Visual Exploration of Simulated FPGA Architecures in Odin II

by

K. Nasartschuk, K. B. Kent and R. Herpers

TR 12-220, August 20, 2012

Faculty of Computer Science
University of New Brunswick
Fredericton, NB, E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
http://www.cs.unb.ca

Abstract

Field Programmable Gate Array (FPGA) research became more and more important during the last decades. The FPGA technology is being used in many fields and offers the main features scalability, flexibility and the low costs of prototyping. The functionality of FPGA devices is developed using hardware description languages such as Verilog. Those descriptions are translated to boolean circuits which can be programmed on the FPGA devices using Computer Aided Design (CAD) Flows which optimize the circuit for the specific features offered by the FPGA device. The VTR CAD flow is such a workflow consisting of three tools: Odin II, ABC and VPR6.0. In order to handle the growing scale of circuits and features offered by FPGA devices the tools in the CAD flow are being improved constantly. This report describes the development of simulation features within the Odin II circuit visualization software. The software was created in order to visualize the netlist created by Odin II to offer developers exploration functionality. Being able to simulate the circuit in the visualization creates a verification feature which can be used for debugging and research of new FPGA architectures. Another goal of the report is to improve the usability and exploration comfort of the visualization software.

Contents

List of Figures

List of Tables

1 Introduction

Field Programmable Gate Arrays (FPGA) are widely used for prototyping and in areas where requirements of an application may change frequently. Instead of creating hundreds or thousands of devices which functionality cannot be changed developers tend to use reprogrammable devices.

Verilog HDL [1] and VHDL [2] are hardware description languages used for development of FPGA applications. The languages provide the ability to define functions which are later translated into boolean circuits. These circuits are programmed onto an FPGA device. The translation from a description file to the final boolean circuit is performed by Computer Aided Design(CAD) tools.

The VTR [3] project provides a CAD workflow consisting of three tools where each tool performs improvements. The workflow was created in order to assist the developers of FPGA architectures to manage the growing complexity and size of FPGA architectures. Evaluation and verification tend to be very complex and time consuming during the development phase and are mostly done manually. Odin II is part of the VTR project and represents the first stage of the VTR workflow. It performs Verilog elaboration and the front end synthesis. Simulation and visualization of the circuit in Odin II were created in order to assist developers during evaluation and verification processes.

This report describes the improvement of the visualization component in Odin II by combining the abilities of the Odin II its simulator and the visualization tool. The main goal is to to explore the simulation process of the created netlist visually. Feedback by developers who work with Odin II is used to improve the usability of the visualization tool in general and to create new functions in order to assist developers during the exploration process.

The report is divided into 6 Chapters. Chapter 2 introduces Field Pro-

grammable Gate Arrays, gives a short introduction to the history of FPGA and describes the basics of graphical user interface (GUI) development.

Chapter 3 describes the VTR [3] project, the tools it consists of and the workflow in general. The chapter describes what Odin II is used for in the workflow, what functions already exist to assist during evaluation and verification and how they can be improved. This improvement is the main goal of this project. The design of the proposed solution is explained in Chapter 4. Improvements to different parts of Odin II are pointed out and an evaluation strategy is defined in this chapter.

The implementation of the additional functions is covered in Chapter 5. Chapter 6 compares the goals which were defined during the design phase and the results using the evaluation strategy. Chapter 7 summarizes the project and gives an outlook into future developments in Odin II and its components.

2 Basics

The project describen in this report combines the FPGA techonology with the field of graphical user interfaces. This section gives a short introduction in both fields of research. Section 2.1 explains the development of FPGAs, describes it's history and fields of research in this area. The history of graphical user interfaces is given in Section 2.2.

2.1 FPGA Introduction and History

During the 1990s the development of hardware became more and more risky as the production cost of chips ranged from $20,000 to $200,000 [4]. The production Application Specific Integrated Circuits (ASICs) is very time consuming and expensive. The goal of the hardware industry was to reduce the costs of hardware prototypes and the time between programming to testing. A solution for this problem was to use a technology which was introduced in 1986 by Xilinx Inc. named "Field Programmable Gate Arrays (FPGA)" [4][5].

FPGAs have two main advantages which lead to the conclusion that this technology is a good solution to prototype hardware. At first prototypes can be produced in small quantities and "no facility must be tooled to begin production of a mask-programmed device which incurs a large overhead cost" [4]. The second reason is that the programming process is finished within minutes and can be tested, "whereas mask-programmable devices must be manufactured by a foundry over a period of weeks or months" [4].

An FPGA is basically a collection of logic blocks on a hardware device, which can be programmed and connected. There are many different architectures, which place the logic blocks and possible interconnections. As can be seen in Figure 2.1 the basic design of an FPGA consists of logic units, interconnection resources and I/O-cells, which are arranged as a two-dimensional array. The FPGA design is always "a trade off in the complexity and flexibility of both the logic blocks and the interconnection resources [4]". The most common design is called *island-style* and consists of evenly distributed

I/O-cells surrounding a basic structure of logic blocks. The main advantage of this structure is the flexibility allowing a single application to "be used in multiple products all with different capacity, pin count, package etc [6]". The structure can also be used in other applications by embedding the design with all its logic blocks and their interconnections [6]. Figure 2.2 shows an example of an *island-style* designed FPGA, which looks very similar to the conceptual FPGA in Figure 2.1.

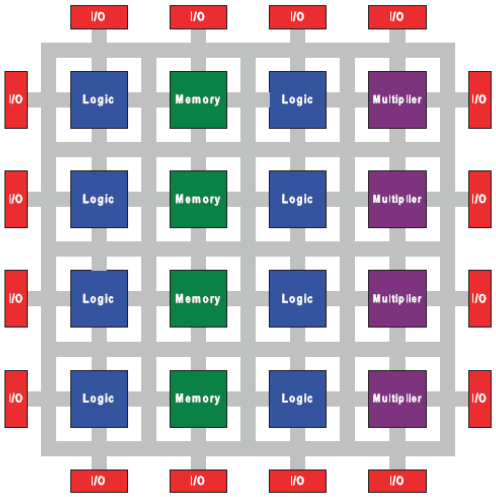

Figure 2.1: A Conceptual FPGA [7]

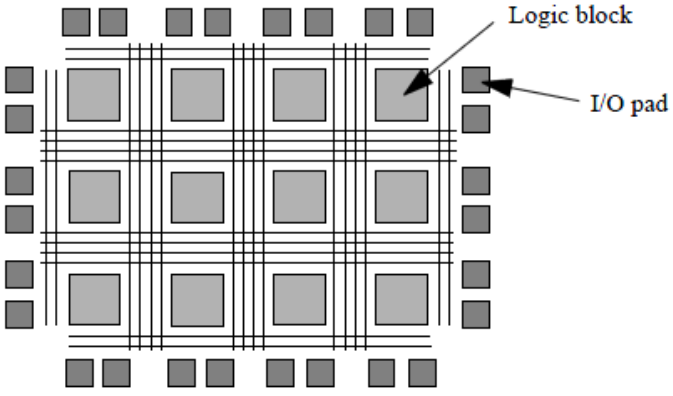

Figure 2.2: "Island-Style" [5]

There are a variety of manufacturers which produce FPGAs. The archi-

tecture and design differs widely. Logic units on an FPGA differ and can represent different levels of granularity. Predefined logic blocks are often embedded in FPGA devices. A more granular logic block is more effective in speed, but less flexible compared to a block with finer granularity [8]. Examples for more granular hard blocks are multipliers or memories.

The first FPGA, presented by Xilinx contained "64 logic blocks and 58 inputs and outputs [7]". Nowadays the size of FPGAs grows every year and in 2007 they contained "approximately 330,000 equivalent logic blocks and around 1100 inputs and outputs [7]". This leads to different problems. The architecture of such a device is a problem which is addressed in the next chapter. Other challenges are the programming and understanding of processes that happen in such a device.

2.2 GUI Development

Computer applications perform actions or computations, which the user has ordered them to do. Most of the applications are very specific and users are capable of using them by entering commands on a command console. The basic flow of such an application can be seen in Figure 2.3a. It receives the input parameters and terminates after results are delivered.

Example advantages of such an application are, that no resources are needed in addition to the actual computation and that computations can be performed on servers if human interaction is not necessary. The main disadvantage of a classical batch-processing application is the way of interaction with a human user. The user has to learn how to address the program and how to interact with it. To simplify this process programs with high human contact are offering graphical user interfaces (GUI), which encapsulate the complexity of the program into an intuitive interface.

The structure of such an application differs from that of a batch application (Figure 2.3) as it adds an event loop, which awaits user commands and passes those commands to the actual program. Such graphical user

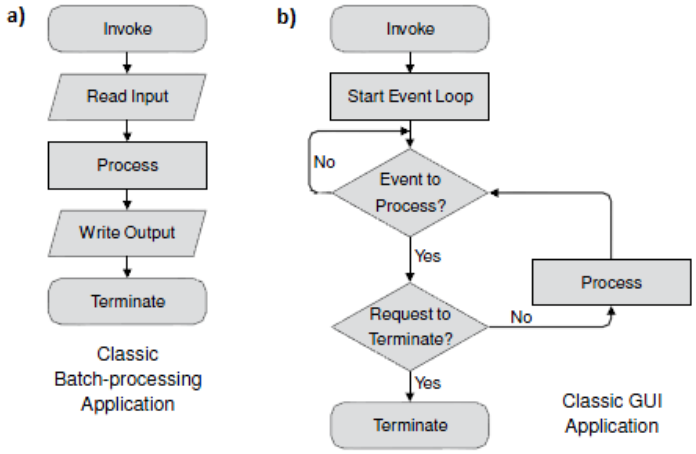

Figure 2.3: GUI Application Flow and Classic Batch-processing Application in Comparison [9].

interfaces can be implemented using different languages and pre-compiled libraries. The Qt [10] library is used for this project and is introduced in the next chapter.

