Development of a VHDL Generator for Scheduled Data Flow Graphs

MASTER THESIS

submitted by

HAGEN STÜBING

Honour declaration

Hereby I assure that the presented thesis was made without any help of third persons and only with the indicated sources and means. All the figures and paragraphs taken from the sources are clearly marked.

Darmstadt, 21.11.2006

Preface

The presented work is created in contents of a master thesis that I have prepared for the "Universitat Politecnica de Catalunya" in Spain. At the "Technical University of Darmstadt" this thesis is submitted as a "Studienarbeit".

The thesis consists of three parts. The enclosed CD-Rom contains the VHDLGenerator program with the corresponding installation files as well as the source code. The documentation of the VHDLGenerator Program is the paper at hand. For quick reference a user manual is available. The user manual is a copy of the most important notes, helpful when starting to work with VHDLGenerator.

The structure of this thesis is closely related to the execution order of the program. The development of the VHDLGenerator program required knowledge of different fields of programming. For streaming the text files you have to know the Java streaming concepts, while the VHDLGenerator error detection is based on Java exception handling. Because the output file is written in a hardware description language (VHDL) one has to deal with the constraints of hardware design. Furthermore the theoretical background of minimization algorithms has to be understood in order to realize the Left Edge Algorithm that is used for register minimization. So as heterogeneous as the knowledge base is, as heterogeneous is the structure of this thesis. The different topics are addressed in the order they are executed inside the VHDLGenerator program.

Every chapter is related to one working step of VHDLGenerator. It always starts with a general explanation and theoretical background of a used concept and then explains how the theory is adapted to the specific case.

Hagen Stübing

Contents

List of Figures

1 Introduction

Common hardware design flows that are in use today, can not satisfy the correctness requirements of cryptographic applications, where an implementation error might violate the security. For this reason the *Faculty of Integrated Circuits and Systems* at the *Technical University of Darmstadt* started to develop a new design flow, appropriate for cryptographic applications. In contents of this work a VHDLGenerator is designed, which is part of that new developed hardware design flow. The objective of this VHDLGenerator is to convert a Data Flow Graph description into synthesizable VHDL code.

A central aspect of the design flow is the desired support for automated verification after code entry. Although it is aimed towards cryptography, most aspects can be adopted to other domains. We will discuss first how the new design flow looks like, and then we explain how this work is related to it. Finally the overall structure is explained.

The proposed design flow is shown in Figure 1.1. The different design phases are represented in the left column. The middle column denotes the actions required in the corresponding design phase, while the right column gives concrete examples for each design phase.

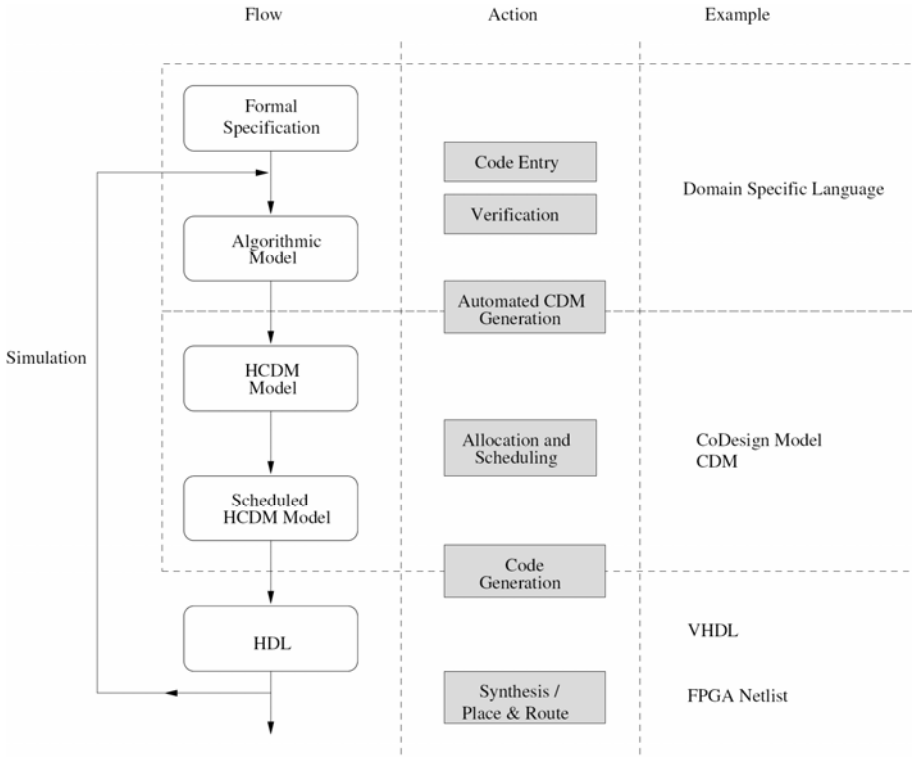

Figure 1: proposed Design Flow

The first step consists of the Code Entry Phase. There the developer generates an algorithmic implementation in the design language. The implementation has to follow the requirements of a formal specification.

An automated verification tool is used to test the implementation against the specification given by the developer.

The next step in the design flow is an automated scheduling and allocation process. For this purpose the HCDM tool is used. The HCDM Generator is a tool that has been created by Stephan Klaus [1] in contents of a dissertation at the Technical University of Darmstadt. HCDM is used to describe data dependencies between different tasks in form of task graphs. Scheduling and resource allocation is done automatically. A more detailed description of the HCDM tool is given in chapter 2.

The last step of the design flow is the Code Generation. Right at this point starts the work of the implemented VHDLGenerator. Its purpose is exactly to parse the results of HCDM, i.e. the data flow graph together with the scheduling information and generate synthesizable VHDL source code.

When starting to design the VHDLGenerator program one has to partition the incidental converting work. It turned out to be of great advantage to introduce three different working stages.

The structure of this thesis mirrors directly the internal stages of the VHDLGenerator program. Composed out of three processing stages, the input is parsed, then pre-processed and finally converted to VHDL code. All three stages are written in Java [24].

The first stage parses the necessary files in order to extract all relevant information. The Parser stage is described in chapter 3.3.

The second stage is the PreConverter stage, described in chapter 3.4. This stage contains the *heavy weighted* methods that do most of the work. Register minimization, Exception handling and the setting of the case-statements are only some tasks that are handled by the PreConverter stage.

The third and final stage converts the pre-processed data to VHDL code. Because of the extensive work of the PreConverter, the methods of this stage are more *light weighted*. The VHDL generator stage is subject of chapter 3.4.

VHDLGenerator offers a fourth optional stage. Depending on the users settings a testbench for the converted VHDL code can be created. How the testbench is constructed is explained in chapter 3.4.2.

The VHDLGenerator usage is greatly simplified by creating a Graphical User Interface (GUI) that is presented in chapter 4.

To demonstrate the correct behaviour of VHDLGenerator a benchmark program is constructed. Chapter 5 tells how the test program and its results look like.

2 The Data Flow Graph Presentation

In the next sections it is explained why we use Data Flow Graphs to describe high level hardware applications. The background of graph theory is explained more in detail, than it would have been necessary for this chapter. But when it comes to explain Register Optimization this additional information will be helpful.

The HCDM tool is introduced as an easy and comfortable way to create task graphs. The basic differences between Task Graphs and Data Flow Graphs are described. It is explained how the Task Graphs, generated by HCDM, are manipulated in order to obtain Data Flow Graphs.

2.1 Data Representation Form

When generating VHDL code it is of great importance to use an appropriate representation form for implementation, that reflects the basic nature of hardware description languages. Hardware implementations allow for example parallel computation. So our chosen data representation form should support parallel structures also. And second, we target dataflow intensive algorithms. While this is a severe limitation in general, for HW implementations it is acceptable. The Control Flow is described implicitly by the presetting for scheduling and allocation inside HCDM.

Furthermore the representation form should be easy to extract by means of a parser.

Given the two criteria of parallel data representation and simplicity of extraction, a graph based description turns out to be the best solution. We will use Data Flow Graphs, which are a subtype of graphs that also allow representing external inputs and outputs. In fact Data Flow Graphs show similar characteristics as digital hardware components. In the DFG a task can only fire if all of its inputs are ready and, the scheduled component is free to process. In the same manner VHDL Components are constructed.

A general description of graph based data representation is following in the next section.

To cover our second goal of extraction simplicity the previous mentioned HCDM tool could be used. This tool allows drawing Task Graphs graphically via a GUI. It

generates automatically a textual graph description, which follows a well-defined grammar that could be used to construct a pattern for the parser. Moreover HCDM also includes timing aspects. It performs scheduling of different tasks on predefined resources. That issue covers the synchronisation aspect of common VHDL design. With respect to these aspects HCDM is considered to be an appropriate tool. The last sections of these paragraphs are dedicated to the HCDM tool.

2.2 Graphs in general

A graph G(V,E) consists of vertices V and its relations E. Vertices are often called nodes. The relations E are the edges between the vertices. Graphs can be categorized into two groups: directed graphs and undirected graphs.

Undirected graphs are graphs that express a slack relation between vertices with temporal order between them. A compatibility graph is a good example for an undirected graph. We will work with compatibility graphs in chapter 3.3.2 in context of register optimization.

A directed graph got a directed dependency between consecutive vertices. In short, a directed graph is an undirected graph but with a temporal order of its vertices. The HCDM Tool uses directed graphs.

Directed graphs can be used to represent procedural languages with imperative semantics. That means we want to express an algorithm with a language that takes care of the execution order. Calculation of one step is therefore based on results of the previous steps, which gives us a temporal order.

The representation of the calculation flow is done graphically in terms of its vertices (the tasks, represented as nodes) and its relations (the dependencies, represented as branches). The direction is indicated by drawing an arrow.

We discuss the differences between two types of sequencing graphs: the task graph and the data flow graph.

2.3 Task Graphs

The Task Graph is the one implemented forhe HCDM Generator. This sequencing graph has only task vertices. That includes, that also the dependencies only relate tasks. Task Graphs have two important characteristics: They are acyclic and polar. Acyclic means that the graph includes a partial order between its tasks. Polar means that it got a source vertex at the beginning and sink vertex at the end of the graph. In the HCDM tool the source vertex and the sink vertex are joined to the Root task.

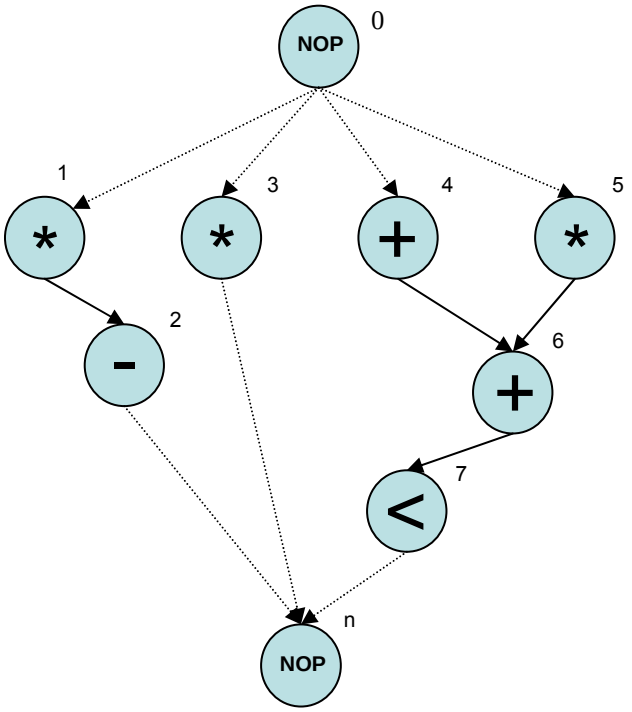

Figure 2: Task Graph example

This task does not implement any function but indicates the begin and the end of the graph. This fact will be important later when it comes to parsing of the tasks.

All the initial tasks are successors of the Root Task and all final tasks are predecessors of the Root Task. Figure 2.1 shows an example of a task graph.

Corresponding Tasks graphs designed by HCDM could look like the example in Figure 2.2.

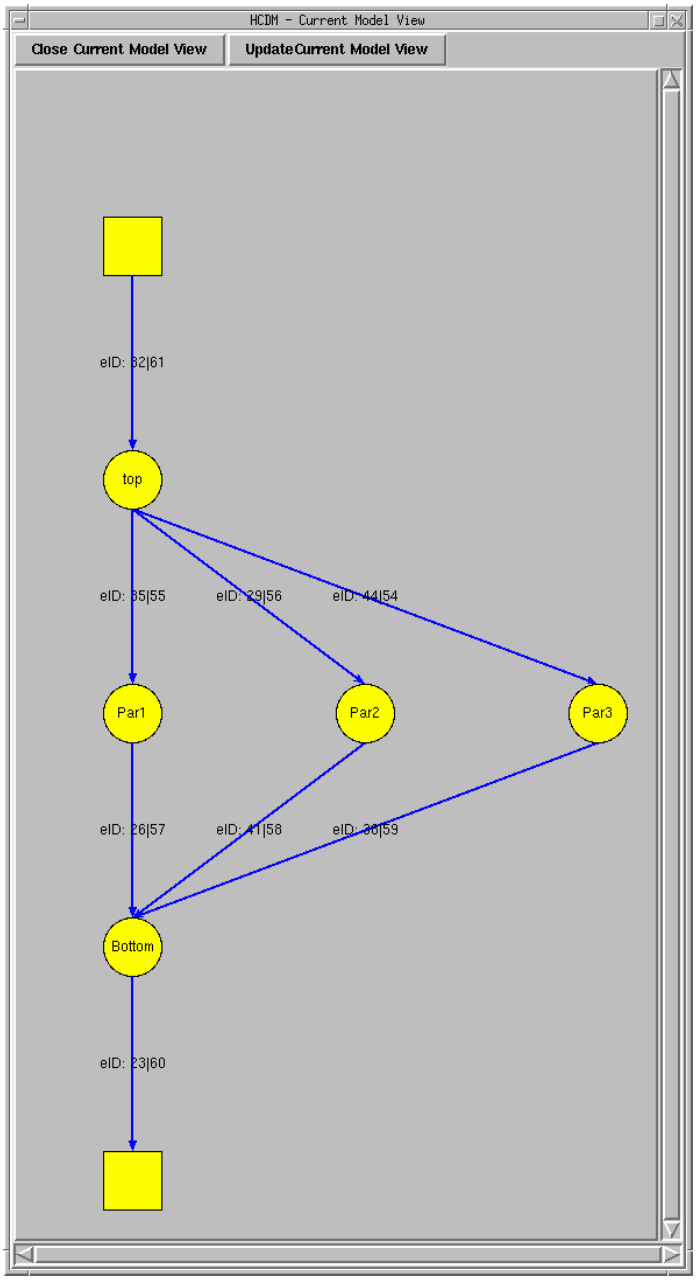

Figure 3: HCDM Task Graph example screenshot

2.4 Data Flow Graphs

Compared to the Task Graph the Data Flow Graph (DFG) includes additional vertices. To every task in the graph possible external operators are appended. So a Data Flow Graph does not only give the data dependency between two consecutive tasks, but also the dependency of every task on external inputs.

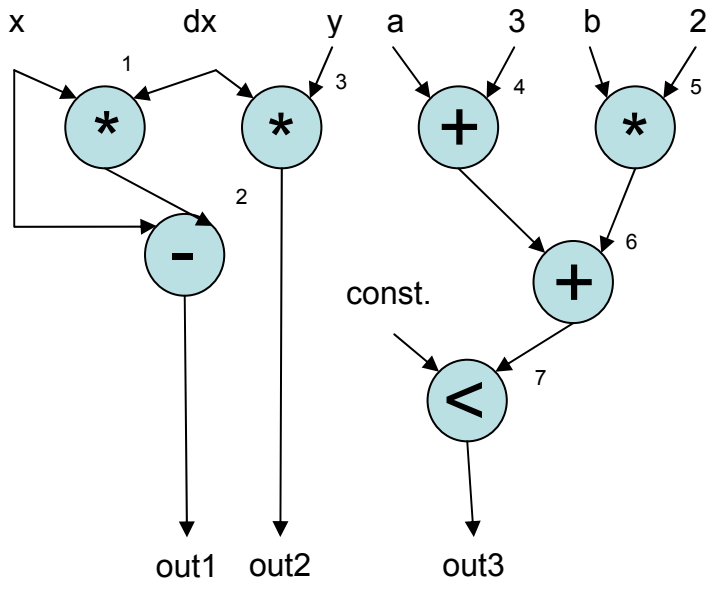

Figure 4: Data Flow Graph example

The destination for the task result can be either another component or an external output port. So a Data Flow Graph could be considered as an extended Task Graph. Figure 2.3 shows an example of a Data Flow Graph.

2.5 HCDM in general

The HCDM tool implements a set of genetic algorithms to generate an optimal resource binding and scheduling for a set of allocated resources and a CoDesign Model (CDM). A CDM is an extended Task Graph Model.

The HCDM flow graph is basically described in terms of processes, their resources and the relations between them. Processes can be considered as tasks that run on a specified resource. A resource could be a physical component on which the process

runs. To every pair of process and resource and execution time is associated, which describes for which period of time the process is running on the resource. The different processes can be connected by branches. Branches represent data dependencies, i.e. a process followed by a second process connected by a branch indicates that the calculation of the second process is dependant on the results of the first process.

In that way a directed hierarchical graph for representing algorithms of any kind can be drawn. The Task Graph is entered graphically via a graphical user interface. From the graphical representation a text file is generated that represents the textual description of the graph. Note that the drawn HCDM data flow graph does only specify the data dependencies of the implementation. It does not contain a sequential ordering of the operations. The order is generated during the scheduling and does not require any further user interaction. The designer has to define the resources and the tool will generate a corresponding scheduling and allocation.

2.6 HCDM Grammar

HCDM produces two different output text files. One is a file with the textual description of the graph. The other text file describes the scheduling for the different tasks on the specified components. For parsing these two files later on, the meaning of the grammar of these two files is important.

The whole HCDM grammar and an example of a graph text file written with the HCDM grammar are added to the appendix of this thesis. For demonstration we state here an example of a task description written in HCDM. Further examples are following in the chapter on parsing.

```
TASK 14 {
               NAME { Par2 }
               OPTIMIZATIONTYPE { 1 }
               IMPORTANCE { 1 }
               FAILURE_PROBABILITY { 0.5 }
               LEVEL { -3 }
               PRIORITY { -4 }
               TIMING { 0 }
               RESOURCES {  (2,20 ) (5,20 ) }
               IORELATIONS {
                  IORELATION {
                     INPUT { 84 181  }
                     OUTPUT {78  }
                     CONDITION {  true  }
                  }
               }
               SUBTASKS {
               }
            }
```

Figure 5: Task description in HCDM

One can see that the graph description always starts with a declaration of the used resources. Shortly after, the tasks are defined. Each task contains several fields for setting task parame-ters. The HCDM grammar contains a PRIORITY field where the parameter for list schedul-ing has to be entered. Safety critical task can be marked inside the IMPORTANCE field. Furthermore the failure probability of a task can be taken into account inside the FAILURE_PROBABILITY field. In the last section the relation between the tasks are de-fined. Tasks are connected over branch numbers inside IORELATION. These are all fields that are important for the original purpose of HCDM which is to perform scheduling and binding for embedded systems. When using HCDM for generating VHDL code most of these parameters will be ignored. In fact we only consider here the ones that are relevant for our VHDL conversion. More to this in the chapter on parsing.

