

MASTER
Template file for IDaSS to HDL-Verilog generatio
Lin, X.
Award date: 1997

Link to publication

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.

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



Faculty of Electrical Engineering Section of Information and Communication Systems

Master's Thesis:

### Template file for IDaSS to HDL-Verilog generation

X. Lin

Coach : Dr. Ir. A.C. Verschueren Supervisor : Prof. Ir. M.P.J. Stevens

Period: March 1997 - October 1997

### Preface

This report contains the result of my graduation project carried out at the section Information and Communication Systems, Faculty of Electrical Engineering of the Eindhoven University of Technology.

I would like to thank Prof. Ir. M.P.J. Stevens for giving me the opportunity to do my graduation project in the section Information and Communication Systems, my coach Dr.Ir. A.C. Verschueren for his guidance. Last but not least, I would like to thank my fellow students, and the other members of the section who made my project not only instructive, but also very pleasant.

### Abstract

This report describes an implementation of a converter for IDaSS (Interactive Design and Simulation System) to Hardware Description Language Verilog. With IDaSS a digital system can be designed and simulated interactively at Register Transfer Level or higher level languages. With a Hardware Description Language, a real chip layout of the digital system can be generated.

The converter consists of Verilog language optimized conversion instructions. The file generated by the converter will be the input file for a Verilog simulator or silicon compiler. The latter can generate files for manufacturing chip.

The complete IDaSS system will consist of several interconnected tools. Different tools have been implemented successfully. The implementation details of the Verilog converter, Expressions, Unary Operators and Binary Operators are the main subject of this report.

### **Contents**

Chapter 1	Introduction				
-	1.1 IDaSS tools	4			
	1.2 IDaSS converters	6			
Chapter 2	Hardware Description Languages and Simulators				
	2.1 Introduction HDLs	8			
	2.2 HDL simulators	9			
Chapter 3	VHDL and HDL-Verilog	10			
_	3.1 VHDL	10			
	3.2 HDL-Verilog	12			
Chapter 4	HDL-Verilog implementation				
	4.1 The layout of the template	14			
	4.1.1 Introduction template	14			
	4.1.2 The template sections	16			
	4.1.3 Expression optimization	17			
	4.2 Precedence level	18			
	4.3 Functions for Unary Operators	20			
	4.4 The Binary Operators	23			
Chapter 5	Instructions for the target language file generation	on 26			
Chapter 6	Conclusions and recommendations	27			
References		28			
Appendix 1	Verilog precedence levels	31			
Appendix 2	Functions in Verilog code	32			
Appendix 3	Simulation and Implementation of Signed Multiple				
Appendix 4	Example of Verilog code for a shiftregister	46 48			
Appendix 5	5 The template file				

### Chapter 1

### Introduction

The Interactive Design and Simulation System (IDaSS) is a research project of the section Information and Communication Systems (ICS) of the Faculty of Electrical Engineering of the Eindhoven University of Technology. IDaSS is a graphics and text based editing system to design and simulate digital systems at Register Transfer Level (RTL) or higher level languages. IDaSS is built in the Smalltalk environment.

#### 1.1 IDaSS tools

The complete system will consist of several interconnected tools. In *figure 1*, *IDaSS Help File* contains the online help for the digital system designer. In a later version, online help for the template file writer will be implemented. The template help file *template.txt* will be converted into IDaSS help file.

IDaSS image file is an important file for a digital system designer. When IDaSS is started for the first time, the technology file and the IDaSS help file will be automatically attached to the image file. During a digital system design, a snapshot of the design can be saved as an image file. It contains all parameters at the moment you save the image. The next time, when the IDaSS system is started, this image will be loaded and the IDaSS design session can be continued.

The *Technology File* describes IDaSS technology parameters: delays approximation, definitions for RAM's, ROM's and other memories. The *Log File Description* is a test vector file generator. It will generate a complete test environment from within IDaSS containing test vector resulting from a simulation.

The Compass Template File tells IDaSS how to convert digital system design into Compass compatible VHDL. In a later IDaSS version, the Verilog Template File will be added to the system. It will enable conversion into Verilog. VHDL and Verilog are Hardware Description Languages (HDLs). In this way, the template files allow translation of IDaSS designs into the languages which are suitable for commercial silicon compilers.

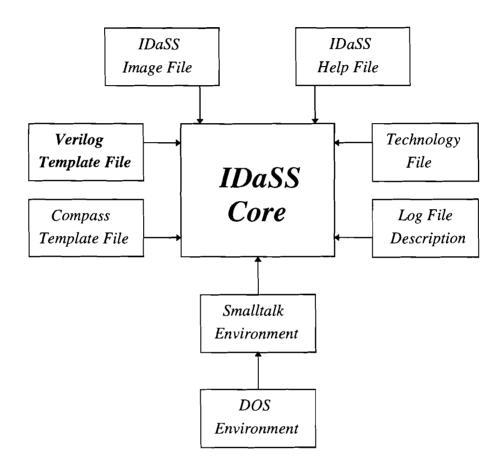


Figure 1. The IDaSS software package

Many digital systems have been designed and simulated successfully with IDaSS tools. Microprocessors, PCM telephone exchange switching matrix and Scaleable 'Batcher-Banyan' ATM switching matrix are designed. The IDaSS software package can be downloaded from the website of the section ICS. (http://www.eb.ele.tue.nl/proj/idassfly.html).

#### 1.2 IDaSS converters

VHDL and Verilog are chosen as the target languages for the IDaSS converters. The converters will generate optimized files for different simulation and synthesis tools. After this conversion, a synthesis tools will be able to generate the netlist of the original digital system designed in IDaSS. Finally, the IC manufacture will be able to produce the chips in cleanrooms.

The conversion control files are named Alien File Templates (AFT). The AFT for IDaSS to Compass compatible VHDL is implemented. The AFT for IDaSS to Verilog, containing the conversion of IDaSS Unary Operators and Binary Operators is the main subject of this report.

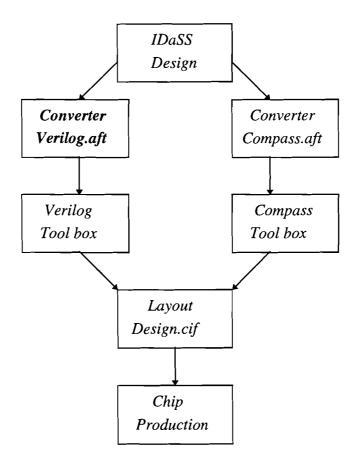


Figure 2. Steps from the IDaSS digital system design to chip production

In *figure 2*, the blocks describe the steps from an IDaSS digital system design to chip production. The blocks are:

◆ IDaSS design
 + Verilog.aft
 : the digital systems designed in IDaSS
 • the converter, the subject of this report

◆ Verilog Tool box : tools to generate design.cif file

◆ Compass.aft : the converter for Compass compatible VHDL

◆ Compass Tool box : tools to generate design.cif file
 ◆ Design.cif : the file for the IC manufacture

In the next chapter, starting with an overview of different HDLs, and then a comparison between VHDL en Verilog. After this, the implementation of the template file will be followed. Conclusions and recommendations will be given.

### Chapter 2

# Hardware Description Languages and Simulators

#### 2.1 Introduction HDLs

There are many Hardware Description Languages and Simulators on the market. Some HDLs are public domain languages. Four of these HDLs which will be described here are:

- ♦ VHDL
- ♦ HDL-Verilog
- M
- ♦ UDL/I

VHDL (VHSIC-HDL) was started in 1981, based on the United States Department of Defense's Very High Speed Integrated Circuit (VHSIC) program. It was developed by IBM and Texas Instruments in 1983. The first version of VHDL was released in 1985. It was standardized by IEEE in 1987 and was updated in 1993. Today it is known as IEEE standard 1076. It is a public domain language.

HDL-Verilog was launched by Gateway in 1983. It has been used extensively since then. After *Cadence Design Systems, Inc.* bought Gateway in 1989, verilog has been used as the language of Cadence Verilog-XL simulator. Verilog became a public domain language in 1990 and *IEEE standard 1364* in 1995. Currently many Verilog-XL simulator licenses have been sold. Universities in the US and elsewhere teach and research Verilog. By the way, a Verilog simulator of the Wellspring Solutions, *Veriwell*, can be download from the website: *http://www.wellspring.com* 

M is the language of the Lsim simulation system which is developed by Silicon Design Labs, then by Silicon Compiler Systems, later merged with Mentor Graphics. A lot of models and libraries have been written in this language. M is not a public domain language.

UDL/I (Unified Design Language for Integrated Circuits) was started in the Japanese LSI-Design Language Standardization Project in 1987. The purpose of UDL/I is for VLSI designs to be compatible among semiconductor manufacturers, chip user, and design centers.

VHDL and HDL-Verilog are public domain languages and are used worldwide. They are industry-standard HDLs for chip design. They have been chosen as the target language of the IDaSS converters.

#### 2.2 HDL Simulators

There are various Simulators for Hardware Description Languages. For the same HDL, many Logic Simulators with different performance and cost are available. Examples of HDL Simulators are:

- ♦ VeriBest (http://www.veribest.com)
- ♦ FinSim
- ◆ PureSpeed
- ◆ SILOS-3 (http://www.simucad.com)
- ◆ VeriWell (http://www.wellspring.com)
- ♦ Viper

VeriBest VHDL is a high performance simulation system for ASICs. *SILOS-3* simulation environment supports the HDL-Verilog for simulation at different levels of design abstraction with a good performance. The free version of *Veriwell* can compile a limited lines of Verilog instructions. In the future, maybe, a powerfull simulator is needed to simulate a converted complex digital system design in IDaSS.

### Chapter 3

### VHDL and HDL-Verilog

#### 3.1 VHDL

In VHDL an entity is a substructure of a design and can be compiled into a working library. An entity has two parts:

- ♦ An entity declaration which specifies the interface between a design entity and the outside word.
- An architecture body which defines the function of a design entity.

All process statements within an architecture body are concurrent. A process statement defines an independent sequential behavior of a part of the design.

There are three levels of abstraction for describing digital systems in an architecture body:

- ♦ Behavioral-level (algorithmic level)
- ◆ Data-flow-level (RTL level)
- ♦ Structural-level (netlist level)

There are two basic data types: scalar and composite. Scalar types are: integer, floating point, Boolean, bit, character and (physical) time. Composites are constructed form scalar types. The user can also define own data types.

Compass (V8R4.7.0) compatible VHDL is used in our University to generate the real chip layout.

#### To Compare the structure of VHDL with Verilog, this is an example of VHDL code:

```
-- This is the comment of the VHDL code
LIBRARY
                                         -- The library described here will be used.
ENTITY MyDesign IS
                                         -- Start with entity declaration which specifies the interface
        PORT (CLK: IN DataType;
                                         -- Other I/O connections of MyDesign
                DATA: OUT DataType);
END MyDesign;
ARCHITECTURE ... OF MyDesign IS
                                         -- The function of the entity MyDesign
                                         -- Behavior or structure of MyDesign
BEGIN
                                         -- State, logic...
PROCESS (CLK)
VARIABLE var_name : DataType;
                                         -- Other variables
BEGIN
                                         -- This can be a statement
        IF (CLK'event) and (CLK = '1')
                                         -- Execute this if the conditions is true
        END IF;
END PROCESS ...
                                         -- This is the end of the process ...
END ...; -- OF MyDesign
                                         -- This is the end of MyDesign
-- The hierarchical perspective is:
                                         -- This is the CONFIGURATION
-- This is the end of VHDL sample code
```

#### 3.2 HDL-Verilog

In Verilog, a set of modules are used to describe a digital system. The modules are reusable as a component and have an I/O interface with other modules. The description of the function of the module can be structural, behavioral, or a mix. A module in Verilog is equivalent to the combination of entity declaration and architecture body of VHDL.

