TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

A converter from IDaSS design files to synthesizable VHDL

Pont, J.F.

Award date:
1995

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

EB 589

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Section of Digital Information Systems

Master's Thesis:

A converter from IDaSS design files to synthesizable VHDL.

J.F. Pont

Coach : Dr. Ir. A.C. Verschueren
Supervisor : Prof. Ir. M.P.J. Stevens
Period : January 1995 - August 1995

The Faculty of Electrical Engineering of Eindhoven University of Technology does not accept any responsibility regarding the contents of Master's Theses.

ABSTRACT

This report describes the work that has been done to implement a converter that converts IDaSS design files to VHDL files. IDaSS is an Interactive Design and Simulation System that makes it possible to test and simulate the designs at register transfer level. To shorten the design time from an IDaSS design to a real chip layout it is necessary to quickly transfer the design files to a language used as the input of a silicon compiler. The implementation of the convereter is started by W.M. Kruytzer but some problems were not solved when he left the TU Eindhoven. These problems mostly are caused by the fact that the silicon compilers available by now do not accept the complete VHDL syntax.

The IDaSS operator blocks were transferred with the use of VHDL procedures and functions, the latter for the complex operators that are not able to be written in a simple assignment. The procedures are used to describe the different IDaSS functions. The functions written, are constructed of VHDL standard statements so they can be used in combination with every silicon compiler wanted.
Control connectors are written in case statements. These control connectors and finite state machines control amongst others the behaviour of the three state output connectors. These three state outputs are written in separate VHDL processes.
These three state outputs can be connected together to one three state bus when the VHDL files are used as the input of most silicon compilers. Compass however, used at the TU Eindhoven, requires that the behavioural files that describe a three state bus, are written differently. Three possible solutions for this problem are presented in this report, but none of them is implemented in the current compiler.
A last improvement to the converter is the addition of a file containing all user defined options. This file is read by the converter in the beginning of the program execution and makes the converter suitable for all wanted silicon compilers. The users interaction is now very much improved, because the complete C-code does not have to be re-compiled every time the user changes an option.

CONTENTS

1 INTRODUCTION 5

2 IDASS 7
- 2.1 IDaSS components 8
  - 2.1.1 Schematic 8
  - 2.1.2 State machine controller 8
  - 2.1.3 Operator 9
  - 2.1.4 Register 10
  - 2.1.5 Buffer 10
  - 2.1.6 Constant generator 10
  - 2.1.7 RAM 10
  - 2.1.8 ROM 11
  - 2.1.9 FIFO 11
  - 2.1.10 LIFO 11
  - 2.1.11 CAM or associative memory 12
- 2.2 IDaSS description files 14
  - 2.2.1 Basic data items 14
  - 2.2.2 Some important switch characters 15
  - 2.2.3 Connectors 15
  - 2.2.4 Buses, Signals and Blocks placed on a schematic 16
  - 2.2.5 Complete file formats 17

3 LEXICAL ANALYSING AND PARSING 18
- 3.1 Global description of LEX 18
- 3.2 Lexical analyzer for IDaSS 19
- 3.3 Global description of YACC 20
- 3.4 Parsing IDaSS des-files 21

4 THE DATA STRUCTURE 23
- 4.1 Lists 23
- 4.2 Some important constructs 23
- 4.3 Expressions 25
- 4.4 Tree of a simple design 26

5 VHDL 28
- 5.1 Entity specifications 28
- 5.2 Structural style architectures 28
- 5.3 Behavioural style architectures 29
- 5.4 Library and USE clauses 30

6 IDASS CONSTRUCTS IN VHDL 31
- 6.1 IDaSS names and VHDL names 31
- 6.2 Registers 31
- 6.3 Operators 33
- 6.4 Finite state machines 37
- 6.5 Buses with multiple drivers 40

7 VHDL CONFIGURATION FILE 44
- 7.1 File format and contents 44

8 CONCLUSIONS 45

LITERATURE 46

APPENDIX 1 IDASSLEX.L 50

APPENDIX 2 TYPES.H 53

APPENDIX 3 STRUCTURAL VHDL OF THE RUNLIGHT 58

APPENDIX 4 RESERVED WORDS 60

APPENDIX 5 VHDL FUNCTIONS FOR COMPLEX IDASS OPERATORS 61

APPENDIX 6 8 BIT MSOMASK 68

APPENDIX 7 FIRST POSSIBILITY FOR A THREE STATE BUS 69

APPENDIX 8 SECOND POSSIBILITY FOR A THREE STATE BUS 70

APPENDIX 9 THIRD POSSIBILITY FOR A THREE STATE BUS 71

APPENDIX 10 SWITCHES OF THE FILE USER_DEF WITH THEIR DEFAULT SETTINGS 72

1 INTRODUCTION

To fully automate the design trajectory from specification to the actual implementation of a design, at the Digital Information Systems Group of the Eindhoven University of Technology a lot of work has been done to create different tools, which all handle a part of this design trajectory. One of these tools is the interactive design program IDaSS (Interactive Design and Simulation System). With this tool it is possible to specify and simulate a certain digital system at register transfer level.

To make the design time shorter from an IDaSS specification to the real chip we would like to translate the design files of IDaSS to files written in a certain language that can be used as the input of a silicon compiler. The target language we use for this purpose is VHDL (Very High Speed Integrated Circuits Hardware Description Language). This language originally was designed to simulate the behaviour of different digital systems, but the language is currently used more and more as the input of a silicon compiler. Because VHDL was not designed for this 'new' application, certain problems are rising. For example, not all the constructs that are allowed in VHDL will be accepted by the silicon compilers or will not generate an IC that works properly.

This report handles the implementation of the converter that translates the IDaSS design files in VHDL. This converter is an extension of the converter that was implemented by W.M. Kruytzer. The existing compiler was not able to translate every operator correctly to VHDL, even as the behaviour of the three state ports present in registers and operators. Also problems with the translation of registers, finite state machines, IDaSS names used for different objects, buses, etc. and the behaviour of three state buses are solved or solutions are presented. The little bugs found in the converter are not handled here. If more information about the previous work is wanted, see the reports of W.M. Kruytzer ([Kru 94] and [Kru 95]). The C-code of the converter contains lots of comment making the code more understandable, so if more information is wanted about the details of the implementation of the converter, check the C-code.

The current converter first reads the design file coming from IDaSS. The converter uses the tools LEX and YACC for this purpose. After reading the design a complete data tree is present in memory containing all the important information to write the IDaSS design to correct synthesizable VHDL. This design tree is passed to different C-functions that do amongst others certain calculations, add information, optimizes the expressions and last but not least writes the structural and behavioural VHDL files. In figure 1 the converter is presented in a simple flow diagram. The structure of this figure can be recognized back when the C-code is read of the file des2vhdl.c. The code in this file calls the different functions and passes the design tree to the next function.

The last chapter of this report describes the implementation of the use of an user definitions file. This file contains all the possible things that can be changed by the user without recompiling the complete C-code and is read every time a translation from IDaSS to VHDL is made.

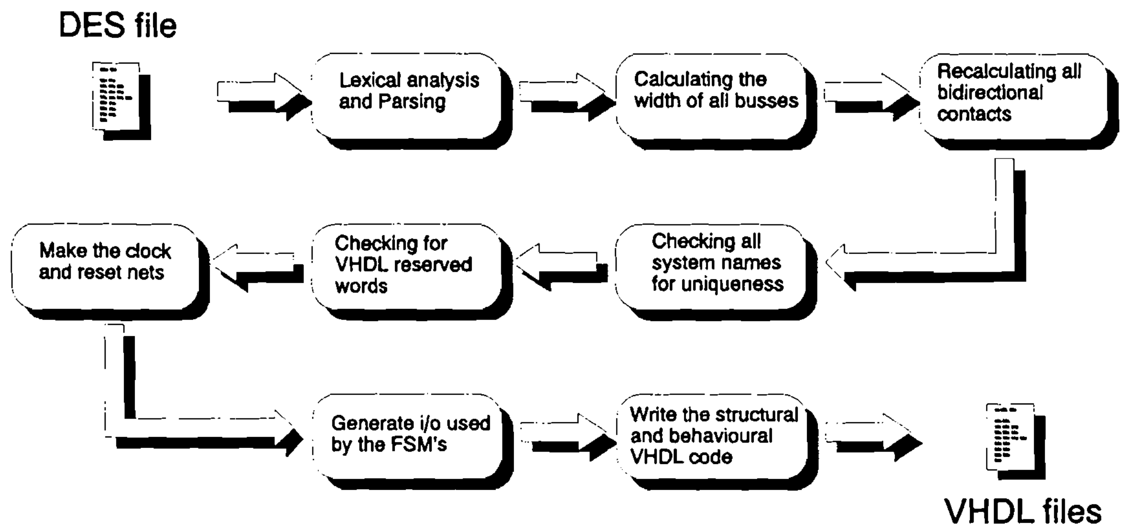


Figure 1 Flow diagram of the DES2VHDL compiler.

Readers interested in the further progresses made, by the Digital Information Systems Group, on this project have to wait for another episode of this marvellous piece of work, that is present in the graduation report of M. Dassen.

2 IDASS

IDaSS for ULSI (Interactive Design and Simulation System for Ultra Large Scale Integration) Is an interactive design and simulation system that is used for digital circuits. The program describes a design as a tree like hierarchy of schematics. These schematics can contain registers, memories, buffers, state machines and so on. The different 'blocks' of a schematic and the schematics themselves can be connected with buses that are displayed as lines on the screen. The behaviour of the designed machine can be simulated one clock step at a time and the contents of the different registers, buses and other elements can be watched with the help of viewers, you are able to connect wherever you like. A screendump of an IDaSS session is given in the next figure.

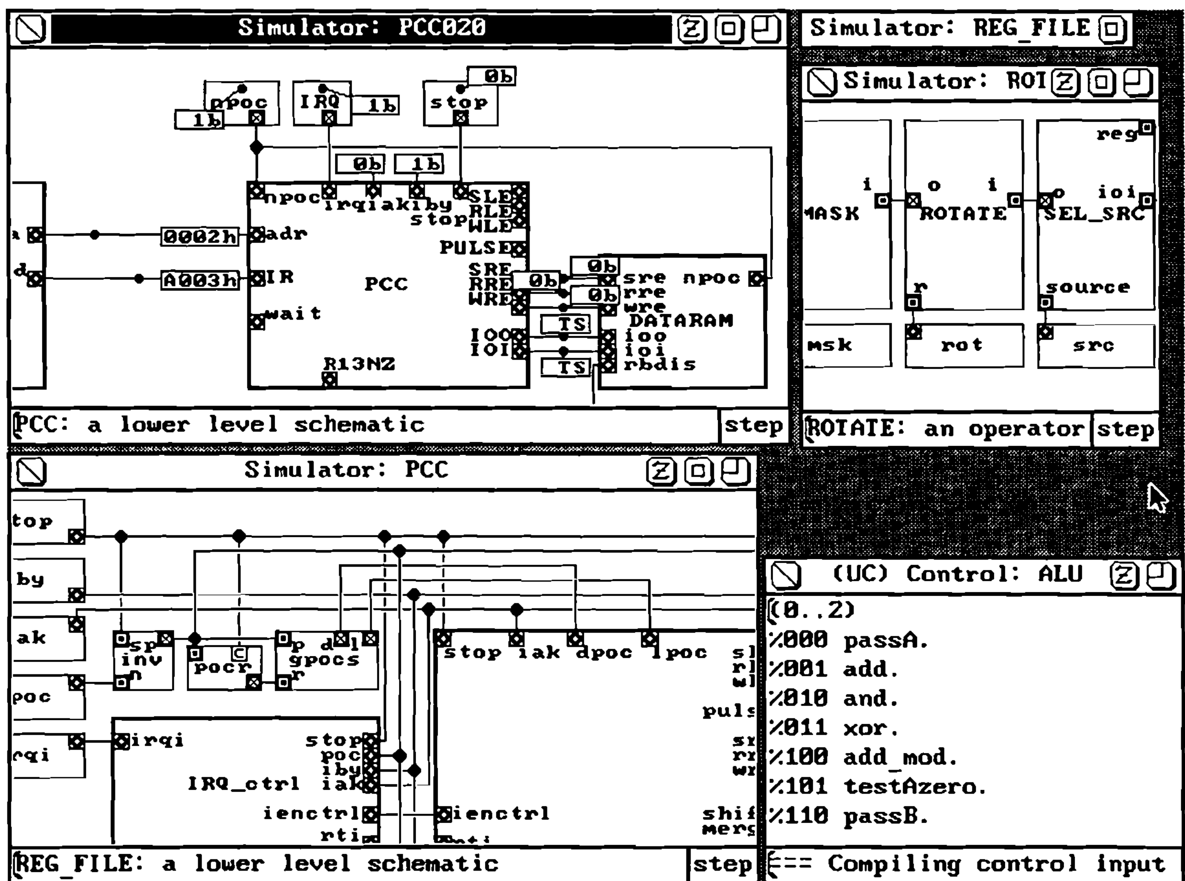


Figure 2 Screen dump of an IDaSS session.

2.1 IDaSS components

The different components that can be used within an IDaSS design are:
- **schematic,**
- **state machine controller,**
- **operator,**
- **register,**
- **buffer,**
- **constant generator,**
- **RAM,**
- **ROM,**
- **FIFO,**
- **LIFO** and
- **CAM** or **associative memory**.

These components will be discussed in this chapter. For a more thorough description see [Ver 90a].

2.1.1 Schematic

Because IDaSS is hierarchical it is possible to place complete schematics inside another schematic. In such a schematic every IDaSS component can be placed, so it is also possible to place another lower level schematic inside a schematic and so on. The data transfer across the boundaries of a schematic is done with bidirectional connectors, which connect the buses outside the schematic and the buses inside the schematic.

2.1.2 State machine controller

The state controllers are described by a number of states with an optional label that can be used for referencing during state transitions. In the description of the states, test points can be included to make conditional program execution possible. These test points (register values, register semaphores and some other values) are clocked directly, and are therefore stable during the evaluation of a state description.

Every state controller can have a stack for the storage of return addresses or states, this is necessary when a subroutine is called. The dept of this stack is user defined and the contents will be adjusted when states are inserted in or removed from the stack.

Its possible to control a block that is in a hierarchically lower schematic, or use the value of such a block as a test point. Controllers are able to communicate with the help of '**signals**'. These are one bit semaphores that can be tested, reset and set by every state controller in the design. Four types of signals are available:
- **Pulsed only**.
  These signals are able to set themselves high the next clock period and then fall back automatically.

- **Level only**.
  These signals can be set high or low and will keep their value indefinitely.
- **Level/Pulse**.
  These signals are a combination of the first two signals. The level mode is considered to be the most important.
- **Pulse/Level**.
  These signals are a combination of the first two also but the difference with the third signal is that the pulse mode is considered to be the most important.

The controllers can be controlled by the use of the '**run/halt**' flip-flop which, when in halt, disables every control output and keeps the state of the controller stable. Controllers can influence the behaviour of each other and themselves by using the commands: **hold, stop, start, goto:** <*label*> and **reset**.
The next figure shows a state machine controller together with one of its state descriptions as it is displayed during an IDaSS session.

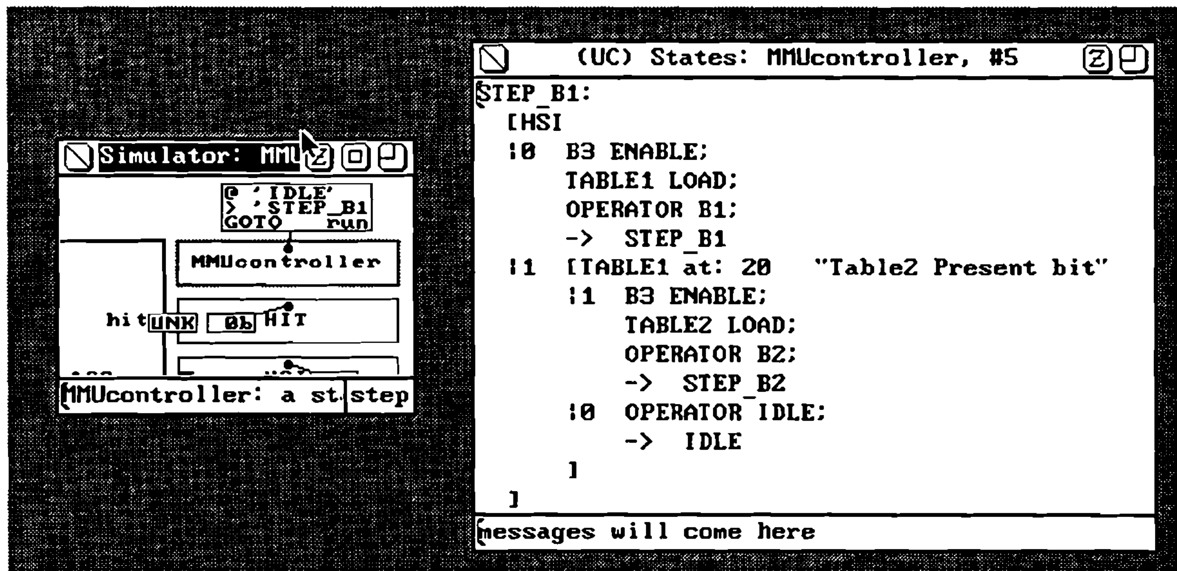


Figure 3 IDaSS screen dump of a state machine controller together with one of its state descriptions.