The Scheduling File produced by HCDM is quite self-explanatory and does not really need a specified grammar. The first line defines the start time and stop time of every task. In the second line the binding of every task with its resource is stated.

2.7 HCDM and Data Flow Graphs

As mentioned before HCDM has been created to design Task Graphs. That means that the tool does not provide any methods to represent external operators, needed for the Data Flow Graph.

Without changing the HCDM program itself, it is still possible to adapt its functionality to create DFGs, which will be described next:

For creating the Task Graph, HCDM provides two graphical elements: Circles that represent the Tasks and directed branches that represent the data dependencies. If we want to represent external operators, the task circles could be used.

We use "pseudo" tasks to implement external inputs. External Inputs are defined as Tasks that are direct successors of the root task. We can identify them by comparing the branch numbers.

In quite the same manner the external outputs are simulated. Outputs are defined to be the last tasks inside the DFG. They have outgoing branches back to the root task, which is an adequate characteristic for filtering them out.

The name of the external operators is handed over the same name field as used for the task names (see HCDM grammar). In the later converted VHDL code these inputs also own a certain bit length.

Inside the HCDM grammar for the Task construct there are several possible fields where to hand over the data type. In principle it is possible to enter the bit length into any of the predefined Task fields like IMPORTANCE, LEVEL or TIMING that are not used for our DFG converting. Because of simplicity and particularly because of consistential reasons we decide to enter the bit length right behind the variable name, separated by brackets. That implies, that when parsing the Task NAME field to VHDL, the name itself and the bit length have to be separated and stored individually.

3 The Data Flow Graph Compiler

The nomination and structure that is applied to the VHDL Converter program is basically the same that is also used to describe the behaviour of a compiler for high level languages. In fact if we define a compiler accordingly to [5] as "a program that translates programs written in a high-level programming language into native machine language of a digital computer" the VHDL Converter can be considered as a complier. In fact the program shows similar characteristics as a compiler and we will orientate on the principles of compiler construction.

3.1 Compiler in general

In the next section the basic components of a common compiler are stated. Figure 6 shows the model of a common compiler.

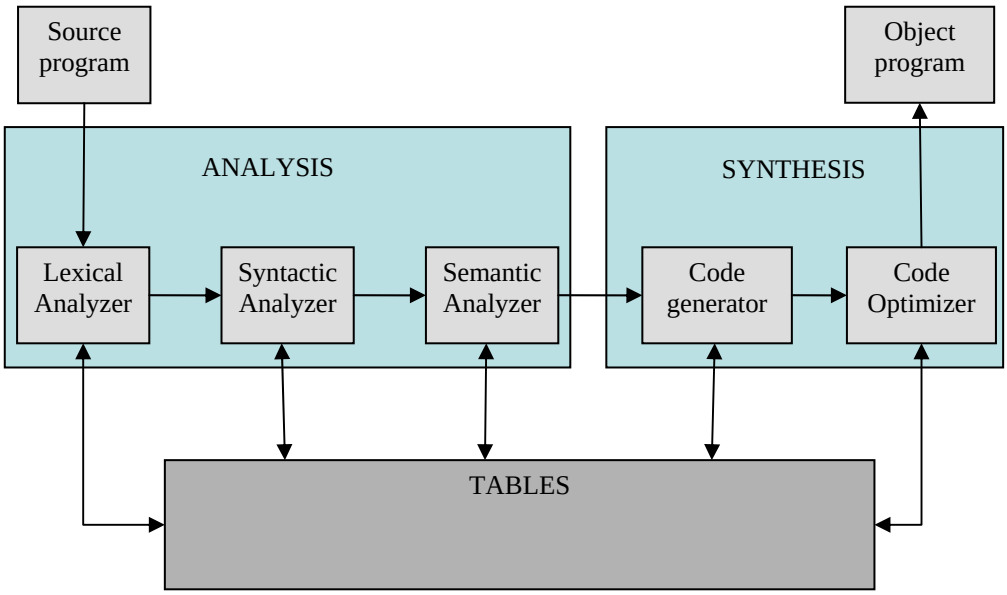

Figure 6: Model of a classical compiler

The compilation process is composed out of two parts: The analysis of the source program and the synthesis of its corresponding object program. During the analysis part the source program is fragmented into its basic parts and processed via a lexical, syntactic and semantic analyser. The pattern to distinguish between between keywords and relevant information are taken out of tables. The Lexical Analyzer needs tables to look up the Keyliterals, while the Syntactic Analyzer looks up the Keywords.

The extracted raw material is forwarded then to the synthesis which builds the equivalent object program modules.

For our VHDLGenerator program we orientate on that classical compiler model. Thereby the basic structure is kept the same to some extend. Figure 7 shows the adjusted compiler model for VHDLGenerator.

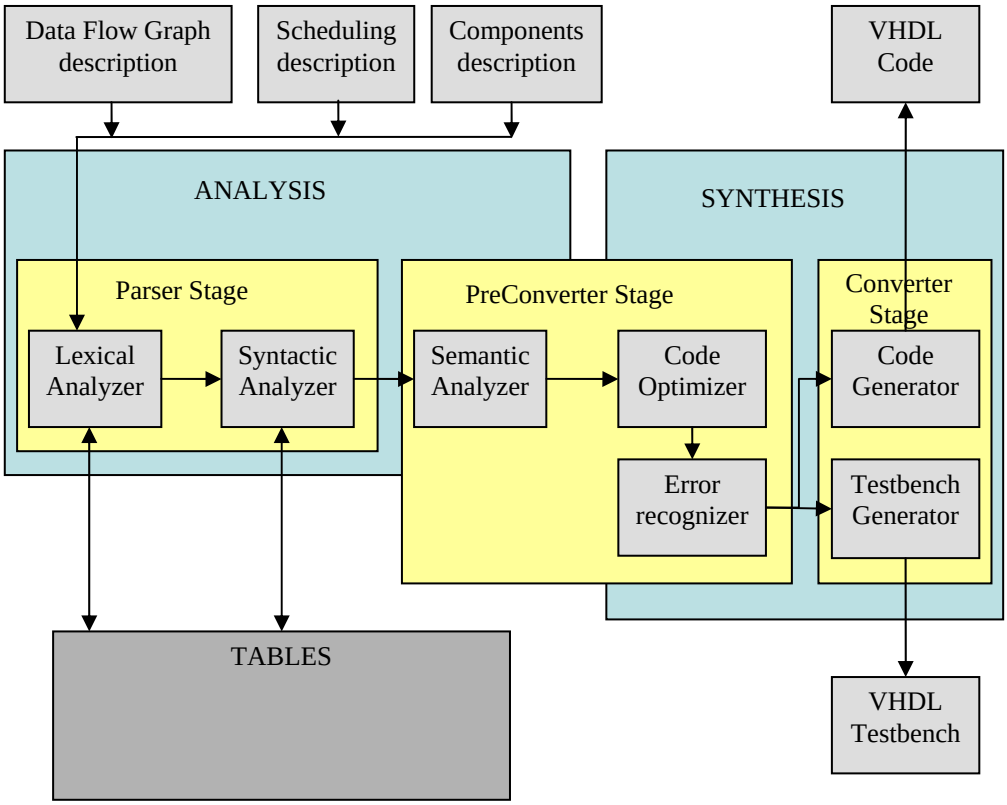

Figure 7: Model of the Data Flow Compiler

As it can be seen, instead of one source program, we receive three text files that need to be extracted. In the drawing the analyzers are related to the Java classes in which they are executed. The boxes in beige indicate the membership to each class. Before the VHDL Code is generated an output register optimisation is performed via Left Edge algorithm in the code optimization phase during the PreConverter stage. Possible errors are detected and printed out. Optional a testbench could be generated.

In the following we will explain more in detail how the different stages work.

3.2 The Parser stage

In this section the design of the HCDM parser will be developed. First an introduction to parsing theory is given, where the different concepts and term declarations are explained. It is shown how these concepts are applied to the actual problem. A state chart diagram is developed to demonstrate the behaviour of the implemented Java Parser.

3.2.1 Parser Theory

In general parsing is defined as a process that analyzes a given sequence of literals and tries to extract the desired information according to a predefined grammar. That is the so called syntax analysis. Before the information could be analyzed, the given sequence needs to be sampled literal by literal, to extract the keywords.

The elementary operation of every syntax analysis is called *lexical analysis*. The lexical analysis got as an input the literal sequence of the to be parsed data. It identifies literals that belong together and passes them as *Lexemes* one layer up to the syntax analysis.

The *syntax analysis* then uses the given grammar to figure out the meaning of every Lexeme and the relation between adjacent Lexemes.

So a Parser can be thought of having two layers that work in parallel. On the first layer a *Lexemer* scans the input sequence and produces valid Lexemes. To distinguish between two successive Lexemes the Lexemer needs to know the predefined separation literals. Separation literals can be set for example as white space or closing braces. According to *Maximal Munch Rule* literals are assembled until such a separation literal appears.

On the second layer a *Tokenizer* absorbs the Lexemes and produces tokens according to the grammar. Tokens are considered to be a pair of Lexemes. First Lexeme indicates the token type and second the token value. In most cases these two Lexemes are consecutive in the parsed text. The tokens are stored for further processing.

If we pick up the idea of seeing the VHDLGenerator as a Complier there has to be a third stage following, the semantic analysis. During this step the actual meaning of

the collected tokens is determined and an intermediate form of source code is generated. The semantic analysis is done inside the PreConverter class and will be explained in the following sections.

3.2.2 The HCDM Parser

The implemented HCDM Parser is orientated on the previously introduced parser theory. We only make some slight changes in the nomenclature for the parser program code. Instead of naming the assembled literals as *Lexemes* they are named as *Words*. For the documentation we will use both synonyms.
Furthermore it is not possible to run the two processes for *Syntax Analysis* and *Lexical Analysis* in parallel as it is recommended. That is basically due to the fact that Java is interpreted sequentially. So we need to serialize the two processes. How the serialization is realized can be seen from the state chart below.
To convert the Data Flow Graph into synthesizable VHDL code the necessary information from three text files has to be parsed. Before we come to discuss the parser itself, basic concepts of streaming text files in Java need to be explained.

3.2.2.1 Java streaming concepts in general

In order to parse, the available data has to be serialized previously. This means, we need to find a method to read the data in and give it out literal by literal. In our case the data will be at text file and a literal will be a single char value.

Java offers the possibility to handle data in form of streams that means to read and create streams. A stream can be considered as a batch of data on its way from some source to some specified destination. The big advantage is, that we can abstract from the kind of data that is streamed. That way the destination process does not have to care about where the stream comes from and the source process does not have to take care where the string is going to. In our case we are parsing from a text file into a storage variable of type Array List.

All inputs and outputs in Java are realized as streams. To use streams the package `java.io.*` has to be included. The streams descend all from a common abstract class. The class *InputStream* implements the interface *Reader*. *OutputStream*

implements the interface *Writer*. All classes that inherit from one of the two classes provide the basic same functionality.

The class *Reader* offers interfaces to open, read and close files. The class *InputStreamReader* and *StringReader* implement that interface. For our purpose we will use these two classes and another subclass called *FileReader*, which inherits from *InputStreamReader*, to read the text files produced by HCDM. To create and read streams with these classes is quite simple. If the path of the file is given to the constructor of FileReader, the file is opened in the read modus automatically. If the opening fails, a FileNotFoundException is thrown. So the method call always has to be placed inside a *try-catch* block. More on the topic error and exception handling in section 3.3.3.

Within a loop the method read() applied to the FileReader object returns the text, sampled character by character. The return value is of type *int* and typcasted to *char*. This will be the smallest data unit the parser will work with. According to parser theory this single char value is nominated as *Literal*.

Inside the Parser code it will not only be necessary to parse a whole text file but also to parse tokens out of a string. For this issue we use an instance of the class StringReader. A string can be streamed in the similar way as a file.

The Parser only absorbs information out of text files and therefore we only need instances of type Reader. But later on it will be required to stream the converted VHDL code back into a text file. The output streaming in Java is done simultaneously as for the input stream. The used instance is of type FileWriter and implements the interface Writer. Like previously, the full path of the destination file is handed over by the constructor. The method write() appends a given string to the end of the text. The call has to be done inside a *try catch* block.

3.2.2.2 Parsing inside VHDL Converter

To construct the VHDL Code basically three different text files need to be parsed: The HCDM file that describes the tasks, ports and the hierarchical relation between them. The scheduling file sets start and stop times for the tasks and how they are bind to the components. The scheduling is necessary because we are dealing with limited

resources. We also need to know which components are available and how their interface looks like. This information is contained in the components file.

The parsing procedure is quite the same for every text file of the three, so we will explain it only once. First it is explained how the parsing is realized in general. Second, the information we want to extract out of every file is stated.

The parsing process can be explained in terms of a deterministic finite state machine that is shown in figure 9. To understand the state chart some explanation of the expressions needs to be given:

Expression	Meaning
valid Literal	a literal that is a possible part of a Lexeme (word)
KeyLiteral	A literal that determines the end of a Word
Word	Lexeme
Word_ID	Stores the meaning of the consequent Lexeme to built the token pair
not-relevant Lexem	Lexeme that is not part of a Token
KeyWord	First Lexeme of a Token (indicates the type of the token)

Figure 8: State Chart expressions

The program enters into the *checkLiteral* state every time a new literal is pushed out of the stream.

If the current literal is not a *valid* literal, that means not a KeyLiteral or part of Lexeme, the parser breaks off the current iteration and takes the next literal in line. If the literal is valid but not a KeyLiteral, the literal is accumulated to the current Lexeme. KeyLiterals are defined for example as white space or closing braces. If one of these characters comes out of the stream, the Lexeme accumulation is aborted and the WordID is checked.

If a matching WordID is set, the current Lexeme has to be the second Lexeme of a token pair. So the Token is stored. Analogous if the WordID is not set, the current Lexeme is neither a non-relevant Lexeme nor a KeyWord of a token pair. In the last case, the corresponding WordID is being set and the state machine continues accumulating literals.

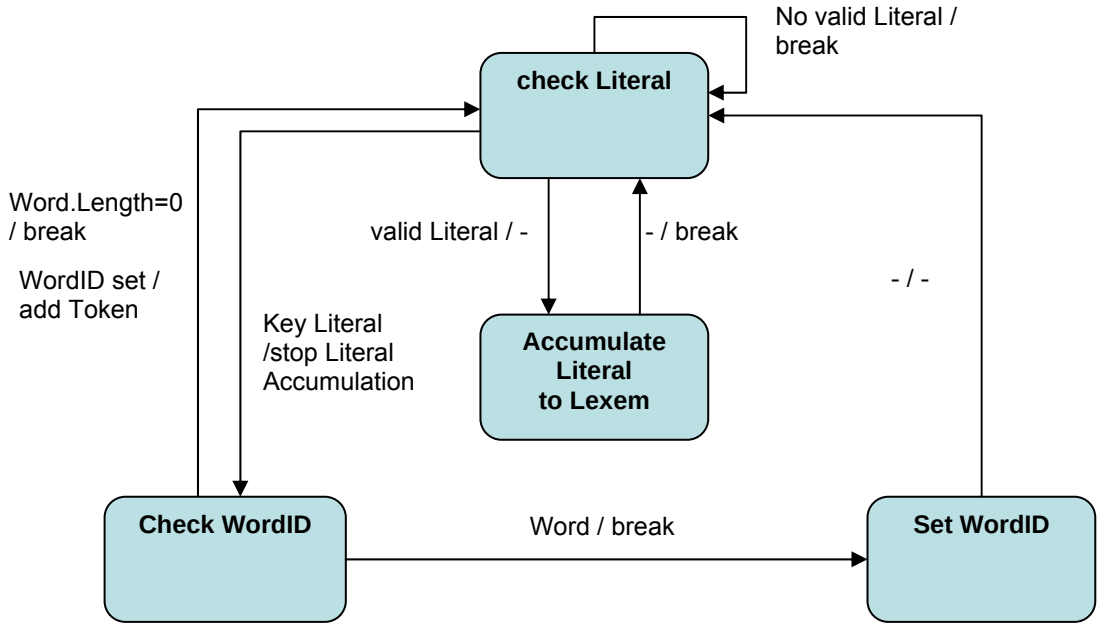

Figure 9: Parser State Chart

3.2.2.3 Parsing the HCDM file

The file that contains most of the needed information is the HCDM.hcdm txt-file. The HCDM file is, as already described in chapter 2, a textual description of the data flow graph. So once the graph is drawn inside HCDM, the tool creates this file and saves it into a specified directory. We now want to use the explained Java streaming concepts and the Parser state machine to extract the relevant information out of the HCDM File. The HCDM file contains a lot of not needed or redundant information. Figure 10 demonstrates how Tasks are stated inside the HCDM grammar.