2.2.1 QT

The first public version of Qt (pronounced *cute*) was published in May 1995 [10]. It was developed at the Norwegian Institute of Techonlogy in Trondheim by two students, Haavard Nord and Eirik Chambe-Eng. The development began in the late 90s, but was published after the students founded the company Quasa Technologies, which later became Troll Tech and is known today as Trolltech [10]. Trolltech is a subsidiary of Nokia. Graphical interfaces developed using Qt are "KDE, the web browser Opera, Google Earth, Skype, Qtopia and OPIE" [11].

The library consists of modules whose functionality is clearly structured and separated. An application only uses the parts of the framework, which are necessary. Dialogue based and also window based applications are possible [12].

The library is written in C++ and utilizes object oriented standards. Parts of the application implement classes which already exist in the library starting with the main class `QObject` and becoming more and more specific to classes such as `QLine` or `QPolygon`.

Qt consists of 22 modules, which are used for different purposes. The `QtGui` module consists of classes which are used to create graphical user interfaces. `QtCore` is a collection of core non-graphical classes used by other models.

Xml, WevKit, OpenGL, Svg can also be handled using modules, which are provided by Qt. A full list of modules and an API can be found on the website of Qt [13].

Development in Qt in the basic version only allows C++, but many projects created bindings for the library. The reason was mostly to be able to use the functionalities of Qt in different programming languages. Qt bindings for more than 15 languages are available. Examples include PyQt (Python), QtRuby (Ruby) and QtLua (Lua).

3 Related Work

The Verilog to Routing (VTR) [3] project is a collaboration project consisting of three core tools Odin II [14], ABC [15] and VPR [16]. "Odin II is responsible for Verilog elaboration and front-end hard-block synthesis [3]", ABC for logic synthesis and VPR for physical synthesis and analysis. The structure of the workflow is shown in Figure 3.1.

A Verilog HDL design and a FPGA description are required as inputs for the workflow. The result of the stages in the workflow is an output bit stream which is used to program the FPGA device. The communication between the stages is performed using the Berkeley Logic Interchange Format (BLIF) [17]. It contains the current state of the circuit. A more detailed description of the tools involved in the VTR project is found in the following sections.

3.1 Odin II

Odin II is a "framework for Verilog Hardware Description Language(HDL) synthesis [14]" which is an improved version of Odin [18]". Its main purpose is the "the front-end conversion of a HDL design into a netlist of basic gates and more complex logic functions [18]". The inputs which are required for Odin II are a Verilog HDL design and an architecture description file. Odin II uses the Verilog HDL description to create an "abstract syntax tree [14]". The architecture description is used to map functionalities onto hard blocks which are available on the specific FPGA device.

As shown in Figure 3.2 there are multiple stages in Odin II which lead to a flat netlist. In order to create this netlist Odin II performs a first optimization and partial mapping. The architecture file is used to discover functionalities which are already implemented as hard blocks on the device and can be used for the circuit. [14] gives the example of a 8x8 multiplier which is used in the design and is available on the device. Odin II binds those to speed up the computation and decrease the number of soft logic

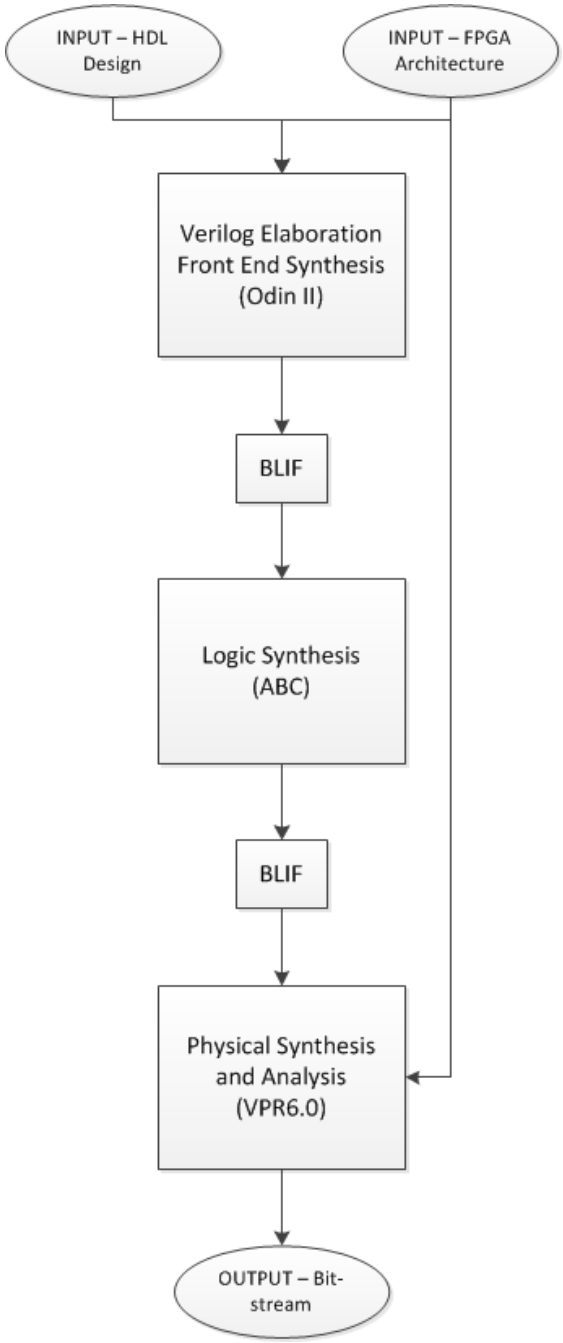

Figure 3.1: The VTR Workflow

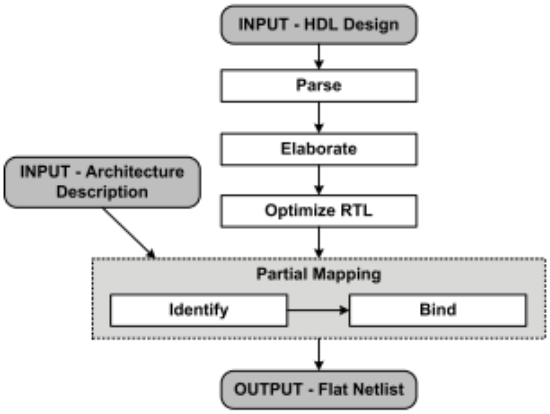

Figure 3.2: "The Processing Stages in Odin II [14]"

blocks used.

"Odin II Verilog elaboration front end has four key roles in the VTR framework [3]". The roles are the interpretation and conversion of "some of the Verilog syntax into a logical netlist targeting the soft logic on the FPGA [3]", synthesis of constructs into hard blocks, "to be responsive to the architecture description of the FPGA" and "to provide a framework for the verification of the correctness of the software flow [3]".

3.1.1 Simulation in Odin II

Simulation in Odin II was created to "verify that the produced netlists are functionally correct [19]". Also the verification of the "future work on the entire tool flow [19]" is computed using the simulator in Odin II.

The simulation algorithm traverses through the netlist starting at the topmost nodes which are the inputs and the constant nodes. The input values for each cycle are taken either from a predefined file or generated randomly. Using the input values the follow up nodes are enqueued and computed if all parent nodes are ready. This procedure iterates through the whole netlist

until all nodes are processed [19].

After the queue is empty and there are no more nodes to compute, "the simulator writes the values of the netlist output nodes to an output vector file [19]". This procedure is repeated until the desired number of input and output vectors is created. The combination of input and output vectors can be used to verify the correctness of the circuit [19].

3.2 Visualization in Odin II

The visualization software was created in 2011 and introduced in [20]. Its main purpose is to assist the exploration and development of new FPGA architectures. It "aims to improve the productivity of development" and "support the evaluation process" in the VTR [3] workflow.

The application is developed using the Qt [13] framework which allows developers to use a subset of modules required for the specific application. The input for the visualization is a Berkeley Logical Interchange Format (BLIF) [17] file. The input file represents a boolean circuit created during the workflow. The nodes in this structure are visualized using objects of classes `LogicUnit` or `Wire` which store information and parameters necessary for the visualization and exploration process.

The visualized graph provides functions such as zoom, arrangeable node structure or the ability to insert notes into the graph. The visualization software was designed using object oriented principles to be extendible in order to add new functionalities.

