 1991.

[6] Bryant, R.E. "MOSSIM: a Logic-Level Simulator for MOS LSI User's Manual" MIT, VLSI Memo No. 80-21, July 1980.

[7] Hitchcock, R.B. "Timing Verification and the Timing Analysis Program" Proceedings, 19th Design Automation Conference, 1982.

[8] Holt, D., and Hutchins, D. "A MOS/LSI Oriented Logic Simulator" Proceedings 18th, Design Automation Conference, 1981.

[9] H. Chen, "Event driven logic simulator", 1992

[10] Korsh, J.F., and Garrett, LJ. Data Structures, Algorithms, and Program Style Using C. Boston: PWS-Kent Publishing Co., 1988.

[11] Johnson, W.A. "Behavioral-level Test Development" Proceedings 16th Design Automation Conference, 1979.

[12] Maczo, A Digital Logic Testing and Simulation. New York: John Wiley & Sons, 1986.

[13] Ruehli, AE. "Survey of Analysis, Simulation and Modeling for Large Scale Logic Circuits" Proceedings 18th Design Automation Conference, 1981.

[14] Szygenda, S.A. "TEGAS2 -- Anatomy of a General Purpose Test Generation and Simulation System for Digital Circuit" 25 Years of Electronic Design Automation,1988.

[15] VLSI editors "1987 Survey of Logic Simulators" VLSI Systems Design, Vol. 8, No.2, Feb. 1987.

[16] Alexander Miczo, "Digital Logic Testing", Second Edition

[17] M. Bushnell, V. Agrawal, "Essentials for electronic testing for digital, memory and mixed-signal circuits"

[18] L. Wang, C. Wu, X. Wuen, "VLSI Test Principles and Architectures”

[19] C. Kalonakis, D. Ntioudis, "Analysis and Development of Digital Circuits Optimization Algorithms in Terms of Speed and Power Dissipation"