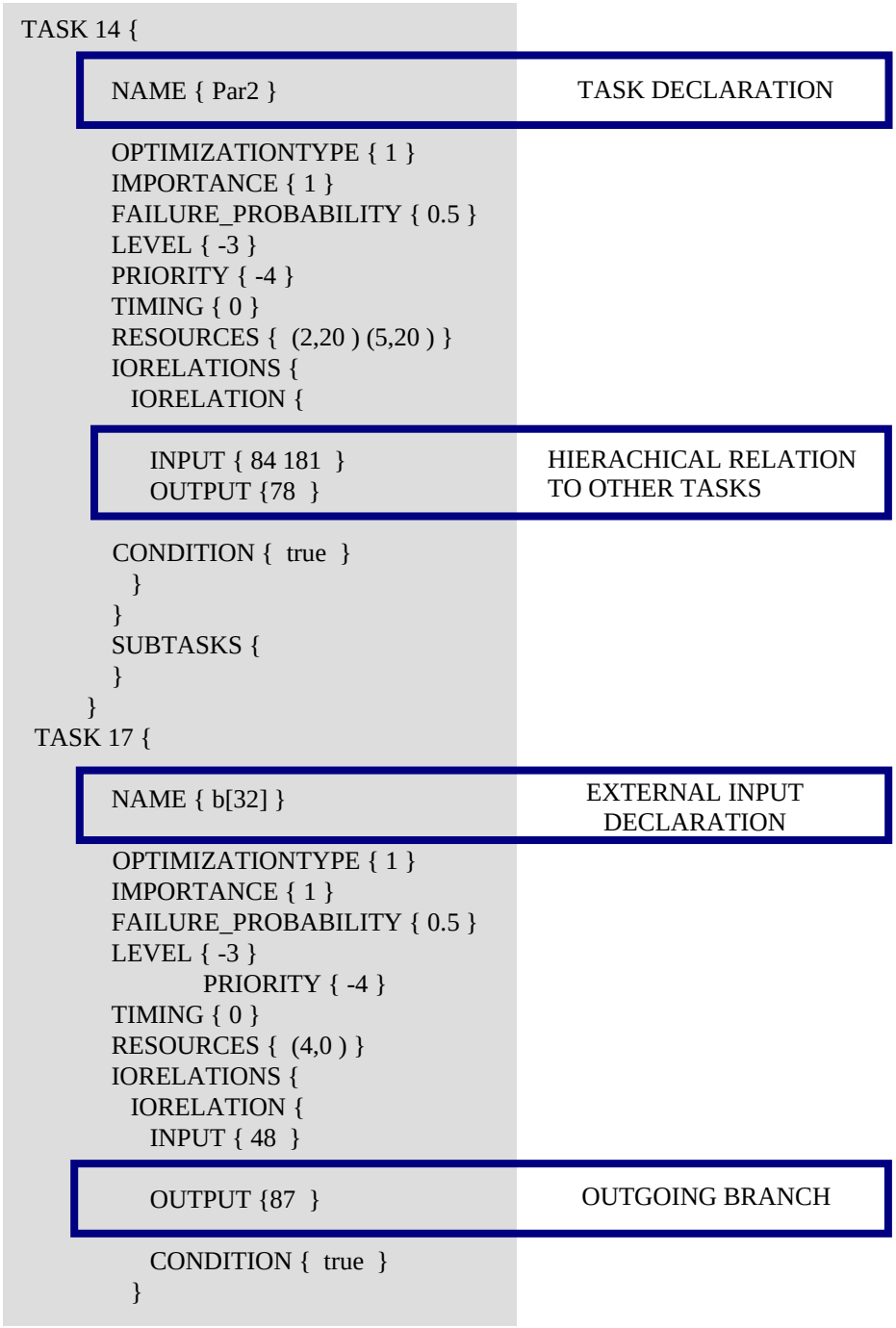

Figure 10: Task declaration inside HCDM

In fact the only information we want to get is: the *name* and *bit width* of external inputs, the task names and the hierarchical relation between them. So out of the HCDM grammar we choose the keywords NAME, INPUT and OUTPUT. When the state machine is in the *WordIDcheck* state, the Token is constructed when the WordID is set. So during the transition from this state back to the initial state, the token has to be stored somewhere. We decided to build a class with name Task in order to store all tokens. Tokens are stored inside attributes of this class.

The Parser pattern can be described best in terms of a Backus-Naur Form (BNF). The BNF in figure 11 is referenced to the HCDM grammar stated in the appendix and only describes the fields that are extracted by the parser.

```
<name>       →   'NAME { '<taskname> | <port>'}'.

<taskname>   →   0| <string>.

<port>       →   0| <portname> <leftdelimiter> <integer>
                    <rightdelimiter>.

<portname>   →   0| <string>.

<iorelation> →   'INPUT {' <idlist> '}'

                 'OUTPUT {' <idlist> '}'.

<idlist>     →   0| <integer> ' '  <idlist>.

<integer>    →   <digit> | <integer> <digit>.

<string>     →   <letter> | <string> <letter>.

<letter>     →   'a' | 'b' | 'c' | ...| 'z' | 'A' | 'B' | 'C'
                 | ... |'Z'.

<digit>      →   ' 1 ' | ' 2 ' | ' 3 ' | ' 4 ' | ' 5 ' | ' 6 '
                 | ' 7 '| ' 8 ' | ' 9 ' | ' 0 ' .

<leftdelimiter>        → '['.

<rightdelimiter>       → ']'.
```

Figure 11: BNF description of Parser pattern

Two important not yet mentioned design rules have to be taken into account when drawing DFG's with HCDM:

1) When we model a component with a task symbol in HCDM, we have to take care that we are drawing the corresponding inputs of the task in exactly the same order as they are stated inside the component entity. If we do so HCDM will assign ascending branch numbers to the task and VHDLGenerator can parse the associations correctly.
2) If a task got more than one proceeding task, HCDM will generate an output branch for each proceeding task. In VHDL these outputs should only be related to one VHDL component output. Note that VHDLGenerator is aware of that and discards the different branches in a way that it produces a single output that is routed to every following component.

That way the parser goes from task to task, extracts the relevant information and puts everything into a variable of type ArrayList. Once all tasks are extracted the external input and output ports need to be separated from the real tasks to generate the top level interface. So an algorithm is started that identifies input and output ports, allocates them to the respective task and deletes them out of the ArrayList. To filter out the external ports we make use of the previously mentioned fact that external Inputs are designed tasks that are direct successors of the root task and external Outputs have outgoing branches back to the root task .

The bit length of every input or output port is parsed from the name by using the discussed Java StringReader concepts.

3.2.2.4 Parsing the Scheduling File

The second file produced by HCDM is a scheduling file like it is shown in figure 12. The scheduling file is parsed in the same manner as the HCDM file.

```
true : b[32] ( 0 ,0 ), a[32] ( 0 , 0 ), top ( 0 , 20 ), Par2 ( 20 , 40 ), Par3 ( 20 , 40 ),
Bottom ( 60 , 70 ), result[32] ( 70 , 70 ),
-----------------------------------------------------
 BINDING { (Par2-myAnd_2) (b[32]-Input_2) (Bottom-myOr_1) (top-
          myXor_1) (a[32]-Input_3) (Par3-myAnd_1)  (result[32]-Output_1)  }
```

Figure 12: Scheduling File example

In the first line of the scheduling file the start and stop times of the tasks are listed. Keywords are here the task names of the task that are already stored in the ArrayList variable. To every task the start time and stop time token is assigned. Later on one single time unit produced by HCDM will mapped to a single clock cycle inside the VHDL code.

Note that the first entries in the scheduling file are the external inputs and outputs. These have to be removed out of the task array afterwards. Also the word "true" is only used as delimiter and is therefore ignored during the parsing process.

When the schedule parsing has finished, the parsing of the component-binding starts. The trigger for the binding line is the keyword "BINDING" which is the first word in line. Extracted Tokens are associated to the respective task.

3.2.2.5 Parsing the Components File

The components on which the tasks run have to be specified somewhere. For that reason a text file is created to enter the needed components. To simplify the parsing work we can completely abstract from the components behavior. In fact it is only necessary to define the interface of a component. The behavior will be included later as a library inside the generated VHDL code. Figure 13 shows how the components file looks like. Note that the commentary will at the beginning will be ignored by the parser.

```
/* NOTE: Enter all used components, with their input and output ports,except the
"clk","reset" and "enable" which are generated automatically */

NAME {myXor}  INPUT {a[32],b[32]}  OUTPUT {result[32]}
NAME {myAnd} INPUT {a[32],b[32]}  OUTPUT {result[32]}
NAME {myOr}    INPUT {a[32],b[32],c[32]} OUTPUT {result[32]}
```

Figure 13: Components File example

So to every component the following information has to be entered: the components name, the inputs ports and output ports with the adequate bit width. Every component is expected to have a *clock*, a *reset* and an *enable* input. So thesis parameters are not entered by hand but generated automatically.
The components are parsed separately from the tasks and stored inside a different ArrayList called *UsedComponents*. We will need them later for components declaration inside the VHDL code. Furthermore they are used for error detection, which will be subject of chapter 3.3.3.

3.3 The PreConverter stage

The Parser stage delivers two objects for further processing. One is called *Tasks* and contains all assigned tasks with their specified attributes. The other is called *AllComponents* and contains an ArrayList of all used VHDL components. The goal of the PreConverter stage is now to prepare the collected data for the final VHDL Converter stage.
In the following chapter the theory of the different processing steps of the PreConverter are explained. Special attention is taken on the part about register optimization.
The concepts are discussed in the same order as the related methods are executed inside the PreConverter code.

3.3.1 Sorting of the Tasks

During the execution of the VHDLGenerator program it occurs quite often that we have to sort arrays. Ports need to be sorted by their branch number, case-statements by their execution time and tasks by their start time. So because of the high usage of

sorting algorithm it is worth to have a deeper look in it, in order to choose the most appropriate one.

When we want to perform the Left Edge algorithm it is of great importance to have the tasks sorted in ascending order to their Start Times. So the first step of the PreConverter stage is the sorting of the tasks.

An appropriate algorithm for sorting has to be selected. Appropriate algorithm means here that we want to find an algorithm that sorts a given set of items with the lowest complexity. We define complexity here as number of steps and time units.

In general one can categorize sorting algorithms either by their algorithm structure (Divide&Conquer form [9]) or by their complexity [10]. Using the second categorization most of the common sorting algorithms are divided further into basically two classes of algorithms with respect to their execution time. For the first class of algorithms, the execution time increases quadratic ally with the number of items. The second class needs n*log n complexity with being n the number of items. Figure 14 and 15 [10] demonstrate how typical algorithms of each class behave for large n.

Big-*O* notation is used, where the *O* represents the complexity of the algorithm, *n* stands for the number of items to be sorted and the whole expression inside parenthesis determines a measure for complexity.

Inside the complexity class the algorithms may still vary in their constant runtime factor (how much time each of the n*log n steps takes).

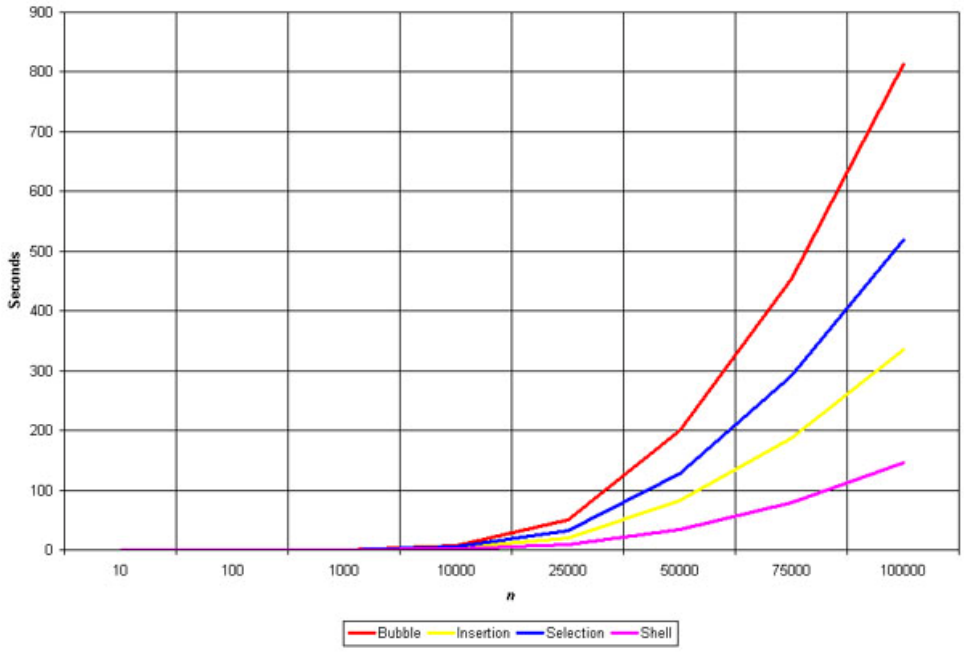

Figure 14: O(n^2) sorts

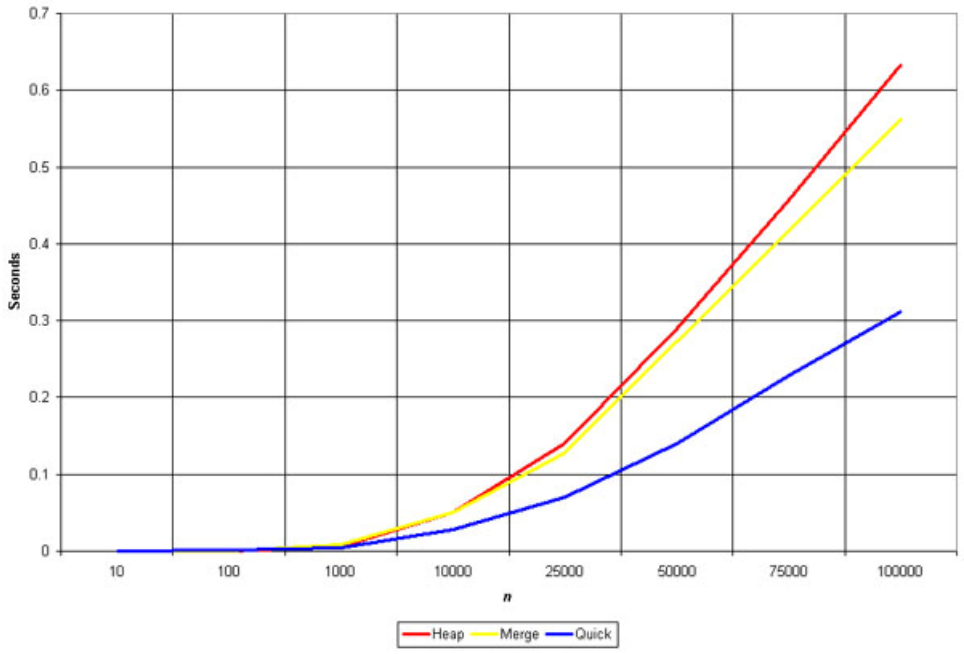

Figure 15: O(n log b) sorts

As it can be seen the Quick Sort algorithm yields the best performance. Quick sort works onto the divide-and-conquer principle and uses recursive structures. That may cause problems for applications with resource limitation. For VHDLGenerator we

expect to have sufficient computation power and memory space, so that recursion is not a drawback for us.

The dividing (partitioning) works in a way that we first choose a pivot element. All elements that are greater than the pivot element are put into a new partition. All elements that are smaller than the pivot are put into a second partition. Then the algorithm is repeated for each partition separately. At the end all partitions are concatenated.

Figure 16 [9] gives pseudo code for the Quicksort algorithm:

```
Quicksort(X,l,r)
1 if l<r
2    then split ← PARTITION(X,l,r)
3           Quicksort(X,l,split)
4           Quicksort(X,split+1,r)
PARTITION(X,l,r)
1 pivot ← X [l]
2 i ← l-1
3 j ← r+1
4 while TRUE
5     do repeat j←j-1
6           until X [j] <= pivot
7           repeat i←i+1
8           until X [i] >= pivot
9           if i<j
10                then exchange X [i] ↔ X [j]
11          else return j
```

Figure 16: Quicksort pseudo code

3.3.2 Register Optimization

The implemented VHDL Converter has to deal with resource constraints. On one hand the number of available instances of one component type is limited. This implicates a scheduling of the tasks which is done by the HCDM tool and will not be discussed further. On the other hand we want to use a minimum amount of registers. A result coming out of one of the components has to be stored in a register until it is further processed. A register in VHDLGenerator is considered to be a signal of type std_logic_vector with the same bit length as the result vector. Depending on the task hierarchy not every result needs a private register. A register could be reused once the previous value has been passed to the input of the next component. So we would like to find an algorithm for yielding an optimum number of registers.

A result of a component can be considered as a variable with a certain *lifetime*. The Lifetime of a variable is defined as "the interval from its birth to its death, where the former is the time at which the value is generated as an output of an operation and the latter is the latest time at which the variable is referenced as an input to another operation"[2].

So first step would be to find the BirthTime and DeathTime of every task result. As it has been already turned out during the parsing of the Tasks it is always quite useful to extract the desired information and store it into an object ArrayList for further processing. We will proceed here in the same way. For these purposes a class TaskOuputLifetime is created that contains the following attributes which have to be set for every task: TaskName, BirthTime, DeathTime, RegisterName and RegisterBitWidth. So the ArrayList is filled with TaskOutputLifetime objects corresponding to every task.

The BirthTime attribute is identically with the Stoptime attribute inside the task object and can be taken over directly. Note that the Stoptime is stated inside the scheduling file and has been parsed in the parser stage before.

The DeadTime is found out with a bit more effort. If an output variable is used as input for more than one component, we need to find the component with the **latest** StartTime, i.e. we are looking for the maximum of all possible proceeding DeathTimes. So we first check which are the successor tasks of our current task. Then the StartTime of each is collected into an array. After having done this we go

through this array, compare all elements with each other, pick up the highest value and assign this value to the DeathTimes attribute of the current TaskOutputLifetime object. The procedure is then repeated for every task until all lifetimes of the task outputs are found.

With this data now the optimum number of needed output register can be calculated using a minimization algorithm. There are basically two different methods to proceed: Finding the minimum number of registers either using *Clique Partitioning* or using *Graph Coloring*.

Clique Partitioning is described in [2] and has been first implemented by Tseng [15]. We observe that solving the register optimization with clique partitioning is NP - Complete. It does not guarantee an optimal result.

The second alternative is the Graph Colouring approach. This algorithm promises optimal results with complexity O(n^2). An appropriate representative of Graph Colouring algorithms is the **Left-Edge Algorithm**.

We will first explain the theoretical background of the left edge algorithm, talk about its development and then show how it is realized inside VHDL Converter.

Accordingly to [2] the underlying theory of the Left Edge Algorithm is the Graph Colouring optimization problem. The Graph Colouring searches a vertex colouring with a minimum number of colours. In the algorithm below the different colours are represented by integer numbers.

```
VERTEX_COLOR(G(V,E)){
1     for(i =  1 to |V|){
2           c=1;
3           while( ∃a vertex adjacent to v_i with color
c) do {
4                 c= c + 1;
5           }
6     Label v_i with color c;
7     }
```

Figure 17: Vertex Coloring pseudo code

At the beginning every node got a colour number of 0. The algorithm is applied to the compatibility graph in figure 18. Note that compatibility graphs are undirected graphs.

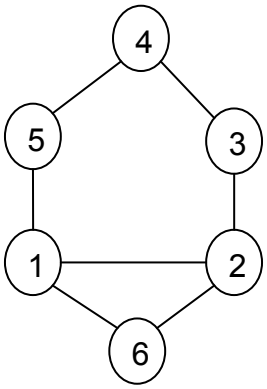

Figure 18: Compatibility Graph

The result is shown in Figure19. In comparison figure 20 shows the optimum colouring. So the colouring graph algorithm does not yield optimal results. One way to improve the algorithm is the swapping colour method. In our example we reach the optimal solution either by backtracking or by swapping the colours of vertex v5 and v4.