2.1.3 Operator

Operators can model all the asynchronous elements in a schematic. The operators can have an unlimited amount of in- and outputs and are able to execute one or more user defined functions. These functions are mathematical descriptions and can vary from a simple adder to a complete ALU. The function that has to be executed can be controlled by a state controller or by a control connector. Also a default can be defined. The control connector acts just like a normal input in the way that its contents can be used in the calculations that are executed in the operator. The control connector will select a function asynchronously.

2.1.4 Register

The register models a master-slave register with a maximum of 64 bits. Simple operations the register can execute are:

- **hold,**
- **load,**
- **inc,**
- **dec,**
- **loadinc,**
- **loaddec,**
- **setto:** <*value*>,
- **ressem** and
- **reset**.

The **reset** command overrules every action. The contents of the register after a reset command are defined by the '**synchronous reset contents**'. Every register carries a semaphore bit that is set by a load command (also **loaddec** and **loadinc**). This semaphore can be tested and reset by the controller(s). The **ressem** command resets the semaphore but has lower priority than a set caused by a load action.
The register can simulate a battery backed register (that keeps the same contents during a system reset) by typing **hold** for the '**asynchronous reset contents**' instead of a value or '**UNK**' for an unknown value.

2.1.5 Buffer

A buffer models a simple three-state buffer that is unidirectional and asynchronous. Whether the three-state output is enabled or not is controlled by a controller. The in- and output must have the same width.

2.1.6 Constant generator

A constant value that is generated by a controller can be injected into the schematic by using a constant generator. This block has one optional three-state output and one command (**setto:** <*value*>). This command generates the wanted value at the output for the current clock period asynchronously.

2.1.7 RAM

The RAM is a model of a random access read/write memory with the possibility to have more than one read or write port. Writing is done synchronous with the clock and the output of a read port follows the address input directly (is asynchronous). A read port can be used to read a fixed memory location. In this case an address input is not necessary. The commands a RAM can be given are:

- **write**.
  Writing at all the write ports is enabled. This command can only be given if at least one port has a default function **nowrite**.

- **write: <*DataInputName*>.**
  This command enables writing at the specified port (only possible if the default function is **nowrite**)
- **nowrite.**
  Writing at all the write ports is disabled. This command can only be given if at least one port has a default function **write**.
- **nowrite: <*DataInputName*>.**
  This command disables writing at the specified port (only possible if the default function is **write**).

2.1.8 ROM

A ROM actually is a RAM without the possibility of attaching write ports. The only possible command a ROM can be given is to enable or disable its outputs (if these are three-state outputs of course).

2.1.9 FIFO

The FIFO models a First In First Out buffer which may have a single write port to add data words at the tail of the list and a read port that reads the head of the list. Extra asynchronous read ports can be added to the FIFO, addressing the memory relative to the head (address = 0) of the list. These ports could have an address input or they could be attached to a fixed address.
A controller is able to check the contents of the head of the list as well as the length of the queue (the number of words in the memory).
The commands a FIFO can be given are:

- **write.**
  At the next clock the value present at the write port will be written in the memory. It is allowed to give the command **read** at the same time.
- **write: <*value*>.**
  The given value is stored in the memory the next clock. A **read** at the same time is allowed.
- **read.**
  The value at the head of the list is removed the next clock.
- **reset.**
  All the contents of the FIFO will be lost at the next clock. **Reset** overrules the other commands.

2.1.10 LIFO

This block gives a model of a Last In First Out memory and can be seen as a stack. The main read port is connected to the head of the stack as well as the single write port that adds the new words to the top of the stack. Extra (asynchronous) read ports can be connected to the memory with relative addresses (fixed or with an address input).
The LIFO is controlled by making use of the next commands.

- **reset.**
  This command overrules all the commands and will reset the LIFO at the next clock.

- **push**.
  The data that is available at the write port is placed at the top of the stack at the next clock (this is the memory location before the current head).
- **push: <*value*>**.
  The given value is placed at the top of the stack at the next clock.
- **pop**.
  The value at the top of the stack will be removed the next clock.
- **replace**.
  This function acts the same way as a **pop** followed by a **push** in just one clock cycle.
- **replace: <*value*>**.
  This function acts the same way as a **pop** followed by a **push: <*value*>** in just one clock cycle.
- **poprepl**.
  This function is a combination of the three functions **pop**, **pop** and **push** in just one clock cycle.
- **poprepl: <*value*>**.
  This function is a combination of the three functions **pop**, **pop** and **push: <*value*>** in just one clock cycle.
- **swap**.
  This function switches the contents of the head of the stack with the contents of the memory location just below the head.
- **pop2**.
  This function is a combination of two **pop**'s in just one clock cycle.
- **pushcopy**.
  This function makes a copy of the head of the stack and places this copy at the top of the stack at the next clock.

2.1.11 CAM or associative memory

The Contents Addressable Memory models a type of memory in which data words can be addressed by comparing bits in these words with a given reference word. The CAM is synchronous except for the optional extra read ports.
A memory word is matching when the memory word matches the match data in the bits that are '1' in the match mask word. The match data and the match mask can be set by the controllers or by the use of a control connector or can come from an input port. The number of matching memory words even as the address of the first matching word is returned and can be tested by a controller. The first matching address and the contents of that memory address can also be the result of an output.
The commands a CAM recognizes are:

- **reset**.
  Reset overrules all other commands and replaces the contents of the CAM with the contents as they are defined by the user (synchronous reset values). **Reset** also executes the default function.
- **match**.
  This function checks the contents of the CAM by using the match mask and the match data. The next clock step the number of matches is counted, the data output (if present) outputs

the contents of the first matching memory location and the address output will give the address of the first matching memory location[1].

- **wrfirst**.
  According to which port is available or which command has been given, **wrfirst** adds the following to the **match** function. If a match write data port is available or a **mdata: <*value*>** command is given, the contents of the first matching cell are replaced by the given value at the next clock.
  If match write set/reset ports are available or **mset: <*value*>** and/or **mres: <*value*>** commands are given, the contents of the first matching cell are replaced by the given value at the next clock using the following formula:

  $$contents := (contents \wedge (\neg resetmask)) \vee setmask \qquad \textbf{(2.1)}$$

  So setting bits has a higher priority than resetting.
- **wrall**.
  This function behaves like the function **match** and changes all the matched cells like **wrfirst**. The cells that do not match are changed also like **wrfirst** but now using the NOmatch data or the set/reset specifications.
- **rdaddr**.
  With this function a normal read can be established. If the address output works in the 'first match' mode, it will output the value of the address input. If it works in the 'match mask' mode, a bitmask containing a 'one' corresponding to the addressed location, will be carried.
- **wraddr**.
  This function works like **rdaddr** and writes at the addressed location, using the algorithms of **wrfirst**.

The following commands are not overruled by a reset and can be used to set match and write values which are not provided by the input ports.

- **mask: <*value*>**.
  This command sets the bitmask that is used to select the bits that have to be matched against the match data.
- **data: <*value*>**.
  This command sets the data bits used for matching.
- **mset: <*value*>**.
  This command sets a mask for bits that will be set during all write actions.
- **mres: <*value*>**.
  This command sets a mask for bits that will be reset during all write actions.
- **mdata: <*value*>**.
  This command sets the data to write during all write actions.
- **nmset: <*value*>**.
  This command sets the mask for bits that will be set during the **wrall** action.
- **nmres: <*value*>**.
  This command sets the mask for bits that will be reset during the **wrall** action.
- **nmdata: <*value*>**.
  This command sets the data to write during the **wrall** action.

[1] This is only the case if the address output is in firts match mode. Is the port in match mask mode then it will carry a bitmask containing '1' bits for each matched address. This mode is only allowed when the CAM has not more than 64 memory locations.

2.2 IDaSS description files

IDaSS makes use of two file formats. The first and simplest one is a format to store the contents of a memory that is present in a design. For these files the standard Intel HEX format is used. These files are currently not used in the conversion of IDaSS designs and therefore not handled here.
The other file format is a textual description of the design. In this file all the design and simulation state information is stored.
Each line in the design file (des-file) starts with a kind of switch. Following this switch, a second and even a third switch may be present, depending on the meaning of the line. Data items may be placed after these switch characters.
The following paragraphs describe the basic IDaSS file format elements and handles the format of an example of an IDaSS block (the operator). The following description of the des-files makes use of the EBNF[2] notation that is commonly used to specify formal grammars or programming languages and describes version 0.07 and 0.08 of the IDaSS file format, which the des2vhdl compiler currently supports.
For a more thorough description see [Ver 90b].

2.2.1 Basic data items

IDaSS makes use of five basic data items:

- **Integer**
- **String**
- **Point**
- **Integer with a given number of bits**
- **Nil**

The integer and string are standard and points are used for graphics information and therefor not important for the des2vhdl compiler.
The fourth item is used for the internal calculations within IDaSS. The values used are called "BoundedInteger" because they are bounded by the number of bits present. Because these BoundedIntegers are used very often, a special data item is introduced.

```
<boundedInt> ::= <value>'W'<width>.
<value> ::= <Integer>.
<width> ::= <Integer>.
```

If the value is unknown then -3 denoting 'UNK' for the value is used.

Nil is used to flag empty objects or to indicate that something is not used.

```
<nil> ::= '?'.
```

2 Extended Backus-Naur Form

2.2.2 Some important switch characters

The use of switch characters is an important construct to correctly parse an IDaSS des-file. The following characters are important switch characters.

'#' Starts the description of any major object in the system. This switch is followed by two names. The first is the type of object and the second is the name of the object.
The following objects are used within IDaSS:

'SuperBlock'	describing a schematic.
'SuperConnector'	describing an object placed in a schematic to provide a logical connection to one of the connectors on the schematic symbol.
'Bus'	describing a bus.
'StateControl'	describing a state machine controller object.
'Signal'	describing a system wide signal for communication between controllers.
'Buffer'	describing a three-state buffer object.
'Constant'	describing a constant generator object.
'Register'	describing a register.
'Operator'	describing an operator.
'RAM'	describing a random access read/write memory.
'ROM'	describing a read only memory.
'FIFO'	describing a first in first out memory.
'LIFO'	describing a last in first out memory.
'CAM'	describing a contents addressable memory.

'.' This character concludes the description of any major object in the system. The switch is followed by the same two names as the corresponding '#'. Together with the '#' line, this line forms a pair of braces.

'/' This switch starts a line containing graphics information. This information can be skipped by the des2vhdl compiler.

''' This single quote starts a line containing textual information which is compiled by IDaSS and describes the behaviour of an object.

'"' This double quote starts a line containing comments. The comment lines can be skipped by the compiler (although it could be of use to copy this text to the generated VHDL output).

2.2.3 Connectors

Five types of connectors are used in IDaSS (Input, Output, Three-state, Bidirectional and Control Input). The description of a connector always follows one or more switch characters which are interpreted by the block when loading from the des-file. The connector description starts with another switch that describes the type of the connector.
The following rules describe an Input connector. The other types of connectors are almost the same as this Input connector and therefore not handled here in particular.

```
<inputConnector>  ::= 'I'<connName> <nameNeeded> <connValue> <LF>[3]
                      {'"'<comment><LF>}
                      '/P'<connPos> <namePos> <LF>.

<connName>        ::= <name> |
                      '*i'.
<nameNeeded>      ::= <boolean[4]>.
<connValue>       ::= <boundedInt>.
```

Whether a name is mandatory is indicated by <nameNeeded>. If the designer did not give a name to this connector the name is '*i'. The lines with the comments and the graphics information are not used by the compiler so they are not explained any further.

2.2.4 Buses, Signals and Blocks placed on a schematic

Buses, signals and the blocks that can be placed on a schematic all have their own set of description rules. Also a schematic itself which may contain any of the other blocks or even a schematic again, has a set of rules. As an example the rules of an operator are shown here. An operator is used to describe combinatorial logic (see 2.1.3).

```
<operator> ::=      '#Operator' <blockname> <LF>
                    '/P'<blockPos> <blockSize> <LF>
                    {'"'<comment><LF>}
                    ['Z'<operatorControl>]
                    {'I'<operatorInput>}
                    {'O'<operatorOutput>}
                    {'F'<functionName> <compiledFlag> <LF>
                     {'''<functionText><LF>}}
                    'S'<function> <defFunc> <LF>
                    '.Operator' <blockname> <LF>.

<blockName>         ::= <name>.
<operatorControl>   ::= <controlConnector>.
<operatorInput>     ::= <inputConnector>.
<operatorOutput>    ::= <outputConnector>.
<functionName>      ::= <name>.
<compiledFlag>      ::= <boolean>.
<functionText>      ::= {<anyPrintableCharacter>}.
<function>          ::= <integer> | <name>.
<defFunc>           ::= <integer> | <name>.
```

An operator starts with a line beginning with '**#Operator**' and ends with '**.Operator**' as was said earlier. The optional control input and the lines starting with '**F**', ''' and '**S**' contain the

---

[3] A <LF> describes the end of a line, as well in DOS format as in UNIX format.

[4] A boolean is an integer with value 0 for 'false' and 1 for 'true'.

behavioural information of this operator, the other lines and again the line describing the control connector are used for the structural description (a control input can now be seen as a normal input).

2.2.5 Complete file formats

Three different combinations of the used objects can be present in a des-file. These depend upon the way the file was created.

- At first it is possible to detect a file only containing signals. These files can be made from each signal editor or from an entry in the '*signals...*' menu on the top level simulator window. It is also possible to file out a single signal from a signal editor. A signal file has to follow the rule:

```
<signalFile> ::= <signal> {<signal>}.
```

- The second possibility is a file containing data for a single block. This kind of file can be created from each of the block menu's (except for the super connector) and has to follow the rules:

```
<singleBlockFile>   ::=  <stateMachineController> |
                         <TSbuffer> |
                         <constantGenerator> |
                         <register> |
                         <operator> |
                         <RAM> |
                         <ROM> |
                         <FIFO> |
                         <LIFO> |
                         <CAM> |
                         'K'<clockCount> <LF>
                         <schematic>.

<clockCount>        ::=  <integer>.
```

- The last possibility is a complete system file. These files are saved while making use of the '*save system*' entry in the top level simulator window menu. These files must follow the next two rules:

```
<completeSystemFile>  ::=  'K'<clockCount> <LF>
                           {<signal>}
                           <schematic>.

<clockCount>          ::=  <integer>.
```

3 LEXICAL ANALYSING AND PARSING

This chapter describes the lexical analyzer and the parser used to construct the des2vhdl compiler. The tools used are Lex and Yacc, because these tools are available on most UNIX work stations and versions for most other operating systems are widely spread. At first the tool Lex, used for lexical analysis (lexing for short) and the design of the IDaSS tokenizer is described. The second part of this chapter describes the tool Yacc that is used to parse the des-files. The last section of this chapter handles the problems that arise while parsing a des-file. For more information about Lex and Yacc see [LMB 92]. This book was used during the design of the lexical analyzer and the parser used for the IDaSS des-files.

3.1 Global description of LEX

Lex is a tool for building Lexical analyzers. A lexer takes an arbitrary input stream and divides this stream into lexical tokens. To create a set of patterns which Lex matches against the input stream, a Lex specification has to be written. The provided C-code, which does something with the matched text, is invoked every time a pattern matches. Lex does not produce an executable but it translates the Lex specification into a file containing a C-routine called yylex(). To run the lexer the program has to call yylex().

A Lex program consists of three parts: the definition section, the rules section and the section with the user defined subroutines.

```
... definition section ...
%%
... rules section ...
%%
... user subroutines ...
```

Only the rules section is mandatory the other two sections may be empty.

The definition section can include a literal block, definitions, internal table declarations, start conditions and translations. The user subroutines section is copied verbatim to the C-file and typically includes routines that are called from the rules. The rules section is the most important one. It contains pattern lines and C-code. When a Lex scanner runs, it matches the input against the patterns in the rules section and when a match is found the corresponding C-code is executed.

The rules are created while making use of regular expressions. The characters that form these regular expressions are (only the characters used in the lexer for the des-files are handled):

- . Matches any single character except the newline character ("\n").
- * Matches zero or more occurrence of the preceding expression.
- [] A character class which matches any character within the brackets. If the first character within the brackets is a "^" the meaning is changed to match any character except the ones within the brackets. A dash indicates a range of characters, e.g., "[0-9]" has the same meaning as "[0123456789]".

^ Matches the beginning of a line.
\ Used to escape characters like . or * that form the regular expressions and as part of the usual C-escape sequences.
+ Matches one or more occurrence of the preceding regular expression.
? Matches zero or one occurrence of the preceding regular expression.
| Matches either the preceding regular expression or the following regular expression.
"..." Matches everything within the quotation marks literally.
/ Matches the preceding regular expression but only if followed by the following regular expression. The material matched by the pattern following the slash is not consumed and remains to be turned into subsequent tokens. Only one slash is permitted per pattern.
() Groups a series of regular expressions together into a new regular expression.