Processes are used to build concurrency in Verilog. With the keyword *always*, a process can be designed. This process will continuously repeat itself. Using the keyword *fork-join*, a concurrency within a processes can be constructed.

Four levels for describing digital systems inside the module are:

- ◆ Algorithmic-level (algorithm in high-level language)
- ♦ RTL-level (data flow between registers)
- ♦ Gate-level (logic gates and interconnections)
- ♦ Switch-level (transistors and interconnections)

In a module a function (between the keywords *function* and *endfuction*) can be defined. A function is a logically connected piece of program, which returns exactly one value by its name. A function will be executed without delay. In a module a task (between the keywords *task* and *endtask*) can be defined. A task is a piece of program with time controls.

Verilog has easy to use data types. With the keyword *reg* the register data type can be defined. Another net data type is the data type *wire* which connects the signal of the different modules. Data between the modules are connected with the interface type *input*, *output* and *inout*.

#### This is a example of the Verilog code:

```
// This is the comment of the Verilog code
module NameOfSubmodule;
                                            //This is the submodule
                                            //The statements of the submodule
endmodule
                                            //The end of the submodule
module MyDesign;
                                            // This is the start of the main module
                                            // The interface specification
input
        [ k:0 ]
                           Var;
output [m:0]
                          Data;
inout
        [ n:0 ]
                          Bidir;
                                            //Integer k,m,.n specify the range in bits
                          Temp_var;
        [p:0]
reg
wire
        [ q:0 ]
                          CLK;
                                            //Other declarations
function Fx;
                                            // A function to compute a result from its arguments without delay
                                            // Input and output variables of the function can be defined here
begin
         if ( Temp_var )
                                            // If Temp_var has at least one bit equals 1 then
                  begin
                                            // Instructions will be executed
                  end
                                            //Other statements in the function block
end
endfunction
                                            // This is the end of the function Fx
task Tx;
                                            //Task with logically connected program may contain time control
                                            // Input and output variables of the task can be defined here
always @ (negedge CLK)
                                            //Always execute the statement each time
                                            //when CLK signal falling
                                            //@ waits for an event to occur
         begin
                                            //Here come the instructions
         ...;
         end
                                             //Other statements in the task block
endtask
                                             //This is the end of the task Tx
                                             //Other statements in the main module
always @ (posedge CLK)
                                            //Always execute the statement each time
                                             //when CLK signal rising
                                            //@ waits for an event to occur
begin
                                             //Here comes the instructions
         ...;
end
initial
                                            //Initial will execute the statement exactly once
begin
         CLK = 0; #1;
                                             //CLK is low and wait for 1 unit of time
         CLK = 1;#1;
                                             //Now rise the CLK so that the always @ block will be executed
         $finish
                                            //This is the end of The instructions
end
endmodule
                                             //This is the end of MyDesign
```

### Chapter 4

### HDL-Verilog implementation

The Alien File Template-Verilog (AFT-Verilog), is a text file. It contains instructions to convert each of the IDaSS constructs and operators, It will generate Verilog code directly from IDaSS. In this chapter the details of the implementation will be described.

#### 4.1 The layout of the template

#### 4.1.1 Introduction

The template file contains all conversion rules from the source to the target. The source is IDaSS and the target is Verilog.

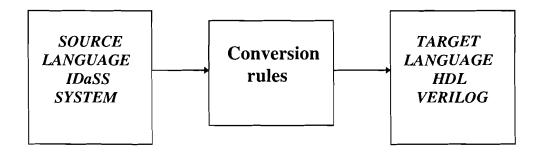


Figure 3. The conversion rules from source to target

In the template file the comment will be given after a double quote. The template file ends with the last line which is started with a '.'. If words or expressions have to be written to the output file, a single quote will be used. With the command 'cr', a line feed will be inserted. Together with the combination of the 'markindent' and 'exitindent' the desired layout of the output file can be generated.

The template has many sections. Each of those sections, we handled starts with '#' which will be followed by an identifier: Expression, UnaryOp and BinaryOp. Those names indicate respectively the sections which handle:

- ♦ Expressions
- ♦ Unary Operators
- ♦ Binary Operators

The sections with the name Expression have many subsections. The combination of the names will describe the function of the subsections.

The names of the subsection of the Expression are:

conversion the conversion between data types
 typing the indication for different IDaSS values

♦ raiseprecedence to place braces

♦ root to generate the root of an expression

The sections with the name UnaryOp will convert all IDaSS Unary Operators. Each unary operator has one source value type and the result value type.

The sections with the name BinaryOp will convert all IDaSS Binary Operators. A binary operator has the left expression part and the right expression part. It has two source value types and the result value type.

To generate the output file, passes are defined. Pass 0 is an initial pass, it is used to generate the lists of separate piece of text to be inserted in backwards order, temporary variables and functions for later passes. Pass 0 is also used to start the file generation by calling the root subsection. For the file generation, the current 'pass' is indicated with the 'generate X' with an integer 'X'.

#### 4.1.2 The template sections

In a section, different names are used as a subsection markers. After a subsection marker, the contents of a subsection are defined.

'Source' and 'Result' are subsection markers. The source and result value types are: constant, Reg\_bit and reg\_bit\_vector. The constant has no bit width. A conversion must take place before a bits operator can be used. Reg\_bit contains one bit. Reg\_bit\_vector contains at least 2 bits. Reg\_bit\_c and reg\_vector\_c are the complement types.

'inline code' can have different meaning in different sections. In the binary operator sections, an operator has two operands. The 'par' code indicates which part of the operand and what is the precedence level of the section. The left part of the expression of a binary operator has the index 1. The right part has the index 2.

For example: par 1 5 means the left part expression of a binary operator and this section has the precedence level 5. The Verilog precedence level are ordered from low to high. It will be described late in this chapter. The list of the precedence level is included in *Appendix 1*.

'guard' creates the condition for the execution of the section. If the expression of the guard is not zero, then this section can be executed.

'root' generates the root of the expression. For example, Output <= InputA + InputB is an IDaSS expression with root. It has two parts, the first part is the target assignment 'Output <=', the second part 'InputA + InputB' is the top node of the expression with the '+' operator. There are also expressions with no root: For example in: 'IF InputA > InputB THEN instructions ...'. In this expression there is no target assignment.

#### 4.1.3 Expression optimization

Optimal conversion means the reduction of the sub-expression tree. The expressions are generated with different operations. The combinations of operators can generate different expressions with the same result. Reducing the expression tree can be done by forward operations. 'forward' removes the sub-expression tree which do not generate output code. For example:

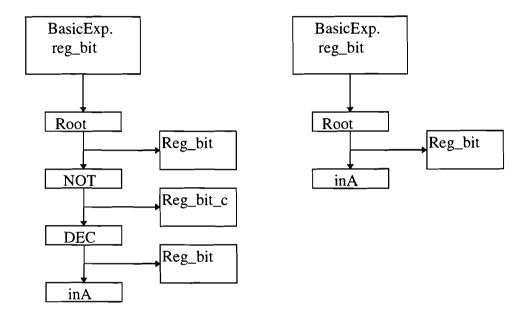


Figure 4. The reduction of the sub-expression tree

In this example, the 'NOT' operator generates the complement type of the variable with one bit width. The 'DEC' operator decrements a value which for 1 bit is a complement of that bit. The result of the two operations is not changed. The result expression tree will be shorter.

Optimal conversion also means the reuse of the same construction in the different level of the design. The different levels of the design can contain sub-level schematics. These schematics may have the same IDaSS data structure. A link is made between IDaSS data structure and the translated data objects.

The specific description about the sections will be found in the IDaSS manual file *template.txt*.

#### 4.2 Precedence level

The *inline X* indicates the precedence level of the section. The precedence level of the subexpressions of the binary operator is given with 'par X Y'. With X is integer 1 or 2 depend on left or right part of the expression of the operator and Y the precedence level.

IDaSS operators have different precedence levels compared to Verilog operators. An expression will be scanned from left to right, and it will be executed in this order, if there are no braces in it. If the braces are added to one part of the expression, this part of the expression with braces will be executed first, then the rest of the expression.

To convert an IDaSS expression into Verilog, at first, all IDaSS operators in the expression have to be converted in Verilog operators. If there isn't any Verilog operator with the same operation as IDaSS operator, special functions in Verilog have to be constructed to carry out those operations. For example: a lot of Unary Operators in IDaSS are converted with function that have been written in Verilog language.

After IDaSS operators are converted into Verilog operators, the precedence level of the operators are changed. This means each part of the expression with binary operators must be checked regarding the precedence level and compared with the Verilog precedence level. If the Verilog operator has a higher precedence level, then braces must be put around that part of the expression.

#### An example:

- ♦ IDaSS expression A ∧ B < 11
- ♦ First expression A && B will be generated in Verilog. This expression has the *inline level 2*, because && operator in Verilog has precedence level 2. This is the left part expression of the operator '<'.
- ♦ The operator '< 'in Verilog has precedence level 7. This is indicated by par 17.
- Conclusion: *inline* is lower than *par*, braces must be put around it.
- ◆ Converted result in Verilog: (A && B) < 11

A list of HDL-Verilog operator precedence levels is included in *Appendix 1*.

#### 4.3 Functions for Unary Operators

The unary operators are the operators with a single operand. IDaSS has powerful unary operators. With these operators, bit vectors will be manipulated easily.

dec decrement value (subtract 1)

epty even parity bit

inc increment value (add 1)
lsomask least significant one bit mask
lsone least significant one bit position
lszmask least significant zero bit mask
lszero least significant zero bit position

maj majority gate

most significant one bit mask msomask most significant one bit position msone most significant zero bit mask mszmask mszero most significant zero bit position neg two's complement negative not complement bits (logical NOT) count number of ONEs in word onecnt ones generate all ONEs constant

opty odd parity bit

rev reverse all bits MSB ←→ LSB width return number of bits in value zerocnt count number of ZEROes in word zeroes generate all ZEROes constant

Verilog has limited Unary Operators. Many functions have been constructed in Verilog language to deal with Unary operators in the Verilog template file. Those functions are tested with the Verilog simulator *Veriwell*.

The function for *most significant one bit mask* (*msomask*) will be described here. In this example, the width of the bit vector is 8 bits. In the template file it will be dependent on the variable. The expression 'par1 width' will calculate the bit width of the variable par1.

```
module fn_msomask;
                                  // This is the module fn_msomask
function [7:0] d2vMSOMASK;
// This is the start of the function d2vMSOMASK;
// The simulation for most significant one bit mask test in HDL-Verilog
input
        [7:0]
                 par1;
                                  // Test input
integer
                 index;
        [7:0]
                 result;
                                  // Test result for output
reg
                 found;
reg
begin
result = 8'b000000000;
found = 0;
for (index = 7; index \geq 0; index = index - 1)
  if (\simfound && par1[index]==1)
   begin
   result[index]=1;
   found=1;
   end
 end
d2vMSOMASK=result;
                                  // This is the output of the test result
end
                                  // This is the end of the function d2vMSOMASK;
endfunction
initial
begin
$display ("function (d2vMSOMASK) = %b", d2vMSOMASK(8'b01000000));
                                  // Calling the function and display the result, when par 1 = 01000000
end
endmodule
                                  // This is the end of the module fn_msomask
```

After this function is tested with the simulator *Veriwell*. It is rewritten for the template file to generate *msomask* function:

```
#UnaryOp msomask
" Most significant one mask priority on bit vector uses function:
sources 'reg_bit_vector'
result 'reg_bit_vector'
function
' 'markindent
'function [' (par1 width - 1) decimal ':0 ] d2vMSOMASK;' cr
'input [' (par1 width - 1) decimal ':0] par1;' cr
' // Equivalent of an IDaSS most significant one mask priority' cr
' // (msomask) operator on a ' (parl width) decimal ' bits word.' cr
'integer index;' cr
'reg [' (par1 width - 1) decimal ':0] result;' cr
'reg found;' cr
'begin' cr
' result =' ((par1 width zeroes) width) decimal "b' (par1 width zeroes) binary ';' cr
  found = 0; cr
  for (index =' (par1 width - 1) decimal '; index >=0; index = index - 1)' cr
  begin' cr
    if (~found && par1[index]==1)' cr
    begin' cr
    result[index]=1;' cr
    found=1; cr
    end' cr
' end' cr
' d2vMSOMASK=result;' cr
'end' cr
'endfunction // function d2vMSOMASK' cr
exitindent cr cr
inline 30
generatefunction
'd2vMSOMASK' (par1 width) decimal '(' markindent par 1 0 exitindent ')'
```

Verilog programs for other Unary Operators is included in *Appendix* 2.