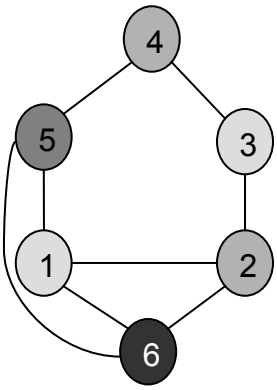

Figure 19: Non-minimum Coloring

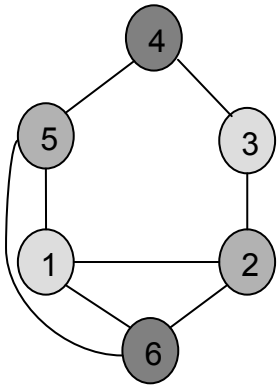

Figure 20: Minimum Coloring

Fortunately the Left Edge Algorithm holds an interesting property. The underlying graph built with our concept of lifetimes of variables belongs to a subgroup of graphs called interval graphs. Colouring the intervals is equivalent to colouring the vertices. For interval graphs the colouring algorithm needs polynomial time for solving and

one important note: the result is **optimal**. This is based on the fact that interval graphs have *perfect vertex elimination scheme.* For detailed description on the definition of perfect graphs and *perfect vertex elimination scheme* take a look inside [2]. For Perfect Vertex Elimination Scheme there is no need of backtracking or colour swapping.

The original Left Edge Algorithm as proposed by Hashimoto and Stevens[13] has been developed to solve channel routing problems. The goal was to assign wires to a minimum number of routing tracks.

As a first step the wires are sorted with increasing order of their left end points from the left edge of the channel. That is the reason why this algorithm is called "Left-Edge". Now the assignment starts. The first wire at the left is assigned to the first track. Then we find the first wire whose left edge is to the right of the last selected wire and assign this one to the current track. If we reach the last column, the assigned wires are removed and a new track is started.

This algorithm is repeated until no more wires can be assigned to tracks. Figure 21 [14] shows the pseudo code for the proposed algorithm

```
Algorithm LEA
Begin
1     Sort all nets on their left most end positions:
2     Initialize the tracks t1, t2, ... , td
      (t1 is the lowest track);
3     for each net  nj ∈ sorted list
4           for each ti, i from 1 to d
5                 if nj doesn´t overlap with any nets
in ti
6                 then assign nj to ti
7           endfor
8           delete nj from the list
9     endfor
END.
```

Figure 21: Left Edge pseudo code

As it can be seen Left Edge algorithm uses a Greedy approach. Nevertheless the Left Edge Algorithm gains an **optimal** solution and is of complexity $O(n^2)$.[14].

Kurdahi and Parker [14] grabbed this idea and used the same principle for register minimization. There the wires correspond to the previous discussed lifetime of variables and the routing tracks are the to be minimized registers. The left and right edges of the wires are considered to represent the birth and death time of a variable formally known as lifetime. Accordingly to [14] all different lifetimes have to be collected inside a table. We have already discussed the procedure at the beginning of the chapter. Having done this, the goal of the Left Edge Algorithm is to assign output variables(wires) to registers(tracks) so as to minimize the total number of registers(tracks) to store the output value. Two wires cannot share a track if they overlap in space, whereas two variables cannot share a register if they overlap with their lifetimes.

The presented algorithm is used inside the program REAL (Program for REgister ALlocation) which has been developed by Kurdahi and Parker. Very similar to

VHDLGenerator, REAL gets as input a data flow graph whose operations have been scheduled, along with a lifetime table of the values in the DFG. REAL also has to deal with resource constraints, like number and type of operators used to implement the operations. Scheduling can be overlapping. In the paper of Kurdahi and Parker also register allocation for conditional branches is discussed. Conditional branches are not supported by HCDM so far and will not be discussed in contents of this work. As REAL is optimal for non-pipelined designs with no conditional branches we also expect VHDLGenerator to give optimal results when using the same algorithm.

As implemented inside VHDLGenerator, the Left Edge Algorithm works in a quite simple but efficient way. The encoding in Java is realised based on the pseudo code shown in figure 22.

Lifetimes are extracted as explained before.

Note that the first step of the Left Edge algorithm, the ordering of the lifetimes is done inherently when the tasks are sorted by their lifetimes, as shown in part 3.1.1.

The next step would be to pick up the first lifetime, allocate it to a register (colour) and check for overlapping with other lifetimes. If we consider the tasks to be sorted in ascending order to their Start Times a criterion for non overlapping lifetimes can be formulated quite easy:

> *Two Lifetimes do not overlap, when the death time of the first is smaller or equal the start time of the next lifetime.*

Outputs with non overlapping lifetimes are packed into one register. Then the first element is deleted and the algorithm starts again. The whole procedure is repeated until all lifetimes are processed. As a final step the found lifetime register pairs are transferred to the tasks in a way that inside the task object the corresponding *OutputRegister* attribute is set.

```
ALGORITHM VHDL_GENERATOR_LEA
BEGIN
1     SORT tasks in ascending order to their start times
2     EXTRACT all Lifetimes and store them.
3     LOOP
4           Take first element out of lifetimes
6           LOOP  //Compare to all other lifetimes
7                 IF(Deathtime of first Lifetime <=
                     Birthtime of next Lifetime)
8                       ASSIGN both tasks to same register

9                 END IF
10                REMOVE first lifetime out of list
11          END LOOP
12     END LOOP
```

Figure 22: Left Edge pseudo code for VHDLGenerator

Note that PreConverter also offers another method with name *allocateOutputRegister()***.** This method allocates non minimized registers to the tasks, i.e. every task gets a separate output register. This method could be used instead of the *doLeftEdgeAlgorithm()* method when we do not have to deal with resource constraints. The user of VHDLGenerator has can choose between these two modes in the graphical user interface.

3.3.3 Error-detection and recovery techniques

Although various tests have shown that VHDLGenerator program itself accomplishes its job in a correct way, you are never aware of faults that are introduced from the user's side. What is desirable is software that reacts robust to the users input. Software "is robust if it describes reasonable behaviour even when it is misused or used in error "[12]. Of course "reasonable" is an expandable item. We will define it here for the VHDLGenerator in a way that we say: *the program should catch inputs that lead to a failure. The user should be informed where and why the error occurred.*

Furthermore it is often of use, to test the obtained results against the specification for faults. How to write system level tests for VHDL programs will be part of chapter 3.4.

We categorize possible faults into three abstraction levels on which they may appear. Figure 23 illustrates the different error levels where the intensity of the color indicates the impact of the error on the design.

On **first** level there are the faults that are inserted on the syntactical level. That means for some reason the given input is faulty and cannot be further processed. In our case a level one fault could be a mistake due to the grammar of the to be parsed text. Level one faults can be detected as an error very soon during the processing of the program and therefore do not lead to a failure.

On the **second** level we have to deal with faults that entered in a way that they are not detected as an error by the program itself. The faults are handed over through the whole process and end in a complete failure when the final operation is put into process. These kinds of faults are the worst because they may end in production errors. So it is highly recommendable to catch these faults and transform them into an error before they end up into a failure.

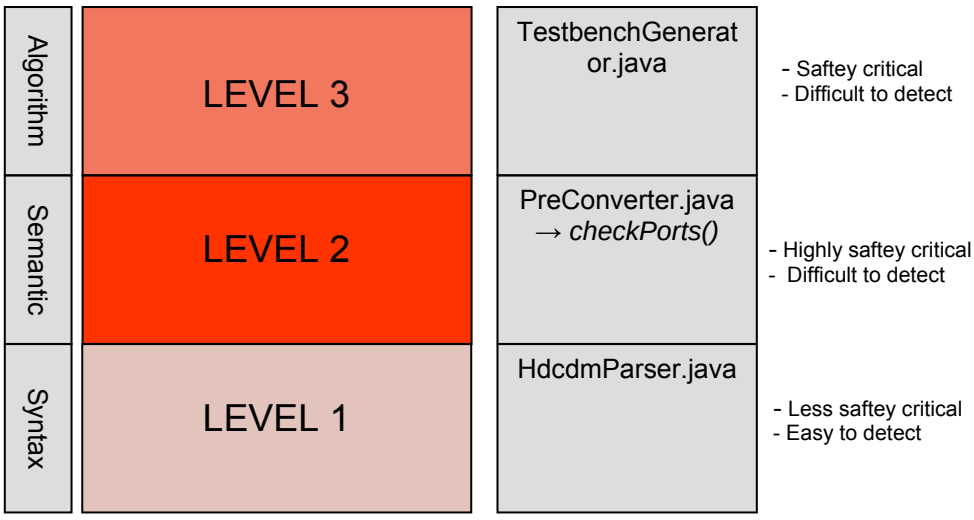

Figure 23: Error Level classification

On the **third** level we want to handle faults that result neither into an error nor into a failure. These faults are made at the behavioural description of the input. In our case this means that the entered algorithm graph does not follow its specification. Like in a compiler these kinds of faults cannot be detected, because they take into account the given specification on which the entered algorithm is based. VHDLGenerator has no knowledge about that. The only chance to get rid of these

faults is to demonstrate the programmer how his implemented algorithm works. The programmer himself has to compare the result with the desired specification and perform possible changes.

In this chapter we will only discuss the faults of level one and two because these are the kind of faults that are treated by the HCDMParser and PreConverter stage. Third level faults are subject of chapter 3.4.

3.3.3.1 Java Exception handling in general

Java offers comfortable constructs to catch and handle errors in form of exceptions in programs. In general an exception in Java is an event, which occurs during the execution of a program, and disrupts the normal flow of the program's instructions. The program flow is then bended over to the exception handling routine.

All exceptions are thrown inside the execution of a method. Methods that are able to throw exceptions got the extension *throws Exceptions* next to the method identifier.

Exception is a Java class of its own. There exist several classes that inherit from Exception and specify the different types of Exceptions. Note that one can also create an own exception class and define a specific behaviour there.

So throwing an exception means creating an instance of this class and hand it over to the runtime system. The runtime system has to find an appropriate handler for the exception. As first step the runtime system gives the exception to the method above that calls the method in which the exception is thrown. The calling method either has to catch the exception and handle it or forward it to the method above. In the last case a new exception object has to be created that encapsulates the original exception message. That way an exception can be handed from the method it occurred up to the first method executed in the program. If not before, the exception has to be definitely handled there. The scheme is shown in the figure 24.

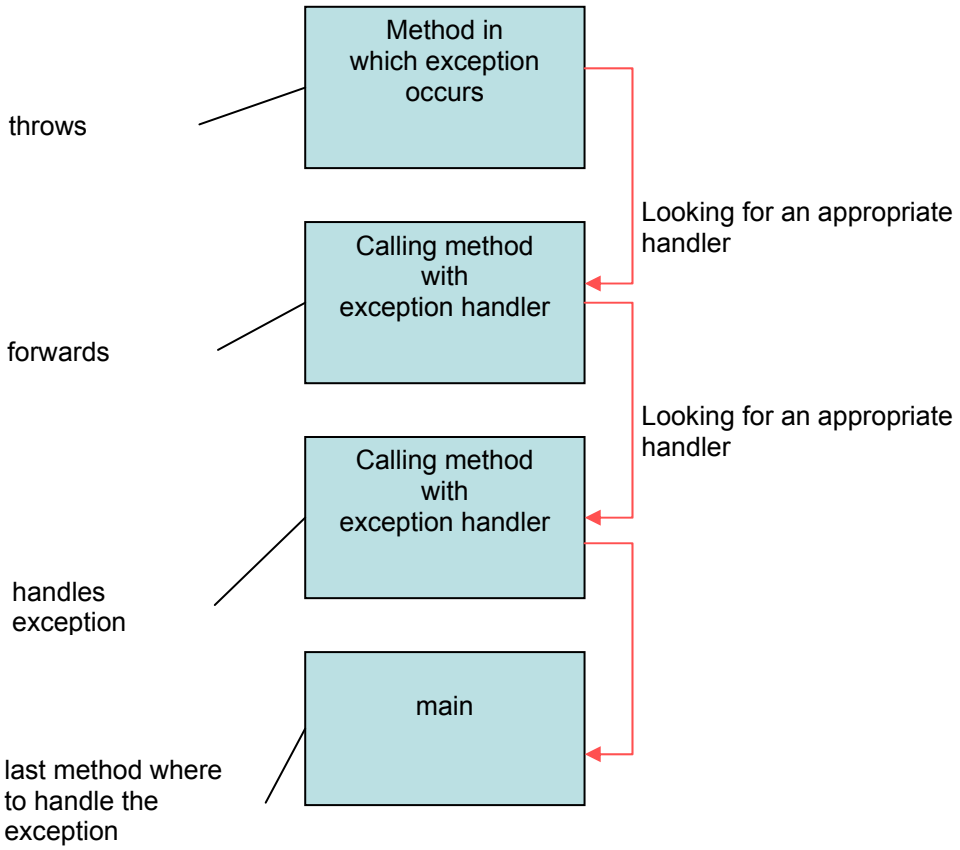

Figure 24: Exception handling in Java

Exceptions objects are created and thrown using the keywords *new* and *throw*. Messages are added over the constructor field. The throw statement could be embedded for example into an *if*-clause like:

```
public void example() throws ExceptionType{
      if(condition){
            throw new ExceptionType(Message);
      }
}
```

Figure 25: throw statement

In the calling method, all methods that might throw an exception have to be placed inside a try-catch block. Methods following after a try statement are executed and in case of an exception the runtime system jumps into the appropriate catch block. The stress lies upon "appropriate" catch block. The calling method can only handle exceptions that are part of one of its catch blocks. If for example a method throws an

exception of type IOException, then the calling method needs a case statement that covers this type. An example is shown in figure 26.

```
try{
      example(); // throws a
ExceptionType
} catch (FileNotFoundException f){
  ...
} catch (ExceptionType e){
  ...
}
```

Figure 26: try-catch

There exist other concepts like the *finally* block, which will not be discussed here. When writing an Exception handler for VHDLGenerator we will need to know how to create own Exception classes and how to throw, catch and handle Exceptions in Java.

3.3.3.2 Exception handling inside VHDLGenerator

As mentioned in the introduction of this chapter, one of the goals of the HCDMParser and PreConverter stage is to catch and handle faults of level one and two.

First we like to figure out all possible faults that might appear when executing VHDLGenerator and categorize them. The table in Figure 27 shows one out of several possibilities how to classify the different faults. The table is thought as a reference for the user to get a more detailed documentation of the error message printed out by VHDLGenerator.

Every error got an error code that is printed out. Errors belonging to level one are of format 1.x, level two starts with 2.x. The Java Exception Type divides the levels further into the different Exception classes. All possible exceptions could be only generated in classes that contain executable methods, namely HCDMParser, PreConverter and VHDLGenerator. The location column gives information about working step in which the exception was thrown. Finally a description of the error and short proposition for solving is given.

Note that all exceptions thrown are forwarded up to the very first calling method, where they are handled.

Level one exceptions occur when input data is faulty and the runtime system cannot proceed. These exceptions are safety non-critical because they result directly in an error and do not sneak through the whole process. They are generated by the Java runtime system automatically. An object is created and a message is added. All we have to do is write an appropriate handler that catches the exception, add an individual text and finally print out the whole message.

Level two faults are safety critical and also more difficult to detect. Let us see why and how they could appear:

VHDLGenerator does not take care whether the produced VHDL code is compilable or not. It just converts given input files into text file which contents could be interpreted as VHDL code. No guarantee of syntactical or semantic correctness is given. Some of these errors inside the VHDL code are detected by the VHDL Compiler and can therefore be corrected afterwards. But we could also imagine a scenario, where errors are not even recognized during the synthesis.

So in order to reduce possible error sources VHDLGenerator should adapt to at least two faulty "level two" user inputs:

Every tasks runs on a component that got a specified number of input ports. So a task needs as many inputs branches as the corresponding component got input ports. We should beware of assigning more ports, because then one will be unused. Assigning fewer ports will lead to floating ports which causes undetermined system behaviour.

Once it is checked whether the number of input ports fit, we further have to verify the bit length of each port. In the Components File the bit length of every component port is stated in brackets next to the identifier. The Input ports bit length has to match with this bit length. Otherwise similar problems than for the previous case appear. We have to check for over and under assignment of bit length.

All level two faults are detected by the method *checkPorts()* inside the PreConverter stage.

checkPorts() throws an exception of type PortException. PortException is a self-made Exception type, that does nothing more than specify the level two exceptions and store messages.

The procedure for finding the errors is straightforward. One loop iterates all tasks and compares the number of ports with the associated component ports. If a mismatch occurs a PortException with error report is thrown. The same procedure applies for a bit width mismatch.

Error Code	Java Exception	Location	Description
1.1	IOException	HcdmParser	The desired Hcdm File is not found in the given filepath. Make sure that the path is correct and the HCDM File really exists inside the specified directory. Note that directories have to be separated by "//". New entered paths always have to be confirmed with "Enter"
1.2	IOException	HcdmParser	The desired Scheduling File is not found in the given filepath. Make sure that the path is correct and the Scheduling File really exists inside the specified directory. Note that directories have to be separated by "//". New entered paths always have to be confirmed with "Enter"
1.3	IOException	HcdmParser	The desired Components File is not found in the given filepath. Make sure that the path is correct and the Components File really exists inside the specified directory. Note that directories have to be separated by "//". New entered paths always have to be confirmed with "Enter"
1.4	IOException	VHDLConverter	The desired path to which the Vhdl File should be created is not valid. Please enter a reachable directory. Note that directories have to be separated by "//". New entered paths always have to be
1.5	IOException	TestbenchGenerator	The Testbench could not be created. TestbenchGenerator uses the same destination path as already used for the VHDL file. Look further for Error Code 1.4

Error Code	Java Exception	Location	Description
			the VHDL file could not be found on the computer. Make sure that the spelling is correct and that the specified program is able to read text files. Note that the code is a simple text file and can always be opened by a text editor like "notepad" or "Kedit"
1.7	Exception	PreConverter	HCDM File Syntax incorrect. The information is entered in the wrong way. Recheck the grammar
1.7	Exception	PreConverter	Scheduling File Syntax incorrect. The information is entered in the wrong way due to the grammar. Verify that the scheduling file got the correct format. The correct format is stated in the VHDLGenerator Reference Manual
1.7	Exception	PreConverter	Components File Syntax incorrect. The information is entered in the wrong way Recheck the grammar
1.8	Exception	VHDLConverter	The converting to VHDL failed. Make sure that the data flow graph is specified in a correct way (no loops, no tasks without inputs).
1.9	Exception	TestbenchGenerator	Testbench could not be created. Recheck if VHDL code is testable.
2.0	PortException	PreConverter	More Ports are assigned to the component than allowed. Increase either number of ports or decrease number of component ports
2.2	PortException	PreConverter	The Input port bit width does not fit with the specified component input bit width. Change either Input Bit Width of Port or component
2.3	PortException	PreConverter	The output port bit width does not fit with the specified component output bit width. Change either Output Bit Width of Port or component

Figure 27: Error Type Reference

3.4 The Converter stage

The VHDL Converter stage finally converts all collected and pre-processed data to synthesizable VHDL code. We make use of the Java streaming concepts already discussed in Chapter 3.2.2 to write a character stream into a text file.