3.2 Lexical analyzer for IDaSS

The file named idasslex.l is the lexical analyzer that recognizes all basic data items and switch characters used in an IDaSS design file.
The comment lines, the lines containing graphics information and the white spaces in a des-file can be skipped. The definitions that are in the definition section for this purpose are:

```
graphics      \/.*
comment       \".*
ws            [\t \r]+
```

The first two definitions start with a \ because the character that has to be matched is a special Lex character. Every empty string, character or series of characters that follow this first character is matched by .* until the end of a line ("\n"). The corresponding rules are:

```
{ws}          ;
^{graphics}   ;
^{comment}    ;
```

No actions are taken when one of these rules is matched and that is exactly what was intended. The other used definitions that are placed in the definition section are:

```
name          [A-Za-z][A-Za-z0-9_]*
no_iname      \*"i"
no_oname      \*"o"
no_cname      \*"c"
val_start     [%&$]
decimal       [0-9]+
value         [0-9a-fA-F]*
val_end       [bBdDhHoOqQ]
xvalue        [0-9a-fA-FxX]*
binaryop      [+\-*/\\><=~]+
integer       -?[0-9]+
boundedint    {integer}"W"{integer}
nil           "?"
```

The rules are much easier to understand while making use of these definitions.
Another feature that is used while making idasslex.l is the use of start states. These states limit the scope of certain rules. Two kinds of start states exist, regular and exclusive start states. The difference is that a rule without a start state is not matched when an exclusive state is active and it is matched when a regular start state is active. The start states used in the des2vhdl compiler are defined in the definition section by the line:

```
%x connector filespec comptext comment
```

The x means that the specified states are exclusive start states (%s would have meant regular states). The states can be entered by the command:

```
BEGIN(state_name);
```

The following set of rules recognizes and handles a line in an IDaSS des-file that defines a file specification.

```
^"L"/.*              { BEGIN(filespec);     /* other C-code */ }
<filespec>[^\n]*     { BEGIN(INITIAL);      /* other C-code */ }
```

To return to the normal situation without the limitation of any start states the statement BEGIN(INITIAL); is used. Every character is matched against this second rule only in the case that the first character of the line was an L, because that is the only way to come into state filespec. So context information can be saved while making use of start states.
Because we want to tokenize the complete IDaSS des-file we have to return a special token to the calling procedure or function, every time Lex recognizes an IDaSS switch character or a basic data item. This is simply done by the C-command:

```
return(token_name);
```

The last part of idasslex.l is the part with the user subroutines. This section only contains a procedure that will properly handle the errors that are able to occur (invalid characters in the des-file). The total file idasslex.l is shown in Appendix 1.

3.3 Global description of YACC

Yacc takes a grammar that you specify and writes a parser that recognizes valid sentences in that grammar. A grammar is a series of rules that the parser uses to recognize syntactically valid input.

Similar to a Lex grammar a Yacc grammar consists of three sections: the definition section, the rules section and the user subroutines section.

```
... definition section ...
%%
... rules section ...
%%
... user subroutines ...
```

The first two sections are required, although a section may be empty. The third section and the preceding "%%" may be omitted.
The definition section can include a literal block (this C-code is copied verbatim to the beginning of the generated C-file), usually containing declarations and #include lines. Also all kinds of declarations may be present using the Yacc terms:
%union, %start, %token, %type, %left, %right and %nonassoc.
The user subroutines typically includes routines called from the actions and is copied verbatim to the generated C-file. In the parser for IDaSS des-files this third section is empty and all the called routines are in separate files to keep the complete written program well organized.
The rules section contains the rules and the actions to be taken when a rule is matched. Each rule starts with a non-terminal symbol, a colon and a possible empty list of symbols, literal tokens, and actions. Every rule ends with a semicolon. A rule that defines a date could be:

```
date : month '-' day '-' year ;
```

(The month, day and year symbols must be defined elsewhere in the grammar.) The left-hand side of this rule is date : the rest is the right-hand side of this rule and therefore may be empty. When this rule is matched an action can be executed like:

```
date : month '-' day '-' year { printf("Date recognized.\n"); } ;
```

The C-code in these actions may have some special constructs starting with a "$". The name "$$" would refer to the left-hand side of the rule and "$*n*" to the *n*-th component of the right-hand side of the rule.
Some other special characters used in Yacc are:

% All of the declarations in the definition section start with a %.
'...' Literal tokens are enclosed in single quotes.
< > In a value reference in an action, you can override the value's default type by enclosing the type name in angle brackets.
| When two consecutive rules have the same left-hand side, the second rule may replace the left-hand side by a vertical bar. The next rule therefore gives two possible ways to write a date:

```
date :    month '-' day '-' year
     |    month '/' day '/' year
     ;
```

3.4 Parsing IDaSS des-files

Having a lexer that returns tokens it is not very complicated to construct the rules belonging to the IDaSS des-files. The start state is defined in the definition section by the line:

```
%start r_CompleteSystem
```

This is the first rule the parser uses. As an example of a rule used in the parser, the rule used to parse an operator is explained (see section 2.2.4 for the way the operator is written in a des-file).

```
r_Operator:    T_Operator_begin T_Name
               o_GenericControl
               l_OperatorInput
               l_OperatorOutput
               l_FunctionName
               T_Sswitch r_Operfunction r_Operfunction
               T_Operator_end T_Name
```

Most of the parts of the operator rule are defined themselves in other rules. Only the four elements starting with a T are tokens returned by the lexer. These tokens therefore are defined in the definition section of the file idass.y in the lines:

```
%token <string> T_Name ...
%token ... T_Operator_begin T_Operator_end ...
```

4 THE DATA STRUCTURE

The previous chapter described in what way the des-files are parsed. The recognized data items and IDaSS elements are to be collected in a data structure. This data structure then can be passed to the next parts of the program that do certain calculations and transformations on it (see the figure in the introduction). This chapter will describe the way the different lists, used in the data structure, are built. Some important structures used are discussed and an example of an expression tree and of design tree of a little system is given.

4.1 Lists

Instead of making use of a simple linked list with in each element one field that points to the next element, the compiler makes use of a somewhat more complex kind of list (figure 4). The advantage is that the length of the list is easy to find without stepping through the complete list. To find the end of the list its not necessary to visit every element too. Another advantage of this kind of list is that every wanted structure can be chosen as the elements of the list. A disadvantage is that this implementation uses more memory than the simple kind of list.
The pointers in the dptr fields point to the listed data and the first field points to the next element in the list.

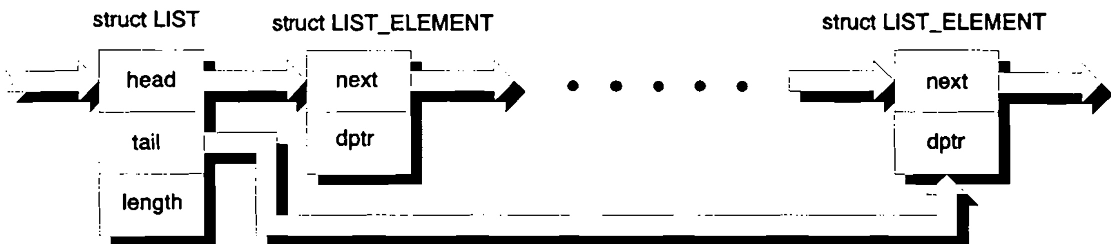


Figure 4 The used structure for all lists.

To create such a list, list_init() has to be called. This function returns a pointer to a struct LIST. The structures of type LIST_ELEMENT are created when an element is added to the list. To add, insert or delete elements and to obtain pointers to elements with for instance a certain value or name, different functions are available.

4.2 Some important constructs

Because IDaSS designs are hierarchically built the data structure of a design in the compiler is hierarchically too. The data structure makes use of the linked list of the previous section and saves all the inputs, outputs, buses, contacts, systems (IDaSS buffers, registers, operators, etc.) and lower level schematics and everything else that is important to create correct VHDL files.

The complete design is a linked list of structure LEVEL (see the next figure).

lower	struct LIST *
name	char *
busses	struct LIST *
systems	struct LIST *
contacts	struct LIST *
postfix	char
prefix	char *

Figure 5 struct LEVEL.

The fields postfix and prefix are not added to field name because the original name is used in all kinds of references (like in bus connections) and therefore has to be saved. The other fields in this structure all contain a pointer to the beginning of a list. These lists contain lower level schematics, the buses on the current level, the IDaSS objects (no schematics) on the current level and the contacts of this schematic. These contacts are always bidirectional initially. The width and the exact direction of these connectors are determined in one of the next steps in the program.
An IDaSS object is saved in a structure SYSTEM (see the next figure).

name	char *
inputs	struct LIST *
outputs	struct LIST *
type	enum SYSTEMTYPE
ctrlcon	struct CTRLCON *
commands	struct LIST *
sys	union { struct REGISTER *reg; struct OP *op; struct FSM *fsm; }
postfix	char
prefix	char *

Figure 6 struct SYSTEM.

This structure also contains a separate field to save the pre- and postfix. When the IDaSS object that is collected in this structure is a register, an operator or a finite state machine then the

field sys contains a pointer to another structure containing specific information of these objects (states, functions, default functions, etc.).
The file types.h (Appendix 2) shows all the other structures and types used by the compiler.

4.3 Expressions

The expressions used in an operator are saved in a structure ASSIGNMENT with two fields. The first saves the name of the variable and the second the expression belonging to this assignment. In a state machine expressions are only used in condition blocks. The structure EXPR contains the type of the used operator and a left and a right argument (and some other fields to make writing VHDL files somewhat easier). The expression tree ends with constants or operands at the leaves. As an example the tree of the next expression is drawn in figure 7.

```
(bit)
if0:  (in1 /\ in2)
if1:  (in1 \/ in2).
```

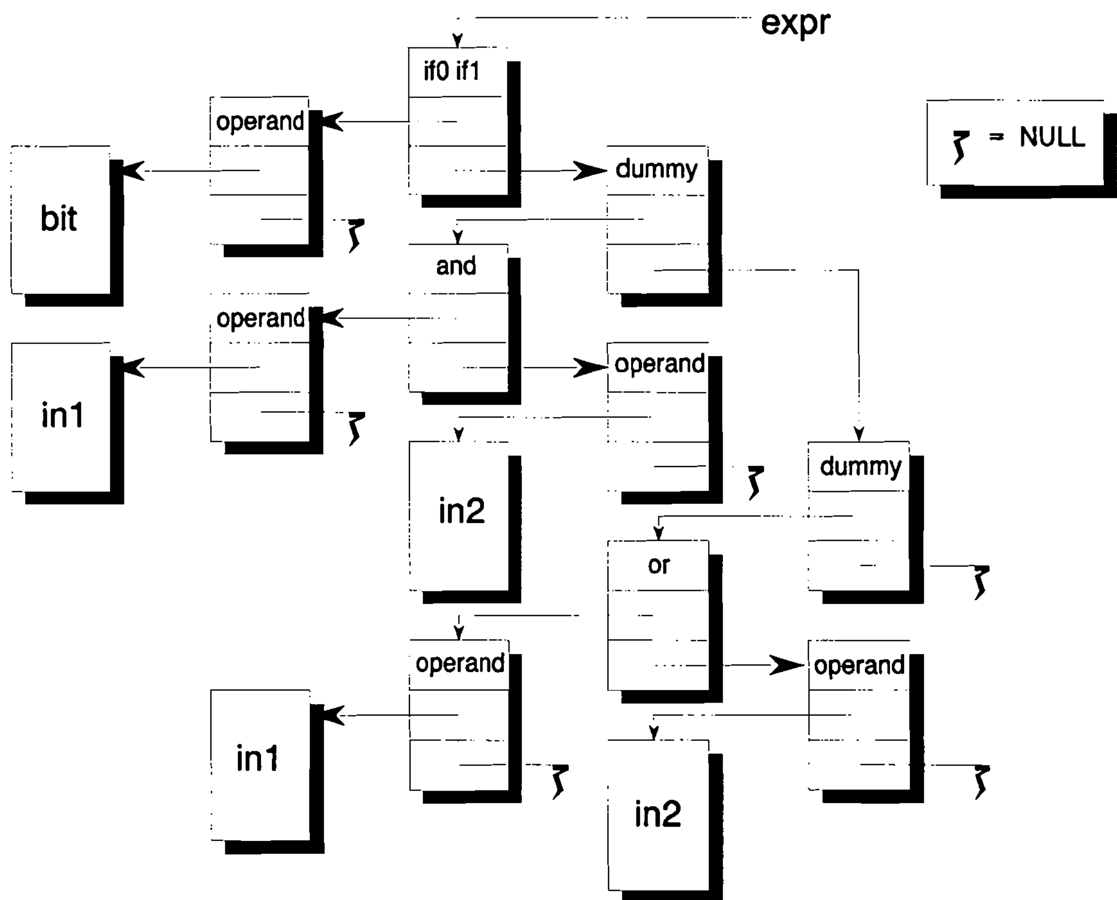


Figure 7 Simplified expression tree.

The structures with the type dummy are also used to make a list of all the if0if1 and if1if0 operators. This list makes use of the fact that the right argument pointer in the second dummy points to NULL. This pointer is now used to point to the next if0if1 or if1if0. In this way its possible to first write all the temporary VHDL variables used while writing these operators in the VHDL files without searching the complete expression trees for these operators.

4.4 Tree of a simple design

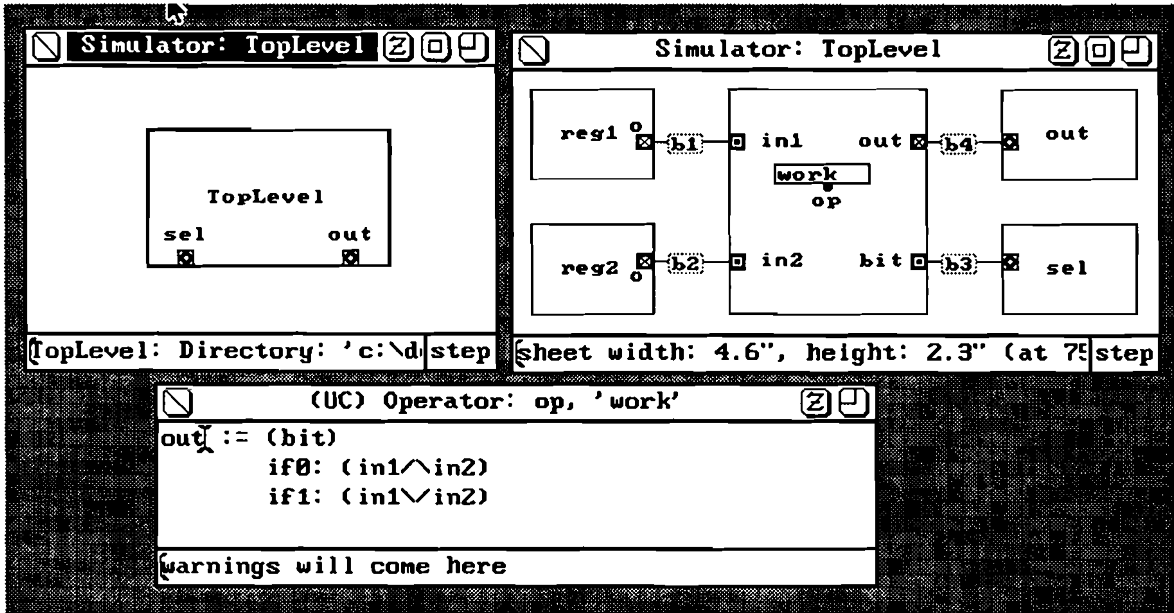


Figure 8 Screendump of a simple design.

The design displayed in this figure, puts an **OR** or an **AND** function of the internal register contents to its output, depending on the value present at the select input. The registers have the functions **inc** and **dec**. The operator makes use of the function that is described in the previous section so the expression tree belonging to this expression is not depicted in the next figure but a reference (exprtree) is displayed at the place this expression tree has to be included in the real tree. Also fields in a structure like prefix or postfix are neglected. Every part of a linked list is displayed in grey.