#### 4.4 The Binary Operator concatenation

Each IDaSS binary operator has two operands. This is the list of binary operators. They are converted into Verilog operators.

```
add
                 subtract
                 unsigned multiply
                 right hand signed multiply
                 left hand signed multiply
                 signed multiply
                 logical AND
٨
                 logical NAND
~/\
                 logical OR
~V
                 logical NOR
                 logical XOR
><
                 logical XNOR
<>
                 unsigned 'equal'
=
                 unsigned 'not equal'
~=
                 unsigned 'less than'
<
                 unsigned 'less than or equal'
<=
                 unsigned 'less than or equal'
                 unsigned 'greater than'
>
                 unsigned 'greater than or equal'
>=
                 unsigned 'greater than or equal'
=>
                 signed 'equal'
+~=+
                 signed 'not equal'
+<+
                 signed 'less than'
                 signed 'less than or equal'
+<=+
                 signed 'less than or equal'
+=<+
                 signed 'greater than'
                 signed 'greater than or equal'
+>=+
+=>+
                 signed 'greater than or equal'
```

The template file sections for these operators is included in the *appendix 5*. Each binary operator has many sections. Many sections are dealing with the optimization of the conversion. In this subchapter, optimization of the *concatenation operation* will be described.

The concatenation operator in Verilog are '{,}'. To concatenate two variables, like A and B, it will be done with the expression '{A,B}' directly. But it is a problem, when one of these expression already contains an concatenation operation.

For example, If C is the concatenation of A and B, the concatenation of C and D will be: '{C,D}', and the result of the nested concatenation will be '{{A,B},D}'. This means more concatenation operators do the same operation. Because C is only a temporary expression and not the final expression for the output file, only a ',' is needed here, there is no need to insert '{' and '}'. The correct result is '{A,B,C}'.

In general the expression  $\{...\{\{A,B\},C\},D\},...\}$  is generated instead of  $\{...A,B,C,D,...\}$ .

To solve this problem, temporary variable type concat\_mltbit is introduced. The rule of the correct expression generation will be: Insert only ',' if the result type is a concat\_mltbit type.

There are 6 different source types: reg\_bit, reg\_bit\_vector, concat\_mltbit, reg\_bit\_c, reg\_bit\_vector\_c and concat\_mltbit\_c. There are 18 possibilities. The 18 sections for the implementation of the concatenation operator are described in the table.

Source type1	source type2	result type
Par1	par2	
concat_mltbit	concat_mltbit	concat_mltbit
concat_mltbit	reg_bit_vector	concat_mltbit
reg_bit_vector	concat_mltbit	concat_mltbit
reg_bit_vector	reg_bit_vector	concat_mltbit
reg_bit	concat_mltbit	concat_mltbit
reg_bit	reg_bit_vector	concat_mltbit
concat_mltbit	reg_bit	concat_mltbit
reg_bit_vector	reg_bit	concat_mltbit
reg_bit	reg_bit	concat_mltbit
concat_mltbit_c	concat_mltbit_c	concat_mltbit_c
concat_mltbit_c	reg_bit_vector_c	concat_mltbit_c
reg_bit_vector_c	concat_mltbit_c	concat_mltbit_c
reg_bit_vector_c	reg_bit_vector_c	concat_mltbit_c
reg_bit_c	concat_mltbit_c	concat_mltbit_c
reg_bit_c	reg_bit_vector_c	concat_mltbit_c
concat_mltbit_c	reg_bit_c	concat_mltbit_c
reg_bit_vector_c	reg_bit_c	concat_mltbit_c
reg_bit_c	reg_bit_c	concat_mltbit_c

Table 1 Implementation of the concatenation operator in the template file.

Example: the result type is a concat\_mltbit type, insert only ',' between the sources.

#BinaryOp , "Start of the new section"
"Concatenation between vectors:
sources 'concat\_mltbit' 'concat\_mltbit' "two source expressions
result 'concat\_mltbit' "result is temporary type concat\_mltbit,"
inline 30 "precedence level of the section
par 1 0 ',' par 2 0 "only ',' is needed here

### Chapter 5

# Instructions for the target language file generation

Starting the AFT-Verilog is done from the menu of any schematic window in any IDaSS session. Entry miscellaneous, followed by alien file template... and attach new template, then HDL-Verilog template can be chosen. By the way, if an other AFT is loaded in the IDaSS session, like AFT-Compass, then this template file has to be removed first from the RAM of the computer with the same menu entry.

It is also possible to attach the template automatically during the startup of the IDaSS. The files delivered with the IDaSS system package, a file is named *idass.cnf*. In this file the line 'template = Verilog.aft' has to be added to load the template automatically. Load the template AFT-Verilog may take a minute on a Pentium machine. It will use about a half megabyte of the memory.

### Chapter 6

### Conclusions and recommendations

In this report a converter is constructed to generate Verilog code directly from IDaSS. Functions have been implemented to convert IDaSS unary operators and binary operators into Verilog language. Verilog language optimized instructions have been implemented to generate the output file.

The converter, the file *Verilog.aft*, is a large text file. Beside the sections that contains the work described in this report, (Expression, Unary and Binary Operators Conversion, totally approximately half the size of Verilog.aft), also contains other code. These code has to be modified. The conversion of the IDaSS keyword operators in Verilog statement will be the next challenge after which the actual building blocks of IDaSS can be converted.

### References

[Lee 97] James M. Lee

Verilog Quickstart.

Dordrecht: Kluwer, 1997. + 1 diskette (3.5").

[Smi 97] Smith, D.J.

HDL basic training: top-down chip design using Verlog and VHDL. EDN [European Edition], Vol:41 1996 Iss:22, p. 103-4, 104, 110, 112.

[Gol 96] Golze, U.

VLSI Chip Design with the Hardware Description Language VERILOG: an introduction based on a large RISC processor design.

Berlin: Springer, 1996. + 1 diskette (3.5").

[Tom 96] Tomson, P.

VHDL for hardware design.

Dr. Dobb's Journal, Vol. 21 1996 Iss: 6, p. 46,48,50,53,55,86.

[Han 95] Hannan, J.

Operational semantics-directed compilers and machine architectures.

ACM Transactions on Programming Languages and Systems,

Vol: 16 1994 Iss: 4, p.1215-47.

[Pont 95] Pont, J.F.

A converter from IDaSS design file to synthesizable VHDL.

Eindhoven University of Technolgy, Faculty of Electrical Engineering, Section of Information and Communication Systems, Eindhoven,

Netherlands, 1995 Master thesis report ICS-EB 589.

[ABK 94] Anderson, P. and D. Bolton, P. Kelly.

Paragon specifications: structure, analysis and implementation.

Future Generation Computer Systems, Vol. 10 1994 Iss. 1, p. 137-48.

[Ber 94] Berman, Victor.

Standard Verilog-VHDL interoperability.

In: Proceedings of the 1994 International Verilog HDL Conference. Los Alamitos, CA. IEEE, Computer Society Press, 1994. p 2-9.

[CoTh 94] Coumeri, Sari L., and, Donald E. Thomas.

Benchmark descriptions for comparing the performance of verilog and VHDL simulators.

In: Proceedings of the 1994 International Verilog HDL Conference. Los Alamitos, CA. IEEE, Computer Society Press, 1994. p 37-42.

[SKKK 94] Sankarshanan, P.N. and H. Kobayashi, P.Kukkal, H. Kanbara. VHDL, verilog-HDL, and UDL/I-feature description and analysis. *IEICE Transactions on information and systems*, Vol: E76-D, 1993 Iss: 9, p. 1055-65.

[HuQu 93] Huaiming Sun and Qun Liang.

A theory of automatic logic programming based on second order term rewriting technique.

IFIP Transactions A [Computer Science and Technology, Vol: A-19 1992, p. 165-76.

[Mag 93] Maginot, Serge.

Evaluation criteria of HDLs: VHDL compared to Verilog, UDL/I & M. In: European Design Automation Conference EURO VHDL 92. Los Alamitos, CA. IEEE, Computer Society, 1992. p 746-751.

- [SGC 93] Schafers, M. and U. Golze, E. Cochlovius.
   Verilog-HDL models of a large RISC processor.
   In: Proc. 4<sup>th</sup> EUROCHIP Workshop, Toledo, 1993. p. 242-246.
- [SST 93] Sternheim, E. and R. Singh, Y. Trivedi.
   Digital design and synthesis with Verilog-HDL.
   Automata Publishing Campany, Cupertino, CA., 1993.
- [Wal 93] Wall, D.W.

  Experience with a software-defined machine architecture.

  ACM Transactions on Programming Languages and Systems,
  Vol: 14,1992 Iss: 3, p. 299-338.
- [BHK 91] Bolton, D. and C. Hankin, P. Kelly.

  An operational semantics for Paragon: a design notation for parallel architectures.

  New Generation Computing, Vol. 9, 1991 Iss. 2, p. 171-97.

[Con 91] Conner, D.

Logic-synthesis tools speed ASIC designs

EDN, Vol. 35 1990 Iss. 19, p. 97, 99-100, 102, 104, 106.

[ThMo 91] Thomas, D.E. and P. Moorby.

The Verilog hardware description language.

Boston: Kluwer, 1991.

[BaRe 90] Balou, A.T., and A.N.Refenes.

Designing a parallel object-oriented compiler target language (TOOL).

Microprocessing & Microprogramming. Vol. 30, 1990 Iss: 1-5 p. 457-66.

[Mey 90] Meyer, E.

Test raises questions about VHDL/Verilog interoperability.

Computer Design, Vol. 29,1990 Iss: 3, p. 30, 34, 38.

[Ver 90-1] Verschueren, A.C.

IDaSS for ULSI (Manual).

Eindhoven University of Technolgy, Faculty of Electrical Engineering, Section of Information and Communication Systems, Eindhoven,

Netherlands, 1990.

[Ver 90-2] Verschueren, A.C.

An object oriented design and simulation system for VLSI.

Microprocessing & Microprogramming, Vol. 30, 1990 Iss. 1-5, p. 241-6.

[Wen 90] Wendt, A.L.

Fast code grneration using automatically-generated decision tree.

SIGPLAN Notices, Vol. 25, 1990 Iss. 6, p. 9-15.

[Wat 89] Waters, R.C.

Program translation via abstraction and reimplementation.

IEEE Transactions on Software Engineering,

Vol: 14,1988 Iss: 8, p. 1207-28.