3.4.1 The VHDL Code Generator

In the DFG a task can only fire if all of its inputs are ready and, the scheduled component is free to process. When generating the VHDL code, these semantics should be implemented carefully in the code so that the resulting hardware has got the same behaviour intended by the original DFG. The conversion is done correctly, when the DFG description and the synthesized hardware have the same input to output characteristics.

The ports define the interface of our entity and are generated first. The whole design is synchronous, which means inputs and outputs of components are written and read synchronously to the clock. So first input will be *clk*. The clock should work at the same frequency used for the DFG scheduling and will be distributed to all components later on.

For synchronous design it is also quite common to define a reset signal to set the circuit back to the initial state. The global reset is defined as low active.

Third input is a one bit wide input called *enable* that starts the processing of every component.

These three single bit inputs are the same for every DFG entity and are generated automatically.

Next come the input and output ports of the DFG. Because the PreConverter has already processed the data, the Input and Output ports are easy to find. If a task got an external input or output, it is already stated in one of its attributes. So VHDLGenerator just has to check if some Tasks got same ports and then write the port with corresponding bit length into the output stream. The methods *setInputPorts()* and *setOutputPorts()* are responsible for this work.

Then the underlying architecture has to be written. The architecture starts with the declaration of used components. Fortunately the used components are already

extracted inside the PreConverter stage. They are all stored in the variable *UsedComponents* and with a proper framework of the VHDL syntax they can be pushed directly to the output stream.

Every Input and Output Port of the components needs a buffer variable over which values are entered and red. The results of the components are written to registers. The method for allocating the registers is called *declareRegister().* Depending on the used algorithm in the PreConverter stage, one register per component result or a minimum number of registers according to LEFTEDGE algorithm is generated.

After the begin clause in VHDL the *port map* is done. Named pormapping is used for the generated VHDL code.

The scheduling in VHDL has to be realized inside a process statement, that is sensitive to *clk* and *reset* .

We need a scheduler variable that triggers the corresponding inputs to components and takes the results back at the right point of time. The scheduler is realized as a counter that is incremented at every rising edge of the clock. The case statements are elaborated during the PreConverter Stage and can be put directly into the output stream with the right VHDL framework.

3.4.2 The VHDL Testbench Generator

This chapter treats with the third class of error types not yet discussed in chapter 3.3. These errors appear on the logic level. The user should have a possibility to compare the current implementation to the given specification.

3.4.2.1 Testbenches in general

For VHDL models it is quite common to write so called testbenches for testing purposes. A testbench is a VHDL description itself with the difference that we do not define a port interface. The port declaration inside the entity is kept empty.

The tested model is instantiated as a component and port mapped to internal test signals.

The internal testbench signals receive test vectors at different points in time using concurrent statements. The keyword *after* followed by a time notation tells the simulator tool to assign that value at the specified point in time. Note that the *after* clause is not synthesizable and only used for simulation purposes.

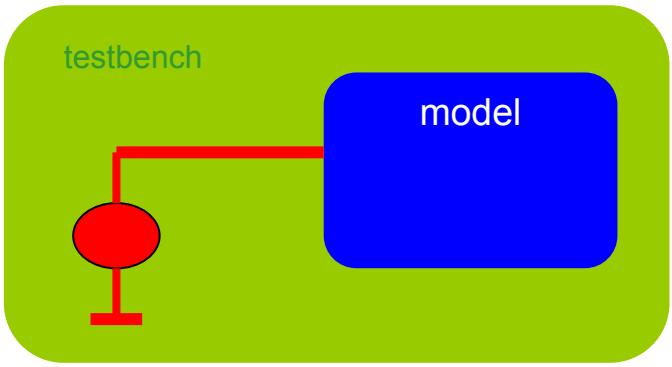

Figure 28: VHDL Testbench

The testbench and the proper VHDL code are handed over to the simulator, which represents the circuit behaviour in graphical form as a wave.

3.4.2.2 Testbench creating for VHDL Converter

The name as well as the directory are constructed using the entered VHDL File path. The testbench uses basically the same concepts of streaming and declaration methods for the VHDL statements as the VHDLConverter stage. For assigning the

testbench variables it is quite helpful that the VHDLConverter class hands over an ArrayList called EntityInputPorts that contains the interface of our VHDL model. If we do so the testbench variables can be allocated much easier.

The assigned testvectors are constructed per default using a random pattern. The algorithm for designing the random pattern is slightly tricky. The Java API provides a class called *Random*. When applying the method *nextInt()* to an instance of the Random class a pseudo random 32 bit integer number is given back. The method *toBinaryString()* returns the corresponding binary value as a string.

We have to deal with basically two major problems that occur when calling the *nextInt()* method. If the generated random number does not cover all the given port bit width, we get a truncated bit vector, that leads to errors during the VHDL compiling process. That is why we have to pad leading zeros in this case.

Another problem is the limited range of the produced random values. Unfortunately the Java API only provides indexed random methods up to 32 bit integer values. That means we receive a maximum bit width of 31 bit (1 bit reserved for the sign). VHDLGenerator is planned to be used for description of encryption algorithms, where bit length of 128 bit and more are quite common. To create appropriate testbenches we need to be able to extend the vector format to an arbitrary length. This is realised in VHDLGenerator by concatenating the random 32-bit values. The description of the concatenating algorithm can be formulated as described in figure 29.

1. Concatenate n times the 31 bit generated random value
   $n \in N$ is calculated as: $n = PortBitWidth / 31$ rounded down to next integer number.
2. Do zeros padding every time the random number does not go over the full 31 bit range.
3. Pad the remaining bits of port width with random values
   The remainder is calculated as: $remainder = PortBitWidth - n \times 31$
4. Do zero padding if generated random number does not go over the full range of the remainder width.
5. Assign concatenate random value to testbench variable.

Figure 29: concatenating algorithm description

With the described algorithm random test vectors of arbitrarily size can be constructed.

The corresponding assignment time for every test vector is generated out of the *Runtime* variable handed over from the previous stage.

The testbench serves as evaluation for the structural design. A functional verification is not possible, because VHDLGenerator does not have information on the functionality. Thus for example CRC (cyclic redundancy check) could not be tested because the *Random* class does not provide methods, that generate correct checksums.

4 The Graphical User Interface

Using the Graphical User Interface (GUI) is a comfortable way to work with VHDLGenerator. The following section gives a short introduction into the concepts of GUI programming in Java.

In the last section the VHDLGenerator GUI itself is discussed. That section is also part of the reference manual.

4.1 Java Swing vs. AWT

The Java Foundation Classes (JFC) offer two basic sets of components, used for building a GUI: The Abstract Window Toolkit (AWT) and Swing. Before starting we have to figure out which one is best to use for our case. Let us emphasize some properties of both sets and point out the main differences.

The AWT classes provide a rich set of user interface components and a robust event handling model for GUI programming. The included Layout Manager allows the creating of flexible window layouts which do not depend on a particular window size or screen resolution. The AWT components depend on native code counterparts (called peers) to handle their functionality. Therefore these components are also called *heavyweight* components in contrast to *lightweight* components which are used inside the Swing classes. The AWT delegates the painting of components as well as monitoring and controlling to the runtime system. Actual graphical operating systems like Windows XP or MacOSX got complex libraries that contain the desired components. These libraries are only available in a compiled form as binary data. Therefore you do not have the possibility to change or extend the given components.

The Swing classes contain all features of the AWT. The big difference is that the Swing components are operating system independent. All used components inherit from the class ComponentUI and are therefore purely written in Java. Components provided by the operating systems are no longer used.

We decide to take most of the components out of the Swing classes because compared to AWT, Swing got the following advantages.

- Swing Components are available for all operating systems

- Programs using the swing components look the same for all operation systems. That is useful with respect to the portability of our VHDLGenerator program. We want VHDLGenerator GUI to run also on different operating systems like Linux or Windows XP without a significant change of the GUI panel.
- Swing Components are 100% coded in Java, that implicates that we are also compatible to other hardware-platforms.
- Swing Components uses *Pluggable Look and Feel* that allows an interchangeable graphical representation.
- Swing contains more than four times as many components as AWT .

Beside all advantages there exist only some small drawbacks that are negligible. Because all components are emulated in Java the execution reduces a slightly in speed. But with regard to the working power of actual computers, the difference is not really noticeable.

4.2 The Model-View-Controller Architecture in Java

To make full use of all the Java Swing GUI power, one has to understand the underlying architecture. Swing uses a modified Model-View-Controller-Architecture (MVC). We will first explain the MVC and then show how these concepts are modified for Swing.

As it can be seen from Figure 30, the architecture is composed of the interaction of three instances:

Model: The Model describes the properties of the component. Properties are for example the colour, size or labelling.

View: The View is responsible for representing the component graphical, depending on the adjustments made before. Because the view is separated one can easily interchange the Look & Feel.

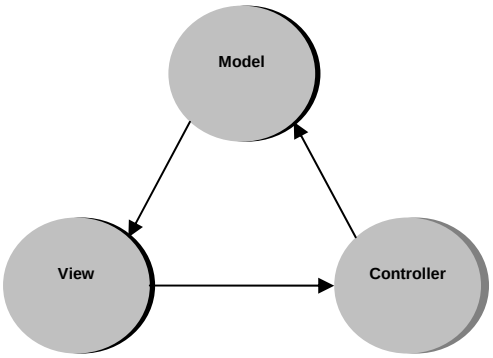

Figure 30: classical MVC architecture

Controller: The Controller is responsible for the interaction with the user. It receives an input, like a mouse click or a menu select and processes it. Processing means here that a proper action inside software takes place. The Controller also notifies the Model to update the View.

In Practice the classical division between View and Controller turned out to have a to high communication complexity. Therefore the Swing architecture uses a modified architecture.

View and Controller are merged together to the *Delegate*. Inside the Delegate, the Controller is called Listener. The Listener listens for events that take place on the component and perform the desired action.

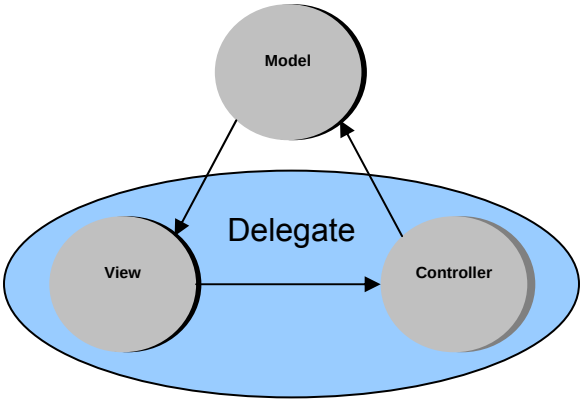

Figure 31: Swing MVC architecture

4.3 The VHDLGenerator GUI

The VHDLGenerator GUI is composed of the two classes GUIPanel and GUIFrame. GuiFrame contains the main method from which the program is started.

The entry method is the constructor of GUIPanel. Inside the GUIPanel the used components and labels are declared. We need to choose a layout manager to determine the position and size of every component.

Java Swing offers several layout managers. We choose the GridBagLayout which is a bit more complex but also more powerful layout manager. With GridBagLayout it is possible to place components at any desired horizontal or vertical position.

The panel is organized in rows and columns that form cells of arbitrary size. Each component is placed inside a cell. Figure 32 [23] shows an example how the cell organisation could look like.

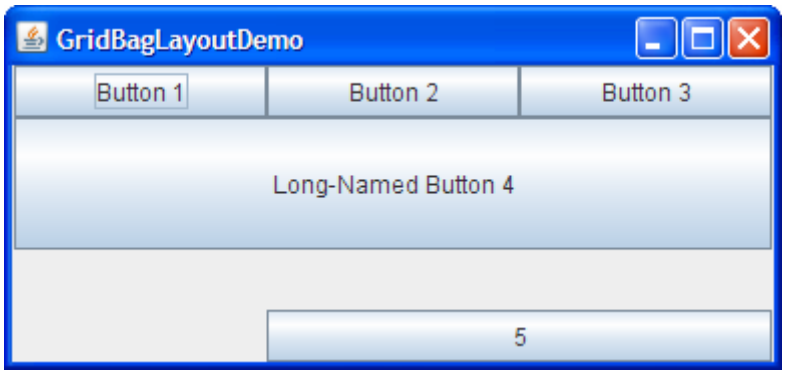

Figure 32: GridBagLayout example screenshot

GridBagLayout works basically with two objects of type GridBagLayout and GridBagConstraints that implement the model inside the MVC architecture.

The GridBagLayout is the layout manager that organizes the components on the panel based on predefined constraints. The constraints are set with the GridBagConstraints object. GridBagConstraints objects got several attributes that could be set to define the exact position and size of every component individually. We are making most use out of the attributes *gridx* and *gridy*. These two are used to define a number of the cell in x- and y-direction where you want to place the current component. With the *insets* variable the border space to adjacent components could be set. Once all desired constraints are set, the GridBagConstraints objects and our component are handed over to the GridBagLayout manager via the method *setConstraints(component,GridBagConstraints)*. Then the constraints of the next

components are set. That way we "draw" the GUI step by step with all the components we would like to include.

When designing the GUI one has to take care that the different fields and buttons are arranged in a logic and concise way. More on this topic inside the user manual in the last section of this chapter.

The execution of the model does not do much but draw the specified button, field and switches on the panel. When pressing a button nothing happens, because we have not implemented yet the Controller, i.e. accordingly to the Swing model: the Delegate.

The Delegate is formed out of methods called listeners. For every component a listener is created that reacts on actions that are performed by the user. If for example a button is pressed, the corresponding method is initiated and a specified action takes place.

4.4 GUI Reference Manual

In the following a short reference manual is given how to work with VHDLGenerator. Although the meaning of the different options or fields are pretty self-explanatory, some detailed information for documentation reason is given.

When starting the VHDLGenerator the user interface looks like in Figure 33. In VHDLGenerator the order of the program execution is directly mirrored to the GUI. That means, that information is entered in the same order, as it is processed later on.

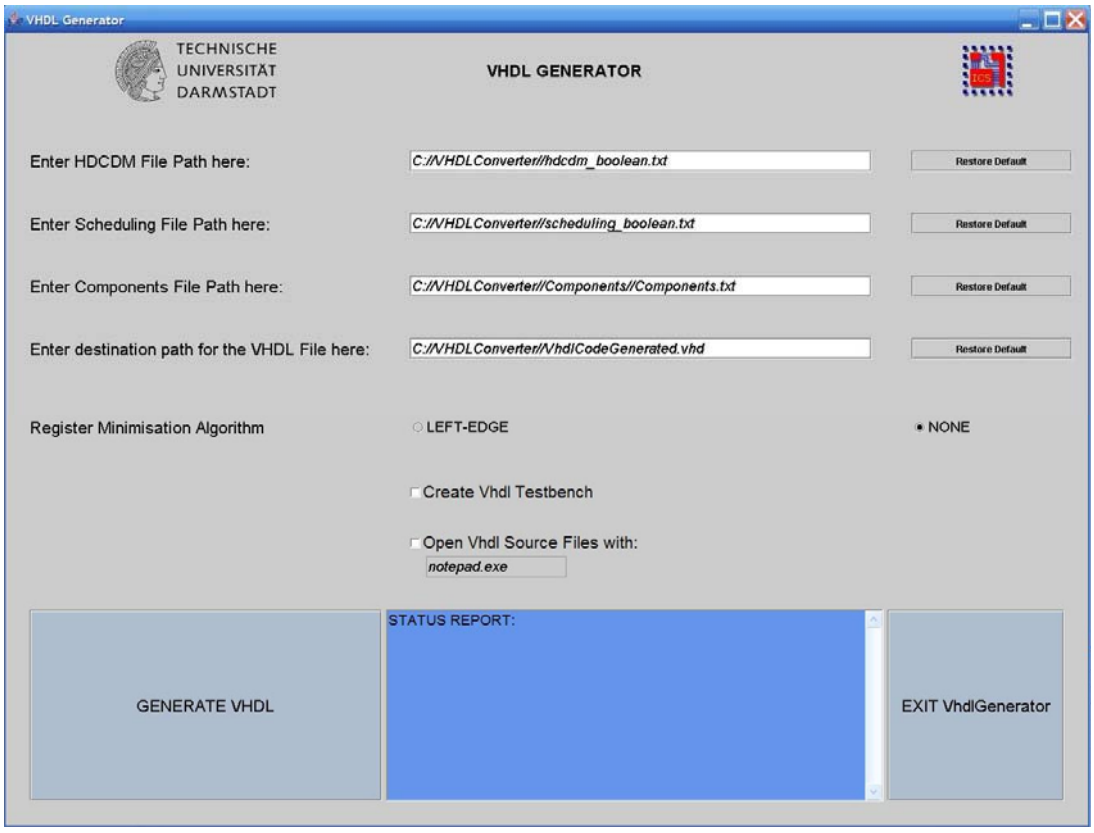

Figure 33: VHDLGenerator GUI screenshot

HCDM File Path field

In the first field the path of the HCDM file has to be entered. A default path is given, that is taken automatically, when no other path is specified. Note that when entering a new path, the directories have to be separated by "//". Otherwise the path is not taken correctly. Confirm the new entered path with "Return". The actual path is given out inside the message box.

Errors that are related with this field contain error code: 1.1. For further information please take a look at the exception table in Chapter 3.3.3.

Scheduling File Path field

The second field is for entering the directory of the Scheduling file. Here the same properties as already explained for the HCDM File Path Field hold. Errors that are related with this field contain error code: 1.2

Components File Path field

The third field is for entering the directory of the Components file. Here the same properties as already explained for the HCDM File Path Field hold. Errors that are related with this field contain error code: 1.3

VHDL File Path field

This field specifies the directory and name of the VHDL code file produced by VHDLGenerator. Errors that are related with this field contain error code: 1.4

Default Button

The Default Button sets back the file path fields to its initial value. After pressing "Enter" the path is taken. There exist four Default Buttons, one for every text field.

Register Minimization Algorithm

With this radio button one can select either Left edge algorithm or none minimization algorithm. Left edge algorithm allocates a minimum number of result registers accordingly to chapter 3.3.2. When choosing none minimization algorithm, every component result receives a separate register. The register minimization is

done inside the PreConverter stage, so possible errors that are related to this button contain error code: 1.5.

The Create VHDL Testbench checkbox

When having marked this field a testbench for the current VHDL model is created. The test bench is placed into the same directory as the generated VHDL code. For every input port five different random vectors are created.

The Open VHDL Source File checkbox

When having marked this field the produced text file specified at *VHDLFilePathfield* is opened using the "notepad.exe" of Windows or some other specified program. The File is opened after having pressed the "Generate VHDL" Button and no error occurred. If the previous "Create Testbench" checkbox is marked, then the also the generated Testbench file is opened.