3.3 VPR6.0

The toolset VPR [21] was developed at the University of Toronto and is "used to perform FPGA architecture and CAD research [22]." VPR performs placing of logic structures on FPGA devices and designes routing based on

experimental results. In 2011 a beta of VPR6.0 was introduced in [16]. Since the release of VPR5.0 [22] the development has focused on the following features:

- Single Driver Routing Architectures
- Heterogeneous Logic Blocks
- Optimized Circuit Design in released Architecture Files
- Robustness

Improvements in VPR6.0 are "new constructs that allow the description and packing of far more complex logic blocks [3]" and "timing driven functionality [3]." In the VTR workflow VPR is mainly used for the specification of architecture descriptions, packing, placing and routing. The flow structure of VPR is shown in Figure 3.3

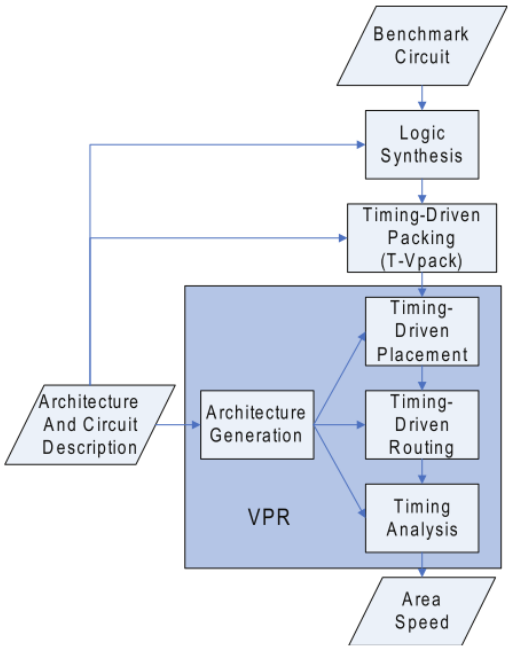

Figure 3.3: "The Processing Stages in VPR [22]"

4 Design

This chapter describes the planning and designing process of the project. Different approaches are discussed and chosen for implementation. The main procedure in this chapter is to specify the problem the project is aimed to solve, create an approach how to solve the problem and to create an evaluation strategy in order to figure out how well the problem was solved. The problem specification can be found in Section 4.1. The basic design of the approach is located in Section 4.3 and the evaluation strategies are created in Sections 4.4 and 4.2.

4.1 Problem Specification

The contribution of this project is to enhance the exploration tool in Odin II by adding extended functionality. The main part of the project is to include the simulation functionality into the software and provide the functions in the graphical user interface.

The simulation is performed on a netlist which is created by Odin II according to a Verilog HDL or a BLIF file. This project mainly focuses on BLIF files. The simulator creates random input vectors or uses a file with predefined input values in order to compute the states and values of all nodes in the netlist.

Initially all values are set to *undefined*. The netlist is traversed starting from the inputs and following all outgoing connections. The output of the simulation is a file which represents all outputs at all stages during the simulation. Values for the nodes that are not output nodes are not stored.

In order to visualize the simulation process, Odin II and the visualization tool need to be connected. The netlist is used to assign output values during the simulation. This requires every node in the visualization to be paired with a node in the netlist.

The simulation is processed in *waves*. A usual wave in Odin II is 16 cy-

cles. The value can be changed in the makefile of Odin II. The simulator computes a whole wave at a time before it stores the output values and computes the next wave. This reduces the file access times as the result does not have to be stored after each cycle. From the perspective of the visualization it is possible to simulate different numbers of cycles which number is usually a multiple of the *wave* length.

The second part of the project is to improve the usability of the BLIF explorer by adding functionality requested by developers. The main goal of this process is to enhance the exploration ability and the readability of a visualized circuit.

4.2 Use cases

The project is created from the perspective of the user and is implemented top down. Use cases are defined to represent the functions. Those are divided into exploration and simulation according to the structure of this project.

- Simulation
  - Perform simulation of the currently viewed circuit
  - See current states of outputs in the visualization
  - *Undefined stage*, *0* and *1* have to be recognizable visually
  - Navigate through simulation steps for raising and falling clock edges
  - Start simulation of the next *wave* and continue exploring simulation
- Exploration
  - Search by name function for nodes
  - Highlight a node and all incoming and outgoing connections

- Recognize latches (flipflops) visually
- Recognize clock node visually
- Open large BLIF files in reasonable time

4.3 Basic Design

In order to connect Odin II and the visualization software two designs were created. The one with the better performance is determined in Chapter 6.3. The initial application design, which was presented in [20] was changed by adding a new module which includes Odin II. This module is addressed after the BLIF file is processed by the BLIF explorer and the current circuit is created within the application. Odin II creates its own netlist and those two structures are connected by an *Odin interface* using the unique names of the nodes.

The second design utilizes Odin II for all functions regarding the circuit except for the visualization itself. This means parts of the initial application are not used any more but are provided by Odin II. The implementation and and selection of the more efficient approach is described in chapters 5 and 6.

4.4 Think Aloud Evaluation Concept

The usability evaluation will be performed using the *think aloud test* [23]. Five persons will be asked to participate in the test and to solve a number of tasks which will cover the use cases listed in Section 4.2. The list of tasks will be the same for all users. The defined list is shown below:

1. Start the application
2. Open a BLIF file
3. Zoom out to 50%
4. Search for the `top^clock` node using the search function

5. Locate the search result
6. Reset the highlighting
7. Pick a node in the circuit and highlight it using the *Highlight connections* function
8. Start the simulation
9. Navigate through the simulation
10. Close the application

5 Implementation

This chapter describes the implementation process in detail. It explains the software engineering part of the program as well as the specifics of the implementation, problems which occurred and algorithm that were used during this process.

5.1 Software Engineering

This project extends the class structure which was described in [20]. In the first step the structure was refactored to split up functions which were not necessarily in one class object. This caused the creation of the class `BlifParser` which is responsible for BLIF file handling. The class contains functions that return the circuit to the `Container` class. The `Container` uses this information to create all logic units and their connections.

As described in Chapter 4 the integration of the Odin II simulator was performed in two different ways. The class structure of those approaches is slightly different. The first approach is shown in Figure 5.1. The second approach uses the same structure with slight changes (Figure 5.2).

Both approaches have a new class called `OdinInterface` which establishes the communication with Odin II. It provides and controls the functions which are called in Odin II, requests information and returns references to nodes in the netlist. The functions in this class are different for two approaches as they are used for different purposes.

The first approach uses the extracted class to read in the BLIF file and connects the circuit created, to the netlist in Odin II. That means the BLIF file is read in by using this class before Odin II processes the file and references to all nodes in the circuit are requested by the `Controller` class.