## Appendix 1 Verilog precedence levels

level 0	? : (ternary)			
level 1	(logical OR)			
level 2	&& (logical AND)			
level 3	(bitwise OR)			
level 4	^ (bitwise XOR)	^~ (bitwise XNOR)		
level 5	& (bitwise AND)			
level 6	== (equal)	=== (equal also x,z)	!= (not equal)	!== (not equal also z, x)
level 7	< (less)	<= (less or equal)	> (great)	>= (great or equal)
level 8	<< (shift left)	>> (shift right)		
level 9	+ _ (addition)	- (subtraction)		
level 10	* (multiplication)	/ (division)	% (gives remainder)	
level 11	! (logical NOT)	~ (bitwise not)		

Table 2. HDL-Verilog standard operator precedence levels (low to high):

## Appendix 2 Verilog program

```
module fn_lsomask;
                                // This is the module fn_Isomask
function [7:0] d2vLSOMASK;
                                // This is the start of the function d2vLSOMASK
                                // Simulation for least significant one bit mask test in HDL-Verilog
input
        [7:0]
                par1;
                                // Test input
integer
                index;
                                // Test result for output
        [7:0]
reg
                result;
reg
                found;
begin
result = 8'b00000000;
found = 0;
  for (index = 0; index \leq 7; index = index + 1)
  begin
  if (~found && par1[index]==1)
   begin
    result[index]=1;
   found=1;
    end
  end
d2vLSOMASK=result;
                                // This is the output of the test result
end
endfunction
                                // This is the end of the function d2vLSOMASK
initial
$display ("function (d2vLSOMASK) = %b", d2vLSOMASK(8b11101110));
                                // Calling the function and display the result, when par1 = 11101110
end
endmodule
                                // This is the end of the module fn_Isomask
```

```
module fn_lsone;
                                // This is the module fn_lsone
function [7:0] d2vLSONE;
                                // This is the start of the function d2vLSONE
                                // Simulation for least significant one bit position test in HDL-Verilog
input
        [7:0]
                par1;
                                // Test input
integer
                index;
reg
        [7:0]
                result;
                                // Test result for output
                found;
reg
begin
result = 8'b00000000;
found = 0;
for (index = 0; index \leq 7; index = index + 1)
  begin
  if (~found && par1[index]==1)
   begin
    result=index;
   found=1;
    end
  if (~found) result=8;
  end
d2vLSONE=result;
                               // This is the output of the test result
end
endfunction
                                // This is the end of the function d2vLSONE
initial
begin
$display ("function (d2vLSONE) = %d", d2vLSONE(8'b00000100));
                                // Calling the function and display the result, when par1 = 00000100
end
endmodule
                                 // This is the end of the module fn_lsone
```

```
module fn_iszero;
                                // This is the module fn_lszero
function [7:0] d2vLSZERO;
                                // This is the start of the function d2vLSZERO
                                // Simulation for least significant zero bit position test in HDL-Verilog
input
       [7:0]
                par1;
                                // Test input
integer
                index;
        [7:0]
                result;
                                // Test result for output
reg
                found;
reg
begin
result = 8'b00000000;
found = 0;
for (index = 0; index \leq 7; index = index + 1)
 begin
  if (~found && par1[index]==0)
   begin
   result=index:
   found=1:
   end
  if (~found) result=8;
  end
d2vLSZERO=result;
                                // This is the output of the test result
end
                                // This is the end of the function d2vLSZERO
endfunction
initial
begin
$display ("function (d2vLSZERO) = %d", d2vLSZERO(8'b11111011));
                                // Calling the function and display the result, when par1 = 11111011
end
endmodule
                                // This is the end of the module fn_lszero
```

```
module fn_lszmask;
                                // This is the module fn_lszmask
function [7:0] d2vLSZMASK;
                                // This is the start of the function d2vLSZMASK
                                // Simulation for least significant zero bit mask test in HDL-Verilog
input
        [7:0]
                par1;
                                // Test input
integer
                index;
                                // Test result for output
reg
        [7:0]
                result;
                found;
reg
begin
result = 8'b00000000;
found = 0;
for (index = 0; index \leq 7; index = index + 1)
  if (~found && par1[index]==0)
   begin
    result[index]=1;
   found=1:
    end
  end
d2vLSZMASK=result;
                                // This is the output of the test result
end
endfunction
                                // This is the end of the function d2vLSZMASK
initial
begin
$display ("function (d2vLSZMASK) = %b", d2vLSZMASK(8'b01011111));
                                // Calling the function and display the result, when par1 = 01011111
end
endmodule
                                // This is the end of the module fn_Iszmask
```

```
module fn_maj_even;
                                // This is the module fn_maj_even
function [1:0] d2vMAJ;
                                // This is the start of the function d2vMAJ
                                // The simulation for majority gate even test in HDL-Verilog
input
        [7:0]
                par1;
                                // Test input
integer
                index,cnt0,cnt1;
reg
        [1:0]
                result;
                                // Test result for output
begin
result = 2'b00;
cnt0=0;cnt1=0;
for (index = 0; index \leq 7; index = index + 1)
if ( par1[index]==1) cnt1=cnt1+1;
else if (par1[index]==0) cnt0=cnt0+1;
if (cnt1==cnt0) result=2'b00;
                                // No majority
if (cnt1>cnt0) result=2'b10;
                               // 1 majority
else result=2'b01;
                                // 0 majority
d2vMAJ=result;
                                // This is the output of the test result
end
endfunction
                                // This is the end of the function d2vMAJ
initial
begin
$display ("function (d2vMAJ) = %b", d2vMAJ(8'b11101010));
                                // Calling the function and display the result, when par1 = 11101010
end
endmodule
                                // This is the end of the module fn_maj_even
```

```
module fn_maj_odd;
                                // This is the module fn_maj_odd
function d2vMAJ;
                                 // This is the start of the function d2vMAJ
                                 // The simulation for majority gate odd test in HDL-Verilog
                                 // Test input
input
        [6:0]
                par1;
integer index,cnt0,cnt1;
                                // Test result for output
reg
                result:
begin
result = 0;
cnt0=0;
cnt1=0;
for (index = 0; index \leq 6; index = index + 1)
if ( par1[index]==1) cnt1=cnt1+1;
else if (par1[index]==0) cnt0=cnt0+1;
if (cnt1>cnt0) result=1;
                                // One is majority
else result=0;
                                // Zero is majority
d2vMAJ=result;
                                // This is the output of the test result
end
endfunction
                                 // This is the end of the function d2vMAJ
initial
begin
$display ("function (d2vMAJ) = %b", d2vMAJ(7'b1110100));
                                 // Calling the function and display the result, when par1 = 1110100
end
endmodule
                                 // This is the end of the module fn_maj_odd
```

```
module fn_msomask;
                               // This is the module fn_msomask
function [7:0] d2vMSOMASK;
                                // This is the start of the function d2vMSOMASK;
                                // Simulation for most significant one bit mask test in HDL-Verilog
input
       [7:0]
                par1;
                                // Test input
integer
                index:
reg
        [7:0]
                result;
                                // Test result for output
               found;
reg
begin
result = 8'b00000000;
found = 0;
for (index = 7; index  = 0; index = index - 1)
 begin
 if (~found && par1[index]==1)
   begin
   result[index]=1;
   found=1;
   end
 end
d2vMSOMASK=result;
                               // This is the output of the test result
end
endfunction
                                // This is the end of the function d2vMSOMASK;
initial
begin
$display ("function (d2vMSOMASK) = %b", d2vMSOMASK(8'b01000000));
                                // Calling the function and display the result, when par1 = 01000000
end
endmodule
                                // This is the end of the module fn_msomask
```

```
module fn_msone;
                                // This is the module fn_msone
function [7:0] d2vMSONE;
                                // This is the start of the function d2vMSONE
                                // Simulation for most significant one bit position test in HDL-Verilog
                                // Test input
input [7:0]
                par1:
integer
                index;
reg
        [7:0]
                result:
                                // Test result for output
                found;
reg
begin
result = 8'b00000000;
found = 0;
for (index = 7; index  = 0; index = index - 1)
  if (~found && par1[index]==1)
   begin
   result=index;
   found=1;
   end
  if (~found) result=8;
  end
d2vMSONE=result;
                                // This is the output of the test result
end
                                // This is the end of the function d2vMSONE
endfunction
initial
begin
$display ("function (d2vMSONE) = %d", d2vMSONE(8'b01000000));
                                // Calling the function and display the result, when par1 = 01000000
end
endmodule
                                // This is the end of the module fn_msone
```

```
module fn_mszero;
                                // This is the module fn_mszero
function [7:0] d2vMSZERO;
                                // This is the start of the function d2vMSZERO
                                // Simulation for most significant zero bit position test in HDL-Verilog
                                // Test input
input [7:0]
                par1;
                index;
integer
       [7:0]
                result;
                                // Test result for output
reg
reg
                found;
begin
result = 8'b00000000;
found = 0;
for (index = 7; index >=0; index = index - 1)
 begin
  if (~found && par1[index]==0)
   begin
   result=index;
   found=1;
   end
  if (~found) result=8;
 end
d2vMSZERO=result;
                                // This is the output of the test result
end
endfunction
                                // This is the end of the function d2vMSZERO
initial
begin
$display ("function (d2vMSZERO) = %d", d2vMSZERO(8'b10111011));
                                // Calling the function and display the result, when par1 = 10111011
end
endmodule
                                // This is the end of the module fn_mszero
```

```
module fn_mszmask;
                               // This is the module fn_mszmask
function [7:0] d2vMSZMASK;
                               // This is the start of the function d2vMSZMASK
                               // Simulation for most significant zero bit mask test in HDL-Verilog
       [7:0]
                               // Test input
input
                par1:
integer
                index:
reg
       [7:0]
                result;
                               // Test result for output
reg
                found:
begin
result = 8'b00000000;
found = 0;
for (index = 7; index >= 0; index = index - 1)
 if (~found && par1[index]==0)
 begin
 result[index]=1;
 found=1;
 end
end
d2vMSZMASK=result;
                               // This is the output of the test result
end
endfunction
                               // This is the end of the function d2vMSZMASK
initial
begin
$display ("function (d2vMSZMASK) = %b", d2vMSZMASK(8'b10111010));
                               // Calling the function and display the result, when par1 = 10111010
end
endmodule
                                // This is the end of the module fn_mszmask
```

```
// This is the module fn onecnt
module fn_onecnt;
function [7:0] d2vONECNT:
                                // This is the start of the function d2vONECNT
                                // The simulation for count ones in a word test in HDL-Verilog
input
        [7:0]
                par1;
                                // Test input
                index;
integer
reg
        [7:0]
                cnt;
                                // Test result for output
begin
cnt=0:
for (index = 0: index \leq 7: index = index + 1)
if (par1[index]==1) cnt=cnt+1;
d2vONECNT=cnt;
                                 // This is the output of the test result
end
endfunction
                                // This is the end of the function d2vONECNT
initial
beain
$display ("function (d2vONECNT) = %d", d2vONECNT(8b11110011));
                                 // Calling the function and display the result, when par1 = 11110011
end
                                // This is the end of the module fn_onecnt
endmodule
module fn zerocnt;
                                 // This is the module fn zerocnt
function [7:0] d2vZEROCNT;
                                 // This is the start of the function d2vZEROCNT
                                 // The simulation for count zero bits in a word test in HDL-Verilog
        [7:0]
                                 // Test input
input
                par1;
integer
                index;
        [7:0]
                                 // Test result for output
reg
                cnt;
beain
cnt=0:
for (index = 0; index \leq 7; index = index + 1)
if (par1[index]==0) cnt=cnt+1;
d2vZEROCNT=cnt;
                                 // This is the output of the test result
end
endfunction
                                 // This is the end of the function d2vZEROCNT
initial
$display ("function (d2vZEROCNT) = %d", d2vZEROCNT(8'b10110001));
                                 // Calling the function and display the result, when par1 = 10110001
end
endmodule
                                 // This is the end of the module fn_zerocnt
```

```
// This is the module fn_epty
module fn_epty;
function d2vEPTY;
                                 // This is the start of the function d2vEPTY
                                 // The simulation for even parity test in HDL-Verilog
        [7:0]
input
                                 // Test input
                par1;
                index;
integer
                                 // Test result for output
                result;
reg
begin
  result = 1;
  for (index = 0; index \leq 7; index = index + 1)
  result=result^par1[index];
  d2vEPTY=result:
                                 // This is the output of the test result
end
                                 // This is the end of the function d2vEPTY
endfunction
initial
begin
  $display ("function (d2vEPTY) = %d", d2vEPTY(8'b10111110));
                                 // Calling the function and display the result, when par1 = 10111110
end
endmodule
                                 // This is the end of the module fn_epty
module fn_opty;
                                 // This is the module fn_opty
function d2vOPTY;
                                 // This is the start of the function d2vOPTY
                                 // The simulation odd parity bit test in HDL-Verilog
        [7:0]
input
                                 // Test input
                 par1;
integer
                 index;
                                  // Test result for output
reg
                 result;
begin
result = 0;
for (index = 0; index \leq 7; index = index + 1)
result=result^par1[index];
d2vOPTY=result;
                                 // This is the output of the test result
end
endfunction
                                 // This is the end of the function d2vOPTY
initial
begin
$display ("function (d2vOPTY) = %d", d2vOPTY(8'b11100000));
                                 // Calling the function and display the result, when par1 = 11100000
end
endmodule
                                 // This is the end of the module fn_opty
```