The Generate Button

The Convert Button starts the VHDL converting process in the following order:

1) A HCDMParser object is created and the three text files are parsed in the order they are represented on the graphical user interface.
2) An object of type PreConverter is created. The HcdmParser object is handed over as an argument to the constructor. The status of the register minimization radio button is stored into an attribute.
   The method *parse()* applied to PreConverter object does the pre-converting job.
3) An object of type VHDLConverter is created. The PreConverter object is handed over as an argument to the constructor.
   The method *convert()* applied to VHDLConverter converts the data into VHDL code
4) Depending on the status of the "Open VHDL File" checkbox, the VHDL file is opened by "notepad.exe" or some other specified program.

The Message Field

The Message Field gives all information of the current state of the program. When new directories are entered into one of the four path fields and confirmed with "Enter", then the actual valid path of the corresponding field is given out.

When pressing the "Generate VHDL " button the user is informed about the start of the convert process. If everything performs well, a message appears, that tells that the converting process succeeded. If not an error message with the according error code is given out. The user can correct the error by means of the two error tables in chapter 3.3.3.

The Exit VHDLGenerator Button

The Exit Button closes the VHDLGenerator program.

5 Conclusions

To demonstrate the correctness of the VHDLGenerator program we construct a basic test example. The test program consists of a boolean equation and is constructed in a way that for the reader it is easy to read the results but still the full functionality of VHDLGenerator is demonstrated.

5.1 The Test Program

The realized Boolean equation contains four external inputs with 32 bit each. The Inputs are processed and combined via three different components: XOR, AND and OR component. All components are available as VHDL code and are registered inside the components entry file. Allocated are 2 AND, 1 XOR and 1 OR component.

The equation is entered into HCDM Converter as follows:

$$F(a,b,c,d,e) = (a \otimes b) \cdot c + (a \otimes b) \cdot d + (a \otimes b) \cdot e$$

The corresponding data flow graph as well as the HCDM description are shown in figure 34 and figure 35. For clarity reasons the HCDM description is reduced to its functional parts.

Notice again that all external inputs are drawn as successor tasks of the *Root* task, which is an essential condition for the parser. External outputs share the same output branch with the Root task.

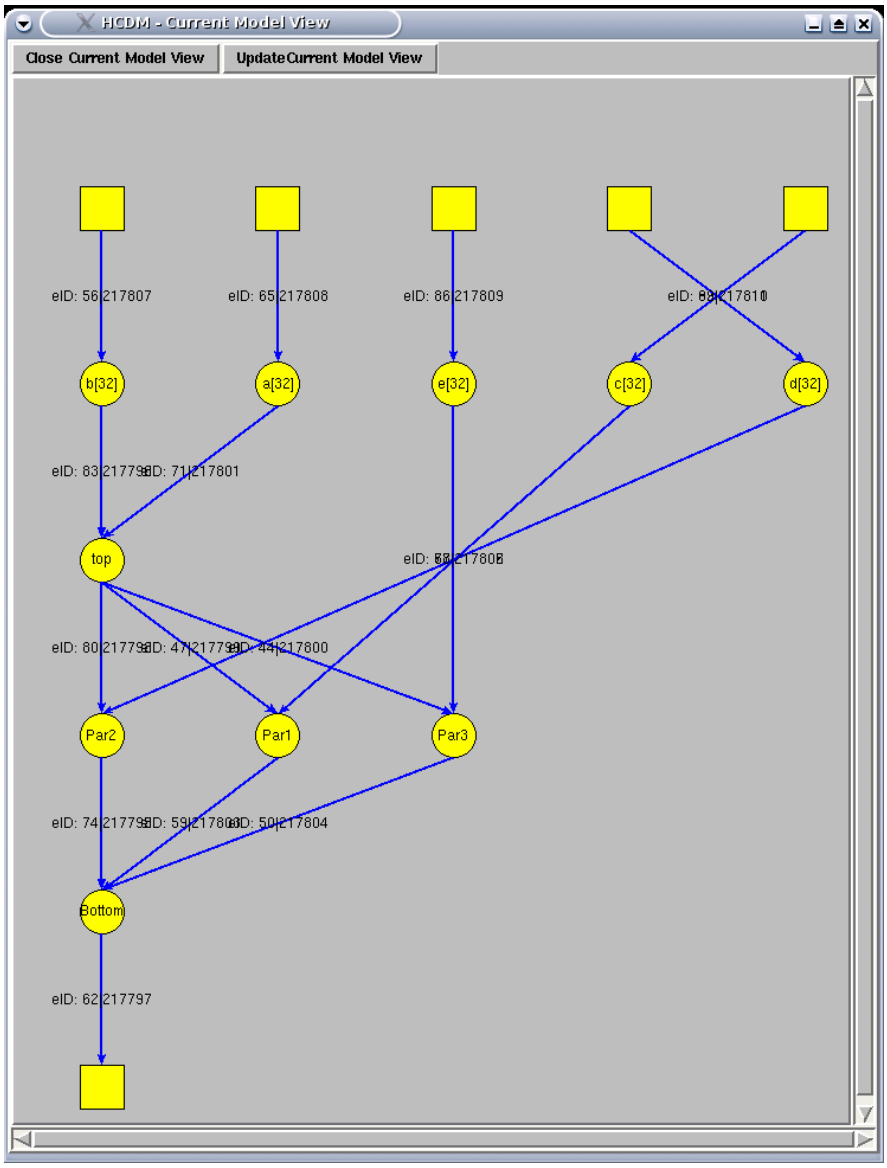

Figure 34: HCDM Boolean DFG screenshot

```
AUFGABES {
 }TASKS {
    TASK 11 {
      NAME { ROOT }
      OPTIMIZATIONTYPE { 1 }
      IMPORTANCE { 1 }
      FAILURE_PROBABILITY { 0.5 }
      LEVEL { -3 }
      PRIORITY { -4 }
      TIMING { 0 }
      RESOURCES {  }
      IORELATIONS {
         IORELATION {
            INPUT { 48 69 93 90 96  }
            OUTPUT {66  }
             CONDITION {  true  }
         }
      }
      SUBTASKS {
        TASK 14 {
           NAME { Par2 }
           OPTIMIZATIONTYPE { 1 }
           IMPORTANCE { 1 }
           FAILURE_PROBABILITY { 0.5 }
           LEVEL { -3 }
           PRIORITY { -4 }
                TIMING { 0 }
           RESOURCES {(2,20 )(5,20 ) }
           IORELATIONS {
              IORELATION {
                 INPUT { 84 181  }
                 OUTPUT {78  }
                 CONDITION {  true  }
              }
           }
            SUBTASKS {
             }
         }
         TASK 41 {
            NAME { d[32] }
            OPTIMIZATIONTYPE { 1 }
            IMPORTANCE { 1 }
            FAILURE_PROBABILITY { 0.5 }
            LEVEL { -3 }
            PRIORITY { -4 }
            TIMING { 0 }
            RESOURCES {  (3,0 ) }
            IORELATIONS {
               IORELATION {
                  INPUT { 93  }
                  OUTPUT {181  }
                 CONDITION {  true  }
              }
            }
            SUBTASKS {
                        }
```

```
TASK 17 {
  NAME { b[32] }
  OPTIMIZATIONTYPE { 1 }
  IMPORTANCE { 1 }
  FAILURE_PROBABILITY { 0.5 }
  LEVEL { -3 }
  PRIORITY { -4 }
  TIMING { 0 }
  RESOURCES {  (4,0 ) }
  IORELATIONS {
     IORELATION {
        INPUT { 48 }
        OUTPUT {87 }
        CONDITION { true  }
     }
  }
    SUBTASKS {
    }
 }
 TASK 20 {
   NAME { Bottom }
   OPTIMIZATIONTYPE { 1 }
   IMPORTANCE { 1 }
   FAILURE_PROBABILITY { 0.5 }
   LEVEL { -3 }
   PRIORITY { -4 }
   TIMING { 0 }
   RESOURCES {  (7,10 ) }
   IORELATIONS {
      IORELATION {
         INPUT { 45 51 78  }
         OUTPUT {57  }
         CONDITION { true  }
      }
   }
     SUBTASKS {
     }
  TASK 23 {
    NAME { top }
    OPTIMIZATIONTYPE { 1 }
    IMPORTANCE { 1 }
    FAILURE_PROBABILITY { 0.5 }
    LEVEL { -3 }
    PRIORITY { -4 }
    TIMING { 0 }
    RESOURCES {  (8,20 ) }
    IORELATIONS {
       IORELATION {
          INPUT { 75 87  }
          OUTPUT {84 63 60  }
          CONDITION {  true  }
       }
    }
     SUBTASKS {
     }
```

```
TASK 35 {
   NAME { Par3 }
   OPTIMIZATIONTYPE { 1 }
   IMPORTANCE { 1 }
   FAILURE_PROBABILITY { 0.5 }
   LEVEL { -3 }
   PRIORITY { -4 }
   TIMING { 0 }
   RESOURCES {  (2,20 ) (5,20 ) }
     IORELATIONS {
        IORELATION {
           INPUT { 60 81  }
           OUTPUT {45  }
           CONDITION {  true  }
     }
   }
     SUBTASKS {
}

TASK 38 {
   NAME { c[32] }
   OPTIMIZATIONTYPE { 1 }
   IMPORTANCE { 1 }
   FAILURE_PROBABILITY { 0.5 }
   LEVEL { -3 }
   PRIORITY { -4 }
   TIMING { 0 }
   RESOURCES {  (6,0 ) }
     IORELATIONS {
        IORELATION {
           INPUT { 96  }
           OUTPUT {72  }
           CONDITION {  true  }
     }
   }
     SUBTASKS {
              }
           }
 TASK 44 {
    NAME { result[32] }
    OPTIMIZATIONTYPE { 1 }
    IMPORTANCE { 1 }
    FAILURE_PROBABILITY { 0.5 }
    LEVEL { -3 }
    PRIORITY { -4 }
        TIMING { 0 }
        RESOURCES {  (159,0 ) }
          IORELATIONS {
             IORELATION {
                INPUT { 57  }
                OUTPUT {66  }
                CONDITION {  true  }
             }
          }
            SUBTASKS {
            }
           }
```

```
TASK 26 {
  NAME { a[32] }
  OPTIMIZATIONTYPE { 1 }
  IMPORTANCE { 1 }
  FAILURE_PROBABILITY { 0.5 }
  LEVEL { -3 }
  PRIORITY { -4 }
  TIMING { 0 }
  RESOURCES {  (10,0 ) }
    IORELATIONS {
       IORELATION {
          INPUT { 69  }
          OUTPUT {75  }
          CONDITION {  true  }
       }
     }
     SUBTASKS {
     }
  }
  TASK 29 {
    NAME { e[32] }
    OPTIMIZATIONTYPE { 1 }
    IMPORTANCE { 1 }
    FAILURE_PROBABILITY { 0.5 }
    LEVEL { -3 }
    PRIORITY { -4 }
    TIMING { 0 }
    RESOURCES {  (9,0 ) }
    IORELATIONS {
                IORELATION {
          INPUT { 90  }
          OUTPUT {81  }
          CONDITION {  true  }
    }
  }
    SUBTASKS {
    }
  }
  TASK 32 {
    NAME { Par1 }
    OPTIMIZATIONTYPE { 1 }
    IMPORTANCE { 1 }
    FAILURE_PROBABILITY { 0.5 }
    LEVEL { -3 }
    PRIORITY { -4 }
    TIMING { 0 }
      RESOURCES {  (2,20 ) (5,20 )
}
       IORELATIONS {
          IORELATION {
             INPUT { 63 72  }
             OUTPUT {51  }
             CONDITION {  true  }
          }
       }
     SUBTASKS {
     }
  }
```

Figure 35: HCDM description

HCDM schedules the different tasks on the components as follows:

```
true : d[32] ( 0 , 0 ), e[32] ( 0 , 0 ), c[32] ( 0 , 0
), b[32] ( 0 ,0 ), a[32] ( 0 , 0 ), top ( 0 , 20 ),
Par2 ( 20 , 40 ), Par3 ( 20 , 40 ), Par1 ( 40 , 60 ),
Bottom ( 60 , 70 ), result[32] ( 70 , 70 ),
   ------------------------------------------------
   BINDING { (Par2-myAnd_2) (d[32]-Input_1) (b[32]-
Input_2) (Bottom-myOr_1) (top-myXor_1) (a[32]-Input_3)
(e[32]-Input_4) (Par1-myAnd_1) (Par3-myAnd_1) (c[32]-
Input_5) (result[32]-Output_1)  }
```

Figure 36: HCDM Boolean example scheduling

The data is extracted and parsed like explained before. Depending whether we have chosen register minimization or non-register minimization, the received VHDL code looks different. The complete generated VHDL Code as well as the corresponding testbench are added to the appendix B.

5.2 Simulation Results

Let us first verify the correct circuit behaviour of the basic version, where we do not make use of the register minimization algorithm. We simulate the VHDL code together with the needed components by means of the generated testbench. Waveforms demonstrate the correct behaviour.

The encircled parts of the waveforms in figure 37 to 39 indicate the results of the single components. At this point in time we only acknowledge that the results are shifted to the right outputs at the specified point of time. So we can adhere that the VHDL code for scheduling and register allocation is generated correctly.