The second approach does not require the `BlifParser` class. It uses the

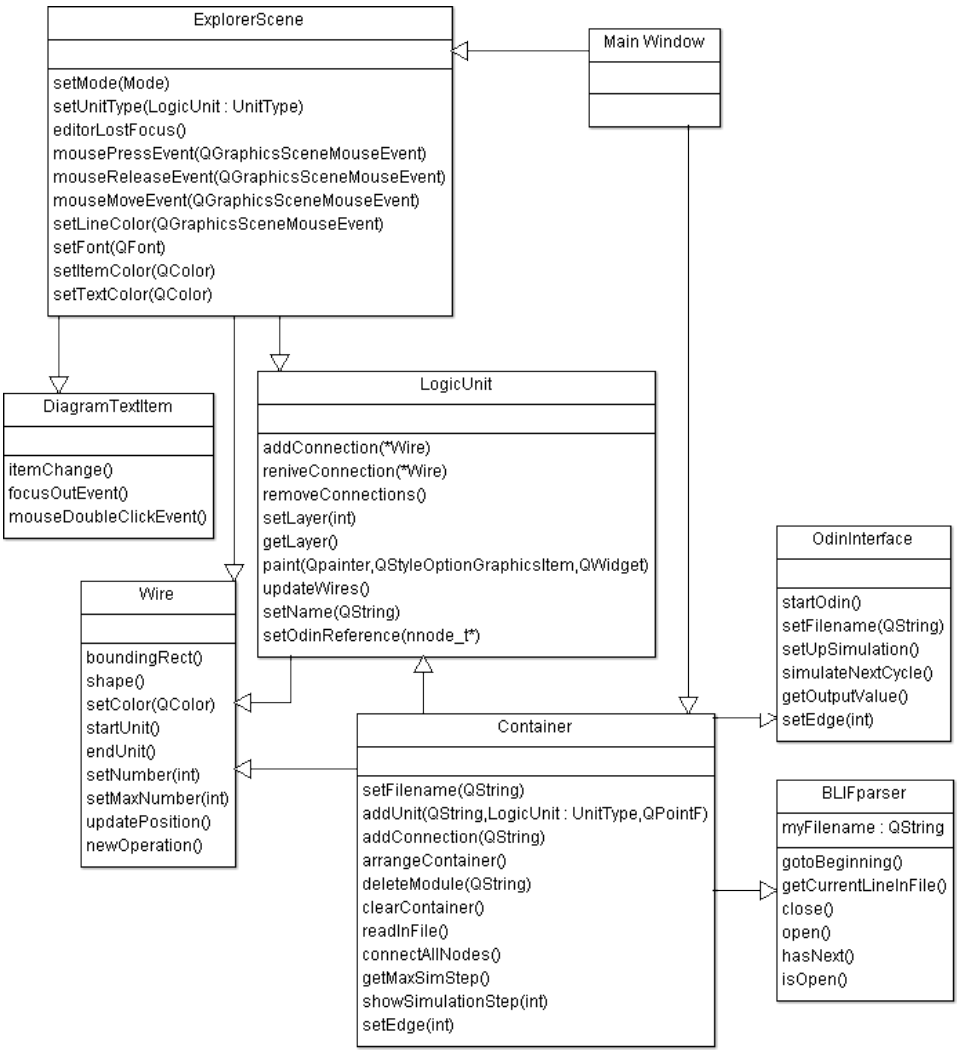

Figure 5.1: Class Diagram of the First Approach

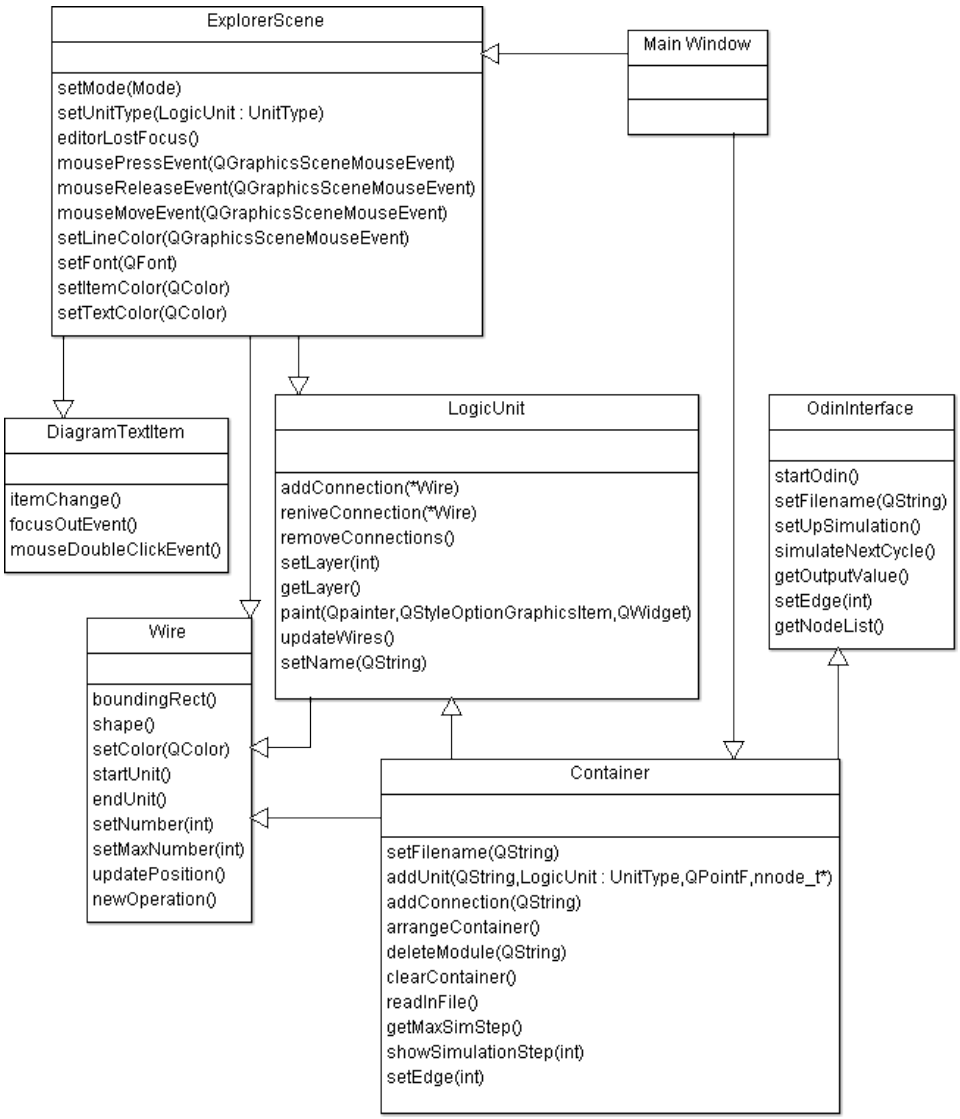

Figure 5.2: Class Diagram of the Second Approach

read BLIF function of Odin II to create the netlist and creates all visual objects according to Odin II's netlist. This causes the `OdinInterface` and `Container` classes to find all nodes in the netlist and to create a visual object for all nodes and connections depending on their type and functionality. The benefits and disadvantages are discussed in Section 5.2 and Chapter 6.

5.2 Visual Simulation

The implementation of the visual simulation was divided into substeps. The steps were created in small packages which could be tested and debugged as easily as possible to make sure the development of a new component will not suffer from previous mistakes. The steps were the following:

1. Refactor the `Container` class.
2. Adjust the classes `LogicUnit` and `Wire` to support simulation.
3. Create a class `OdinInterface` and connect it to Odin II.
4. Adjust `LogicUnit` to store an Odin II node reference.
5. Connect Odin II netlist and the circuit stored in the visualization.
6. Simulate circuits in Odin II using `OdinInterface`.
7. Access actual state of the simulated circuit and visualize it.
8. Provide implemented functions in the BLIF explorer.

The class `Container` which stores all nodes and connections is also responsible for the creation and orchestration of all processes that involve the visualized circuit. A BLIF file was used initially to be parsed in by the `readInFile()` method and to create the visual structure.This function was refactored to be better readable and more scalable. The `BlifParser` class was created to read the file line by line and return information about the logic units and their interconnections.

This structure allows developers not only to better navigate in the source code but also implement different file structures which might be interesting to use as input files for the BLIF explorer. Such an example is a Verilog HDL file with an architecture file as used in Odin II. The interface, which is defined by the class `BlifParser` can be used in order to read in new file structures.

The status of each node in the circuit is saved in a `LogicUnit` object. Each regular node in the netlist and the BLIF file structure has exactly one output, which is named by the node itself. This means the output value which is received by all following nodes can be stored in one place and all outgoing Wires only have to check which status it is at the moment. Nevertheless the BLIF format also allows hard blocks, which can have multiple outputs with different names. To allow such a structure, the output status variable is a vector of values.

For cases representing the soft logic of the circuit, the vector has the length `1`. Only in case of a hard block with multiple outputs is the status vector bigger. The class `LogicUnit` therefore was given one new variable and two new methods which set and get the required values:

```
public:
    int getOutValue(int outputNumber);
     int setOutValue(int value, int outputNumber);
private:
QList<int> myOutVal;
[...]
```

The visualization of the output status uses the width of the connection wires. A `1` is thicker than a `0`. The status is updated using the `updateWire()` method. This method is called by two different components in the application. The `ExplorerScene` detects mouse events and user inputs. If the zoom level is changed and everything is scaled down, the wires are updated. To increase the performance of the application `ExplorerScene`, which inherits `QGraphicsScene`, only updates the actually visible part of the visualized

graph.

A `LogicUnit` is notified by `ExplorerScene` if the user drags it, changes wires etc. The `Container` object notifies it every time a simulation step is computed by Odin II. Each of those cases requires the `LogicUnit` not only to store the new data but also to notify it's outgoing connections to update themselves.

The changes for simulation purposes in the `Wire` header class are the following:

```
//wire.h
public:
void updateWireStatus();

[...]
private:
int wireOff;
    int wireOn;

[...]
```

The integer variables `wireOn` and `wireOff` store the width of the wires during the status on and off. The values are initialized in the constructor and are the same for all wires. If the difference is not visible enough it can be changed. The default values are `2` and `4` which means wires that are currently representing a `1` are twice as thick as those representing a `0`. The initial value for all outputs and inputs in the netlist that are processed by the Odin II simulator is `-1`. In this case the function `updateWireStatus()` visualizes the status by setting the green value of the connection to the maximal value `255`.

This is useful during the first cycles as green connections are still available and structures in the circuit need some cycles to establish defined values. Flip-flops for example need a defined value and a clock edge in order to have

a defined output value.Obviously the number of cycles is increased if more flip-flops exist in different stages of the circuit.

In order to include and use functions that are provided by Odin II an interface had to be created which establishes communication with Odin II and provides all functions that are needed by the `Container` class. Odin II is written in `C` and the visualization software is written in `C++`. In general both languages are related and it is possible to use most source code that is written in `C` in a `C++` application. The precompiled libraries have to be included in the Qt project file which contains all compiler instructions. In addition the VPR6.0 library is also needed for some functions in Odin II and is included together with the 20 precompiled Odin II files. The list of those files can be found in the appendix.

The use of `C` code in a `C++` environment can be done by including the header files of the required source code. It is possible to inform the compiler and the developers that are not familiar with the source code that a different programming language is used by adding the `extern` keyword. The class `OdinInterface` uses it to include all Odin II headers. An excerpt of this include procedure is shown here:

```
extern "C" {
    #include "globals.h"
    #include "types.h"
    #include "util.h"
    #include "netlist_utils.h"
    #include "arch_types.h"
    [...]
}
```

The only file source file of Odin II which is not included is `odin_ii.c` as it contains a main method of Odin II. This would cause collision with the main function of the visualization software. All initialization procedures which are

performed in `odin_ii.c` are therefore executed in the `OdinInterface`.

The basic structure of the class `OdinInterface` can be found here:

```
public:
    OdinInterface();
    void startOdin();
    void setFilename(QString filename);
private:
void initialize_options();
    void do_high_level_synthesis();
    void do_simulation_of_netlist();
    t_type_descriptor* type_descriptors;
    int block_tag;
    QHash<QString, nnode_t *> nodehash;
    QQueue<nnode_t *> nodequeue;
    QString myFilename;
```

The method structure of `odin_ii.c` is taken and changed to fit the requirements in the visualization software. The naming of the main method which is changed to be `startOdin()` and not necessary functions, needed such as command line interactions with the user are not included as this information is given by the visualization software. This approach has the advantage, that no source code file of Odin II has to be changed in order to use the visualization software. The folder structure is completely separated from Odin II and only uses the provided functions. The disadvantage is that any changes that are made to `odin_ii.c` also have to be adjusted in the class `OdinInterface`.

The connection of the netlist in Odin II and the visualization graph is done using references that are stored in the `LogicUnit` objects.

```
public:
```

```
void setOdinRef(nnode_t* odinNode);
    nnode_t* getOdinRef();
    bool hasOdinReference();
private:
nnode_t *myOdinNode;
    bool hasOdinNode;
```

This is used to assign a connection between the structures and to receive information about the Odin II node such as type, output value, incoming and outgoing connections. The connection process was implemented in two different ways as already described in previous chapters.