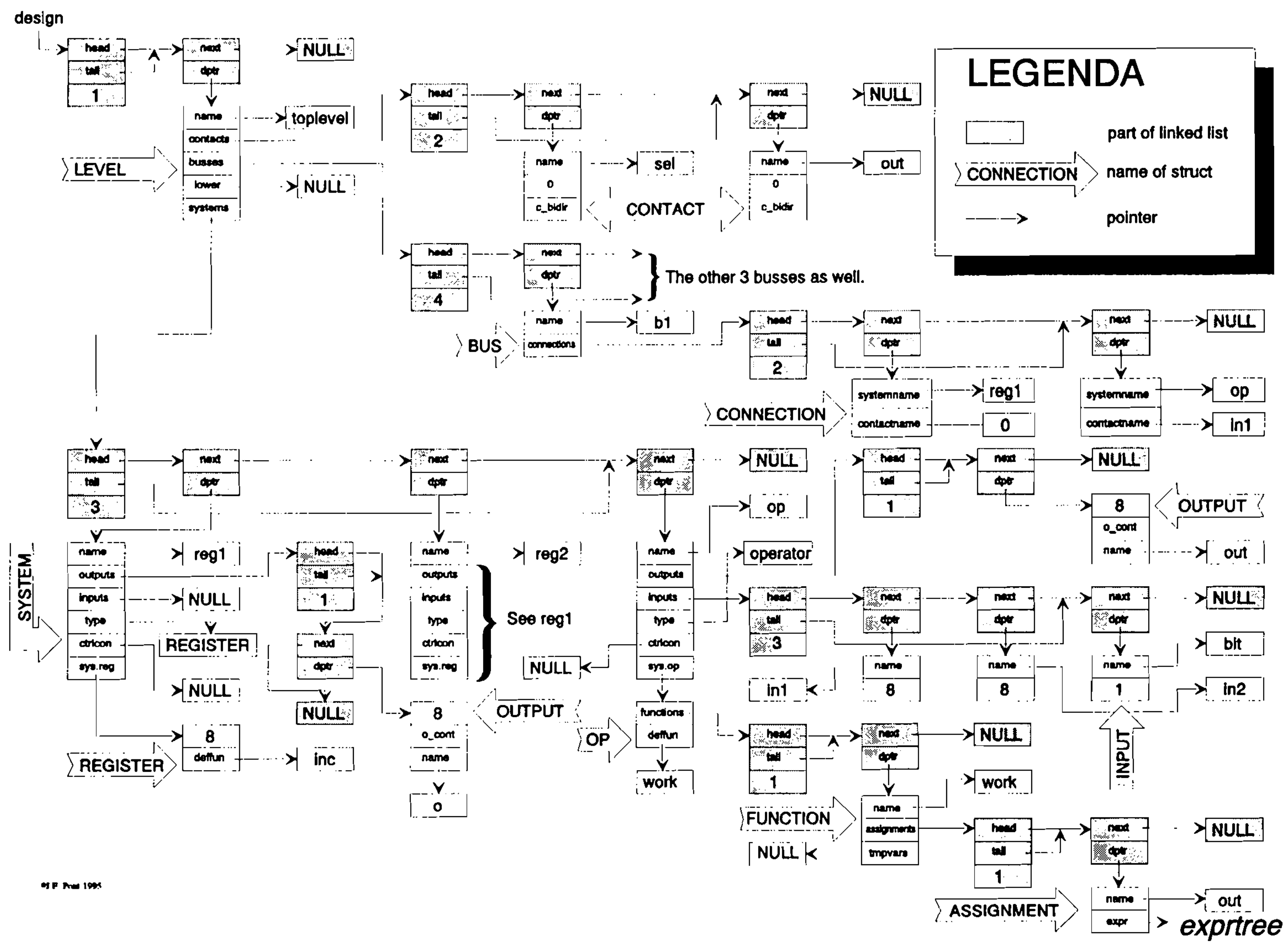


Figure 9 Simplified data tree of a simple design.

5 VHDL

The Very High Speed Integrated Circuit Hardware Description Language is a language that was designed to document the interconnection of components and the behaviour of an digital electronic circuit. The VHDL design description can be the input of a simulator or a logic synthesis tool to produce a physical design. IEEE standardized the language in December 1987 [STD 87]. If a more complete description of the language is required than given in this chapter see for instance [LSU 89] or [ML 92].

5.1 Entity specifications

The design entity defines a new component name, its input and output connections and the related declarations. The entity is the I/O interface to a component design. VHDL separates the external interface to a design from the details of architectural implementation. The entity describes the direction and type of signal. An example of an entity of an adder of two 8 bit inputs is displayed here.

```
ENTITY add IS
   PORT(
      in1,in2:     IN   bit_vector(7 DOWNTO 0);
      out:         OUT  bit_vector(7 DOWNTO 0)
   );
END add;
```

No behavioural information is described just the interface description is given. The actual adder is seen as a black box from this point of view. The behaviour of the different entities is described in the architecture bodies.

5.2 Structural style architectures

The VHDL structural style describes the interconnection of components within an architecture. In a structural architecture, the components to be used are declared and instances of these components are created. Also the mapping of the signal wires to the various pins of the components are created. Component instantiation statements identify the wired connections.
An architecture description has the next general form:

```
ARCHITECTURE name OF entity_name IS
     declarations
BEGIN
     statements of the architecture
END name;
```

The architecture name defines the unique name of this architecture for the entity it refers to. The architecture declarations are items used only in this architecture such as types, subprograms, constants. The actual design description is given by the statements.

An example of a declaration of a component adder is given here:

```
COMPONENT adder
  PORT (
    in1,in2:    IN   bit_vector(7 DOWNTO 0);
    out:        OUT  bit_vector(7 DOWNTO 0)
  )
END COMPONENT;
```

Now this adder is declared it can be used after instantiation. Two kinds of instantiation can be used, one with a positional association

```
instance1: adder PORTMAP (a,b,c);
```

and one with a named association

```
instance1: adder PORTMAP (in1 => a, in2 => b, out => c);
```

In this example the two instance declarations mean the same but only in the second example the possibility exists to change the order of the IN- and OUTputs.
A structural description of the famous runlight example is given in Appendix 3.

5.3 Behavioural style architectures

Behavioural style architectures make use of the same form as the structural style only the statements are different. The statements describe in a program-like or algorithmic manner the behaviour of the design. The statements are sequential but when concurrent statements are necessary, VHDL processes are used containing sequential statements that are executed in the same time as the sequential statements in other processes. A process must be of the form:

```
PROCESS (A,B,C)
BEGIN
  process_statements
END PROCESS;
```

The signals A, B and C are in the *sensitivity list* of this process, meaning that the process is waiting for a change in these signals, before it starts with the execution of the statements.

5.4 Library and USE clauses

Clauses in VHDL select and define declarations. A LIBRARY clause defines logical names for design libraries in the host environment. The USE clause selects declarations made visible by the selection.
The IEEE standard 1164 defines nine "logical" values within a VHDL package. These nine values are more useful for simulation and synthesis than type bit. The nine values are:

U	uninitialized
X	forcing an unknown
0	forcing 0
1	forcing 1
Z	high impedance (three state)
W	weak unknown
L	weak 0
H	weak 1
-	don't care

The available types in this library are:

```
std_logic
std_logic_vector
std_ulogic
std_ulogic_vector
```

The first two types are resolved meaning that it is possible to have multiple drivers for signals of this type. The last two types are unresolved and you cannot have more than one driver.
To use this package and these types the next lines must be present in the VHDL files above every ENTITY that makes use of these types.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all
```

In the structural VHDL file in the Appendix also a library compass_lib is declared to include compass specific things, but this has to become a user defined library.

6 IDASS CONSTRUCTS IN VHDL

The different IDaSS objects and constructs have to be translated into correct, synthesizable VHDL. For the problems W.M. Kruytzer solved see his reports [Kru 94] and [Kru 95].
This chapter discusses the problems that exist with the (re)naming of the IDaSS objects. Also the VHDL equivalents of the registers are given and the different IDaSS operator commands are discussed as well as the translation of the finite state machines. At last possible solutions are discussed for the problems with the three state buses. These buses are still not implemented correctly in the compiler.

6.1 IDaSS names and VHDL names

Some differences between IDaSS and VHDL exist regarding the names of objects, buses etc. In IDaSS nearly everything is allowed. In VHDL however not every name is a correct one. At first, VHDL has reserved words (see Appendix 4). Other IDaSS names can start with an '_' (temporary variables) or with a '*' and that is not allowed in VHDL.
Other problems arise because it is not allowed in VHDL to give a bus or instance a name that has been used somewhere else in the same architecture.
The functions that do the renaming add a postfix to the names or prepend a prefix to it. As mentioned earlier these pre- and postfixes must not be added to the same field as the real name, because this would mean that the "old" name will be lost. This name however is used in all kinds of references and after adding a postfix or a prefix these references could not be resolved any more.

6.2 Registers

The registers found in an IDaSS design are written in the VHDL behavioural files, according to the next template.

```
PROCESS(clk,reset)

   declaration of the internal temporary variable of the register

BEGIN
   IF (reset = '1') THEN
      the optional output, the internal variable and the optional output
      to the FSM will get the asynchronous reset value
   ELSE IF (clk'EVENT AND clk = '1') THEN
      the commands of the register (mostly a case statement to decide which
      command has to be executed)
   ENDIF;
END PROCESS;
```

The register commands are mostly written as a VHDL case statement to select the various functions. If the output of the register is a three state output then a separate process to handle that output is generated. This process follows the next template.

```
<signal declarations> if necessary
PROCESS(<enablectrl>,<internal signal>)
BEGIN
   <test output> <= <internal signal>;
   IF (<enable ctrl> = <enable>) THEN
      <real output> <= <internal signal>;
   ELSE
      <real output> <= (OTHERS => 'Z');
   END IF;
END PROCESS;
```

The signal that controls the three state output (enablectrl), can be a signal coming from a finite state machine or a internal controller signal. The latter case is present if the three state output is controlled by a control connector on the register. In the process describing the commands of the register this signal will be set in the VHDL case statement describing the commands of the register when necessary (see the next example).

As an example the description of a control connector on a register (the name of the entity is reg) is converted to VHDL. In IDaSS the control description is:

```
%0xxx enable.
%xx00 hold.
%xx11 inc.
```

Assuming that the default function of this register is load, the next VHDL text will be present (after removing the overlap in the command encoding (see [Kru 95]) in the first described process in the part where the commands of the register are written:

```
d2v_enable <= '0';          -- describing the default value of this control signal

CASE to_integer(ctrl) IS
   WHEN 3 | 7 =>
      d2v_enable <= '1';
      d2v_registertemp := d2v_registertemp + "00000001";          -- inc
      d2v_reg_int <= d2v registertemp;
   WHEN 0 | 4 =>
      d2v_enable <= '1';
      -- do nothing in hold state
   WHEN 1 | 2 | 5 | 6 =>
      d2v_enable <= '1';
      d2v_registertemp := input;          -- load by default
      d2v_reg_int <=d2v_registertemp;
   WHEN 8 | 12 =>
      -- do nothing in hold state
```

```
    WHEN 11 | 15 =>
      d2v_registertemp := d2v_registertemp + "00000001";          -- inc
      d2v_reg_int <= d2v registertemp;
    WHEN others =>
      d2v_registertemp := input;                         -- load by default
      d2v_reg_int <=d2v_registertemp;
  END CASE;
```

If the register is controlled by a finite state machine these extra internal controller signals are not necessary. The bits defining the three state output are specificly added to the controller bus coming from the FSM at a known bit position (see [Kru 95] for the structure of these controller buses coming from an FSM). This part of the controller bus can be written immediately in the sensitivity list of the second process that was handled here.

6.3 Operators

The function that writes the VHDL code for the IDaSS operator blocks optimizes the expressions of the operator as a first step.
A function of an IDaSS operator is written as the body of a VHDL process. If there is more than one function present in the operator, the assignments of these functions are written in separate VHDL procedures. In the latter case the actual behaviour of the operator, controlled by a control input or an input coming from a finite state machine, is written in the body of the VHDL process in the form of a case statement to decide which procedure has to be executed. In the sensitivity list of the VHDL process all the inputs of the operator have to be present.
An example of an operator in VHDL is given below.

```
ENTITY oper IS
  PORT(
    ctrl: IN std_ulogic;
    in1: IN std_ulogic_vector(7 DOWNTO 0);
    in2: IN std_ulogic_vector(7 DOWNTO 0);
    o: OUT std_ulogic_vector(7 DOWNTO 0)
  );
END oper;

ARCHITECTURE behaviour OF oper IS

BEGIN

  PROCESS(in1,in2,ctrl)

  PROCEDURE a_function

  BEGIN
    o <= in1 AND in2;
  END a_function;
```

```
        PROCEDURE another_function

        BEGIN
          o <= in1 OR in2;
        END another_function;

        BEGIN
          CASE to_integer(ctrl) IS
          WHEN 1 =>
            a_function;
          WHEN OTHERS =>
            another_function;
          END CASE;
        END PROCESS;
      END behaviour;
```

As can be seen the assignments present in an operator are sequentially written and all have the next form:

```
signal <= expression;     or     variable := expression;
```

After writing the signal or the variable name to the VHDL file the expression is passed to another C-function that checks the operators and decides what to do with the expression and how to write it in VHDL.
IDaSS recognizes three kinds of operators in expressions: unary operators, binary operators and keyword operators. Some of the more complex IDaSS operators could not be translated to VHDL immediately, but a VHDL function is necessary. The next tables (table 1, table 2 and table 3) show all the IDaSS operators together with the operator representation in the C-code (**enum** OPERATOR) and the construct that is written in VHDL (expr means the expression that is attached to the operator, exprn means the n-th parameter of the operator).
Also two-input multiplexers have their own treatment but these operators are not explained here, for more information see [Kru 95]. As mentioned in the previous chapter, the real actions of the if0if1 and if1if0 operators have been done earlier. The only thing that has to be done when these operators are found in the expression trees at this moment (writing the expressions) is just to write the name of the already created temporary variable and use it in the expression just as a normal operand.

Table 1 IDaSS unary operators together with their VHDL representation.

IDaSS ope rator	C code repre sen-tation	VHDL construct
dec	uDEC	(expr - '1')
inc	uINC	(expr + '1')
neg	uNEG	NOT (expr + '1')
not	uNOT	NOT expr
epty	uEPTY	(expr(0) XOR expr(1) XOR ... XOR expr(width-1))
opty	uOPTY	NOT (expr(0) XOR expr(1) XOR ... XOR expr(expr-1))
ones	uONES	"111...1"
zeroes	uZEROES	"000...0"
width	uWIDTH	*not written because this operator is already removed and handled while optimizing the expressions*

Table 2 IDaSS binary operators together with their VHDL representation.

IDaSS operator	C code representation	VHDL construct
+	bADD	expr + expr1
-	bSUB	expr - expr1
*	bUNMUL	expr * expr1
*+	bRMUL	expr * signed'expr1
+*	bLMUL	signed'expr * expr1
+*+	bSMUL	signed'expr * signed'expr1
/\	bAND	expr AND expr1
\/	bOR	expr OR expr1
><	bXOR	expr XOR expr1
< >	bXNOR	NOT (expr XOR expr1)
=	bEQUAL	*bit2ulogicbit(boolean2bit*(expr = expr1))
~=	bNOTEQ	*bit2ulogicbit(boolean2bit*(expr /= expr1))
<	bLESS	*bit2ulogicbit(boolean2bit*(expr < expr1))
<=, =<	bLESSEQ	*bit2ulogicbit(boolean2bit*(expr <= expr1))
>	bMORE	*bit2ulogicbit(boolean2bit*(expr > expr1))
>=, =>	bMOREEQ	*bit2ulogicbit(boolean2bit*(expr >= expr1))
+=+	bSIEQUAL	*bit2ulogicbit(boolean2bit*(expr = expr1))
+~=+	bSINOTEQ	*bit2ulogicbit(boolean2bit*(expr /= expr1))
+<+	bSILESS	*bit2ulogicbit(boolean2bit*(signed'expr < signed'expr1))
+<=+, +=<+	bSILESSEQ	*bit2ulogicbit(boolean2bit*(signed'expr <= signed'expr1))
+>+	bSIMORE	*bit2ulogicbit(boolean2bit*(signed'expr > signed'expr1))
+>=+, +=>+	bSIMOREEQ	*bit2ulogicbit(boolean2bit*(signed'expr >= signed'expr1))
,	bCONCAT	expr & expr1

Table 3 IDaSS keyword operators together with their VHDL representation.

IDaSS operator	C code representation	VHDL construct
shl:	kSHL	SHL(expr,expr1)
shr:	kSHR	SHR(expr,expr1)
rol:	kROL	(SHL(expr,expr1) OR SHR(expr,(expr'length-expr1)))
ror:	kROR	(SHR(expr,expr1) OR SHL(expr,(expr'length-expr1)))
at:	kAT	expr(expr1) *or* expr(*logic_type2integer*(expr1))
from:to:	kFROMTO	expr(expr2 DOWNTO expr1)
if0:if1:	kIF0IF1	*The real work has already been done so it is enough to write the*
if1:if0:	kIF1IF0	*temporary variable's name appended with the correct number.*
merge:mask:	kMERGEMASK	((expr AND NOT expr2) OR (expr1 AND expr2))
copiesof:	kCOPIESOF	expr & expr & ... & expr *(expr1 times)*

The operators that have not been handled till now are too complex to write just in the middle of an expression. Or in some special cases it is not possible to write the operators in the process body (the operator "at:" and from:to: can be written as shown in table 3 in the process body except when the left arguments are expressions).
If it is not possible to write the operators in the process body, a function call to a VHDL function will be written. These VHDL functions already have been written at the declaration section of the VHDL process at the top of the architecture description (at the same places as the VHDL procedures are written).
It is possible that an IDaSS operator has different possible VHDL functions, for instance the operator "maj:" has a different behaviour in case the width of the receiver is odd or even.

In Appendix 5 examples are given of the different VHDL functions of the IDaSS operators that have not been handled yet. Most input signals have a width of 8 bit, if its possible to write an operator in different VHDL functions, examples are given of all these possibilities.

Other problems rise when the parameters does not fit the correct types in the function header. To solve this problem, sometimes user defined constructs like:

to_integer(*no_integer*)

are used when the VHDL function is called. These constructs are also used for variable input parameters like the parameter of the operator "at:".

As an example of the hardware that will be generated while making use of these complex IDaSS operators, the VHDL function of an operator with the easy assignment:

out := in msomask.

is compiled by the des2vhdl compiler and a netlist has been generated by COMPASS. The schematic that is drawn by COMPASS is depicted in Appendix 6.

When an operator has a three state output, an extra process has to be generated just like the one used for the three state outputs of the registers (see the previous section). The difference with writing registers is, that it is possible to have multiple three state outputs, meaning multiple processes describing the three state behaviour and in case of a control input on the operator, multiple internal signals. Therefore the internal signals will get as part of the signal name, the name of the output they are controlling, to keep the names unique.

6.4 Finite state machines

Finite state machines have to be described in VHDL by two processes (see [ASIC],[ARJ 93] and [LHY 92]). The first process describes the actual state of the state machine with the help of two signals. These signals are declared to have an enumerated type. The declaration of this type and these signals is done by the next lines, present in the declaration section of the state machine's behavioural architecture (the used names are the default settings the converter uses).