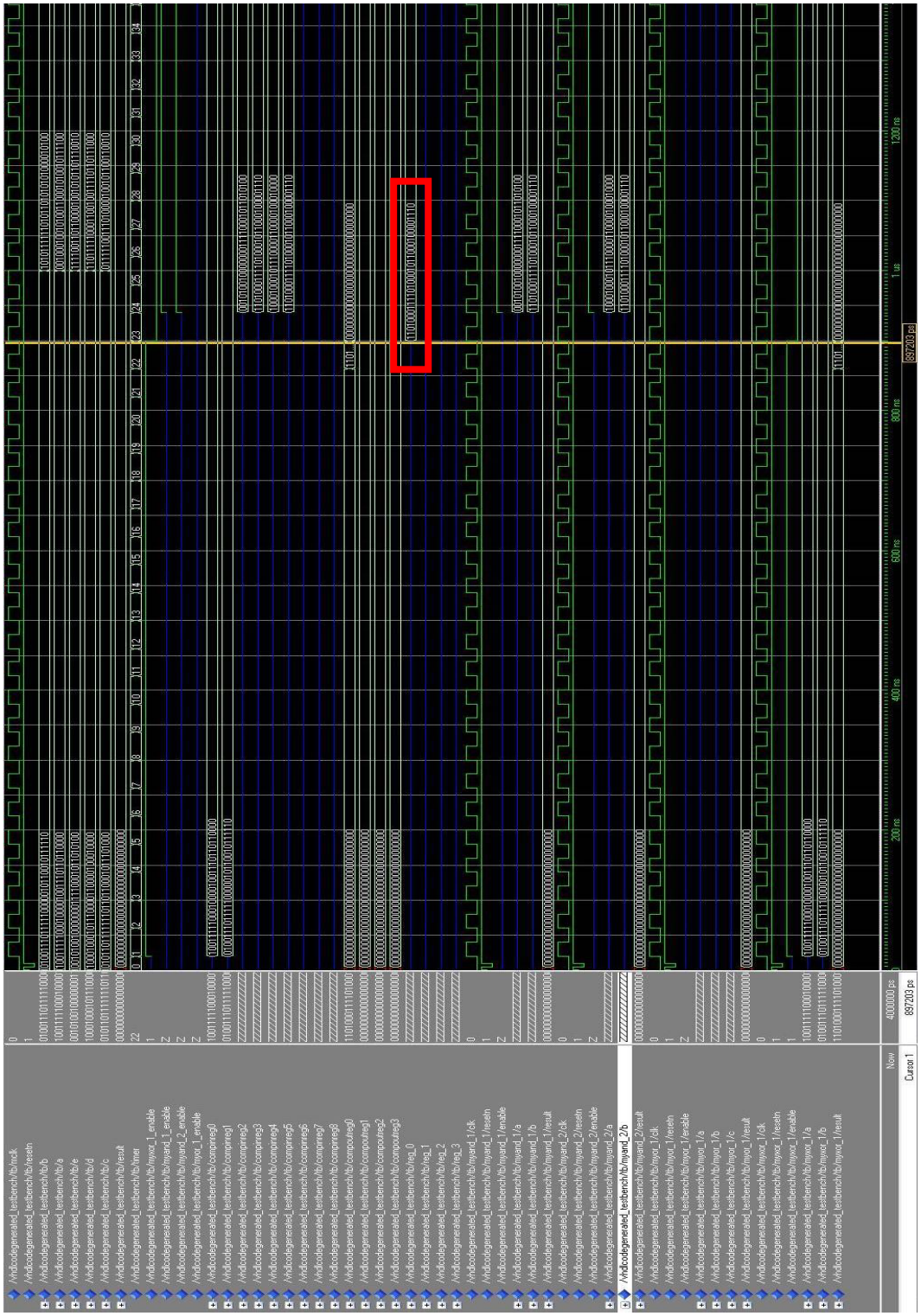

Figure 37: Result of myXOR_1

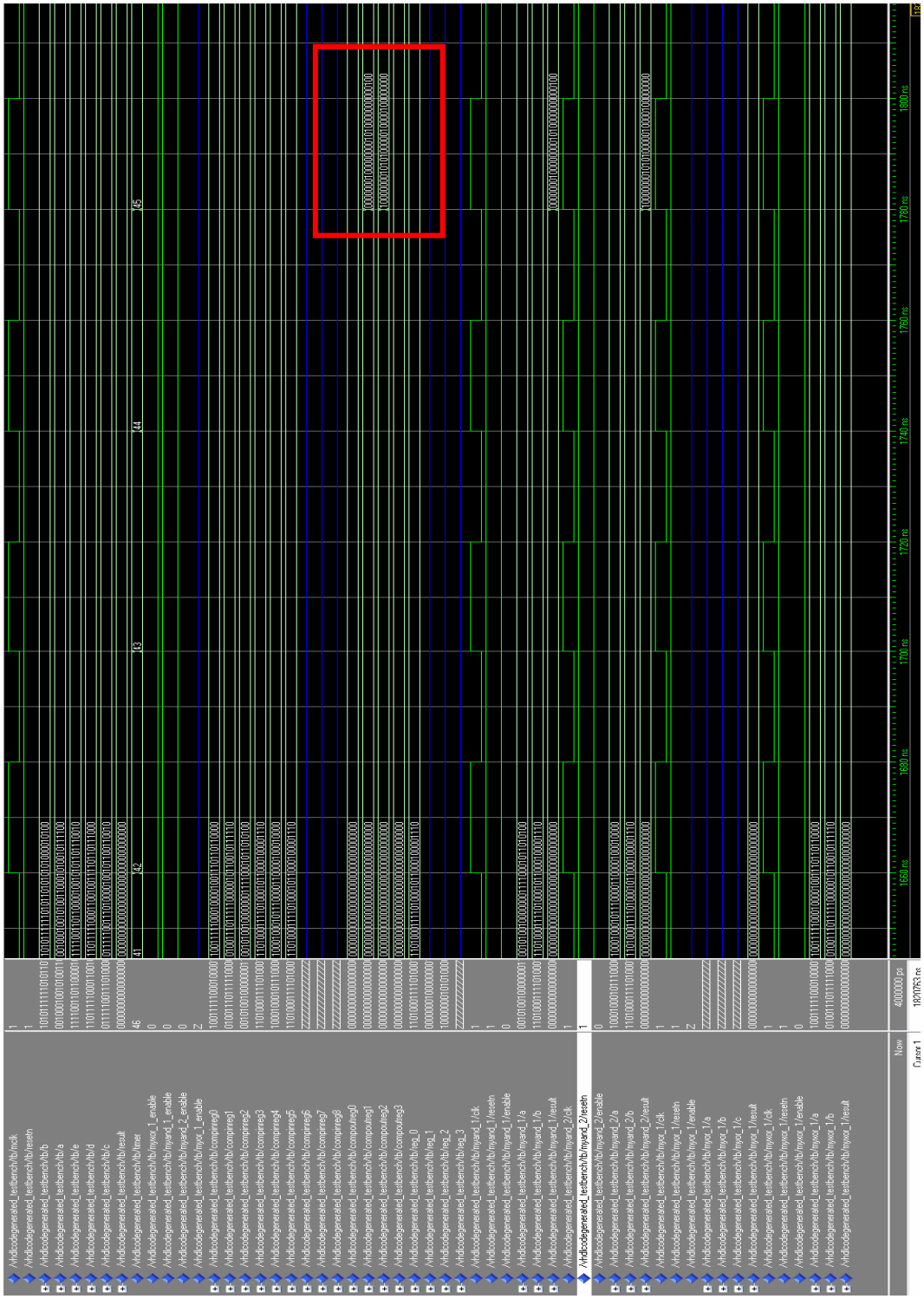

Figure 38: Result of myAND_1 and myAND_2

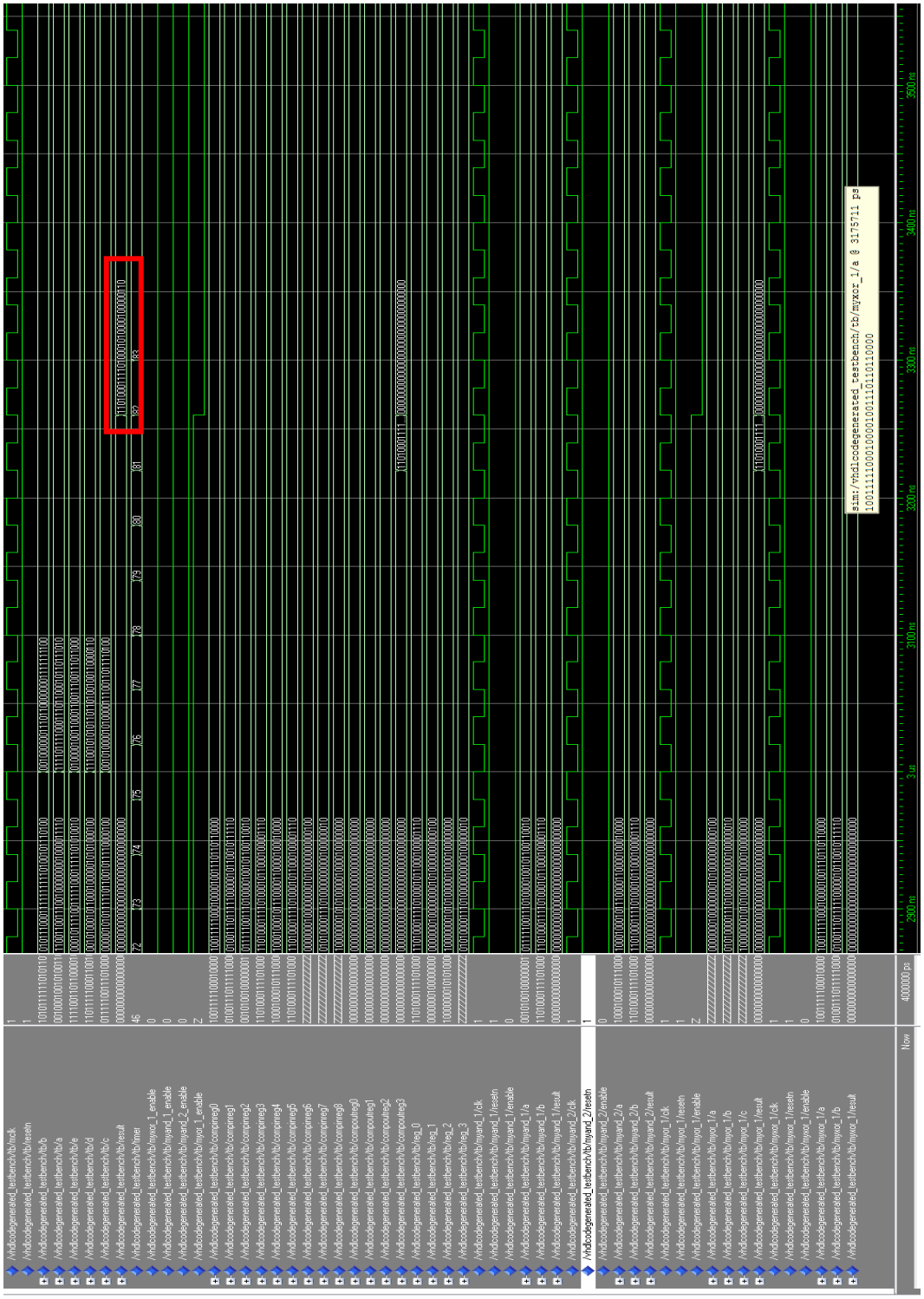

Figure 39: Figure 5.6: Final result of myOR_1

Next step would be to verify the produced program for its arithmetical results. That means we want to know whether the gained results are correct according to the specified boolean equation. For that reason we have to take a look at the random input values that are generated by the *TestbenchGenerator*:

The random port assignments for this example are:

Input Port	Assignment
a	" 10011111000100001001110110110000 "
b	" 01001110111110000010110010111110 "
c	" 01011011111110110110001011101000 "
d	" 10001000101110000110000100010000 "
e	" 00101001000000011110001011010100 "

If we put these values into the proposed boolean equation and calculate by hand we yield:

$$
\begin{aligned}
&F(a,b,c,d,e) = (a \otimes b)\cdot c + (a \otimes b)\cdot d + (a \otimes b)\cdot e \\
&= (a \otimes b)\cdot (c + d + e) \\
&= (10101101111100111111001010011010 \otimes \\
&01001110111110000010110010111110)\cdot \\
&(01011011111110110110001011101000 \\
&+ 10001000101110000110000100010000 \\
&+ 00101001000000011110001011010100) \\
&\\
&= 00111001000001010000000111010110
\end{aligned}
$$

Comparing the result computed by hand with the result delivered by the program we find consistence. Hence we can state that the generated VHDL code works correct for the tested values.

Depending on whether we have chosen register minimization or non-register minimization we obtain different VHDL source code. When comparing the two descriptions in terms of waveforms we can asses that the circuit behaviour is the same for both circuits. Note that for reasons of clarity the waveforms of the register minimized version are not added. So we can state further that using the left-edge register minimization for the VHDL code generation does not have any influence on the functional behaviour of the resulting circuit. That characteristic is a necessary condition for any minimization step you perform on digital circuits.

5.3 Synthesis Results

Next step will be to check whether the generated VHDL description passes the synthesis. In figure 40 and the Synthesis reports of register minimization and register non-minimization are opposed. The relevant parts are highlighted.

```
=========================================================
*                            Final Report
*
=========================================================
Final Results
RTL Top Level Output File Name     : VhdlCodeGenerated.ngr
Top Level Output File Name         : VhdlCodeGenerated
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 194

Cell Usage :
# BELS                             : 371
#      GND                         : 1
#      INV                         : 2
#      LUT1                        : 9
#      LUT2                        : 14
#      LUT2_D                      : 2
#      LUT2_L                      : 1
#      LUT3                        : 178
#      LUT3_D                      : 2
#      LUT3_L                      : 7
#      LUT4                        : 84
#      LUT4_D                      : 13
#      LUT4_L                      : 36
#      MUXCY                       : 9
#      MUXF5                       : 3
#      VCC                         : 1
#      XORCY                       : 9

# FlipFlops/Latches                : 612

#      FDC                         : 160
#      FDCE                        : 32
#      FDE                         : 420
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 193
#      IBUF                        : 161
#      OBUF                        : 32
=========================================================
Device utilization summary:
---------------------------

Selected Device : 3s200ft256-5

 Number of Slices:                401  out of   1920    20%
 Number of Slice Flip Flops:      612  out of   3840    15%
 Number of 4 input LUTs:          348  out of   3840     9%
 Number of IOs:                   194
 Number of bonded IOBs:           194  out of    173   112% (*)
 Number of GCLKs:                 1  out of      8    12%
=========================================================
```

Figure 40: Synthesis report without register

```
=========================================================
*                      Final Report
                       *
   =========================================================
Final Results
RTL Top Level Output File Name             : VhdlCodeGenerated.ngr
Top Level Output File Name                                 :
VhdlCodeGenerated
Output Format              : NGC
Optimization Goal          : Speed
Keep Hierarchy             : NO

Design Statistics
# IOs                                 : 194

Cell Usage :
# BELS                                : 545
#      GND                           : 1
#      INV                           : 2
#      LUT1                          : 9
#      LUT2                          : 7
#      LUT2_L                        : 1
#      LUT3                          : 175
#      LUT3_D                        : 8
#      LUT3_L                        : 9
#      LUT4                          : 236
#      LUT4_D                        : 9
#      LUT4_L                        : 5
#      MUXCY                         : 9
#      MUXF5                         : 64
#      VCC                           : 1
#      XORCY                         : 9

# FlipFlops/Latches                  : 593

#      FDC                           : 173
#      FDCE                          : 32
#      FDE                           : 388
# Clock Buffers                      : 1
#      BUFGP                         : 1
# IO Buffers                         : 193
#      IBUF                          : 161
#      OBUF                          : 32
=========================================================
Device utilization summary:
---------------------------

Selected Device : 3s200ft256-5

 Number of Slices:              434  out of   1920    22%
 Number of Slice Flip Flops:    593  out of   3840    15%
 Number of 4 input LUTs:        461  out of   3840    12%
 Number of IOs:                 194
 Number of bonded IOBs:         194  out of    173   112% (*)
 Number of GCLKs:               1  out of      8    12%
=========================================================
```

Figure 41: Synthesis report with left edge register minimization

A comparison of both reports yields that the left edge register minimization reduces the number of needed flip flops on the target device.

If we take a closer look to the produced codes we can figure why the second version needs less resources. In the figures above the segment where the temporary output registers are stated are shown for each code version.

The non-minimized version needs one output register for each of the four instantiated VHDL component

```
  .....
  signal reg_0: std_logic_vector(31 downto 0);
  signal reg_1: std_logic_vector(31 downto 0);
  signal reg_2: std_logic_vector(31 downto 0);
  signal reg_3: std_logic_vector(31 downto 0);

 begin
    .....
```

Figure 42: VHDL code without register minimization

In comparison the LEFT EDGE ALGORITHM reduces the number of needed registers to three.

```
      .....
      signal reg_0: std_logic_vector(31 downto 0);
      signal reg_1: std_logic_vector(31 downto 0);
      signal reg_2: std_logic_vector(31 downto 0);
begin
   .....
```

Figure 43: VHDL code with Left Edge register minimization

So we conclude that the reduced amount of needed output registers, generated by the LEFT EDGE Algorithm, is the cause for reduced amount of used flip flops.

In contrast the number of LUT4 is increasing compared to the non-optimized variant. The additional resources are used for multiplexing the inputs for the registers.

Depending on the application the user can decide whether to have minimized flip flops or less usage of combinatorial logic. In most applications the Flip Flops will be a more sever resource constraint.

Appendix A

HCDM describes the task graph in terms of bnf-grammar (Backus-Naur form), that allows a flexible number of parameters. Note that here the complete HCDM grammar is stated, although we will only take care of some of the fields. The modified grammar used for the parser is stated in chapter 3.2.2.3 .

$$
\begin{array}{rcl}
< hcdm > & \rightarrow & \text{'HCDM'} < id > \text{'}\{\text{'} < params > \text{'}\}\text{'}. \\
 & & < resourcelist > \\
 & & < behaviorlist > \\
 & & < tasklist > \\
 & & < edgelist > \\
 & & \text{'}\}\text{'}. \\
< resourcelist > & \rightarrow & \text{'RESOURCES }\{\text{'} < resorces > \text{'}\}\text{'}. \\
< resources > & \rightarrow & \emptyset | < resource >< resources >. \\
< resource > & \rightarrow & \text{'RESOURCE'} < id > \text{'}\{\text{'} < params > \text{'}\}\text{'}. \\
< behaviorlist > & \rightarrow & \text{'BEHAVIORS }\{\text{'} < behaviors > \text{'}\}\text{'}. \\
< behaviors > & \rightarrow & \emptyset | < behavior >< behaviors >. \\
< behavor > & \rightarrow & \text{'BEHAVIOR'} < id > \text{'}\{\text{'} \\
 & & \text{'NAME }\{\text{'} < string > \text{'}\}\text{'} \\
 & & \text{'SIZE }\{\text{'} < integer > \text{'}\}\text{'} \\
 & & < params > \text{'}\}\text{'}. \\
< tasklist > & \rightarrow & \text{' TASKS }\{\text{'} < tasks > \text{'}\}\text{'}. \\
< tasks > & \rightarrow & \emptyset | < task >< tasks >. \\
< task > & \rightarrow & \text{'TASK'} < id > \text{'}\{\text{'} < params > \\
 & & \text{'IORELATIONS }\{\text{'} < iorelations > \text{'}\}\text{'} \\
 & & \text{'SUBTASKS }\{\text{'} < tasks > \text{'}\}\text{'} \\
 & & \text{'}\}\text{'}. \\
< iorelations > & \rightarrow & \emptyset | < iorelation >< iorelations >.
\end{array}
$$

$$
\begin{array}{rcl}
< iorelation > & \rightarrow & \text{' IORELATION \{} \\
 & & \text{'INPUT \{'} < idlist > \text{'\}'} \\
 & & \text{'OUTPUT \{'} < idlist > \text{'\}'} \\
 & & \text{'CONDITION \{'} < conditionlist > \text{'\}'} \\
 & & \text{'\}'.} \\
< edgelist > & \rightarrow & \text{'EDGES \{} < edges > \text{\}.} \\
< edges > & \rightarrow & \emptyset | < edge >< edges >. \\
< edge > & \rightarrow & \text{'HEDGE'} < id > \text{'\{} \\
 & & \text{'NODES \{'} < idlist > \text{'\}'} \\
 & & < params > \text{'\}'.} \\
< params > & \rightarrow & \emptyset | < param >< params >. \\
< param > & \rightarrow & < string > \text{'\{'} < string > \text{'\}'.} \\
< condition > & \rightarrow & \text{'true'} \mid \text{'false'} \mid < conditionlist >. \\
< conditionlist > & \rightarrow & \text{'\{'} < statesequences > \text{'\}'.} \\
< statesequences > & \rightarrow & \emptyset | \text{'\{'} < statesequence > \text{'\}'} < statesequences > . \\
< statesequence > & \rightarrow & \emptyset | < cond >< statesequence >. \\
< cond > & \rightarrow & \text{'('} < integer > \text{' '} < integer > \text{')'.} \\
< idlist > & \rightarrow & \emptyset | < integer > \text{' '} < idlist >. \\
< integer > & \rightarrow & \emptyset | < digit >< integer >. \\
< digit > & \rightarrow & \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'} \mid \text{'5'} \mid \text{'6'} \mid \text{'7'} \mid \text{'8'} \mid \text{'9'} \mid \text{'0'.} \\
< string > & \rightarrow & \emptyset | < allascii - codesexl. >< string >.
\end{array}
$$

In the following the parameter, that are provided by each component:

• hCDM

– NAME -The name.

– ROOT - ID of the root task.

– CURRENT ROOT - ID of current root.

– CURRENT LEAFS - list of Task-ID of node of actual detail grade

• Resource

– NAME - The name.

– COST - costs of this resource.

– NUMBER - maximum amount that are allowed to use

– TYPE -communication or functional resource.

– CONNECTED - list of functional resources that are connected via a communication resource.

• Behavior class

– NAMES - Names of each behaviours that are implemented by this class

• Task

– NAME - The name.

– PRIORITY - priority for list scheduling.