The first approach uses the existing structure and adds a new module and functions to it. The flow of the open BLIF process is extended by an Odin II step as shown in Figure 5.3. In order to accomplish this, the class `OdinInterface` has a method to create a hash table of all nodes in the netlist. This information is passed to the `Container` which connects all nodes based on unique names.

The second approach changes the flow of the application. It requests the Odin II netlist before nodes and connections are created.The nodes are created by traversing the netlist based on it's type and child nodes. The flow of the second approach is shown in Figure 5.4.

The traversal algorithm used by the second approach is required to make sure that every node in the netlist is created in the visualization graph. A similar algorithm is used in the simulator to compute the values of every node. It starts with the inputs of the circuit and moves on to the child nodes computing the values based on the new inputs. A node needs all input values to be computed for the actual cycle in order to be created.

The parallel requirement for the visualization is that two nodes which have to be connected have to be existent in order to create the `Wire` object. A

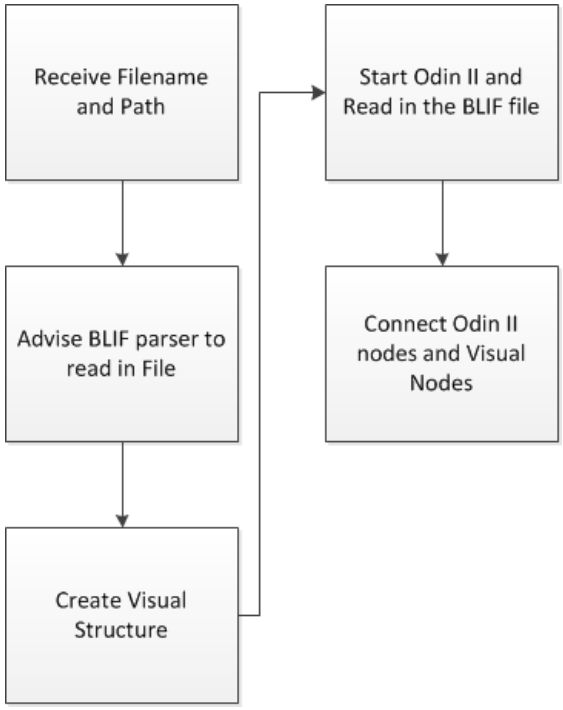

Figure 5.3: Open File by BLIF Parser Flow Diagram

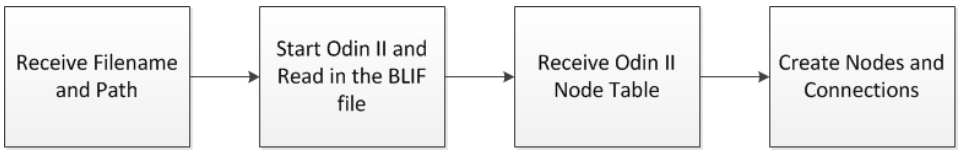

Figure 5.4: Create Visualization Using Odin II Open File Flow Diagram

problem that is found in this procedure are feedback structures as shown in Figure 5.5.

The logic node `AND1` will not be added until `FLIPFLOP1` is available in the

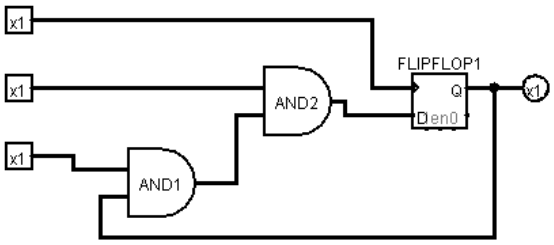

Figure 5.5: Feedback Loop in a Boolean Circuit

structure. Also `FLIPFLOP1` cannot be added as long as `AND2` is waiting for `AND1`. This deadlock is handled using the specification in the BLIF file format that every "feedback loop must contain at least one latch [17]". As `FLIPFLOP1` has only two inputs, one for the clock and one d-in, the flipflop can be added as soon as it is reached from a node which is not the clock. The clock is added in the first iteration and can be assumed as given. Based on this information the pseudo code version of the traversing algorithm is shown below.

```
createNodes(){
    enqueueTopInputNodes(nodequeue);
    enqueueConstantNodes(nodequeue);

    while(!nodequeue.isEmpty()){
        node = nodequeue.dequeue();
        create(node);
        doneTable.add(node);
        childlist = getChildren(node);

        forall(child in childList){
            if(doneTable.contains(child)){
                addConnection(node, child);
```

```
            }
            else if(!inQueue(child) &&
            !parentsDone(node)){
                nodequeue.enqueue(child);
            }
        }
    }
}
```

The next step towards visualizing the simulation is to run the Odin II simulator on the netlist and to store the values in the nodes. The simulation is called using the globally visible method `simulate_netlist(verilog_netlist)`. The simulator computes the values in waves. A simulation of multiple cycles is done at once. The number of cycles is defined in the `Makefile` of Odin II and is 16 per default.

The visualization software copies those 16 cycles after the computation to the `LogicUnit` objects as an array of `short` variables. A `bool` variable would not be sufficient as there are three possible states (*high*, *low*, *undefined*). If a cycle number is requested which is above the number of the last computed cycle so far, another wave is computed by the simulator and the values are added to the array. This allows the user to see the simulation result very quickly even for very big circuits as only 16 cycles are computed and additional cycles only in case they are required.

The simulation function is provided to the user in the graphical user interface in a separate area. At the current state of the application the user has three functions: *run simulation*, *next simulation step* and *previous simulation step*. The graphical interface which shows a circuit being simulated and the user simulation controls is shown in Figure 5.6. The example shows a very small circuit with only nine nodes. Examples of a circuit with more than 2000 nodes is shown in the appendix in the Figures B.1 and B.2.

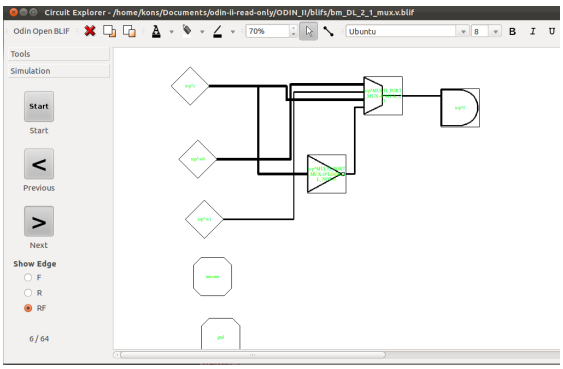

Figure 5.6: Visual Simulation Exploration

5.3 Extending the BLIF Explorer

During the project, developers were able to work with the software and give feedback in form of functions which would be helpful during the exploration process. A wish for better navigation in bigger circuits and a better visual overview of the modules and its placement in the circuit was requested.

New functions were implemented in order to improve and speed up the visualization process. A search function, which allows the user to find specific nodes and to explore their connections, was added. The main menu and the context menu which is shown if the user performs a right mouse click on the visualization offers to find nodes.

In order to perform a search, an interaction menu requests the user to enter the name or part of the name of the node. The algorithm iterates through the node hash table which is stored in the `Container` class. Nodes containing the string are highlighted. The result of an example search is shown in Figure 5.7. In addition, the number of nodes highlighted is presented to the user. In a big circuit the search results can be seen in low zoom levels and zoomed in to inspect the specific nodes.

Another request was to view specific nodes, their connections and neighbours. The right mouse click context menu of the nodes was extended by

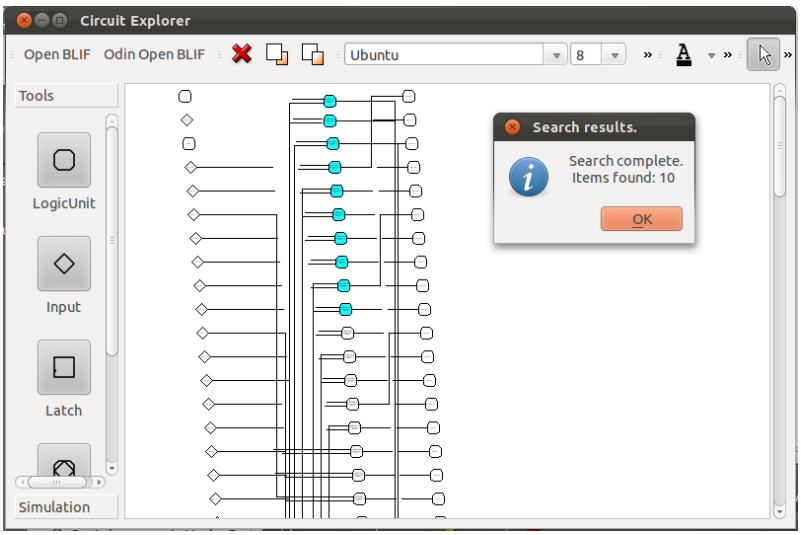

Figure 5.7: Search Result for a Substring in a Circuit With 133 Nodes

the functions *Highlight node* and *Reset highlighting*. If a node is highlighted, the `LogicUnit` and its connections are painted red. The layer of those objects is set to a high value which shows the connection wires above nodes, which are crossed. This allows developers to see and follow all connections a node has and the neighbour structure can be explored. The highlighting is persistent during a possible simulation process. If the value of the connection is currently not defined, only the green value is changed and the connection is shown in yellow until it is *high* or *low* which changes the connection color back to red. One of the first cycles of a simulation while a node is highlighted is shown in Figure 5.8.