## Appendix 3 Simulation and Implementation of Signed Multiply

```
// The simulation for multiply_signed function in Verilog
module multiply;
                                  // This is the module multiply
function [7:0] multiply_signed;
input
        [3:0]
                 l,r;
                                  // Test input
                 index;
integer
        [3:0]
                 result;
                                  // Test result for output
        [3:0]
                 lowbit;
reg
        [4:0]
                 accu;
reg
begin
accu = 5'b000000;
for (index = 0; index \leq 3; index = index + 1)
  begin
  if (r[index] == 1)
        begin
        if (index == 3)
                 accu=accu - {I[3],I};
                 else
                 accu = accu + {I[3],I};
         end
  lowbit [index] = accu [0];
  accu={accu[4],accu[4:1]};
result={accu[3:0],lowbit};
multiply_signed=result;
                                  // This is the output of the test result
end
endfunction
                                  // This is the end of the function multiply_signed;
initial
begin
$display ("function (multiply_signed) = %b", multiply_signed(4'b0001,4'b0001));
                                  // Calling the function, display the result, when I = 0001 and r=0001
end
endmodule
                                  // This is the end of the module multiply_signed
```

## "This is the implementation of signed multiply function in Verilog

```
'function [' ((par1 width) + (par2 width) -1) decimal ':0 ] multiply_signed;' cr
 "This is the signed multiply function for HDL-Verilog
 'input [' (par1 width - 1) decimal ':0 ] par1;' cr
 'input [' (par2 width - 1) decimal ':0 ] par2;' cr
 'integer index;' cr
 'reg [' ((par1 width) + (par2 width) -1) decimal ':0 ] result;' cr
 'reg [' (par1 width - 1) decimal ':0 ] lowbit;' cr
 'reg [' (par1 width + 1) decimal ':0 ] accu;' cr
 'begin' cr
 'accu =' ((par1 width) + 1) decimal "b' (((par1 width) + 1) zeroes ) binary ';' cr
 'for (index = 0; index <=' (par1 width - 1) decimal '; index = index + 1 )' cr
    if (par2[index] ==1)' cr
    begin' cr
       if (index == ' (par1 width - 1) decimal ')' cr
         accu= accu - {par1 [' (par1 width - 1) decimal '], par1};' cr
         accu= accu + {par1 [' (par1 width - 1) decimal '], par1};' cr
       end' cr
    end' cr
    lowbit [index] = accu[0];' cr
   accu=[accu[' (par1 width ) decimal '], accu[' (par1 width) decimal ':1]];' cr
 ' end' cr
 'result={accu[' (par1 width - 1) decimal ':0], lowbit};' cr
 'multiply_signed=result;' cr
  'end' cr
 'endfunction // function multiply_signed;' cr
```

## Appendix 4 Example of Verilog code for a shiftregister

```
// This is a example of a shiftregister
module vshftreg1;
                              //The name of the module Verilog shiftregister
integer [7:0] tel;
reg
               clock,reset;
        [7:0] X_in;
wire
        [7:0] X_out;
wire
reghl reghl(clock,reset,X_in,X_out);
initial
begin
 reset=0;#2;reset=1;#2;//reset=0;#1;
 $display ("INITIAL after reset=1 X_out value= %b", X_out);
 $display ("=========");
 for (tel=1; tel<=25; tel=tel+1)
  begin
  clock=0; #1;clock=1;#1;
  $display ("Clock X_out value= %b", X_out);
  $display ("-----
  end
 $finish;
end
endmodule
// Register
module reghl (clock,reset,X_in,X_out);
input
               clock,reset;
       [7:0] X_in;
[7:0] X_out;
[7:0] d2vCONTENTS;
input
output
reg
        [7:0] d2vOUT;
reg
               d2vCTRL;
reg
               d2vTEST1, d2vTEST2;
reg
               d2vCURRSTATE;
reg
         [7:0] X_out=d2vOUT;
wire
```

```
task Tsk left:
d2vOUT=d2vCONTENTS << 1;
endtask
task Tsk_right;
d2vOUT=d2vCONTENTS >> 1;
endtask
always @ (posedge reset)
begin
 d2vOUT=8'b00010000:
 $display ("Reset d2vOUT waarde= %b", d2vOUT);
 d2vCTRL=0:
 d2vCURRSTATE =0; //ST_left
 $display ("Reset d2vCTRL waarde= %b", d2vCTRL);
 $display ("Reset d2vCURRSTATE waarde= %b", d2vCURRSTATE);
end
always @ (posedge clock)
begin
 d2vCONTENTS=d2vOUT;
 d2vTEST1=d2vCONTENTS[7]:
 d2vTEST2=d2vCONTENTS[0];
 case (d2vCURRSTATE)
 0: begin //ST left
        if (d2vTEST1==1'b0) begin
        d2vCTRL=0;
        d2vCURRSTATE = 0; end //ST_left
        else begin
        d2vCTRL=1;
        d2vCURRSTATE = 1; end //ST_right
   end
 1: begin //ST_right
        if (d2vTEST2==1'b0) begin
        d2vCTRL=1;
        d2vCURRSTATE =1; end //ST_right
        else begin
        d2vCTRL=0;
        d2vCURRSTATE =0; end // ST_left
   end
 endcase
 if (d2vCTRL==1)
  Tsk_right;
  else
  Tsk_left;
endmodule //This is the end of the module shiftregister
```