– TIMING - default time behaviour, as long as no component binding exist
– RESOURCES - a list of resources on which the task can be implemented
– LEVEL - parameter for the graphical representation
– OPTIMIZATIONTYPE - Kommunikations- oder funktionaler Task.
– IMPORTANCE - importance of this task for error free function of the system
– FAILURE PROBABILITY - breakdown probability of this task

Appendix B

The generated VHDL code for the register minimized version and the non-minimized version are stated below. The testbench as well as the used VHDL components are the same for both. First the generated VHDL code **without** register optimization then the code **with** register optimization are stated:

```
library ieee;
use ieee.std_logic_1164.all;

entity VhdlCodeGenerated is

  port( mclk: in std_logic;
        resetn: in std_logic;
        a : in std_logic_vector(31 downto 0);
        b : in std_logic_vector(31 downto 0);
        e : in std_logic_vector(31 downto 0);
        d : in std_logic_vector(31 downto 0);
        c : in std_logic_vector(31 downto 0);
        result : out std_logic_vector(31 downto 0));
end VhdlCodeGenerated;

architecture arc of VhdlCodeGenerated is

   component myXor is
      port(   clk :in std_logic;
              resetn: in std_logic;
                  enable: in std_logic;
              a :in std_logic_vector(31 downto 0);
              b :in std_logic_vector(31 downto 0);
              result :out std_logic_vector(31 downto 0));
   end component myXor;

       component myAnd is
      port(   clk :in std_logic;
              resetn: in std_logic;
                  enable: in std_logic;
              a :in std_logic_vector(31 downto 0);
              b :in std_logic_vector(31 downto 0);
              result :out std_logic_vector(31 downto 0));
   end component myAnd;

   component myOr is
      port(   clk :in std_logic;
              resetn: in std_logic;
                  enable: in std_logic;
              a :in std_logic_vector(31 downto 0);
              b :in std_logic_vector(31 downto 0);
              c :in std_logic_vector(31 downto 0);
              result :out std_logic_vector(31 downto 0));
   end component myOr;
```

```
 signal timer: integer range 0 to 1000;

   signal myXor_1_enable: std_logic := 'Z';
   signal myAnd_1_enable: std_logic := 'Z';
   signal myAnd_2_enable: std_logic := 'Z';
   signal myOr_1_enable: std_logic := 'Z';

   signal CompInReg0: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg1: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg2: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg3: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg4: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg5: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg6: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg7: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg8: std_logic_vector(31 downto 0) := (others=>'Z');

   signal CompOutReg0: std_logic_vector(31 downto 0):= (others => 'Z');
   signal CompOutReg1: std_logic_vector(31 downto 0):= (others => 'Z');
    signal CompOutReg2: std_logic_vector(31 downto 0):= (others => 'Z');
   signal CompOutReg3: std_logic_vector(31 downto 0):= (others => 'Z');


   signal reg_0: std_logic_vector(31 downto 0):= (others => 'Z');
   signal reg_1: std_logic_vector(31 downto 0):= (others => 'Z');
   signal reg_2: std_logic_vector(31 downto 0):= (others => 'Z');
   signal reg_3: std_logic_vector(31 downto 0):= (others => 'Z');

begin

   myXor_1: myXor
      port map (clk => mclk,resetn => resetn,enable => myXor_1_enable,a =>
               CompInReg0,b => CompInReg1,result => CompOutReg0);

   myAnd_1: myAnd
      port map (clk => mclk,resetn => resetn,enable => myAnd_1_enable,a =>
               CompInReg2,b => CompInReg3,result => CompOutReg1);


   myAnd_2: myAnd
      port map (clk => mclk,resetn => resetn,enable => myAnd_2_enable,a =>
               CompInReg4,b =>CompInReg5,result => CompOutReg2);

   myOr_1: myOr
      port map (clk => mclk,resetn => resetn,enable => myOr_1_enable,a =>
               CompInReg6,b => CompInReg7,c => CompInReg8,result =>
                                          CompOutReg3);
```

```
process(mclk,resetn)
   begin

      if resetn='0' then

           timer<= 0 ;
           result <= (others => '0');

      elsif mclk'event and mclk='1' then

           case timer is

                      when 0 =>  CompInReg0 <= a;
                                 CompInReg1 <= b;
                                 myXor_1_enable <= '1';
                                 timer <=timer+1;

                      when 22 =>  reg_0 <= CompOutReg0;
                                 myXor_1_enable <= '0';
                                 timer <=timer+1;

                      when 23 =>  CompInReg2 <= e;
                                 CompInReg3 <= reg_0;
                                 myAnd_1_enable <= '1';
                                 CompInReg4 <= d;
                                 CompInReg5 <= reg_0;
                                 myAnd_2_enable <= '1';
                                 timer <=timer+1;

                      when 45 =>  reg_1 <= CompOutReg1;
                                 myAnd_1_enable <= '0';
                                 reg_2 <= CompOutReg2;
                                 myAnd_2_enable <= '0';
                                 timer <=timer+1;

                      when 46 =>  CompInReg2 <= c;
                                 CompInReg3 <= reg_0;
                                 myAnd_1_enable <= '1';
                                 timer <=timer+1;

                      when 68 =>  reg_3 <= CompOutReg1;
                                 myAnd_1_enable <= '0';
                                 timer <=timer+1;

                      when 69 =>  CompInReg6 <= reg_1;
                                 CompInReg7 <= reg_3;
                                 CompInReg8 <= reg_2;
                                 myOr_1_enable <= '1';
                                 timer <=timer+1;

                      when 81 =>  result <= CompOutReg3;
                                 myOr_1_enable <= '0';
                                 timer <=timer+1;

                      when 83 =>  timer <= 83; -- end of Schedule

                      when others => timer<=timer+1;  --only count up

           end case;
```

```
        end if;

    end process;

 end arc;
```

Figure 44: complete VHDL code without register minimization

Next the VHDL code **with** register optimization is stated:

```
library ieee;
use ieee.std_logic_1164.all;

entity VhdlCodeGenerated is

  port( mclk: in std_logic;
        resetn: in std_logic;
        b : in std_logic_vector(31 downto 0);
        a : in std_logic_vector(31 downto 0);
        e : in std_logic_vector(31 downto 0);
        d : in std_logic_vector(31 downto 0);
        c : in std_logic_vector(31 downto 0);
        result : out std_logic_vector(31 downto 0));

end VhdlCodeGenerated;

architecture arc of VhdlCodeGenerated is

   component myXor is
      port(   clk :in std_logic;
              resetn: in std_logic;
                  enable: in std_logic;
              a :in std_logic_vector(31 downto 0);
              b :in std_logic_vector(31 downto 0);
              result :out std_logic_vector(31 downto 0));
   end component myXor;

   component myAnd is
      port(   clk :in std_logic;
              resetn: in std_logic;
                  enable: in std_logic;
              a :in std_logic_vector(31 downto 0);
              b :in std_logic_vector(31 downto 0);
              result :out std_logic_vector(31 downto 0));
   end component myAnd;

   component myOr is
      port(   clk :in std_logic;
              resetn: in std_logic;
                  enable: in std_logic;
              a :in std_logic_vector(31 downto 0);
              b :in std_logic_vector(31 downto 0);
              c :in std_logic_vector(31 downto 0);
              result :out std_logic_vector(31 downto 0));
   end component myOr;
```

```
signal timer: integer range 0 to 1000;

   signal myXor_1_enable: std_logic := 'Z';
   signal myAnd_1_enable: std_logic := 'Z';
   signal myAnd_2_enable: std_logic := 'Z';
   signal myOr_1_enable: std_logic := 'Z';

   signal CompInReg0: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg1: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg2: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg3: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg4: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg5: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg6: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg7: std_logic_vector(31 downto 0) := (others=>'Z');
   signal CompInReg8: std_logic_vector(31 downto 0) := (others=>'Z');

   signal CompOutReg0: std_logic_vector(31 downto 0):= (others => 'Z');
   signal CompOutReg1: std_logic_vector(31 downto 0):= (others => 'Z');
   signal CompOutReg2: std_logic_vector(31 downto 0):= (others => 'Z');
   signal CompOutReg3: std_logic_vector(31 downto 0):= (others => 'Z');


   signal reg_0: std_logic_vector(31 downto 0):= (others => 'Z');
   signal reg_1: std_logic_vector(31 downto 0):= (others => 'Z');
   signal reg_2: std_logic_vector(31 downto 0):= (others => 'Z');

begin

   myXor_1: myXor
      port map (clk => mclk,resetn => resetn,enable => myXor_1_enable,a
               => CompInReg0,b => CompInReg1,result => CompOutReg0);

   myAnd_1: myAnd
      port map (clk => mclk,resetn => resetn,enable => myAnd_1_enable,a
               => CompInReg2,b => CompInReg3,result => CompOutReg1);

   myAnd_2: myAnd
      port map (clk => mclk,resetn => resetn,enable => myAnd_2_enable,a
               => CompInReg4,b => CompInReg5,result => CompOutReg2);

   myOr_1: myOr
      port map (clk => mclk,resetn => resetn,enable => myOr_1_enable,a
        => CompInReg6,b => CompInReg7,c => CompInReg8,result
        =>CompOutReg3);
```

```
process(mclk,resetn)
   begin

      if resetn='0' then

           timer<= 0 ;
           result <= (others => '0');

      elsif mclk'event and mclk='1' then

           case timer is
                       when 0 =>  CompInReg0 <= b;
                                  CompInReg1 <= a;
                                  myXor_1_enable <= '1';
                                  timer <=timer+1;

                      when 22 =>  reg_0 <= CompOutReg0;
                                  myXor_1_enable <= '0';
                                  timer <=timer+1;

                      when 23 =>  CompInReg2 <= e;
                                  CompInReg3 <= reg_0;
                                  myAnd_1_enable <= '1';
                                  CompInReg4 <= d;
                                  CompInReg5 <= reg_0;
                                  myAnd_2_enable <= '1';
                                  timer <=timer+1;

                      when 45 =>  reg_1 <= CompOutReg1;
                                  myAnd_1_enable <= '0';
                                  reg_2 <= CompOutReg2;
                                  myAnd_2_enable <= '0';
                                  timer <=timer+1;

                      when 46 =>  CompInReg2 <= c;
                                  CompInReg3 <= reg_0;
                                  myAnd_1_enable <= '1';
                                  timer <=timer+1;

                      when 68 =>  reg_0 <= CompOutReg1;
                                  myAnd_1_enable <= '0';
                                  timer <=timer+1;

                      when 69 =>  CompInReg6 <= reg_1;
                                  CompInReg7 <= reg_0;
                                  CompInReg8 <= reg_2;
                                  myOr_1_enable <= '1';
                                  timer <=timer+1;

                      when 81 =>  result <= CompOutReg3;
                                  myOr_1_enable <= '0';
                                  timer <=timer+1;


                      when 83 =>  timer <= 83; -- end of Schedule

                      when others => timer<=timer+1;  --only count up

           end case;


   end if;

      end process;

   end arc;
```

Figure 45: complete VHDL code with register minimization

The used components for the boolean equation look the same for both versions:

The XOR component:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity myXor is
	port (
		clk, resetn : in std_logic;
		enable : in std_logic;
		a: in std_logic_vector(31 downto 0);
		b: in std_logic_vector(31 downto 0);
		result: out std_logic_vector(31 downto 0)
		);
end myXor;

architecture RTL of myXor is

begin

	process (clk, resetn)
		variable delay : integer range 0 to 31;
	begin
		if resetn='0' then
			delay := 0;
			result <= ( others => '0');
		elsif CLK'event and CLK='1' then
			if enable='1' then
			   if delay = 20 then
					result <= a xor b;
					delay := 0;
			   else
					delay := delay +1;
					result <= ( others => '0');
			   end if;
			else
			   delay := 0;
			   result <= ( others => '0');
			end if;

		end if;
	end process;

end RTL;
```

Figure 46: The used XOR component

The OR Component:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity myOR is
	port (
		clk, resetn : in std_logic;
		enable: in std_logic;
		a: in std_logic_vector(31 downto 0);
		b: in std_logic_vector(31 downto 0);
		c: in std_logic_vector(31 downto 0);
		result: out std_logic_vector(31 downto 0)
		);
end myOR;

architecture RTL of myOR is

begin

	process (clk, resetn)
		variable delay : integer range 0 to 31;
	begin
		if resetn='0' then
			delay := 0;
			result <= ( others => '0');
		elsif CLK'event and CLK='1' then
			if enable = '1' then
			   if delay = 10 then
				result <= a or b or c;
				delay := 0;
			   else
				delay := delay +1;
				result <= ( others => '0');
			   end if;
			else
			  delay := 0;
			  result <= ( others => '0');
			end if;
		end if;
	end process;

end RTL;
```

Figure 47: The used OR component

The AND Component:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity myAND is
	port (
		clk, resetn : in std_logic;
		enable: in std_logic;
		a: in std_logic_vector(31 downto 0);
		b: in std_logic_vector(31 downto 0);
		result: out std_logic_vector(31 downto 0)
		);
end myAND;

architecture RTL of myAND is

begin

	process (clk, resetn)
		variable delay : integer range 0 to 31;
	begin
		if resetn='0' then
			delay := 0;
			result <= ( others => '0');
		elsif CLK'event and CLK='1' then
			if enable='1' then
			   if delay = 20 then
				result <= a and b;
				delay := 0;
			   else
				delay := delay +1;
				result <= (others => '0');
			   end if;
			else
			   delay := 0;
			   result <= ( others => '0');
			end if;
		end if;
	end process;

end RTL;
```

Figure 48: The used AND component

The testbench is the same for both versions:

```
library ieee;
use ieee.std_logic_1164.all;

entity VhdlCodeGenerated_Testbench is
end VhdlCodeGenerated_Testbench;

architecture tb_arc of VhdlCodeGenerated_Testbench is

  component VhdlCodeGenerated  is
  port(mclk :in std_logic;
      resetn: in std_logic;
      a : in std_logic_vector(31 downto 0);
      b : in std_logic_vector(31 downto 0);
      e : in std_logic_vector(31 downto 0);
      d : in std_logic_vector(31 downto 0);
      c : in std_logic_vector(31 downto 0);
      result : out std_logic_vector(31 downto 0));
  end component VhdlCodeGenerated ;

  signal tb_clk: std_logic :='0';
  signal tb_resetn: std_logic;
  signal tb_a: std_logic_vector(31 downto 0):= (others => 'Z');
  signal tb_b: std_logic_vector(31 downto 0):= (others => 'Z');
  signal tb_e: std_logic_vector(31 downto 0):= (others => 'Z');
  signal tb_d: std_logic_vector(31 downto 0):= (others => 'Z');
  signal tb_c: std_logic_vector(31 downto 0):= (others => 'Z');
  signal tb_result: std_logic_vector(31 downto 0):= (others =>'Z');

begin

  tb: VhdlCodeGenerated
    port map (mclk => tb_clk,resetn => tb_resetn, a => tb_a, b => tb_b, e => tb_e, d => tb_d, c => tb_c,
result =>                tb_result);

  tb_clk  <= not tb_clk after 20 ns; -- clock working at 50Mhz

  tb_resetn <= '1' after 0 ns,
               '0' after 5 ns,
               '1' after 10 ns;

  tb_a <= "10011111000100001001110110110000" after 0 ns,
          "00100010010100110001010010111100" after 1000 ns,
          "11001100111001000000001000011110" after 2000 ns,
          "11110111100011101100010110111010" after 3000 ns,
          "11010110010001111110100110110110" after 4000 ns;

  tb_b <= "01001110111110000010110010111110" after 0 ns,
          "10101111110101101010101000010100" after 1000 ns,
          "01011100011111111110100101110100" after 2000 ns,
          "00100000011101100000001111111100" after 3000 ns,
          "00100000011011010100000011000110" after 4000 ns;

  tb_e <= "00101001000000011110001011010100" after 0 ns,
          "11110011011000010010101101110010" after 1000 ns,
          "00010111100111110001111101010010" after 2000 ns,
          "01000010011000110011100111011000" after 3000 ns,
          "10101001011010110011100101110110" after 4000 ns;
```

```
  tb_d <= "1000100010111
          "1101111100011
          "0010100010110
          "1110010101011
          "0001100110011

  tb_c <= "0101101111
          "0111110011101
          "0000010101111
          "0010100001010
          "0101100001010

end tb_arc;
```

Figure 49: The generated VHDL testbench

References

[1] Klaus, S. (2006). *System -Level -Entwurfsmethodik eingebetteter Systeme.* Aachen: Shaker Verlag, pp. 101-106, pp. 129-131.

[2] DeMichelli, G (1994). *Synthesis and Optimization of Digital Circuits.* New York: McGraw-Hill Inc, pp. 37- 42, pp. 100-102, pp.119-123, p.240, pp. 61-64, pp. 64-67.

[3] Rozenberg,G. and Salomaa,A. (1997). *Handbook of Formal Languages*. Vol.1 Word Language Grammar. Berlin: Springer-Verlag.

[4] Reps,T. (2001). *Maximal-Munch" Tokenization in Linear Time*. University of Wisconsin, USA, September 2001, pp. 1-3.

[5] Lutz, M. and Schmitt,F. J. (1997). *Vom Prozessor zum Programm*. München: Carl Hanser Verlag, pp.90-103.

[6] Ralston, A. and Reilly,E.D. and Dahlin,C.A. ed. (1993). *Encyclopedia of Computer Science* .Third Edition. New York: Van Nostrand Reinhold , pp. 207-208.

[7] Tremblay,J.-P. and Sorenson,P.G. (1985). *The Theory and Practice of Compiler Writing*. New York: McGraw-Hill, pp.5-8, pp.138- 150, pp. 182-196.

[8] Aho,A.V. and Sethi,R. and Ullman,J.D. (1986). *Compilers Principles,Techniques, and Tools*. Reading: Addison-Wesley, pp.1-15, pp.83-86.

[9] Atallah,M.J. (1999). *Algorithms and Theory of Computation Handbook*. London: CRC Press LLC, pp. 3.1-3.25.

[10] Lamont,M. . *Sorting Algorithms* [online]. Available at: http://linux.wku.edu/~lamonml/algor/sort/sort.html [Accessed 23 July 2006].

[11] C.A.R. Hoare (1962): *Quicksort*. Computer Journal, Vol. 5, 1, pp.10-15.

[12] Hehner,E.C.R. (1993). *A Practical Theory of Programming*. New York: Springer-Verlag, pp.67-68.

[13] A. Hashimoto and J. Stevens (1971). *Wire routing by optimizing channel assignment within large apertures*. **Proceedings of the 8th Design Automation Workshop**. Atlantic City, USA, June 1971.

[14] Kurdahi,F.J. and Parker,A.C. (1987). *REAL: A Program for REgister ALlocation*. **IEEE Design Automation Conference**. LosAngeles, USA, CA 90089-0781.

[15] *Automated Synthesis of Data Paths in Digital Systems*. **IEEE Transactions on Computer-Aided Design,** Volume 5, July 1986.

[16] Lin,T.C. and Cyre,W.R. (1997). *An algorithm based on the Hungarian method for register reduction during complex functional unit allocation*. **Southeastcon'97. 'Engineering new New Century': Proceedings IEEE**. Blacksburg, USA, April 1997.

[17] Zhang,S. and Dai,W.W.-M. (2000). *Linear Time Left Edge Algorithm*. **INT'L Conference on Chip Design Automation: Proceedings of ICDA2000**. Beijing, China, February 2000.

[18] Cooke,J (1998). *Constructing Correct Software - the basics*. London: Springer-Verlag, pp. 10-11.

[19] Oh,M. (1999). *Transformation and VHDL Code Generation from Coarse-grained Data Flow Graph* [online]. Department of Computer Engineering, Seoul National University. Available at: http://peace.snu.ac.kr/publications/data/142/ms_1999_oh_moonwook.pdf [Accessed at 14.September 2006]

[20] Oh,M. and Ha,S. (1998). *Synthesizable VHDL Code Generation from Data Flow Graph*. **Proceedings of the 5th Asia Pacific Conference on Hardware Description Languages**. Seoul, Korea, July 1998.

[21] Fletcher,J. (2001). *AWT vs Swing* [online]. Borland Developer Network. Available at: http://bdn.borland.com/article/ 0,1410,26970,00.html [Accessed 23 August 2006].

[22] Eckstein,R. and Loy,M. and Wood,D. (1998) .*Java Swing*. Sebastopol: O'Reilly & Associates, USA.

[23] Sun Microsystems, Inc. . *How to use GridBagLayout* [online]. Available at: https://cis.med.ucalgary.ca/http/java.sun.com/docs /books/tutorial/uiswing/layout/gridbag.html [Accessed 27 Oktober 2006].

[24] SHOUFAN,A . *Rekonfigurierbare Prozessoren* [online]. Available at: http://www.vlsi.informatik.tu-darmstadt.de/student_area/rp/rekpro/5_6/uebung /loesungen/1.pdf [Accessed 15 November 2006].

[24] Sun Microsystems, Inc. . *JAVA API.* Available at: http://java.sun.com/j2se/1.3/docs/api [Accessed 20 November 2006].