In addition nine new shape types were included into the simulation which allow to recognize *clock*,*latch*,*logical AND*, *logical NAND*, *logical OR*, *logical NOR*, *logical XOR*, *logical XNOR* and *logical NOT* nodes. Also the shape of a standard node without a specific shape was changed to show a more recognizable difference to latch nodes. It is not required any more to create a node shape by using way points, which build a polygon but can be added using an image of the shape on transparent background. The currently included shapes are shown in Figure 5.9.

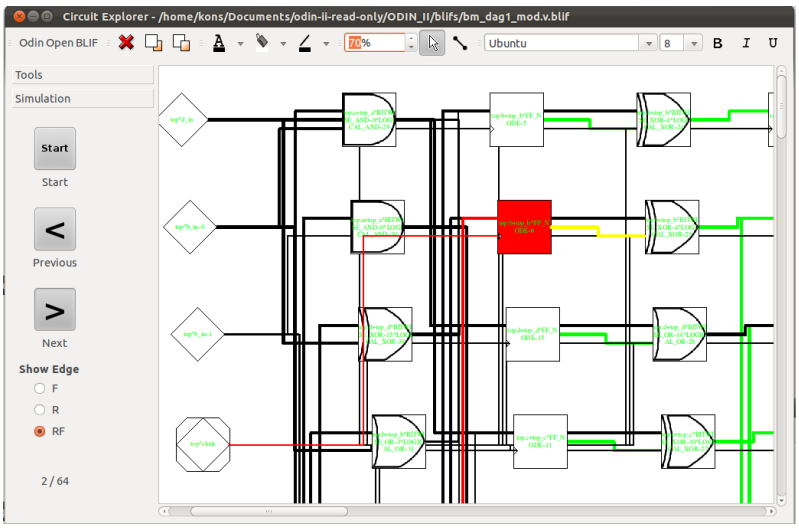

Figure 5.8: Highlighting of a Node During the Simulation Process(green=*undefined*, yellow=*undefined and highlighted*, red=*defined and highlighted*, black=*defined*)

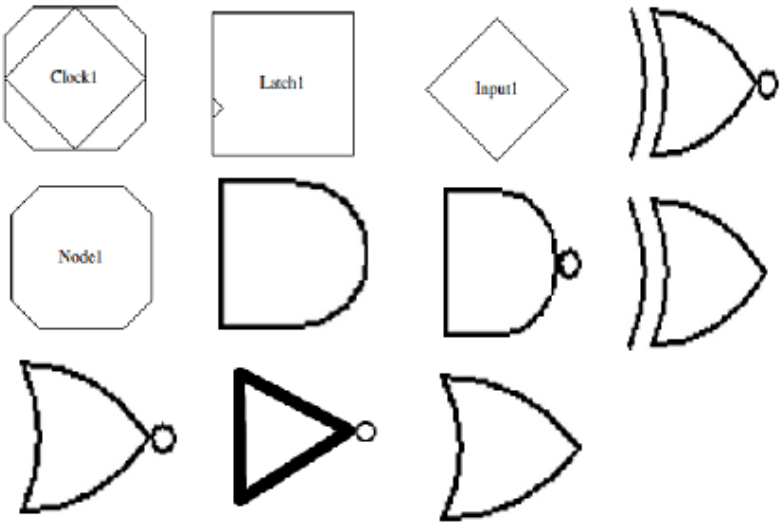

Figure 5.9: Currently Implemented Shapes in the Vizualization

6 Evaluation

This chapter applies the evaluation strategies described in 4 in order to find out how well the problems defined in 4.1 were soved. The evaluation is divided into three subsections. The first subsection analyses the use cases which were defined before the implementation started. The goal of this evaluation is to find out if the defined goals were met, to discuss advantages and disadvantages of the approaches and to point out what kind of development could be done in the future.

The second subsection discusses two approaches which were implemented in order to extend the visualization software by simulation abilities. The third section will evaluate the overall usability by discussing results of a *think aloud* test which was created in Chapter 4 and performed by five different persons. sections.

6.1 Use Case Based Evaluation

Perform simulation of the currently viewed circuit

The BLIF file is opened in Odin II and the visualization software. The nodes are connected and the simulation can be started using a button in the simulation tool section. The user is still able to use the functions that are provided by the exploration tool. The nodes can still be dragged around, highlighted and comments can be added.

In case the user modifies the circuit within the application, the BLIF file remains the same and also the netlist in Odin II is still the same, which was created when the BLIF file was opened. All nodes, that were in the circuit before it was modified will still show their status. The new added structures do not have actual functions.

See current states of outputs in the visualization, *Undefined stage*, *0* and *1* have to be recognizable visually

Wired connections were extended by parameters which influence the width and color of the visual representation which allows the user to see the current state of the node output. All states are represented visually.

Navigate through simulation steps for raising and falling clock edges

The simulation shows both clock edges. In circuits without a clock it causes the circuit to stay in the same position for two steps. The edges, which have to be simulated can be chosen by the user. The simulation toolbox contains three radio buttons, which determine which edges are shown.

Start simulation of the next *wave* and continue exploring simulation

The simulation is started as soon as the user presses the *Start* button. The first wave is simulated and the values are stored. The number of values depends on the wave length which is defined in the Odin II `Makefile`.

As soon as the last cycle was shown and the user clicks the *Next* button the next wave is simulated and the user has one wave length more cycles which can be explored.

Search by name function for nodes

A search was implemented. The user can use the function by pressing the hot key `Ctrl+F`, using the `Item` menu option or the right click mouse context menu to access it. The information which is required for the search is the name or a substring of the node. All nodes, which match the name are highlighted light blue in order to locate them in higher zoom levels. The highlighting can be undone using the *Reset Highlighting* function in the menu.

Highlight a node and all incoming and outgoing connections

Using the right click context menu a node and all incoming and outgoing connections are highlighted red and the layer of each of the objects is set to a high value. Connections which cross other nodes are drawn above them and can be tracked in the circuit. If the highlighting is not needed any more the *Reset Highlighting* function restores the initial color and layer values.

Recognize latches (flipflops) clock node and other logical gates visually

The `LogicUnit` class was extended by new types of nodes. All types were created with standard shapes which allow to see the logical structure of the circuit without reading the names of the nodes. The currently available types are:

- Input
- Clock
- Latch
- Logical AND
- Logical NAND
- Logical OR
- Logical NOR
- Logical XOR
- Logical XNOR
- Logical NOT

Shapes are drawn using way points or images with a transparent background which allows to add complex shapes or change them without accessing the source code. All visual shapes are shown in Figure 5.9.

Node count	Connection count	App. 1 time	App. 2 time
26	43	0.01s	0.01s
66	139	0.01s	0.01s
708	1676	0.04s	0.05s
918	2150	0.05s	0.1s
1537	3692	0.12s	0.26s
2328	5872	0.15s	0.34s
3354	8654	0.26s	0.51s
5027	12900	0.43s	1.01s
10876	35814	4.12s	8.07s
23757	96733	23.76s	65.3s

Table 1: File Opening Times Using Both Approaches

Open large BLIF files in reasonable time

Two different approaches were implemented which handle the reading of an input file. The difference of the opening times is part of the discussion in Section 6.2. The evaluation of the times was performed on a machine with an Intel Core i7 processor and 4GB of RAM.

Table 1 shows that approach 1 which uses Odin II to read in the BLIF file performs faster than to open the file using the `BlifParser`. This can be explained by the fact that the second approach uses the file multiple times. The file is opened and parsed a first time to create all nodes, which are found in the BLIF file. It is parsed a second time to create all interconnections between the files before Odin II opens it in order to create the netlist with all node functionalities. The first approach uses the file only once when Odin II is started. Based on the netlist the visual representations are created. A visual comparison between the times is found in Figures 6.1 and 6.2.

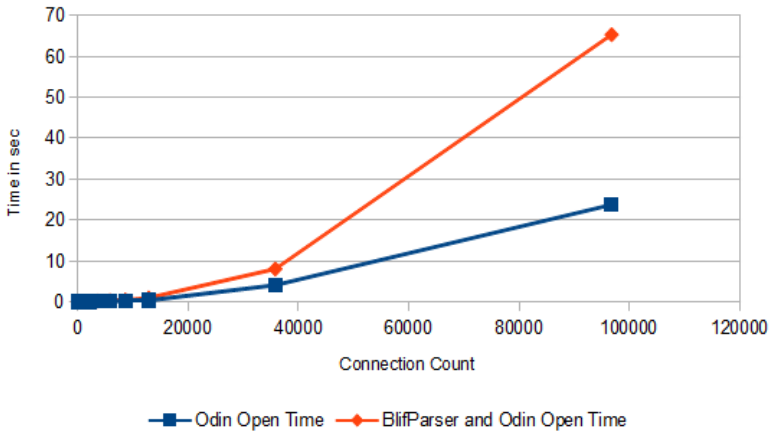

Figure 6.1: Open BLIF Times in Relation to the Connection Count

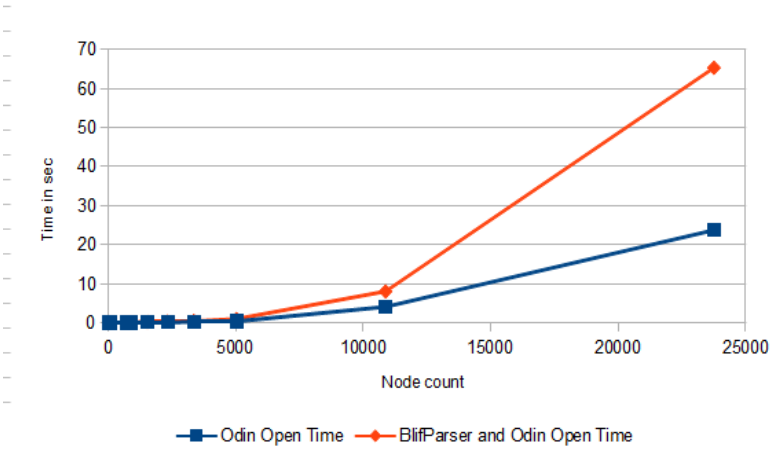

Figure 6.2: Open BLIF Times in Relation to the Node Count

6.2 Approach Discussion

The two approaches which were implemented in Chapter 5 have the same result visually, but the structure and some metrics differ. The flow of the file handling differs as one approach uses a module, which was already existent in the application whereas the other approach uses a completely new flow, which discards the file interface. The benefit of having an interface which is specifically created for the visualization tool is that new file formats can be recognized if a new file handler is created which implements the interface.