```
TYPE state_type IS (names of all possible states);
SIGNAL current_state : state_type;
SIGNAL next_state : state_type;
```

The first process called 'SYNC', follows the next template.

```
SYNC: PROCESS(clk,reset)
BEGIN
   IF (reset = '1') THEN
      current_state <= default state name;
   ELSIF (clk'EVENT AND clk = '1') THEN
      current_state <= next_state;
   END IF;
END PROCESS SYNC;
```

This first process is a registered process to store the state variable. External influences ("goto:" or "reset" commands) and stack actions can be handled in this process after the ELSIF line.
The second process is a combinational process to describe the state transitions and the actions that have to be taken in a certain state. This process, called 'FSM' exists of a case statement to determine the correct state. All the inputs of the FSM's entity have to be present in the sensitivity list of this process, plus the internal signal current_state. For cosmetic reasons the inputs clk and reset are not written in this list because they are already present in the sensitivity list of the process 'SYNC'.

The template the process FSM follows, is given here.

```
FSM: PROCESS(current_state, all the inputs of the entity except clk and reset)
BEGIN
  CASE current_state IS
    WHEN first_state =>
      list of assignments of all the default values
      list of the real state commands of the first state
    WHEN second_state =>
      list of assignments of all the default values
      list of the real state commands of the second state
    WHEN OTHERS =>
      -- do nothing
  END CASE;
END PROCESS FSM;
```

When the FSM processes are always written like this some problems will rise. In this case we assume that in a condition block present in the commands of a state, only one entry is executed at the time. In IDaSS however its possible to have parallel execution of commands (the test values of a condition block have overlapping values). This problem is already handled in [Kru 95] so from now on there are only non overlapping test values in a condition block.
Still one other problem stays, caused by the fact that IDaSS allows nested tests to be performed and the way IDaSS treats state transitions. To illustrate the problem the next state is converted to VHDL.

```
first_state:
[TEST_REG
 | %00    action1;
          [SECOND_TEST_REG
           | %00    action2;
           | %10    action3;
          ];
          action4;
 | %11    action5;
];
action6;
```

If this state is just written to VHDL, a problem rise when for instance *action2* describes a state transition. In IDaSS the simulator immediately steps to that state, in VHDL however only an assignment to the signal next_state is executed followed by *action4* and *action6* (assuming that *action4* is not a state transition). According to the ideas presented in [Ver 95] this has to be transferred to a VHDL case in the FSM: PROCESS like this (assuming *action2*, *action3* and *action5* to describe state transitions):

```
CASE to_integer(TESTREG) IS
   WHEN 0 =>
      action1;
      CASE to_integer(TESTREG2) IS
         WHEN 0 =>
            action2; -- state transition
         WHEN 2 =>
            action3; -- state transition
         WHEN OTHERS =>
            -- do nothing
      END CASE;
      IF NOT ((TESTREG2 = 0) OR (TESTREG2 = 2)) THEN
         action4;
      END IF;
   WHEN 3 =>
      action5; -- state transition
   WHEN OTHERS =>
      -- do nothing
END CASE;
IF NOT (((TESTREG = 0) AND ((TESTREG2 = 0) OR (TESTREG2 = 2))) OR
      (TESTREG = 3)) THEN
   action6;
END IF;
```

As can be seen the IF statements following the CASEs are already very complex and not practical. In more complex state machines these guards will be even more incomprehensive and very complex to calculate. A better solution to overcome this complexity is the next that has been implemented in the des2vhdl compiler.

```
jump := FALSE; -- default
CASE to_integer(TESTREG) IS
   WHEN 0 =>
      action1;
      CASE to_integer(TESTREG2) IS
         WHEN 0 =>
            action2; -- state transition
            jump := TRUE;
         WHEN 2 =>
            action3; -- state transition
            jump := TRUE;
         WHEN OTHERS =>
            -- do nothing
      END CASE;
      IF (jump = FALSE) THEN
         action4;
      END IF;
```

```
      WHEN 3 =>
         action5; -- state transition
         jump := TRUE;
      WHEN OTHERS =>
         -- do nothing
   END CASE;
   IF (jump = FALSE) THEN
      action6;
   END IF;
```

The calculation of the guards is now left to the VHDL synthesizer and the VHDL text is much more understandable. The only thing we have to add to the VHDL code to make this approach work, is the declaration of the used boolean.

This boolean that has just been introduced is not necessary in all cases. If the guards of the CASE statements cover all possible values and if all these possibilities contain a state transition the new boolean is not necessary to control the state transitions and the following actions are never reached so they can be skipped by the converter. To determine whether the CASE covers all values a C-function is written that makes a list of all <value|mask> pairs of the test values of the CASE in ascending order of value. When this list is generated, the function starts searching for a value in the range 0 to $(2^{(\text{width of the test expression})} - 1)$ that is not covered by any of the <value|mask> pairs. This test makes use of the formulae:

$$(TESTVALUE \wedge (\neg MASK)) = VALUE \tag{6.1}$$

If this formulae holds, a test value is covered by a certain <value|mask> pair. If just one value is not covered, the boolean is necessary, only if all WHEN's seem to end in a state transition.

When the finite state machine only has one state a lot of work can be skipped. At first the registered process containing the current state variable is not necessary any more. In the second process the CASE that makes the decision which state is active can also be forgotten. Only the real actions of the state have to be written in this combinational process.

6.5 Buses with multiple drivers

Despite the work that has been done to implement the three state output connectors correctly, problems are still rising while connecting the buses together coming from these connectors. Although these problems depend on the VHDL synthesizer used, possible implementations for three state buses are presented in this section[5] ([ASIC]).

Three state logic can be created using a resolved signal that has multiple concurrent processes driving the signal. The resolved signal must be of type std_logic or std_logic_vector, as defined in the std_logic_1164 package.
The following example illustrates how to create a signal that is driven by multiple three state drivers.

---

5 These solutions will work under the ASIC Synthesizer V8R4.6 of COMPASS.

```
ENTITY threestate IS
   PORT (x,y,en : IN std_logic;
      trisig : OUT std_logic
      );
END threestate;

ARCHITECTURE threestate OF threestate IS

BEGIN
   PROCESS(x,en)
   BEGIN
      IF en  = '0' THEN
         trisig <= x;
      ELSE
         trisig <= 'Z';
      END IF;
   END PROCESS;

   PROCESS(y,en)
   BEGIN
      IF en  = '1' THEN
         trisig <= y;
      ELSE
         trisig <= 'Z';
      END IF;
   END PROCESS;
END threestate;
```

In this example, the ASIC Synthesizer uses two three state buffers to create the circuit (see Appendix 7). Assuming an active-HIGH three state buffer, the first process creates a three state buffer whose input is connected to x. Its output is connected to trisig, and the (not en) signal is connected to the enable pin of the three state buffer. Similarly, the second process creates a three state buffer whose input is connected to y. Its output is connected to trisig, and the en signal is connected to the enable pin of the three state buffer.

The ASIC Synthesizer performs an approximate test to check if multiple three state drivers are mutually exclusive. If it cannot prove that the drivers are mutually exclusive, it issues a warning message.

Three state flip flops can be similarly created by setting a resolved signal to 'Z' in a clocked process, as in the following example, but in IDaSS a clocked solution is never used so this three state bus is not correct for the IDaSS buses:

```
ENTITY threestateff IS
   PORT (x,y,clk,en : IN std_logic;
      trisig : OUT std_logic
      );
END threestateff;


ARCHITECTURE threestateff OF threestateff IS

BEGIN
   PROCESS(clk)
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
         IF en = '0' THEN
            trisig < = x;
         ELSE
            trisig < = 'Z';
         END IF;
      END IF;
   END PROCESS;

   PROCESS(clk)
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
         IF en = '1' THEN
            trisig < = y;
         ELSE
            trisig < = 'Z';
         END IF;
      END IF;
   END PROCESS;

END threestateff;
```

Two three state flip flops are created in this example (see Appendix 8). One of them has x as its D input, and the other has y as its D input. The outputs of the two flip flops are connected to the trisig signal. The enable pin of these three state flip flops is controlled by two other flip flops that take the en and (not en) signals as inputs to their D pin. The reason for these extra flip flops is to prevent changes on the en signal between clock periods from affecting the outputs of the three state flip flops. The three state flip flop outputs change only on a clock edge. If you do not want these extra flip flops, the behaviour should be rewritten as follows:

```
ENTITY ffandts IS
   PORT (x,y,clk,en : IN std_logic;
      trisig : OUT std_logic
      );
END ffandts


ARCHITECTURE ffandts OF ffandts IS

SIGNAL ffout1, ffout2 : std_logic;

BEGIN
   PROCESS(clk)
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
         ffout1 <= x;
      END IF;
   END PROCESS;

   PROCESS(clk)
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
         ffout2 <= y;
      END IF;
   END PROCESS;

   trisig <= ffout1 WHEN en = '0' ELSE 'Z';
   trisig <= ffout2 WHEN en = '1' ELSE 'Z';
END ffandts;
```

In this example, the output of regular flip flops is connected to three state buffers that then connect to the three state bus trisig (see Appendix 9). This solution will work for IDaSS because the buses are combinatorial.

The des2vhdl compiler does not generate any of the possibilities presented in this section. The converter just connects three state signals coming from different entities together to one bus. This solution makes the Compass tool give several warnings (eg. multiple drivers present on bus ...). This solution however will work, when the user for instance works with Synopsis to synthesize the VHDL code.

7 VHDL CONFIGURATION FILE

To make user interaction a lot easier and to make the generated VHDL code suitable for different VHDL synthesizers, a file containing different program settings is read during program execution. The contents and the format of this file are discussed in this chapter. The file that contains all the information that can be changed by the user is called user_def and is read by the compiler every time a des-file is converted to VHDL. So it is not necessary to recompile the complete C-source of the des2vhdl compiler.

7.1 File format and contents

The settings wanted by the user have to be preceded by a certain switch in the user_def file. The switches each have to start with a '#' character at the beginning of a new line, directly followed by the name of the switch. Three types of information can be read. Integers, strings and multiple strings. A string cannot contain any whitespace characters but a multiple string can. To be clear where the multiple strings start and end all the user information has to be included in double quotes (this is not necessary for normal strings). Examples of the three kinds of user definitions are given here.

```
#INDENT 3            integer
#CLK_NAME clk        string
#FILE_HEADER         multiple string
"
-- This is the file header (starting and ending with a newline)
"
```

Also the reserved words of VHDL are included in this user file. The list of reserved words that have to be collected in a hash table [CLR 90] starts with a switch character like all the other information of the file but the last string of the list has to be !RESERVED to indicate that no other reserved words will follow.
The des2vhdl compiler stops reading the file when it scans the character '.' at the start of a new line. The information after this dot and all the other lines of the file (not starting with a '#') are skipped by the compiler. The different names of the switches that are possible are shown in Appendix 10 together with the used default settings of the variables.

The information, when read, is available in the C-code as a global variable (int for integers and char * for strings and multiple strings). If the user does not want to give a user option for a certain variable the lines containing the switch and the new value have to be removed from the user_def file (easier is to type a certain character in front of the switch in which case the lines are simply not read by the compiler). The compiler will now use a default setting for these variables.

8 CONCLUSIONS

The designs made in IDaSS, can be translated automaticly to synthesizable VHDL while making use of the converter that is now present. Most constructs can be handled and the corresponding VHDL description is organized in such a way that the original design (in IDaSS) is reflected as closely as possible.

In contrary to the converter that was present at the start of this graduation period the current converter can also handle the next IDaSS constructs correctly.
At first, all the names in a IDaSS design are translated to correct names in VHDL, without name clashes between for instance buses and components.
Secondly, the behaviour of registers is translated to correct synthesizable VHDL even when three state outputs are present in the design. The registers make use of a internal variable containing the actual contents of the register. The three state ports are written with the use of an extra VHDL process (also in operators).
Thirdly, all the operators present in IDaSS operator blocks can be written in VHDL. The operators that were to complex to be written in a simple assignment, were written in VHDL functions built out of simple standard VHDL constructs, so the output can be used in combination with every wanted silicon compiler.
Fourthly, the behaviour of the finite state machines is optimized, and written in simpler VHDL code.
Fifthly, three possible solutions for the implementation of three state buses are presented. These solutions are not implemented in the compiler because only Compass needs these constructs. Other silicon compilers are happy with the present solutions.
At last the converter is extended with the possibility for the user to interact much easier in the way the program executes. A file containing user definitions is read during execution so its not necessary to recompile the complete C-code for every minor program change.

The converter is still not able to translate a design correctly that makes use of for instance a RAM, a ROM, a LIFO, a FIFO or a CAM. Also register semaphores and multiple controllers, controlling one block are not implemented correctly, although these last two problems are not very complex to add to the converter.

LITERATURE

[ASIC] VHDL FOR THE ASIC SYNTHESIZER
V8R4.6
(Part of the online Compass manual)

[ARJ 93] Abrahams, M.S. and A. Rushton, P. Johnson
DESIGN USING VHDL FOR SYNTHESIS AND TEST
In: Proceedings of the IEE Colloquium on VHDL - Applications and CAE Advances, London, United Kingdom, April 6, 1993.
London: IEE, 1993, p. 3/1-5.

[Ben 90] Bennet, J.P.
INTRODUCTION TO COMPILING TECHNIQUES. A FIRST COURSE USING ANSI C, LEX AND YACC
London: Mc Graw Hill Book Company, 1990.

[CLR 90] Cormen, T.H. and C.E. Leiserson, R.L. Rivest
INTRODUCTION TO ALGORITHMS
New York: Mc Graw Hill Book Company, 1990.

[CST 91] Composano, R. and L.F. Saunders, R.M. Tabet
VHDL AS INPUT FOR HIGH-LEVEL SYNTHESIS
IEEE Design & Test of Computers, Vol. 8 (1991), no. 1, p. 43-49.

[DO 92] Debreil, A. and P. Oddo
SYNCHRONOUS DESIGN IN VHDL
In: proceedings of the EURO-DAC '92 European Design Automation Conference with EURO-VHDL '92, Hamburg, Germany, Sept. 7-10, 1992.
Los Alamitos: IEEE Comput. Soc. Press, 1992, p. 680-681.

[Guy 92] Guyler, A.
VHDL 1076-1992 LANGUAGE CHANGES
In: proceedings of the EURO-DAC '92 European Design Automation Conference with EURO-VHDL '92, Hamburg, Germany, Sept. 7-10, 1992.
Los Alamitos: IEEE Comput. Soc. Press, 1992, p. 672-678.

[HS 92] Harper, P.L. and K. Scott
TOWARDS A STANDARD VHDL SYNTHESIS PACKAGE
In: proceedings of the EURO-DAC '92 European Design Automation Conference with EURO-VHDL '92, Hamburg, Germany, Sept. 7-10, 1992.
Los Alamitos: IEEE Comput. Soc. Press, 1992, p. 706-712.

[KR 88] Kernighan, B.W. and D.M. Ritchie
C HANDBOEK
Schoonhoven: Prentice Hall, 1990.
Translated from the English: The C Programming Language.
London: Prentice Hall, 1988.

[Kru 94] Kruijtzer, W.M.
GENERATING A VHDL FRAMEWORK FROM IDASS DESIGN FILES
Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, 1994.
Report of practical training period.

[Kru 95] Kruijtzer, W.M.
GENERATING A VHDL FRAMEWORK FROM IDASS DESIGN FILES
Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, 1995.
Master's thesis.

[LHY 92] Lim, S.E. and D.C. Hendry, P.F. Yeung
EXPERIENCES AND ISSUES IN VHDL-BASED SYNTHESIS
In: proceedings of the EURO-DAC '92 European Design Automation Conference with EURO-VHDL '92, Hamburg, Germany, Sept. 7-10, 1992.
Los Alamitos: IEEE Comput. Soc. Press, 1992, p. 646-651.

[LMB 92] Levine, J.R. and T. Mason, D. Brown
LEX AND YACC. 2nd ed.
Sebastopol: O'Reilly & Associates, Inc., 1992.

[LSU 89] Lipsett, R. and C.F. Schaeffer, C. Ussery
VHDL: HARDWARE DESCRIPTION AND DESIGN
Dordrecht: Kluwer Academic Publishers, 1989.

[ML 92] Mazor, S. and P. Langstraat
A GUIDE TO VHDL. 2nd ed.
Dordrecht: Kluwer Academic Publishers, 1992.

[Nav 93] Navabi, Z.
VHDL ANALYSIS AND MODELING OF DIGITAL SYSTEMS
Singapore: Mc Graw Hill Iternational Editions, 1993.

[NBD 92] Nogasamy, V. and N. Berry, C. Dangelo
SPECIFICATION, PLANNING, AND SYNTHESIS IN A VHDL DESIGN ENVIRONMENT
IEEE Design & Test of Computers, Vol. 9 (1992), no. 2, p. 58-68.

[NS 90] Navabi, Z. and J. Spillane
TEMPLATES FOR SYNTHESIS FROM VHDL
In: Proceedings. Third Annual IEEE ASIC Seminar and Exhibit, Rochester, USA, Sept. 17-21, 1990. Ed. by Hsu, K.W. and M.E. Schrader.
New York: IEEE Comput. Soc. Press, 1990, p. P16/1.1-1.4.

[Pee 91] Peerbooms, M.
ANALYSE AND IMPLEMENTATIE VAN IDASS BOUWSTENEN IN VHDL
Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, 1991.
Graduation Report.

[STD 87] IEEE STANDARD VHDL LANGUAGE REFERENCE MANUAL
New York: IEEE, 1988.
IEEE Std. 1076-1987.

[VD 92] Vermeeren, J. and D. Donderman
OMSCHRIJVEN VAN IDASS OPERATOREN NAAR SID
Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, 1992.
Graduation Report.

[Ver 90a] Verschueren, A.C.
IDASS FOR ULSI (IDASS MANUAL)
Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, 1990.

[Ver 90b] Verschueren, A.C.
THE IDASS FILE FORMATS V0.07
Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, 1990.

[Ver 95a] Verschueren, A.C.
FLATTENING IDASS STATE DESCRIPTIONS TO NESTED VHDL IF/CASE STATEMENTS
Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, 1995.

[Ver 95b] Verschueren, A.C.
IDASS V0.09 .DES FILE FORMAT
Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, 1995.

APPENDICES

APPENDIX 1 IDASSLEX.L