## Appendix 5 The template file

```
'reg ' myname tabto 21 '; // Original IDaSS temp var' cr
     'Template' file for generating HDL-Verilog"
                                                                        root
            directly from within IDaSS.
                                                                        attachseparatelist
                                                                        generatetempvars
                                                                         insertseparatelist
                                                                         ' ' markindent target myname ' ='
                                                                         generatetree ';' exitindent cr
                   Expressions
                                                                        inline 31
#BasicExpression Operator
                                                                        myname
" Assigning single bit to output:
                                                                         #TempExpression Operator
guard (prototype width = 1 \Lambda
                                                                         " Assigning direct single bit to temp var:
    prototype isUNK)
                                                                         guard (prototype width = 1 \Lambda
result 'reg_bit' withroot
                                                                             prototype isUNK Λ
                                                                             issimple)
attachseparatelist
                                                                        result 'reg_bit' noroot
insertseparatelist
' 'markindent target generateattached 7 ' = '
                                                                        inline 31
generatetree ';' exitindent cr
                                                                         generatetree
#BasicExpression Operator
                                                                         #TempExpression Operator
" Assigning single constant bit to output:
                                                                         " Assigning single constant bit to temp var:
guard (prototype width = 1 \Lambda
                                                                         guard (prototype width = 1 \Lambda
    prototype isUNK not)
                                                                             prototype isUNK not)
result 'reg_bit' noroot
                                                                         result 'reg_bit' noroot
                                                                         inline 30
  markindent generateattached 7 ' =' cr
                                                                         (prototype) decimal
  (prototype) decimal ';' exitindent cr
                                                                         #TempExpression Operator
#BasicExpression Operator
                                                                         " Assigning calculated bit vector to temp var:
" Assigning bit vector to output:
                                                                         guard (prototype width > 1 \Lambda
guard (prototype width > 1 \Lambda
                                                                             prototype isUNK ∧
    prototype isUNK)
                                                                             issimple not)
result 'reg_bit_vector' withroot
                                                                         result 'reg_bit_vector' withroot
root
                                                                         tempvar 0
attachseparatelist
                                                                         'reg [ '
                                                                          (prototype width - 1 width: 8) decimalleft
insertseparatelist
 'markindent target generateattached 7 '='
                                                                          ':0]' myname tabto 54 '; // Original IDaSS temp var' cr
generatetree exitindent ';' cr
                                                                         root
#BasicExpression Operator
                                                                         attachseparatelist
                                                                         generatetempvars
" Assigning constant bit vector to output:
                                                                         insertseparatelist
                                                                         ' ' markindent target myname ' ='
                                                                         generatetree ';' exitindent cr
guard (prototype width > 1 \Lambda
    prototype isUNK not)
                                                                         inline 31
result 'reg_bit_vector' noroot
                                                                         mvname
root
                                                                         #TempExpression Operator
 'markindent generateattached 7 ' = ' cr
' ' (prototype width) decimal "'b' (prototype) binary ';'
                                                                         * Assigning direct bit vector to temp var:
exitindent cr
                                                                         guard (prototype width > 1 \Lambda
#TempExpression Operator
                                                                             prototype isUNK Λ
                                                                             issimple)
" Assigning calculated single bit to temp var:
                                                                         result 'reg_bit_vector' noroot
guard (prototype width = 1 \Lambda
    prototype isUNK A
                                                                         inline 31
    issimple not)
                                                                         generatetree
result 'reg_bit' withroot
                                                                         #TempExpression Operator
tempvar 0
                                                                         Assigning constant bit vector to temp var:
```

```
issimple not)
guard (prototype width > 1 \Lambda
    prototype isUNK not)
                                                                        result 'reg_bit_vector' withroot
result 'reg_bit_vector' noroot
                                                                        tempvar &10
                                                                        'reg [ '
                                                                         (prototype width - 1 width: 8) decimalleft
inline 30
(prototype width) decimal "b' (prototype) binary
                                                                          ':0 ] d2vTEST' (INDEX1) decimal tabto 54 '; // Test
                                                                        variable' cr
#TempExpression Operator
                                                                        root
                                                                        IF (pass = 0)
" Assigning constant to temp var, dummy entry, as this
will never be actually called:
                                                                        THEN
                                                                         attachseparatelist
                                                                         generatetempvars
guard (prototype width = 0)
                                                                         target 'dummy' generatetree
                                                                        ELSE
result 'constant' noroot
                                                                         IF (pass = 4)
                                                                         THEN " For insertion in CASE..IS..:
inline 30
                                                                           'd2vTEST' (INDEX1) decimal
(prototype) decimal
                                                                          ELSE
#BasicExpression ControlCase
                                                                           insertseparatelist
                                                                           ' 'markindent
                                                                           target 'd2vTEST' (INDEX1) decimal ' ='
" Assigning calculated single bit to test:
                                                                           generatetree ';' exitindent cr
guard (prototype width = 1 \Lambda
                                                                          ENDIF
    issimple not)
                                                                        ENDIF
result 'reg_bit' withroot
                                                                        inline 31
                                                                        'd2vTEST' (INDEX1) decimal
tempvar &00000010
'reg d2vTEST' (INDEX1) decimal
                                                                        #BasicExpression ControlCase
 tabto 21 '; // Test variable' cr
                                                                        " Assigning direct bit vector to test:
root
IF (pass = 0)
                                                                        guard (prototype width > 1 \Lambda
THEN
                                                                             issimple)
 attachseparatelist
 generatetempvars
                                                                        result 'reg_bit_vector' noroot
 target 'dummy' generatetree
ELSE
                                                                        root
 IF (pass = 4)
                                                                        IF (pass = 4)
 THEN " For use in CASE..IS.. : 'd2vTEST' (INDEX1) decimal
                                                                        THEN " For use in CASE..IS.. :
                                                                         generatetree
 ELSE " Actual expression insertion:
                                                                        ENDIF
   insertseparatelist
   ' 'markindent
                                                                        inline 31
                                                                        generatetree
   target 'd2vTEST' (INDEX1) decimal ' ='
   generatetree ';' exitindent cr
 ENDIF
                                                                        #TempExpression ControlCase
ENDIF
                                                                         " Assigning calculated single bit to temp var:
inline 31
'd2vTEST' (INDEX1) decimal
                                                                        guard (prototype width = 1 \Lambda
                                                                             prototype isUNK Λ
#BasicExpression ControlCase
                                                                             issimple not)
" Assigning direct single bit to test:
                                                                        result 'reg_bit' withroot
guard (prototype width = 1 \Lambda
                                                                        tempvar 0
                                                                        'reg ' myname tabto 21 '; // Original IDaSS temp var' cr
    issimple)
result 'reg_bit' noroot
                                                                        root
                                                                        attachseparatelist
                                                                        generatetempvars
root
IF (pass = 4)
                                                                        insertseparatelist
THEN * For use in CASE..IS..:
                                                                         ' ' markindent target myname ' ='
  generatetree
                                                                        generatetree ';' exitindent cr
ENDIF
                                                                        inline 31
inline 31
                                                                        myname
generatetree
                                                                        #TempExpression ControlCase
#BasicExpression ControlCase
                                                                         " Assigning direct single bit to temp var:
" Assigning calculated bit vector to test:
                                                                        guard (prototype width = 1 \Lambda
guard (prototype width > 1 \Lambda
                                                                             prototype isUNK Λ
```

```
issimple)
                                                                       inline 30
                                                                        (prototype) decimal
result 'reg_bit' noroot
                                                                        #BasicExpression CommandCombiner
inline 31
generatetree
                                                                        " Method for writing out the command combiner
                                                                        expression
#TempExpression ControlCase
                                                                        " tree for a single bit combined command channel:
* Assigning single constant bit to temp var:
                                                                        guard (prototype width = 1 \land
                                                                             issimple not)
guard (prototype width = 1 \Lambda
    prototype isUNK not)
                                                                        result 'reg_bit' withroot
result 'reg_bit' noroot
                                                                        tempvar 0
                                                                        'wire d2vINTCMD' tabto 21 '; // Combined internal
inline 30
                                                                        command channel cr
(prototype) decimal
                                                                        root
#TempExpression ControlCase
                                                                        IF (pass = 7)
                                                                        THEN * For source lists:
" Assigning calculated bit vector to temp var:
                                                                         'd2vINTCMD'
guard (prototype width > 1 \Lambda
                                                                         generatetempvars
                                                                           ' markindent target 'd2vINTCMD ='
    prototype isUNK Λ
    issimple not)
                                                                          generatetree ';' exitindent cr
                                                                        ENDIF
result 'reg_bit_vector' withroot
                                                                        inline 31
                                                                        'd2vINTCMD'
tempvar 0
'reg [
 (prototype width - 1 width: 8) decimalleft
                                                                        #BasicExpression CommandCombiner
 ':0]' myname tabto 54 '; // Original IDaSS temp var' cr
                                                                        " Method for writing out the command combiner
root
                                                                        expression
attachseparatelist
                                                                        " tree for a multi bit combined command channel:
generatetempvars
insertseparatelist
                                                                        guard (prototype width > 1 \Lambda
 ' markindent target myname ' ='
                                                                             issimple not)
 generatetree ';' exitindent cr
                                                                        result 'reg_bit_vector' withroot
inline 31
myname
                                                                        tempvar 0
                                                                        'wire [ '
                                                                         (prototype width - 1 width: 8) decimalleft
':0] d2vINTCMD' tabto 54'; // Combined internal
#TempExpression ControlCase
" Assigning direct bit vector to temp var:
                                                                        commands channel' cr
guard (prototype width > 1 \Lambda
                                                                        root
    prototype isUNK Λ
                                                                        IF (pass = 7)
    issimple)
                                                                        THEN " For source lists:
                                                                         'd2vINTCMD'
result 'reg_bit_vector' noroot
                                                                        ELSE
                                                                         generatetempvars
                                                                           ' markindent target 'd2vINTCMD ='
inline 31
generatetree
                                                                           generatetree ';' exitindent cr
                                                                        ENDIF
#TempExpression ControlCase
                                                                        inline 31
" Assigning constant bit vector to temp var:
                                                                        'd2vINTCMD'
guard (prototype width > 1 \Lambda
                                                                        #BasicExpression CommandCombiner
    prototype isUNK not)
                                                                        " Method for writing out the command combiner
result 'reg_bit_vector' noroot
                                                                        expression
                                                                        " for a single bit command channel which is directly for-
                                                                        " warded from either an internal or external control
                                                                        " channel:
(prototype width) decimal "b' (prototype) binary
#TempExpression ControlCase
                                                                        guard (prototype width = 1 \Lambda
                                                                             issimple)
* Assigning constant to temp var, dummy entry, as this
" will never be actually called:
                                                                        result 'reg_bit' noroot
guard (prototype width = 0)
                                                                        root
                                                                        generatetree
result 'constant' noroot
                                                                        inline 31
```

```
generatetree
                                                                      markindent generatetree exitindent
#BasicExpression CommandCombiner
                                                                      #BasicExpression CommandValue
" Method for writing out the command combiner
                                                                      " Method for writing out a command value expression
                                                                      * tree for a single bit constant value:
expression
for a multi bit command channel which is directly for-
" warded from either an internal or external control
                                                                      guard (prototype width = 1 \Lambda
" channel:
                                                                           prototype isUNK not)
guard (prototype width > 1 \Lambda
                                                                      result 'reg_bit' noroot
    issimple)
                                                                      " Called with 'generatevalue':
result 'reg_bit_vector' noroot
                                                                      root
                                                                      (prototype) decimal
generatetree
                                                                      #BasicExpression CommandValue
inline 31
generatetree
                                                                      " Method for writing out a command value expression
                                                                       " tree for a multi bit constant value:
#TempExpression CommandCombiner
                                                                      guard (prototype width > 1 \Lambda
" Assigning calculated single bit to temp var:
                                                                           prototype isUNK not)
guard (prototype width = 1)
                                                                      result 'reg_bit_vector' noroot
result 'reg_bit' withroot
                                                                       " Called with 'generatevalue':
tempyar 0
'reg ' myname tabto 21 '; // Command combiner temp
                                                                      (prototype width) decimal "b' (prototype) binary
var' cr
                                                                       #BasicExpression CommandValue
root
attachseparatelist
                                                                      " Method for writing out a command value expression
generatetempvars
                                                                       " tree for a single bit value input on a connector:
insertseparatelist
 'markindent target myname ' = '
                                                                      guard (prototype width = 1 \Lambda
 generatetree ';' exitindent cr
                                                                           prototype isUNK)
inline 31
                                                                      result 'reg_bit' noroot
myname
                                                                       " Called with 'generatevalue':
#TempExpression CommandCombiner
* Assigning calculated bit vector to temp var:
                                                                      markindent generatetree exitindent
guard (prototype width > 1)
                                                                       #BasicExpression CommandValue
result 'reg_bit_vector' withroot
                                                                       " Method for writing out a command value expression
                                                                       tree for a multi bit value input on a connector:
tempvar 0
reg [
                                                                       guard (prototype width > 1 \Lambda
 (prototype width - 1 width: 8) decimalleft
                                                                            prototype isUNK)
  :0] 'myname tabto 54 '; // Command combiner temp
var' cr
                                                                       result 'reg_bit_vector' noroot
                                                                       " Called with 'generatevalue':
root
attachseparatelist
generatetempvars
                                                                       root
insertseparatelist
                                                                       markindent generatetree exitindent
 ' markindent target myname ' ='
 generatetree ';' exitindent cr
                                                                       #BasicExpression CaseTest
inline 31
                                                                       "Check CASEs in an 'IF...' construct.
                                                                       "To remove the (=1'b1) conversion, call this expression
myname
                                                                       with pass 1
#BasicExpression CommandTest
                                                                        (which is handled in the conversion tree node):
"Check commands in an 'IF...' construct.
                                                                       result 'reg_bit' noroot
" If pass 6 or 7 is used for a multi-bit source, the LHS
(6)/RHS (7)
 will be writeen out separately (for conversion in a
                                                                       markindent generatetree exitindent
CASE ... ):
                                                                       #BasicExpression ValueParameter
result 'reg_bit' noroot
                                                                       Needed to insert (slice) of value parameter input for a
root
                                                                       system
```

```
" parameter (like a reset value for a register):
                                                                    #Expression Conversion
guard (prototype width = 1)
                                                                    * From complemented bit vector into bit vector:
result 'reg_bit' noroot
                                                                    sources 'reg_bit_vector_c'
                                                                    result 'reg_bit_vector'
" Called with 'generateparameter':
                                                                    '~(' markindent par 1 11 ')' exitindent
markindent generatetree exitindent
#BasicExpression ValueParameter
                                                                    #ExpressionTypes conversion
" Needed to insert (slice) of value parameter input for a
                                                                    " True constants (the 'inline' is dummy here):
' parameter (like a reset value for a register):
                                                                    guard (prototype width = 0)
                                                                    result 'constant'
guard (prototype width > 1)
result 'reg_bit_vector' noroot
                                                                    (prototype) decimal
* Called with 'generateparameter':
                                                                    #ExpressionTypes conversion
root
                                                                    Single bit values:
markindent generatetree exitindent
guard (prototype width = 1)
                                                                    result 'reg_bit'
#Expression Conversion
" From concat_mltbit into reg_bit_vector:
                                                                    (prototype width) decimal "b' (prototype) binary
sources 'concat_mltbit'
                                                                    #ExpressionTypes conversion
result 'reg_bit_vector'
                                                                    " Multi-bit values:
inline 30
'{' par 1 0 '}'
                                                                    guard (prototype width > 1)
                                                                    result 'reg_bit_vector'
#Expression Conversion
                                                                    (prototype width) decimal "b' (prototype) binary
" From concat_mltbit_c into reg_bit_vector_c:
sources 'concat_mltbit_c'
                                                                     result 'reg_bit_vector_c'
                                                                    #Expression raiseprecedence
inline 30
'{' par 1 0 '}'
                                                                    sources 'reg_bit'
                                                                    result 'reg_bit'
#Expression Conversion
                                                                    inline 30
" From single bit into complemented single bit:
                                                                    '(' markindent par 1 0 exitindent ')'
sources 'reg_bit'
                                                                    #Expression raiseprecedence
result 'reg_bit_c'
                                                                    sources 'reg_bit_c'
                                                                    result 'reg_bit_c'
inline 11
'~(' markindent par 1 11 ')' exitindent
                                                                    inline 30
#Expression Conversion
                                                                    '(' markindent par 1 0 exitindent ')'
" From complemented single bit into single bit:
                                                                    #Expression raiseprecedence
sources 'reg_bit_c'
                                                                    sources 'reg_bit'
result 'reg_bit'
                                                                    result 'reg_bit'
inline 11
                                                                    tempyar 1
                                                                    'reg d2vTEMP' (INDEX) decimal
'~(' markindent par 1 11 ')' exitindent
                                                                    tabto 21 '; // Inserted to raise precedence' cr
#Expression Conversion
                                                                    separate
" From bit vector into complemented bit vector:
                                                                      d2vTEMP' (INDEX) decimal ' = ' cr
                                                                       ' markindent par 1 0 exitindent ';' cr
sources 'reg_bit_vector'
result 'reg_bit_vector_c'
                                                                    inline 31
                                                                    generatetempvars
                                                                    generateseparate
'~(' markindent par 1 11 ')' exitindent
                                                                    'd2vTEMP' (INDEX) decimal
```