On the other side the visualization tool is created especially for the workflow where Odin II is part of and Odin II is already capable of opening output files from every stage in the workflow and in addition it can open Verilog HDL files in combination with architecture files, which represent a specific FPGA architecture. This functionality of Odin II creates a netlist which can be used in the visualization software to create a visual circuit.

The file handling of the approach, which abstains from using the `BlifParser` has the benefit that the file is only read in by Odin II whereas it is read by the visualization twice before Odin II is started and has to create the netlist based on the BLIF file as well. This also reflects in the file opening times which are shown in Table 1 and the Figures 6.1 and 6.2.

In comparison the approach which is not using the file multiple times has smaller growth rates in relation to the connection count as well as the node count. In case of a change in the BLIF standard the second approach benefits from the fact that only the read in function of Odin II has to be adapted in order to apply the changes also for the visualization.

Table 6.3 lists benefits and detriments of both approaches which leads to the decision to discard the file interface and only use the approach which utilizes Odin II.

	Approach 1	Approach 2	Favoured Approach
Extendability for new formats	Create a new file reader	Adjust Odin II interface to be able to use more of Odin II's functions	2
File opening times	Reads the file 3 times, slower than Appr. 2	Reads the file once	2
Adjusting for new functionality in file format	In the visualization and in Odin II	Only in Odin II	2
Open Verilog and architecture files	would have to use Odin II to create a BLIF first	Could use Odin II netlist	2

Figure 6.3: Comparison Table Between the two Approaches

6.3 Usability Evaluation

For the usability evaluation a think aloud test was created which was performed with five persons who could potentially use it. All of the five persons are students in Computer Science. The test was created during the design phase and a list of tasks was defined in 4. Each of the testers was asked to complete the tasks and to say everything they do and think about the application out loudly. The task list which was completed by the users can be found in Appendix C.

Opening the file using the hot key and using the button on the main button group was found by all users very quickly. Dragging around nodes and arranging them as desired was also done by all users intuitively. Highlighting a node and all connected nodes was not found by all users initially. Two users tried to find this function in the menus or in the tool bar on the left side, which allow to add new nodes into the circuit. After the function was not found on the tool bar the users tried to right click a node and find the function. The comments on this function were that it is easier to see the connections as they are shown above nodes, which are not connected to the highlighted one.

The task of searching for a sub-string which is included in some of the names was found after searching by all users. No user used the hot key but

tried to find the search function in the menus. The function was not found at first sight and it was commented that the naming of the function in the menus was not the name they looked for. Another name of the function and a better placement would help to find it faster.

The simulation was started by all users without problems. Each user clicked the *Start* button before the *Next* button was used. The *low* and *high* values of the connection were recognized as such but the *undefined* status was not. A user proposed to include a legend into the simulation tool bar which explains the coloring and the width changes of the wired connections.

7 Conclusion and Future Work

This report presented the improvement of the visualization software in Odin II by additional features in order to assist the exploration process and the development of visual simulation using the Odin II simulator. The report provided a short introduction to FPGA, GUI development and presented the VTR project as related work. By offering the visual simulation functionality, the visualization software can be used not only for exploration but also for evaluation and verification purposes.

The usability of the application was improved by adding comfort functions such as *highlight all connections* or a search function which allows to find nodes using a sub string of its name. Shapes of simple logical functions were added into the simulation to recognizing the structure of the visualized circuit without the need to see the name of the node.

Future development of the visualization are focused on improving the VTR workflow. Abilities of Odin II which are not used at the current state such as opening Verilog HDL files in combination with a hardware architecture file and to visualize the result of the Odin II compilation will be added.

Power consumption based on activity estimation is part of future development in the VTR CAD flow. Using the simulation, activity of outputs can be estimated to compute three statistical values which are required for activity estimation: static probability, switching probability and switching activity. Using these values power consumption can be approximated and visualized in the graph. Hotspots in the circuit can be found by developers and causes addressed and considered in future development of the workflow.

All stages of the VTR workflow will be added to the visualization in order to explore the changing structure during all stages in the workflow. This allow developers to back track hotspots or other structures, which are not desired. Once the source of this development is found the tool can be improved to address this issue. Being able to visualize the circuit after plavement and

routing by VTR timing on the FPGA device can be computed and metrics such as longest route could be determined and visualized.

References

[1] D. Thomas and P. Moorby, *The Verilog hardware description language*. Springer Netherlands, 2002.

[2] P. Ashenden, *The designer's guide to VHDL*. Morgan Kaufmann, 2008.

[3] J. Rose, J. Luu, C. Yu, O. Densmore, J. Goeders, A. Somerville, K. Kent, P. Jamieson, and J. Anderson, "The vtr project: architecture and cad for fpgas from verilog to routing," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 77–86, ACM, 2012.

[4] S. D. Brown, *Field programmable gate arrays*. Kluwer Academic, 1997.

[5] P. Chow, J. Rose, K. Chung, G. Paez-Monzon, and I. Rahardja, "The design of an SRAM-based field-programmable gate array. I. Architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, pp. 191–197, June 1999.

[6] H. Schmit, "Extra-dimensional island-style FPGAs," *New Algorithms, Architectures and Applications for Reconfigurable Computing*, pp. 3–13, 2005.

[7] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2007.

[8] P. Leong, W. Luk, and S. Wilton, "Floating-Point FPGA: Architecture and Modeling," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, pp. 1709–1718, Dec. 2009.

[9] M. Summerfield, "Rapid GUI programming with Python and Qt: the definitive guide to PyQt programming," *October*, 2008.

[10] J. Blanchette and M. Summerfield, *C++ GUI programming with Qt 4*. Prentice Hall PTR, 2006.

[11] T. Sommer, "Physics for a 3D Driving Simulator Torsten Sommer Bachelor Thesis," *Physics*.

[12] A. Sharma, "White Paper Merits of QT for developing Imaging Applications UI," *System*, pp. 1–8, 2008.

[13] Nokia Corporation, "Qt Reference Documentation." `http://doc.qt.nokia.com/4.7-snapshot/index.html`, Mar. 2012.

[14] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research," *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 149–156, 2010.

[15] A. Mishchenko, "ABC: A System for Sequential Synthesis and Verification." `http://www.eecs.berkeley.edu/~alanmi/abc/`, 2011.

[16] J. Luu, J. Anderson, and J. Rose, "Architecture description and packing for logic blocks with hierarchy, modes and complex interconnect," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 227–236, ACM, 2011.

[17] U. Berkeley, "Berkeley logic interchange format," tech. rep., Technical report, Technical report, University of California at Berkeley, Aug. 1998.

[18] P. Jamieson and J. Rose, "A verilog RTL synthesis tool for heterogeneous FPGAs," in *Field Programmable Logic and Applications, 2005. International Conference on*, pp. 305–310, IEEE, 2005.

[19] J. Libby, A. Furrow, P. O'Brien, and K. Kent, "A framework for verifying functional correctness in odin ii," in *Field-Programmable Technology (FPT), 2011 International Conference on*, pp. 1–6, IEEE, 2011.

[20] K. Nasartschuk, "Visualization support for fpga architecture exploration," tech. rep., Technical report, University of New Brunswick, 2011.

[21] V. Betz and J. Rose, "VPR : A New Packing , Placement and Routing Tool for," *Technology*, pp. 1–10, 1997.

[22] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, and W. M. Fang, "VPR 5 . 0 : FPGA CAD and Architecture Exploration Tools with Single-Driver Routing , Heterogeneity and Process Scaling," *Computer Engineering*, pp. 133–142, 2009.

[23] D. M. Turner-Bowker, R. N. Saris-Baglama, K. J. Smith, M. a. DeRosa, C. a. Paulsen, and S. J. Hogue, "Heuristic evaluation of user interfaces," *Telemedicine journal and e-health : the official journal of the American Telemedicine Association*, vol. 17, no. 1, pp. 40–5, 1990.

[24] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer, "An automated exploration framework for FPGA-based soft multiprocessor systems," *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '05*, p. 273, 2005.