```
%{
#define NEED_STRING
#include "des2vhdl.h"
#include "types.h"

/*** local types ***/
struct BOUNDEDINT {
  int value;
  int width;
};
#include "y_tab.h"
#include "global.h"

/*
 * This is a complete implementation of a tokenizer for the IDaSS design
 * file format as stated in the internal report :
 * The IDaSS file formats V0.08, September 27, 1993 which actually is
 * the same as V0.07, March 19, 1990.
 *
 * The tokenizer skips Lf's, spaces, comments and all graphics info.
 * It recognises all basic data items e.g integer, name, boundedInt and nil
 * and a lot of switch characters.
 */

/*** exported ***/
void yyerror(char *s);

/*** local variables ***/
int lineno =1;

#ifdef __ZTC__                                  /* the flex scanner needs this */
#define fileno(fp)      ((fp)->_file)           /* this is a non-ansi function */
int isatty(int);                                /* non-ansi function (see io.h)*/
#endif                                          /* so we declare it here       */

#define yywrap() 1                              /* this is for flex version 2.4.6 (not needed for 2.3.7) */
%}

graphics                                        \/.*
comment                                         \".*
comptext                                        \'.*
ws                                              [\t \r]+
name                                            [A-Za-z][A-Za-z0-9_]*
no_iname                                        \*"i"
no_oname                                        \*"o"
no_cname                                        \*"c"
val_start                                       [%&$]
decimal                                         [0-9]+
value                                           [0-9a-fA-F]*
val_end                                         [bBdDhHoOqQ]
xvalue                                          [0-9a-fA-FxX]*
binaryop                                        [+\-*/\\><=~]+
integer                                         -?[0-9]+
boundedint                                      {integer}"W"{integer}
nil                                             "?"

%x  connector filespec comptext comment
```

```
%%
{ws}                                    ;
^{graphics}                             ;
^{comment}                              ;
^\'                                     { BEGIN(comptext); }
\n                                      { lineno++; }
"#SuperBlock"                           { return(T_SuperBlock_begin); }
".SuperBlock"                           { return(T_SuperBlock_end); }
"#SuperConnector"                       { return(T_SuperConnector_begin);}
".SuperConnector"                       { return(T_SuperConnector_end);}
"#Bus"                                  { return(T_Bus_begin); }
".Bus"                                  { return(T_Bus_end); }
"#StateControl"                         { return(T_StateControl_begin); }
".StateControl"                         { return(T_StateControl_end); }
"#Signal"                               { return(T_Signal_begin); }
".Signal"                               { return(T_Signal_end); }
"#Buffer"                               { return(T_Buffer_begin); }
".Buffer"                               { return(T_Buffer_end); }
"#Constant"                             { return(T_Constant_begin); }
".Constant"                             { return(T_Constant_end); }
"#Register"                             { return(T_Register_begin); }
".Register"                             { return(T_Register_end); }
"setto:"                                { return(T_Setto); }
"#Operator"                             { return(T_Operator_begin); }
".Operator"                             { return(T_Operator_end); }
"#RAM"                                  { return(T_RAM_begin); }
".RAM"                                  { return(T_RAM_end); }
"#ROM"                                  { return(T_ROM_begin); }
".ROM"                                  { return(T_ROM_end); }
"#FIFO"                                 { return(T_FIFO_begin); }
".FIFO"                                 { return(T_FIFO_end); }
"#LIFO"                                 { return(T_LIFO_begin); }
".LIFO"                                 { return(T_LIFO_end); }
"#CAM"                                  { return(T_CAM_begin); }
".CAM"                                  { return(T_CAM_end); }

^"V"/({name}|{boundedint}|{integer}|{nil})   { return(T_Vswitch); }
^"O"/("O"|"T")({name}|{no_oname})       { BEGIN(connector); return(T_Oswitch); }
^"D"/("O"|"T")({name}|{no_oname})       { BEGIN(connector); return(T_Dswitch); }
^"M"/("O"|"T")({name}|{no_oname})       { BEGIN(connector); return(T_Mswitch); }
^"Q"/("O"|"T")({name}|{no_oname})       { BEGIN(connector); return(T_Qswitch); }
^"Z"/"C"({name}|{no_cname})             { BEGIN(connector); return(T_Zswitch); }
^"C"/"B"{name}                          { BEGIN(connector); return(T_Cswitch); }
^"I"/"I"({name}|{no_iname})             { BEGIN(connector); return(T_Iswitch); }
^"R"/"I"({name}|{no_iname})             { BEGIN(connector); return(T_Rswitch); }
^"A"/"I"({name}|{no_iname})             { BEGIN(connector); return(T_Aswitch); }
^"D"/"I"({name}|{no_iname})             { BEGIN(connector); return(T_Dswitch); }
^"W"/"I"({name}|{no_iname})             { BEGIN(connector); return(T_Wswitch); }
^"M"/"I"({name}|{no_iname})             { BEGIN(connector); return(T_Mswitch); }
^"N"/"I"({name}|{no_iname})             { BEGIN(connector); return(T_Nswitch); }
^"B"/"I"({name}|{no_iname})             { BEGIN(connector); return(T_Bswitch); }
^"C"/"I"({name}|{no_iname})             { BEGIN(connector); return(T_Cswitch); }
^"E"/"I"({name}|{no_iname})             { BEGIN(connector); return(T_Eswitch); }
^"G"/"I"({name}|{no_iname})             { BEGIN(connector); return(T_Gswitch); }
^"C"/{integer}                          { return(T_Cswitch); }
^"T"/{integer}                          { return(T_Tswitch); }
^"P"/{integer}                          { return(T_Pswitch); }
^"W"/{integer}                          { return(T_Wswitch); }
^"X"/{integer}                          { return(T_Xswitch); }
^"K"/{integer}                          { return(T_Kswitch); }
^"F"/({name}|{integer}|{nil})           { return(T_Fswitch); }
^"S"/({name}|{integer})                 { return(T_Sswitch); }
^"L"/.*                                 { BEGIN(filespec); return(T_Lswitch); }

{name}                                  { yylval.string = yytext; return(T_Name); }
{no_cname}                              { yylval.string = yytext; return(T_No_cname); }
{no_oname}                              { yylval.string = yytext; return(T_No_oname); }
```

```
{no_iname}                                  { yylval.string = yytext; return(T_No_iname); }
{boundedint}                                { char *p = strchr(yytext,'W');
                                              yylval.boundedint.value = atoi(yytext);
                                              yylval.boundedint.width = atoi(p+1);
                                              return(T_BoundedInt);
                                            }
{integer}                                   { yylval.value = atoi(yytext); return(T_Integer); }
{nil}                                       { return(T_Nil); }
.                                           { yyerror("invalid character"); }

<connector>"B"/{name}                       { BEGIN(INITIAL); return(T_conBswitch); }
<connector>"C"/({name}|{no_cname})          { BEGIN(INITIAL); return(T_conCswitch); }
<connector>"I"/({name}|{no_iname})          { BEGIN(INITIAL); return(T_conIswitch); }
<connector>"O"/({name}|{no_oname})          { BEGIN(INITIAL); return(T_conOswitch); }
<connector>"T"/({name}|{no_oname})          { BEGIN(INITIAL); return(T_conTswitch); }

<filespec>[^\n]*                            { BEGIN(INITIAL); return(T_Filename); }

<comment>\"                                 { BEGIN(comptext); }
<comment>\n                                 { lineno++; }
<comment>.                                  ;

<comptext>{ws}                              ;
<comptext>\"                                { BEGIN(comment); }
<comptext>\n                                { lineno++; BEGIN(INITIAL); }
<comptext>":="                              { return(T_Equals); }
<comptext>\.                                { return(T_Period); }
<comptext>\.\.                              { return(T_dPeriod); }
<comptext>\(                                { return(T_Leftbrace); }
<comptext>\)                                { return(T_Rightbrace); }
<comptext>"["                               { return(T_Leftbracket); }
<comptext>"]"                               { return(T_Rightbracket); }
<comptext>","                               { yylval.string = yytext; return(T_Comma); }
<comptext>";"                               { return(T_Semicolon); }
<comptext>"|"                               { return(T_Bar); }
<comptext>\/                                { return(T_Slash); }
<comptext>\\                                { return(T_Backslash); }
<comptext>"!"                               { return(T_Exclamation); }
<comptext>"!!"                              { return(T_dExclamation); }
<comptext>"?"                               { return(T_Question); }
<comptext>"??"                              { return(T_dQuestion); }
<comptext>"->"                              { return(T_Jumpstate); }
<comptext>"<<"                              { return(T_Holdstate); }
<comptext>">>"                              { return(T_Nextstate); }
<comptext>{val_start}{value}                { yylval.string = strsav(yytext); return(T_Value); }
<comptext>({decimal}|[0-9]{value}{val_end}) { yylval.string = strsav(yytext); return(T_Value); }
<comptext>{val_start}{xvalue}               { yylval.string = strsav(yytext); return(T_xValue); }
<comptext>[0-9]{xvalue}{val_end}            { yylval.string = strsav(yytext); return(T_xValue); }
<comptext>{binaryop}                        { yylval.string = strsav(yytext); return(T_Binaryop); }
<comptext>":"                               { return(T_Colon); }
<comptext>{name}":"/([^=])                  { yylval.string = strsav(yytext); return(T_Keywordop); }
<comptext>"_"{name}                         { yylval.string = strsav(yytext); return(T_Tempname); }
<comptext>{name}                            { yylval.string = strsav(yytext); return(T_Name); }

%%

void yyerror(char *s)
{
  fprintf(stderr,"%%LEX-E-ERROR, %s at %s in line %d \n", s, yytext, lineno);
}
```

APPENDIX 2 TYPES.H

```
#ifndef _TYPES_H
#define _TYPES_H

#include "number.h"

/*
 * Because I like my own 'boolean type' I undefine already existing
 * definitions. GNU dbm for example defines an own boolean type.
 */
#ifdef TRUE
#undef TRUE
#endif
#ifdef FALSE
#undef FALSE
#endif

typedef enum {FALSE,TRUE} boolean; /* my own boolean definition */

/*
 * The following type definitions are used to construct the overall
 * data-structure that contains the IDaSS design information.
 */
struct CONNECTION {
  char *systemname;
  char *contactname;
};

struct BUS {
  char *name;
  struct LIST *connections;    /* this will become a list of struct CONNECTION */

        /* the following items are convenient for generating VHDL code */
  char postfix;                /* Used as postfix if name is a reserved word */
  int width;                   /* Not in DES-file, calculated after parse */
  boolean need_extrabus;       /* This boolean is TRUE if we have an external contact */
                               /* of type continuous output and the bus has > 2 nodes. */
  struct CONTACT *contact;     /* If the bus is connected to a contact this will */
};                             /* point to that contact, else it will be NULL */

struct INPUT {
  char *name;
  int width;

        /* the following item is convenient for generating VHDL code */
  char postfix;                /* Used as postfix if name is a reserved word */
};

enum OUTPUTTYPE {o_cont,o_TS};

enum DEFSTATE {TS_disable,TS_enable};

struct OUTPUT {
  char *name;
  int width;
  enum OUTPUTTYPE type;
  enum DEFSTATE defstate;      /* Only valid if type == o_TS. */
  char *intname;               /* Only valid if type == o_TS. */

        /* the following item is convenient for generating VHDL code */
  char postfix;                /* Used as postfix if name is a reserved word */
};
```

```
enum CONTACTTYPE {c_bidir,c_input,c_output,c_TS};

struct CONTACT {
  char *name;
  enum CONTACTTYPE type;

        /* the following items are convenient for generating VHDL code */
  char postfix;              /* Used as postfix if name is a reserved word */
  int width;                 /* Not in DES-file, calculated after parse */
};

enum SYSTEMTYPE {FSM,BUFFER,CONSTANT,REGISTER,OPERATOR,RAM,ROM,FIFO,LIFO,CAM};

struct SYSTEM {
  char *name;
  struct LIST *inputs;       /* this will become a list of struct INPUT  */
  struct LIST *outputs;      /* this will become a list of struct OUTPUT */
  enum SYSTEMTYPE type;      /* The kind of system we have. */
  struct CTRLCON *ctrlcon;   /* pointer to control connector (if present). */
  struct LIST *commands;     /* list of commands send from fsm to this system */
  union
  { struct REGISTER *reg;    /* Used to store register specific things */
    struct OP *op;           /* Used to store operator specific things */
    struct FSM *fsm;         /* Used to store fsm specific things */
  } sys;

        /* the following items are convenient for generating VHDL code */
  char postfix;              /* Used as postfix if name is a reserved word */
  char *prefix;              /* Used as prefix if name is not unique */
};

struct REGISTER {
  short width;
  char *deffun;              /* pointer to default function */
  short areset;              /* value to be loaded upon system reset */
  short sreset;              /* value to be loaded for 'reset' command */
};

struct OP {
  struct LIST *functions;    /* This will become a list of struct FUNCTION */
  char *deffun;              /* pointer to default function */
};

struct FSM {
  struct LIST *states;       /* This will become a list of struct STATE */
};

struct LEVEL {
  struct LIST *lower;        /* this will become a list of struct LEVEL */
  char *name;
  struct LIST *buses;        /* this will become a list of struct BUS */
  struct LIST *systems;      /* this will become a list of struct SYSTEM */
  struct LIST *contacts;     /* this will become a list of struct CONTACT */

        /* the following items are convenient for generating VHDL code */
  char postfix;              /* Used as postfix if name is a reserved word */
  char inst_postfix;         /* Used as postfix if the name of the instance
                                of this level is not unique */
  char *prefix;              /* Used as prefix if name is not unique */
};
```

```
/*
 * Used to construct a list of temporary variables used in an operator
 * or state controller.
 */
struct TMPVAR {
  char *name;                   /* Name of temporary. */
  int width;                    /* Width of temporary variable. */
  short fnr;                    /* Holds the number of the function where the var. is declared. */
};

/*
 * This is used to hold the different functions of an operator.
 * Each function can have a number of assignments (pointed to by assignments).
 */
struct FUNCTION {
  char *name;                   /* name of function */
  struct LIST *assignments;     /* This will become a list of struct ASSIGNMENT */
  struct LIST *tmpvars;         /* pointer to a list of struct TMPVAR */
        /* the following items are convenient for generating VHDL code */
  char postfix;                 /* Used as postfix if name is a reserved word */
  char *prefix;                 /* Used as prefix if name is the same as in- or output name */
};

/*
 * This is used to store one complete assignment e.g name and expression-tree.
 */
struct ASSIGNMENT {
  char *name;                   /* outputname or tmpname */
  struct EXPR *expr;            /* pointer to complete expression */
};

/*
 * Used to save all the operators that need a VHDL function description
 */
struct FUNC_OP {
  enum OPERATOR op;             /* Kind of operator */
  int width;                    /* width of the function input variable */
  long reswidth;                /* width of the functions result (kSIGNED and kWIDTH) */
  struct EXPR *expr;            /* the expression tree started with the written operator */
  boolean written;              /* written indicates that the VHDL function
                                   of this operator has already been written */
};

/*
 * The following enumarated type is used to enumerate all operators (currently)
 * known in IDaSS (v0.08m). The following naming convention is used :
 * enumerators that start with an 'a' are used for operands.
 * enumerators that start with a 'u','b' or 'k' are used for resp. unary,binary
 * and keyword operators.
 * Because hard-coded constants are ugly-programming, there are a few dummy enumerators
 * defined to set the end of the different operators. These are aEND, uEND, bEND and
 * kEND. To find for example the number of unary operators one can use (uEND-aEND-1).
 */
enum OPERATOR {
  aOPERAND,aCONSTANT,
  aEND,
  uDEC,uINC,uNEG,uNOT,uEPTY,uOPTY,uMAJ,uLSOMASK,uLSZMASK,uMSOMASK,uMSZMASK,
  uLSONE,uLSZERO,uMSONE,uMSZERO,uONES,uZEROES,uREV,uONECNT,uZEROCNT,uWIDTH,
  uEND,
  bADD,bSUB,bUNMUL,bRMUL,bLMUL,bSMUL,bAND,bOR,bXOR,bXNOR,bEQUAL,bNOTEQ,bLESS,
  bLESSEQ,bMORE,bMOREEQ,bSIEQUAL,bSINOTEQ,bSILESS,bSILESSEQ,bSIMORE,bSIMOREEQ,
  bCONCAT,
  bEND,
  kSHL,kSHR,kSAR,kSOL,kSOR,kROL,kROR,kAT,kFROMTO,kATWIDTH,kIF0IF1,kIF1IF0,
  kMERGEFROMTO,kMERGEMASK,kWIDTH,kSIGNED,kCOPIESOF,
  kEND,kDUMMY
};
```