#Expression raiseprecedence	" Single bit result:
sources 'reg_bit_c' result 'reg_bit_c'	sources 'reg_bit' result 'reg_bit' "
tempvar 1 'reg d2vTEMP' (INDEX) decimal tabto 21 '; // Inserted to raise precedence' cr	root targetassignment cr ' ' markindent par 1 0 exitindent
separate ' d2vTEMP' (INDEX) decimal ' =' cr ' markindent par 1 0 exitindent ';' cr	inline 31 generateroot
inline 31 generatetempvars generateseparate	#Expression root  "Bit vector result:
'd2vTEMP' (INDEX) decimal # #Expression raiseprecedence	sources 'reg_bit_vector' result 'reg_bit_vector' "
sources 'reg_bit_vector' result 'reg_bit_vector'	root targetassignment cr ' ' markindent par 1 0 exitindent
inline 30 '(' markindent par 1 0 exitindent ')'	inline 31 generateroot
#Expression raiseprecedence	"=====================================
sources 'reg_bit_vector_c' result 'reg_bit_vector_c' "	"=====================================
inline 30 '(' markindent par 1 0 exitindent ')' "	" naming 'returnprioritycount'
#Expression raiseprecedence	" contents
sources 'reg_bit_vector' result 'reg_bit_vector' "	markindent IF (par1 width > 2) THEN
tempvar 1 'reg [ '     (par1 width - 1 width: 8) decimalleft     ' :0 ] d2vTEMP' (INDEX) decimal tabto 54 '; // Inserted to raise precedence' cr  separate ' d2vTEMP' (INDEX) decimal ' =' cr ' ' markindent par 1 0 exitindent ';' cr	IF (par1 width - par1 width log2 = 1) THEN  '{0' ELSE  '{' (((par1 width - par1 width log2) zeroes) width) decimal "'b' ((par1 width - par1 width log2) zeroes) binary ENDIF ', count }'
inline 31 generatetempvars generateseparate 'd2vTEMP' (INDEX) decimal	ELSE 'count' ENDIF exitindent " #UnaryOp dec
#Expression raiseprecedence sources 'reg_bit_vector_c' result 'reg_bit_vector_c'	"Decrement on bit vector done with add: sources 'reg_bit_vector' result 'reg_bit_vector'
tempvar 1 'reg [ '	inline 9 par 1 9 ' +' cr ((par1 width ones) width) decimal "'b' (par1 width ones) binary
separate ' d2vTEMP' (INDEX) decimal ' =' cr	#UnaryOp dec
' markindent par 1 0 exitindent ';' cr	" Decrement on single bit equals NOT operator:
inline 31 generatetempvars generateseparate	sources 'reg_bit' result 'reg_bit_c' "
'd2vTEMP' (INDEX) decimal	forward 1
*======================================	#UnaryOp dec
#Expression root	" Decrement on single bit equals NOT operator:

```
sources 'reg_bit_c'
                                                                         ((1 width: par1 width) width) decimal "b' (1 width: par1
result 'reg_bit'
                                                                         width) binary
forward 1
                                                                         #UnaryOp inc
#UnaryOp epty
                                                                         " Increment on single bit equals NOT operator:
                                                                         sources 'reg_bit'
* Even parity on a single bit, done with NOT:
                                                                         result 'reg_bit_c'
sources 'reg_bit'
result 'reg_bit_c'
                                                                         forward 1
forward 1
                                                                         #UnaryOp inc
#UnaryOp epty
                                                                         "Increment on single bit equals NOT operator:
* Even parity on a single bit, done with NOT:
                                                                         sources 'reg_bit_c'
                                                                         result 'reg_bit'
sources 'reg_bit_c'
result 'reg_bit'
                                                                         forward 1
                                                                         "#UnaryOp log2
forward 1
#UnaryOp epty
                                                                         " Log2 operator, never seen here - always returns
                                                                         constant...
* Even parity on two bits, done with NOT XOR:
                                                                         #UnaryOp Isomask
sources 'reg_bit_vector'
                                                                         " Least significant one mask priority on single bit is
guard (par1 width = 2)
result 'reg_bit_c'
                                                                         removed:
inline 4
                                                                         sources 'rea bit'
par 1 31 '[0] ^' cr
                                                                         result 'reg_bit'
par 1 31 '[1]'
                                                                         forward 1
#UnaryOp epty
                                                                         #UnaryOp Isomask
" Even parity on > 2 bits, uses function:
                                                                         " Least significant one mask priority on single bit is
sources 'reg_bit_vector'
                                                                         removed:
guard (par1 width > 2)
result 'reg_bit'
                                                                         sources 'reg_bit_c'
                                                                         result 'reg_bit_c'
function
 ' markindent
                                                                         forward 1
'function [' (par1 width - 1) decimal ':0 ] d2vEPTY;' cr
'input [' (par1 width - 1) decimal ':0 ] par1;' cr
                                                                         #UnaryOp Isomask
 ' // Equivalent of IDaSS even parity (epty) operator' cr
                                                                         " Least significant one mask priority on bit vector uses
' // on a ' (par1 width) decimal ' bits word.' cr
                                                                         function:
'integer index;' cr
                                                                         sources 'reg_bit_vector'
 'reg result;' cr
                                                                         result 'reg_bit_vector'
 'begin' cr
 ' result =1;' cr
                                                                         function
' for (index = 0; index <=' (par1 width - 1) decimal ';
                                                                          ' markindent
                                                                         'function [' (par1 width - 1) decimal ':0 ] d2vLSOMASK;'
index = index + 1)' cr
   result = result ^par1[index];' cr
   d2vEPTY=result:' cr
                                                                         'input [' (par1 width - 1) decimal ':0 ] par1;' cr
 'end' cr
                                                                         cr
 'endfunction // function d2vEPTY' cr
                                                                         ' // Equivalent of an IDaSS least significant one mask
                                                                         priority' cr
 exitindent cr cr
                                                                           // (Isomask) operator on a ' (par1 width) decimal ' bits
 inline 30
                                                                         word.' cr
 generatefunction
 'd2vEPTY' (par1 width) decimal '(' markindent par 1 0
                                                                         'integer index;' cr
                                                                         'reg [' (par1 width - 1) decimal ':0] result;' cr
 exitindent ')'
                                                                         'reg found;' cr
 #UnaryOp inc
                                                                         'begin' cr
                                                                           result =' ((par1 width zeroes) width) decimal "b'
                                                                         (par1 width zeroes) binary ';' cr
 "Increment on bit vector done with add:
                                                                            found = 0; cr
 sources 'reg_bit_vector'
                                                                            for (index = 0; index <=' (par1 width - 1) decimal ';
 result 'reg_bit_vector'
                                                                         index = index + 1)' cr
                                                                            begin' cr
 inline 9
                                                                              if (~found && par1[index]==1)' cr
 par 1 9 ' +' cr
                                                                              begin' cr
                                                                              result[index]=1;' cr
```

```
found=1:' cr
                                                                         #UnaryOp Iszero
    end' cr
                                                                         " Least significant zero bit pos priority on single bit is
  end' cr
  d2vLSOMASK=result;' cr
                                                                         removed:
'end' cr
'endfunction // function d2vLSOMASK' cr
                                                                         sources 'reg_bit'
exitindent cr cr
                                                                         result 'reg_bit'
inline 30
                                                                         forward 1
generatefunction
'd2vLSOMASK' (par1 width) decimal '(' markindent par
                                                                         #UnaryOp Iszero
1 0 exitindent ')'
                                                                         " Least significant zero bit pos priority on single bit is
#UnaryOp Isone
                                                                         removed:
" Least significant one bit pos priority on single bit is
                                                                         sources 'reg_bit_c'
done with NOT:
                                                                         result 'reg_bit_c'
sources 'reg_bit'
                                                                         forward 1
result 'reg_bit_c'
                                                                         #UnaryOp Iszero
forward 1
                                                                         * Least significant zero bit pos priority on bit vector
#UnaryOp Isone
                                                                         uses function:
" Least significant one bit pos priority on single bit is
                                                                         sources 'reg_bit_vector'
done with NOT:
                                                                         result 'reg_bit_vector'
sources 'reg_bit_c'
                                                                         function
result 'reg_bit'
                                                                         ' 'markindent
                                                                         'function [' (par1 width - 1) decimal ':0] d2vLSZERO;'
forward 1
                                                                         'input [' (par1 width - 1) decimal ':0 ] par1;' cr
#UnaryOp Isone
                                                                         cr
                                                                         ' // Equivalent of an IDaSS least significant zero bit
                                                                         position' cr
Least significant one bit pos priority on bit vector uses
                                                                           // priority (Iszero) operator on a ' (par1 width) decimal
function:
                                                                         'bits word,' cr
sources 'reg_bit_vector'
result 'reg_bit_vector'
                                                                         'integer index;' cr
                                                                         'reg found;' cr
                                                                         'reg [' (par1 width - 1) decimal ':0 ] result;' cr
function
 ' markindent
                                                                          'begin' cr
'function [' (par1 width - 1) decimal ':0 ] d2vLSONE;' cr
                                                                           result =' ((par1 width zeroes) width) decimal "b'
'input [' (par1 width - 1) decimal ':0 ] par1;' cr
                                                                         (par1 width zeroes) binary ';' cr
cr
                                                                            found = 0; cr
' // Equivalent of an IDaSS least significant one bit
                                                                            for (index = 0; index <=' (par1 width - 1) decimal ';
position' cr
                                                                         index = index + 1)' cr
 // priority (Isone) operator on a ' (par1 width) decimal '
                                                                            begin' cr
bits word.' cr
                                                                              if (~found && par1[index]==0)' cr
                                                                              begin' cr
'integer index:' cr
                                                                              result=index:' cr
'reg [' (par1 width - 1) decimal ':0 ] result;' cr
                                                                              found=1;' cr
'reg found;' cr
                                                                              end' cr
'begin' cr
                                                                            if (~found) result =' (par1 width ) decimal ';' cr
  result =' ((par1 width zeroes) width) decimal "b'
                                                                            end' cr
(par1 width zeroes) binary ';' cr
                                                                            d2vLSZERO=result;' cr
  found = 0;' cr
                                                                          'end' cr
  for (index = 0; index <=' (par1 width - 1) decimal ';
                                                                          'endfunction // function d2vLSZERO' cr
index = index + 1)' cr
                                                                          exitindent cr cr
   begin' cr
     if (~found && par1[index]==1)' cr
                                                                         inline 30
     begin' cr
                                                                          generatefunction
     result=index;' cr
                                                                          'd2vLSZERO' (par1 width) decimal '(' markindent par 1
     found=1;' cr
                                                                         0 exitindent ')'
     end' cr
  if (~found) result =' (par1 width ) decimal ':' cr
                                                                          #UnaryOp Iszmask
   end' cr
  d2vLSONE=result;' cr
                                                                          " Least significant zero mask priority on single bit uses
'end' cr
                                                                         NOT:
 'endfunction // function d2vLSONE' cr
exitindent or or
                                                                          sources 'reg_bit'
                                                                          result 'reg_bit_c'
inline 30
generatefunction
                                                                          forward 1
'd2vLSONE' (par1 width) decimal '(' markindent par 1 0
exitindent ')'
                                                                          #UnaryOp Iszmask
```

```
" Least significant zero mask priority on single bit uses
NOT:
                                                                         function
                                                                           ' markindent
                                                                         'function [' (par1 width - 1) decimal ':0 ] d2vMAJ;' cr
sources 'reg_bit_c'
result 'reg_bit'
                                                                         'input [' (par1 width - 1) decimal ':0 ] par1;' cr
                                                                         ' // Equivalent of an IDaSS majority (maj) operator on a
forward 1
                                                                           (par1 width) decimal 'bits' cr
#UnaryOp Iszmask
                                                                         ' // word. The odd number of bits gives a single bit