A Appendix

Precompiled files of Odin II and VPR6.0, which are included in to Qt project file:

```
../ODIN_II/OBJ/netlist_visualizer.o \
../ODIN_II/OBJ/multipliers.o \
../ODIN_II/OBJ/partial_map.o \
../ODIN_II/OBJ/errors.o \
../ODIN_II/OBJ/node_creation_library.o \
../ODIN_II/OBJ/netlist_utils.o \
../ODIN_II/OBJ/netlist_stats.o \
../ODIN_II/OBJ/odin_util.o \
../ODIN_II/OBJ/string_cache.o \
../ODIN_II/OBJ/hard_blocks.o \
../ODIN_II/OBJ/memories.o \
../ODIN_II/OBJ/queue.o \
../ODIN_II/OBJ/hashtable.o \
../ODIN_II/OBJ/simulate_blif.o\
../ODIN_II/OBJ/print_netlist.o \
../ODIN_II/OBJ/read_xml_config_file.o \
../ODIN_II/OBJ/outputs.o \
../ODIN_II/OBJ/parse_making_ast.o \
../ODIN_II/OBJ/ast_util.o \
../ODIN_II/OBJ/high_level_data.o \
../ODIN_II/OBJ/ast_optimizations.o \
../ODIN_II/OBJ/netlist_create_from_ast.o \
../ODIN_II/OBJ/netlist_optimizations.o \
../ODIN_II/OBJ/output_blif.o \
../ODIN_II/OBJ/netlist_check.o \
../ODIN_II/OBJ/activity_estimation.o \
../ODIN_II/OBJ/read_netlist.o \
../ODIN_II/OBJ/read_blif.o \
../ODIN_II/OBJ/output_graphcrunch_format.o \
../ODIN_II/OBJ/verilog_preprocessor.o \
```

```
../ODIN_II/OBJ/verilog_bison.o \
../ODIN_II/OBJ/verilog_flex.o \
../libvpr_6/ezxml.o\
../libvpr_6/read_xml_arch_file.o\
../libvpr_6/read_xml_util.o\
../libvpr_6/util.o\
../libvpr_6/ReadLine.o
```

B Visualization Software Screenshots

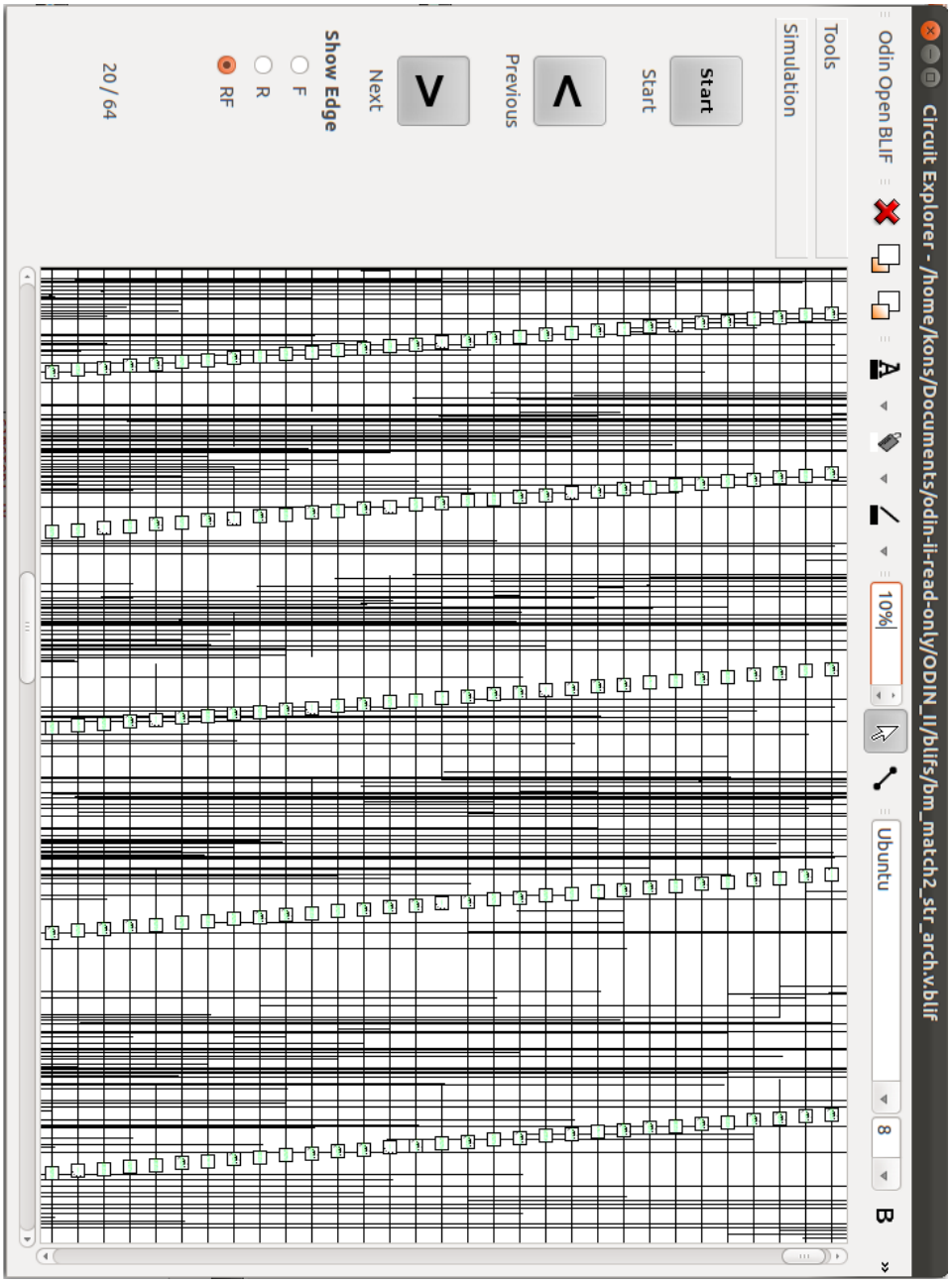

Figure B.1: Simulation of a Circuit with 3354 Nodes. Zoom Level: 10%

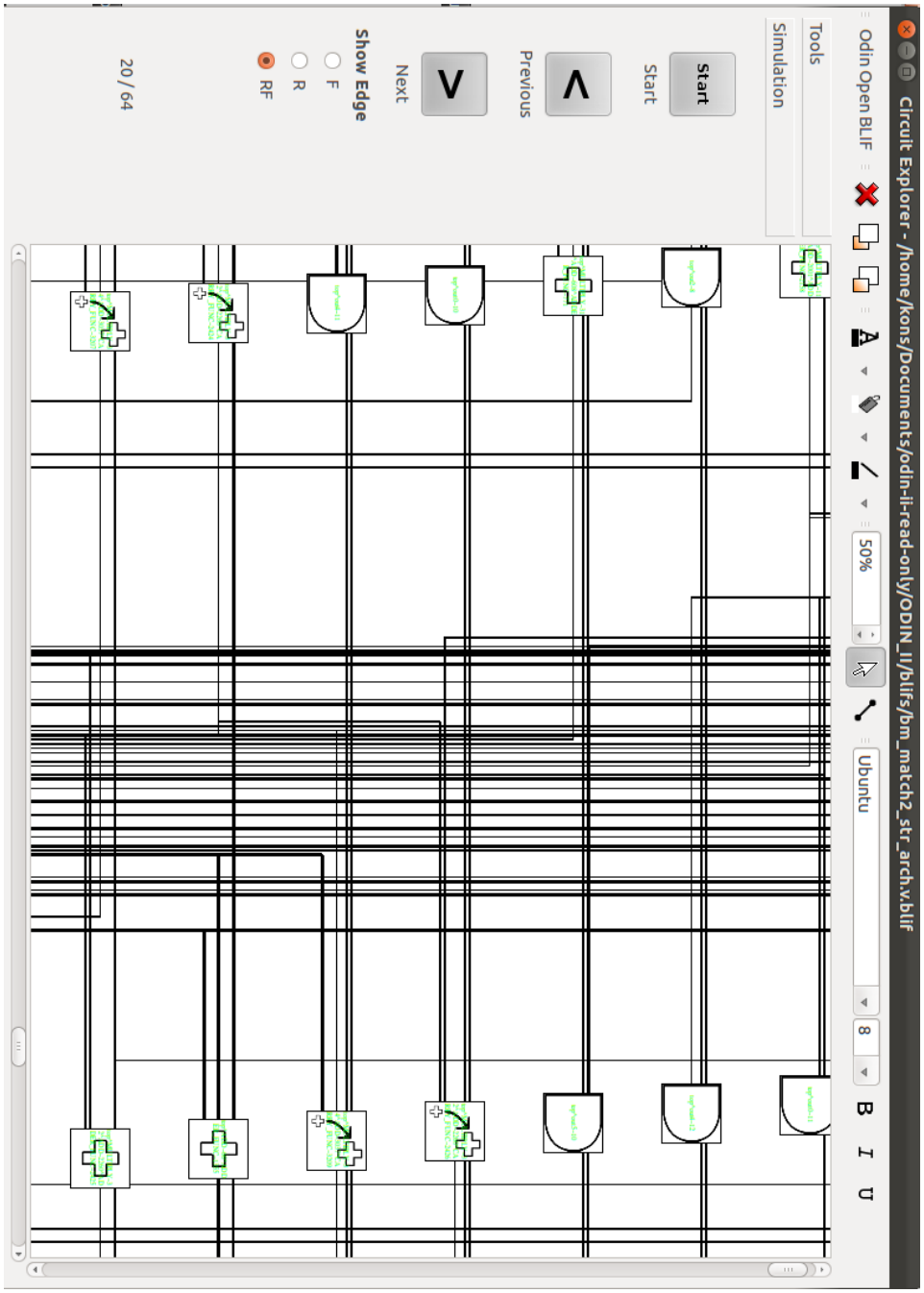

Figure B.2: Simulation of a Circuit with 3354 Nodes. Zoom Level: 50%

C Think Aloud Evaluation Task List

- Start the application.
- Open a BLIF file from the evaluation set.
- Change zoom level to 50%
- Simulate the visualized BLIF
- Step to cycle 17
- Explain what happens for one node
- Use the search function to find node "topĉlock"
- Reset the highlighting
- Highlight a specific node with all its connections
- Reset highlighting
- Close the application