```
/*
 * This type defines a node of an expression-tree.
 * If the operator is aOPERAND or aCONSTANT than the left argument is an operand(
 * these are always leave-cells).
 */
struct EXPR {
  int width;                /* Width of expression. */
  enum OPERATOR op;         /* Kind of operator. */
  struct EXPR *rightarg;    /* Right side of expression. */
  boolean brace;            /* Flag if expr. is within braces */
  union
  { struct EXPR *arg;       /* Left side of expression if not an operand or constant. */
    char *operand;          /* Name of operand. */
    NumberPtr constant;     /* Value of constant (as bitstring). */
  } left;
};

struct CTRLCON {
  struct INPUT *ctrlinp;    /* Pointer to input connector that is used as control. */
  struct LIST *fieldspec;   /* This will become a list of struct FIELDSPEC. */
  struct LIST *ctrlspec;    /* This will become a list of struct CTRLSPEC. */
};

/*
 * The fieldspec can be seen as a 'preprocessor' for the values present on the
 * control connector's bus.
 */
struct FIELDSPEC {
  short firstbit;
  short lastbit;
};

struct CTRLSPEC {
  struct LIST *value_mask;  /* This will become a list of Numbers (Bitstrings). */
  struct LIST *commands;    /* This will become a list of lists of commands. */
};


struct STATE {
  char *name;
  struct EXECSTR *execstr;
  struct LIST *assignments; /* This will become a list of struct ASSIGNMENT */
  struct LIST *tmpvars;     /* This will become a list of struct TMPVAR */

      /* the following items are convenient for generating VHDL code */
  char postfix; /* Used as postfix if name is a reserved word */
};

enum EXEC_CMDTYPE {e_schemcmd,e_condblock};

struct EXEC_CMD {
  enum EXEC_CMDTYPE type;
  union
  { struct SCHEM_CMD *schem_cmd;
    struct COND_BLOCK *cond_block;
  } exec;
};

struct EXECSTR {
  struct LIST *exec_cmds;   /* This will become a list of struct EXEC_CMD */
  struct FLOW_CMD *flow_cmd;
};

enum FLOW_CMDTYPE {f_jump,f_call,f_return,f_hold,f_next};
```

```
struct FLOW_CMD {
  enum FLOW_CMDTYPE type;
  char *state;
};

struct SCHEM_CMD {
  struct LIST *blockname;
  struct COMMAND *block_cmd;
};

struct COND_BLOCK {
  struct EXPR *expr;          /* This is the test expression */
  struct LIST *executor;      /* This will become a list of struct EXECUTOR */
};

struct EXECUTOR {
  struct LIST *exectest;      /* This will become a list of Numbers <value/mask> */
  struct LIST *execstr;       /* This will become a list of struct EXECSTR */
  boolean empty_range;        /* Used to indicate if an empty range is present */
};

enum COMMANDTYPE {cmd_reset,cmd_enable,cmd_disable,cmd_normal};

struct COMMAND {
  enum COMMANDTYPE type;
  char *name;
  boolean par_cmd;            /* indicates if command is a parameter command (TRUE) */
  char *parameter;            /* Only valid if par_cmd == TRUE. */
  short firstbit;             /* first bitpos. in ctrl. vector of command code. */
  short lastbit;              /* first bitpos. in ctrl. vector of command code. */
  short code;                 /* The one and only command code. */
};

#endif /* _TYPES_H */
```

APPENDIX 3 STRUCTURAL VHDL OF THE RUNLIGHT

```
LIBRARY compass_lib,ieee;
USE ieee.std_logic_1164.all;
USE compass_lib.compass.all;

ENTITY DISCO IS
 PORT (
   c2 : OUT std_ulogic;
   c3 : OUT std_ulogic;
   out1 : OUT std_ulogic_vector(7 DOWNTO 0);
   c1 : IN std_ulogic;
   reset : IN std_ulogic;
   clk : IN std_ulogic
 );
END DISCO;

ARCHITECTURE structure OF DISCO IS

  COMPONENT DIV
   PORT (
     clk : IN std_ulogic;
     reset : IN std_ulogic;
     o : OUT std_ulogic_vector(4 DOWNTO 0)
   );
  END COMPONENT;

  COMPONENT SHIFT
   PORT (
     clk : IN std_ulogic;
     reset : IN std_ulogic;
     c : IN std_ulogic_vector(4 DOWNTO 0);
     i : IN std_ulogic_vector(7 DOWNTO 0);
     o : OUT std_ulogic_vector(7 DOWNTO 0);
     d2v_SHIFT : OUT std_ulogic_vector(7 DOWNTO 0)
   );
  END COMPONENT;

  COMPONENT NOT1
   PORT (
     i : IN std_ulogic;
     o : OUT std_ulogic
   );
  END COMPONENT;

  COMPONENT CONTROL
   PORT (
     clk : IN std_ulogic;
     reset : IN std_ulogic;
     d2v_SHIFT : IN std_ulogic_vector(7 DOWNTO 0);
     d2v_SHIFTER_ctrl : OUT std_ulogic
   );
  END COMPONENT;

  COMPONENT NOT3
   PORT (
     i : IN std_ulogic;
     o : OUT std_ulogic
   );
  END COMPONENT;
```

```
  COMPONENT NOT2
    PORT (
      i : IN std_ulogic;
      o : OUT std_ulogic
    );
  END COMPONENT;

  COMPONENT SHIFTER
    PORT (
      d2v_SHIFTER_ctrl : IN std_ulogic;
      i : IN std_ulogic_vector(7 DOWNTO 0);
      o : OUT std_ulogic_vector(7 DOWNTO 0)
    );
  END COMPONENT;

  SIGNAL in1                : std_ulogic_vector(7 DOWNTO 0);
  SIGNAL c21                : std_ulogic;
  SIGNAL c4                 : std_ulogic;
  SIGNAL out2               : std_ulogic_vector(7 DOWNTO 0);
  SIGNAL c                  : std_ulogic_vector(4 DOWNTO 0);
  SIGNAL d2v_SHIFT          : std_ulogic_vector(7 DOWNTO 0);
  SIGNAL d2v_SHIFTER_ctrl   : std_ulogic;

BEGIN

  i_DIV       : DIV PORT MAP (clk,reset,c);

  i_SHIFT     : SHIFT PORT MAP (clk,reset,c,in1,out2,d2v_SHIFT);

  i_NOT1      : NOT1 PORT MAP (c1,c4);

  i_CONTROL   : CONTROL PORT MAP (clk,reset,d2v_SHIFT,d2v_SHIFTER_ctrl);

  i_NOT3      : NOT3 PORT MAP (c21,c3);

  i_NOT2      : NOT2 PORT MAP (c4,c21);

  i_SHIFTER   : SHIFTER PORT MAP (d2v_SHIFTER_ctrl,out2,in1);

  c2    <= c21;
  out1  <= out2;

END structure;
```

APPENDIX 4 RESERVED WORDS

abs
access
after
alias
all
and
architecture
array
assert
attribute
begin
block
body
buffer
bus
case
component
configuration
constant
disconnect
downto
else
elsif
end
entity
exit
file
for
function
generate
generic
guarded
if
in
inout
is
label
library
linkage
loop
map
mod
nand
new
next
nor
not
null
of
on
open
or
others
out
package
port
procedure
process
range
record
register
rem
report
return
select
severity
signal
subtype
then
to
transport
type
units
until
use
variable
wait
when
while
with
xor

APPENDIX 5 VHDL FUNCTIONS FOR COMPLEX IDASS OPERATORS

IDaSS operator	C code representation	VHDL function
at:	kAT	FUNCTION 8at3 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN std_ulogic_vector IS -- *The function name exists of the width of the receiver, the string "at" and the wanted position (prepended with a function prefix of course).* BEGIN RETURN rec(3); END 8at3; *This at-function is only written if the receiver is an expression, all other cases are written at once in the operator's process.*

at:width:	kATWIDTH	FUNCTION atwidth8 (rec : std_ulogic_vector(7 DOWNTO 0):at : integer; width : integer) RETURN std_ulogic_vector IS
		-- *The function name exists of the string "atwidth" and the receiver width.*
		VARIABLE temp1 : std_ulogic_vector(7 DOWNTO 0);
		VARIABLE temp2 : integer;
		BEGIN
		temp1 := SHR("11111111",(8 - width)); -- *8 is the rec'length*
		temp2 := to_integer(SHR(rec,at) AND temp1);
		RETURN (to_stdulogicvector(itobv(temp2,width)));
		END atwidth8;

from:to:	kFROMTO	FUNCTION 8from2to6 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN std_ulogic_vector IS -- *The function name exists of the receiver width, the string "from", the value of from, the string "to" and the value of to.* BEGIN RETURN rec(to DOWNTO from); -- *with to and from the wanted positions* END 8from2to6; *This from:to:-function is only written if the receiver is an expression all other cases are written at once in the operator's process.*

lsomask uLSOMASK

```
FUNCTION lsomask8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                    std_ulogic_vector IS

-- The function name exists of the string "lsomask" and the receiver width.

  VARIABLE temp : integer;
  VARIABLE mask : std_ulogic_vector(7 DOWNTO 0);

BEGIN
  mask := to_stdulogicvector(itobv(1,rec'length));
  temp := rec'length;
  FOR i IN (rec'length - 1) DOWNTO 0 LOOP
    IF rec(i) = '1' THEN
      temp := i;
    END IF;
  END LOOP;
  RETURN SHL(mask,temp);
END lsomask8;
```

lsone uLSONE

```
FUNCTION lsone8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                    std_ulogic_vector IS

-- The function name exists of the string "lsone" and the receiver width.

  VARIABLE temp : std_ulogic_vector(7 DOWNTO 0);

BEGIN
  temp := to_stdulogicvector(itobv(rec'length,rec'length));
  FOR i IN (rec'length - 1) DOWNTO 0 LOOP
    IF rec(i) = '1' THEN
      temp := to_stdulogicvector(itobv(i,rec'length));
    END IF;
  END LOOP;
  RETURN temp;
END lsone8;
```

lszero uLSZERO

```
FUNCTION lszero8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                    std_ulogic_vector IS

-- The function name exists of the string "lsone" and the receiver width.

  VARIABLE temp : std_ulogic_vector(7 DOWNTO 0);

BEGIN
  temp := to_stdulogicvector(itobv(rec'length,rec'length));
  FOR i IN (rec'length - 1) DOWNTO 0 LOOP
    IF rec(i) = '0' THEN
      temp := to_stdulogicvector(itobv(i,rec'length));
    END IF;
  END LOOP;
  RETURN temp;
END lszero8;
```

lszmask uLSZMASK

```
FUNCTION lszmask8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                        std_ulogic_vector IS

-- The function name exists of the string "lszmask" and the receiver width.

   VARIABLE temp : integer;
   VARIABLE mask : std_ulogic_vector(7 DOWNTO 0);

BEGIN
   mask := to_stdulogicvector(itobv(1,rec'length));
   temp := rec'length;
   FOR i IN (rec'length - 1) DOWNTO 0 LOOP
     IF rec(i) = '0' THEN
        temp := i;
     END IF;
   END LOOP;
   RETURN SHL(mask,temp);
END lszmask8;
```

maj (even receiver width) uMAJ

```
FUNCTION maj8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                        std_ulogic_vector IS

-- The function name exists of the string "maj" and the receiver width.

   VARIABLE temp : integer;

BEGIN
   temp := 0;
   FOR i IN 0 TO (rec'length -1) LOOP
     IF rec(i) = '1' THEN
        temp := temp + 1;
     END IF;
   END LOOP;
   IF temp < (rec'length / 2) THEN
     RETURN "01";
   ELSE IF temp = (rec'length / 2) THEN
     RETURN "00";
   ELSE
     RETURN "10";
   END IF;
   END IF;
END maj8;
```

maj (odd receiver width) uMAJ

```
FUNCTION maj7 (rec : std_ulogic_vector(6 DOWNTO 0)) RETURN
                                                        std_ulogic IS

-- The function name exists of the string "maj" and the receiver width.

   VARIABLE temp : integer;

BEGIN
   temp := 0;
   FOR i IN 0 TO (rec'length -1) LOOP
     IF rec(i) = '1' THEN
        temp := temp + 1;
     END IF;
   END LOOP;
   IF temp < (rec'length / 2) THEN
     RETURN '0';
   ELSE
     RETURN '1';
   END IF;
END maj7;
```

merge:from:to: kMERGEFROMTO

```
FUNCTION mergefromto8_8 (rec1 : std_ulogic_vector(7 DOWNTO 0);
                         rec2 : std_ulogic_vector(7 DOWNTO 0); from : integer; to : integer)
                                                                     RETURN std_ulogic_vector IS

-- The function name exists of the string "mergefromto", the width of the first
   receiver and the width of the second receiver.

   VARIABLE temp1 : std_ulogic_vector(7 DOWNTO 0);

BEGIN
   temp := "11111111";
   IF from <= to THEN
      RETURN ((temp XOR (SHR((SHL(temp,(rec'length+from-to-1)) AND
                (NOT rec1)),from))) AND rec1);
   ELSE IF from > to THEN
      RETURN ((temp XOR ((SHL((SHR(temp,(to-from+1)) AND (NOT rec2)),from))
                OR(SHR((SHR(temp,(to-from+1))AND(NOT rec2)),(rec1'length-a)))))
                AND rec1);
   END IF;
   END IF;
END mergefromto8_8;
```

msomask uMSOMASK

```
FUNCTION msomask8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                              std_ulogic_vector IS

-- The function name exists of the string "msomask" and the receiver width.

   VARIABLE temp : integer;
   VARIABLE mask : std_ulogic_vector(7 DOWNTO 0);

BEGIN
   mask := to_stdulogicvector(itobv(1,rec'length));
   temp := rec'length;
   FOR i IN 0 TO (rec'length - 1) LOOP
      IF rec(i) = '1' THEN
         temp := i;
      END IF;
   END LOOP;
   RETURN SHL(mask,temp);
END msomask8;
```

msone uMSONE

```
FUNCTION msone8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                              std_ulogic_vector IS

-- The function name exists of the string "msomask" and the receiver width.

   VARIABLE temp : std_ulogic_vector(7 DOWNTO 0);

BEGIN
   temp := to_stdulogicvector(itobv(rec'length,rec'length));
   FOR i IN 0 TO (rec'length - 1) LOOP
      IF rec(i) = '1' THEN
         temp := to_stdulogicvector(itobv(i,rec'length));
      END IF;
   END LOOP;
   RETURN temp;
END msone8;
```

mszero uMSZERO

```
FUNCTION mszero8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                          std_ulogic_vector IS

-- The function name exists of the string "mszero" and the receiver width.

  VARIABLE temp : std_ulogic_vector(7 DOWNTO 0);

BEGIN
  temp := to_stdulogicvector(itobv(rec'length,rec'length));
  FOR i IN 0 TO (rec'length - 1) LOOP
    IF rec(i) = '0' THEN
      temp := to_stdulogicvector(itobv(i,rec'length));
    END IF;
  END LOOP;
  RETURN temp;
END mszero8;
```

mszmask uMSZMASK

```
FUNCTION mszmask8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                          std_ulogic_vector IS

-- The function name exists of the string "mszmask" and the receiver width.

  VARIABLE temp : integer;
  VARIABLE mask : std_ulogic_vector(7 DOWNTO 0);

BEGIN
  mask := to_stdulogicvector(itobv(1,rec'length));
  temp := rec'length;
  FOR i IN 0 TO (rec'length - 1) LOOP
    IF rec(i) = '0' THEN
      temp := i;
    END IF;
  END LOOP;
  RETURN SHL(mask,temp);
END mszmask8;
```

onecnt uONECNT

```
FUNCTION onecnt8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                          std_ulogic_vector IS

-- The function name exists of the string "mszmask" and the receiver width.

  VARIABLE temp : integer;

BEGIN
  temp := 0;
  FOR i IN 0 TO (rec'length - 1) LOOP
    IF rec(i) = '1' THEN
      temp := temp + 1;
    END IF;
  END LOOP;
  RETURN to_stdulogicvector(itobv(temp,rec'length));
END onecnt8;
```

rev uREV

```
FUNCTION rev8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                    std_ulogic_vector IS

-- The function name exists of the string "mszmask" and the receiver width.

   VARIABLE temp : std_ulogic_vector(7 DOWNTO 0);

BEGIN
   FOR i IN 0 TO (rec'length - 1) LOOP
      temp(i) := rec(rec'length - 1 - i);
   END LOOP;
   RETURN temp;
END rev8;
```

sar: kSAR

```
FUNCTION sar8 (rec : std_ulogic_vector(7 DOWNTO 0); a : integer) RETURN
                                                    std_ulogic_vector IS

-- The function name exists of the string "mszmask" and the receiver width.

   VARIABLE temp : std_ulogic_vector(7 DOWNTO 0);

BEGIN
   temp := "11111111";
   IF rec(rec'length - 1) = '1' THEN
      RETURN SHR(rec,a);
   ELSE
      RETURN (SHR(rec,a) OR SHL(temp,(rec'length - a)));
   END IF;
END sar8;
```

signed: kSIGNED

```
FUNCTION signed8_11 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                    std_ulogic_vector IS

-- The function name exists of the string "signed", the receiver width and the
   resulting width.

   VARIABLE temp : std_ulogic;

BEGIN
   temp := rec(7); -- most significant bit
   RETURN (temp & temp & temp & rec);
END signed8_11;
```

sol: kSOL

```
FUNCTION sol8 (rec : std_ulogic_vector(7 DOWNTO 0); a : integer) RETURN
                                                    std_ulogic_vector IS

-- The function name exists of the string "sol" and the receiver width.

   VARIABLE temp : std_ulogic_vector(7 DOWNTO 0);

BEGIN
   temp := "11111111";
   RETURN (SHL(rec,a) OR SHR(temp,(rec'length - a)));
END sol8;
```

sor: kSOR

```
FUNCTION sor8 (rec : std_ulogic_vector(7 DOWNTO 0); a : integer) RETURN
                                                          std_ulogic_vector IS

-- The function name exists of the string "sol" and the receiver width.

   VARIABLE temp : std_ulogic_vector(7 DOWNTO 0);

BEGIN
   temp := "11111111";
   RETURN (SHR(rec,a) OR SHL(temp,(rec'length - a)));
END sor8;
```

width: (width:3) kWIDTH

```
FUNCTION width3 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                          std_ulogic_vector IS

-- The function name exists of the string "width" followed by the resulting width.

BEGIN
   RETURN rec(2 DOWNTO 0);
END width3;
```

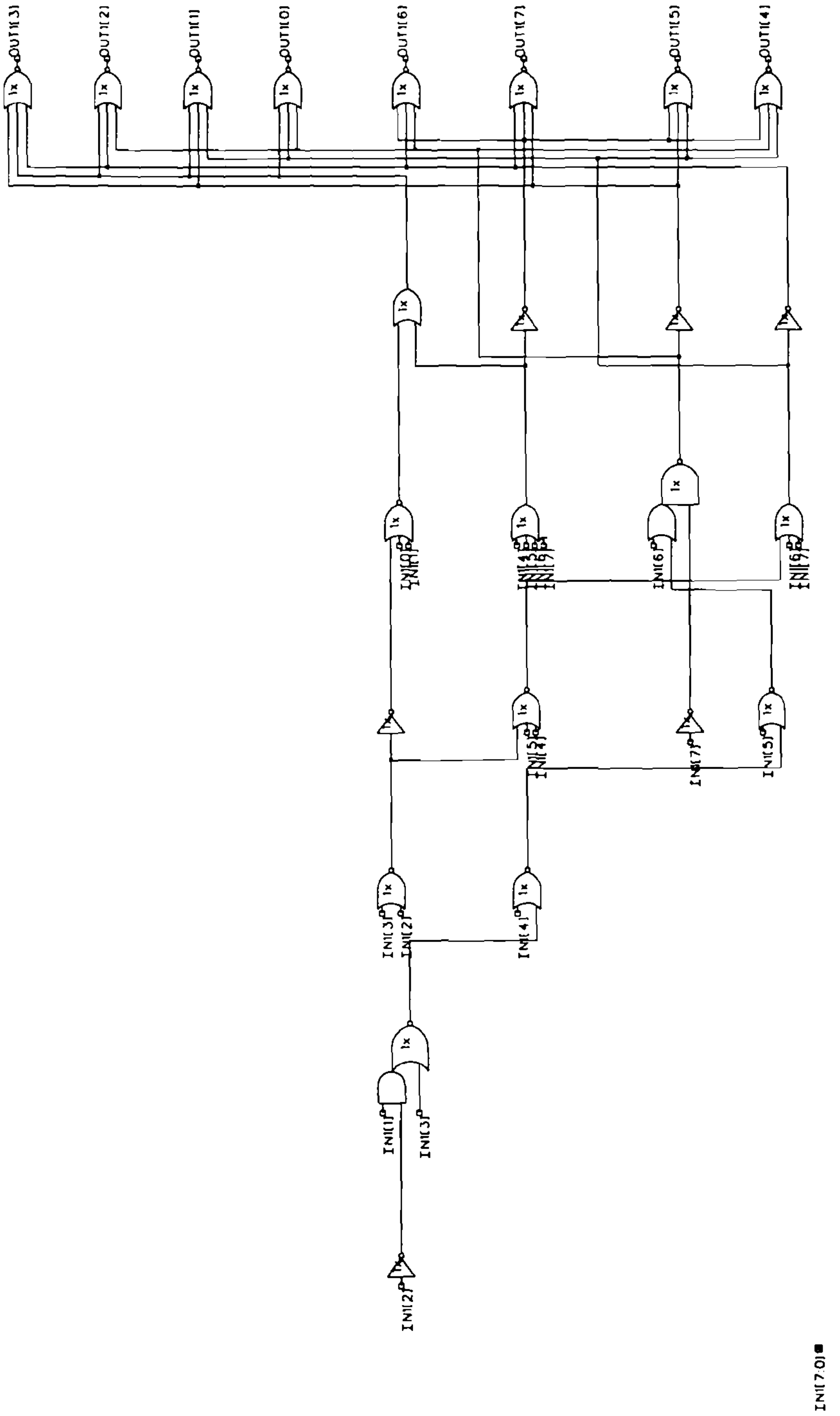
width: (width:12) kWIDTH

```
FUNCTION width12 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                          std_ulogic_vector IS

-- The function name exists of the string "width" followed by the resulting width.

BEGIN
   RETURN "0000" & rec;
END width12;
```

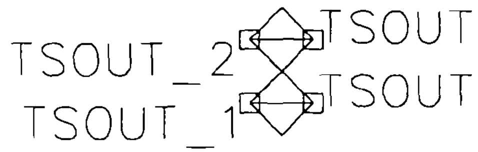
zerocnt uZEROCNT

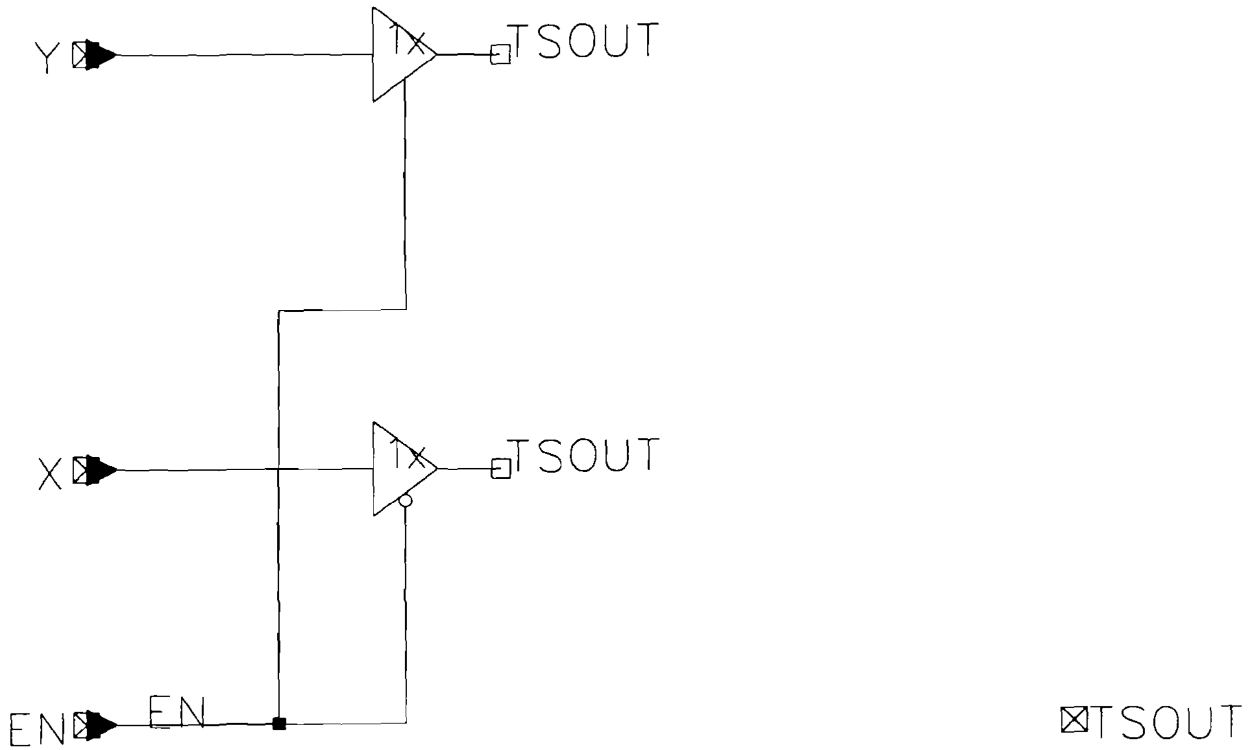
```
FUNCTION zerocnt8 (rec : std_ulogic_vector(7 DOWNTO 0)) RETURN
                                                          std_ulogic_vector IS

-- The function name exists of the string "width" followed by the resulting width.

   VARIABLE temp : integer;

BEGIN
   temp := 0;
   FOR i IN 0 TO (rec'length - 1) LOOP
     IF rec(i) = '0' THEN
       temp := temp + 1;
     END IF;
   END LOOP;
   RETURN to_stdulogicvector(itobv(temp,rec'length));
END zerocnt8;
```

APPENDIX 6 8 BIT MSOMASK

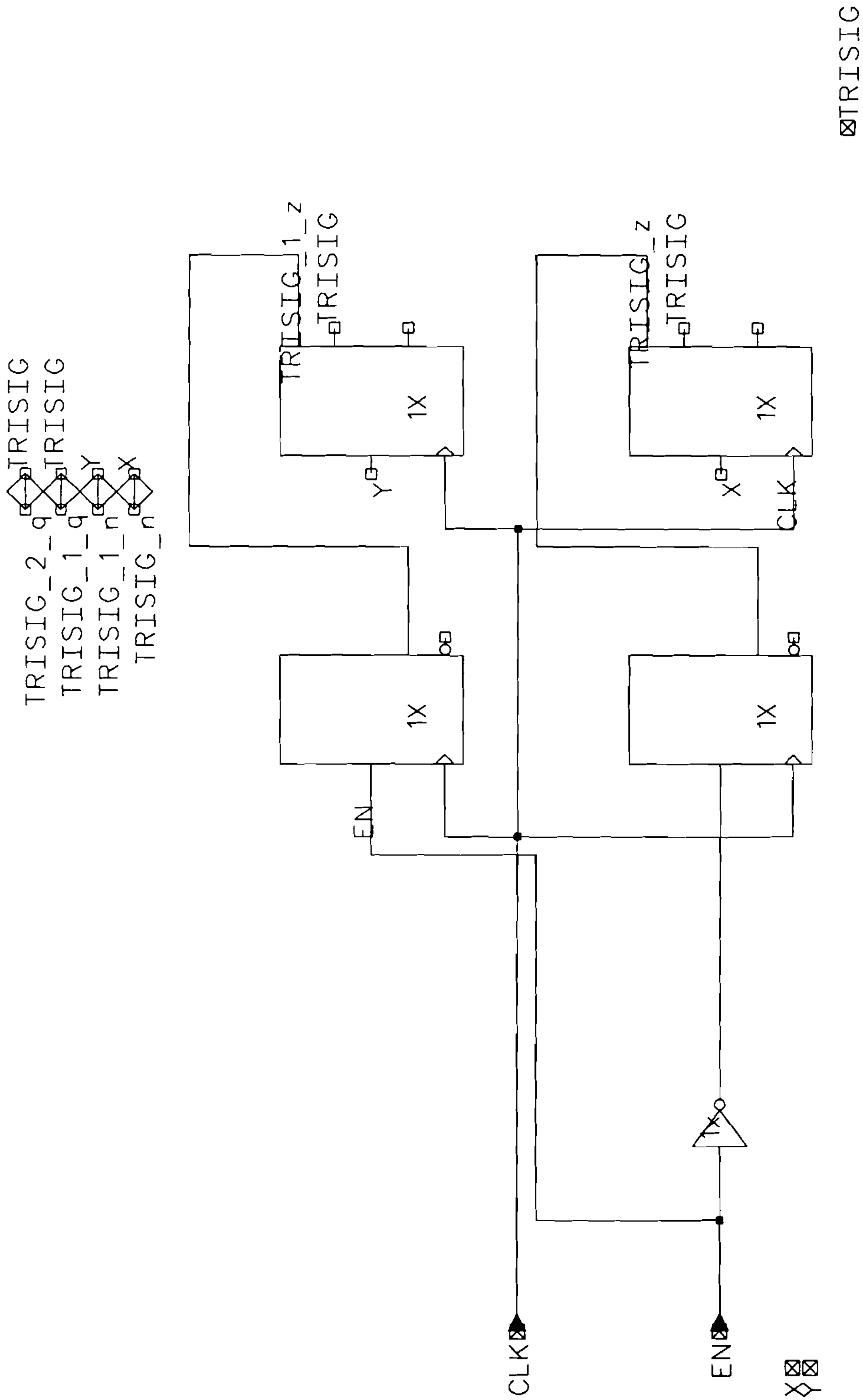


APPENDIX 7 FIRST POSSIBILITY FOR A THREE STATE BUS

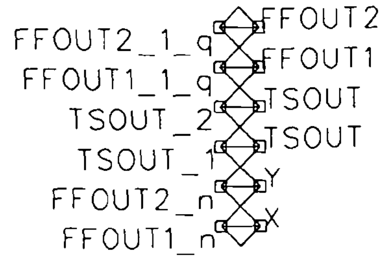


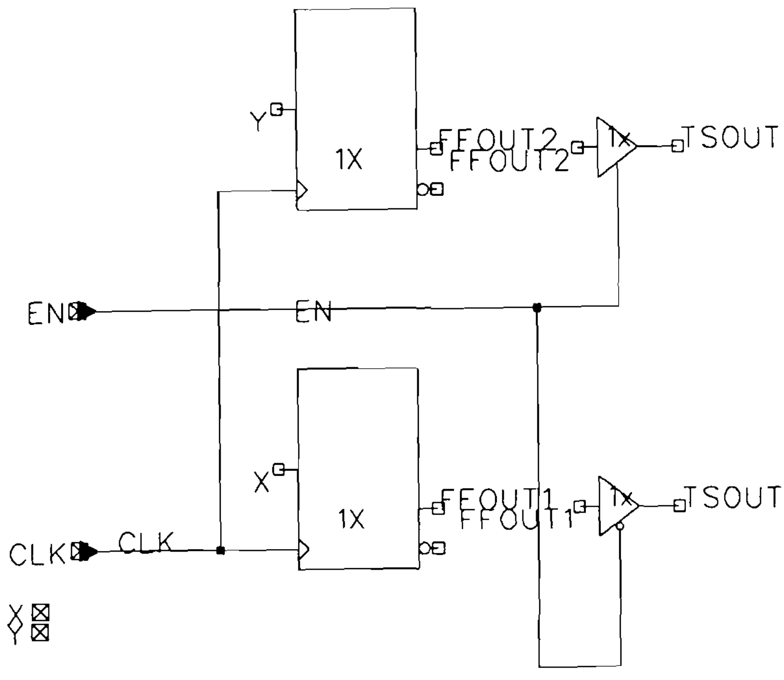


APPENDIX 8 SECOND POSSIBILITY FOR A THREE STATE BUS



APPENDIX 9 THIRD POSSIBILITY FOR A THREE STATE BUS





TSOUT

APPENDIX 10 SWITCHES OF THE FILE USER_DEF WITH THEIR DEFAULT SETTINGS

Switches of integer variables	default value
INDENT	2
USR_MAX_IDENT_LEN	32

Switches of strings variables	default value
INST_PREFIX	i_
PREFIX_SEP	_
FILE_PREFIX	beh
DIR_SEP	/
FILE_POSTFIX	.vhd
MUXVAR_NAME	_if
NAME_PREFIX	d2v
FUNC_PREFIX	d2vf
CTRL_APPEND	ctrl
REGISTERTEMP	registertemp
CLK_NAME	clk
RESET_NAME	reset
STATE_T_NAME	statetype
CUR_STATE	current_state
NEXT_STATE	next_state
JUMP_BOOL	jump
EMPTY_STATE_NAME	empty_name
ULOGIC_BIT_TYPE	std_ulogic
ULOGIC_VECTOR_TYPE	std_ulogic_vector
LOGIC_BIT_TYPE	std_logic
LOGIC_VECTOR_TYPE	std_logic_vector
BIT2ULOGIC_BIT	to_stdulogic
TO_ULOGIC_TYPE	to_stdulogicvector
BIT2LOGIC_BIT	to_stdlogic
TO_LOGIC_TYPE	to_stdlogicvector
BOOLEAN2BIT	to_bit
LOGIC_TYPE2INT	to_integer
INT2BITVECTOR	itobv

Switches of multiple string variables	default value
LIBRARY_HEADER	"LIBRARY compass_lib,ieee;\n"
USE_CLAUSES	"USE ieee.std_logic_1164.all;\n USE compass_lib.compass.all;\n"
FILE_HEADER	"\n"