" Least significant zero mask priority on bit vector uses
                                                                         result.' cr
function:
                                                                         'integer index,cnt0,cnt1;' cr
sources 'reg_bit_vector'
                                                                          'reg result;' cr
                                                                         'begin' cr
result 'reg_bit_vector'
                                                                           result =0;' cr
                                                                           cnt0 =0;' cr
function
                                                                            cnt1 =0:' cr
 ' markindent
'function [' (par1 width - 1) decimal ':0 ] d2vLSZMASK;'
                                                                            for (index = 0; index <=' (par1 width - 1) decimal ';
                                                                         index = index + 1)' cr
                                                                            if (par1[index]==1) cnt1=cnt1+1;' cr
'input [' (par1 width - 1) decimal ':0 ] par1;' cr
                                                                            else if (par1[index]==0) cnt0=cnt0+1;' cr
cr
' // Equivalent of an IDaSS least significant zero mask
                                                                            if (cnt1>cnt0) result=1; // 1 majority' cr
                                                                            else result=0; // 0 majority' cr
priority cr
 // (Iszmask) operator on a ' (par1 width) decimal ' bits
                                                                            d2vMAJ=result;' cr
word.' cr
                                                                          'end' cr
                                                                          'endfunction // function d2vMAJ' cr
'integer index;' cr
                                                                          exitindent cr cr
'reg [' (par1 width - 1) decimal ':0 ] result;' cr
'reg found;' cr
                                                                         inline 30
'begin' cr
                                                                         generatefunction
  result =' ((par1 width zeroes) width) decimal "'b'
                                                                          'd2vMAJ' (par1 width) decimal '(' markindent par 1 0
(par1 width zeroes) binary ';' cr
                                                                         exitindent ')'
  found = 0; cr
  for (index = 0; index <=' (par1 width - 1) decimal ';
                                                                          #UnaryOp maj
index = index + 1)' cr
  begin' cr
                                                                          " Majority operator on an EVEN number of bits:
     if (~found && par1[index]==0)' cr
     begin' cr
                                                                          sources 'reg_bit_vector'
    result[index]=1;' cr
                                                                          guard (par1 width \Lambda 1 = 0)
    found=1;' cr
                                                                          result 'reg_bit_vector'
     end' cr
  end' cr
                                                                          function
  d2vLSZMASK=result;' cr
                                                                           ' markindent
                                                                          'function [' (par1 width - 1) decimal ':0] d2vMAJ;' cr
'end' cr
'endfunction // function d2vLSZMASK' cr
                                                                          'input [' (par1 width - 1) decimal ':0 ] par1;' cr
exitindent cr cr
                                                                           // Equivalent of an IDaSS majority (maj) operator on a
inline 30
generatefunction
                                                                           (par1 width) decimal 'bits' cr
                                                                          ' // word. The even number of bits gives a two bit
 'd2vLSZMASK' (par1 width) decimal '(' markindent par
1 0 exitindent ')'
                                                                          result.' cr
#UnaryOp maj
                                                                          'integer index,cnt0,cnt1;' cr
                                                                          'reg [ 1:0 ] result;' cr
" Majority operator on a single bit is removed:
                                                                          'begin' cr
                                                                           result = 2"b00;' cr
sources 'reg_bit'
                                                                            cnt0 =0;' cr
result 'reg_bit'
                                                                            cnt1 =0;' cr
                                                                            for (index = 0; index <=' (par1 width - 1) decimal ';
                                                                          index = index + 1)' cr
forward 1
                                                                            if (par1[index]==1) cnt1=cnt1+1;' cr
                                                                            else if (par1[index]==0) cnt0=cnt0+1;' cr
 #UnaryOp maj
                                                                            if (cnt1==cnt0) result = 2"b00; // No majority' cr
 " Majority operator on a single bit is removed:
                                                                            if (cnt1>cnt0) result = 2"b10; // 1 majority' cr
                                                                            else result = 2"b01; // 0 majority' cr
 sources 'reg_bit_c'
                                                                            d2vMAJ=result;' cr
result 'reg_bit_c'
                                                                          'end' cr
                                                                          'endfunction // function d2vMAJ' cr
forward 1
                                                                          exitindent cr cr
 #UnaryOp maj
                                                                          inline 30
                                                                          generatefunction
 " Majority operator on an ODD number of bits:
                                                                          'd2vMAJ' (par1 width) decimal '(' markindent par 1 0
                                                                          exitindent ')'
 sources 'reg_bit_vector'
 guard (par1 width \Lambda 1 = 1)
                                                                          #UnaryOp msomask
 result 'reg_bit'
```

```
" Most significant one mask priority on single bit is
                                                                         result 'reg_bit'
removed:
                                                                         forward 1
sources 'reg_bit'
result 'reg_bit'
                                                                         #UnaryOp msone
forward 1
                                                                          Most significant one bit pos priority on bit vector uses
                                                                          function:
#UnaryOp msomask
                                                                          sources 'reg_bit_vector'
" Most significant one mask priority on single bit is
                                                                          result 'reg_bit_vector'
removed:
sources 'reg_bit_c'
                                                                          ' 'markindent
                                                                          'function [' (par1 width - 1) decimal ':0 ] d2vMSONE;' cr
result 'reg_bit_c'
                                                                          'input [' (par1 width - 1) decimal ':0 ] par1;' cr
                                                                         \operatorname{cr} ' // Equivalent of an IDaSS most significant one bit
forward 1
                                                                         position' cr
' // priority (msomask) operator on a ' (par1 width)
#UnaryOp msomask
" Most significant one mask priority on bit vector uses
                                                                          decimal
function:
                                                                           ' bits word.' cr
                                                                          'integer index,' cr
sources 'reg_bit_vector'
                                                                          'reg [' (par1 width - 1) decimal ':0] result;' cr
result 'reg_bit_vector'
                                                                          'reg found;' cr
                                                                          'begin' cr
function
                                                                           result =' ((par1 width zeroes) width) decimal "b'
  ' markindent
'function [' (par1 width - 1) decimal ':0]
                                                                          (par1 width zeroes) binary ';' cr
d2vMSOMASK;' cr
                                                                            found = 0; cr
                                                                            for (index =' (par1 width - 1) decimal '; index >=0;
'input [' (par1 width - 1) decimal ':0 ] par1;' cr
                                                                          index = index - 1)' cr
' // Equivalent of an IDaSS most significant one mask
                                                                            begin' cr
priority' cr
                                                                              if (~found && par1[index]==1)' cr
  // (msomask) operator on a ' (par1 width) decimal '
                                                                              begin' cr
bits word.' cr
                                                                              result=index;' cr
                                                                              found=1;' cr
                                                                              end' cr
'integer index;' cr
'reg [' (par1 width - 1) decimal ':0 ] result;' cr
                                                                            if (~found) result =' (par1 width ) decimal ';' cr
'rea found:' cr
                                                                            end' cr
                                                                            d2vMSONE=result;' cr
'begin' cr
  result =' ((par1 width zeroes) width) decimal "b'
(par1 width zeroes) binary ';' cr
                                                                          'endfunction // function d2vMSONE' cr
  found = 0; cr
                                                                          exitindent cr cr
   for (index = ' (par1 width - 1) decimal '; index >=0;
index = index - 1)' cr
                                                                          inline 30
                                                                          generatefunction
   begin' cr
     if (~found && par1[index]==1)' cr
                                                                          'd2vMSONE' (par1 width) decimal '(' markindent par 1 0
     begin' cr
                                                                          exitindent ')'
     result[index]=1;' cr
     end' cr
                                                                          #UnaryOp mszero
   end' cr
   d2vMSOMASK=result;' cr
                                                                          " Most significant zero bit pos priority on single bit is
'end' cr
                                                                          removed:
'endfunction // function d2vMSOMASK' cr
exitindent cr cr
                                                                          sources 'reg_bit'
                                                                          result 'reg_bit'
inline 30
generatefunction
                                                                          forward 1
'd2vMSOMASK' (par1 width) decimal '(' markindent par
1 0 exitindent')'
                                                                          #UnaryOp mszero
#UnaryOp msone
                                                                          " Most significant zero bit pos priority on single bit is
                                                                          removed:
 " Most significant one bit pos priority on single bit is
                                                                          sources 'reg_bit_c'
done with NOT:
                                                                          result 'reg_bit_c'
sources 'reg_bit'
result 'reg_bit_c'
                                                                          forward 1
forward 1
                                                                          #UnaryOp mszero
#UnaryOp msone
                                                                          " Most significant zero bit pos priority on bit vector uses
                                                                          function:
" Most significant one bit pos priority on single bit is
done with NOT:
                                                                          sources 'reg_bit_vector'
                                                                          result 'reg_bit_vector'
 sources 'reg_bit_c'
```

```
'reg [' (par1 width - 1) decimal ':0 ] result;' cr
function
 ' markindent
                                                                        'reg found;' cr
'function [' (par1 width - 1) decimal ':0 ] d2vMSZERO;'
                                                                        'begin' cr
                                                                         ' result =' ((par1 width zeroes) width) decimal "'b'
'input [' (par1 width - 1) decimal ':0 ] par1;' cr
                                                                         (par1 width zeroes) binary ';' cr
                                                                           found = 0;' cr
' // Equivalent of an IDaSS most significant zero bit
                                                                           for (index =' (par1 width - 1) decimal '; index >=0;
position' cr
                                                                         index = index - 1 )' cr
 // priority (mszero) operator on a ' (par1 width)
                                                                           begin' cr
decimal 'bits word.' cr
                                                                             if (~found && par1[index]==0)' cr
                                                                             begin' cr
                                                                             result[index]=1;' cr
'integer index' cr
'reg [' (par1 width - 1) decimal ':0 ] result;' cr
'reg found;' cr
                                                                             found=1;' cr
                                                                             end' cr
'begin' cr
                                                                           end' cr
 result =' ((par1 width zeroes) width) decimal "b'
                                                                           d2vMSZMASK=result;' cr
(par1 width zeroes) binary ';' cr
                                                                         'end' cr
                                                                         'endfunction // function d2vMSZMASK' cr
  found = 0; cr
  for (index =' (par1 width - 1) decimal '; index >=0;
                                                                         exitindent cr cr
index = index - 1)' cr
  begin' cr
                                                                         inline 30
    if (~found && par1[index]==0)' cr
                                                                         generatefunction
                                                                         'd2vMSZMASK' (par1 width) decimal '(' markindent par
    begin' cr
    result=index;' cr
                                                                         1 0 exitindent ')'
    found=1;' cr
    end' cr
                                                                         #UnaryOp neg
  if (~found) result =' (par1 width ) decimal ';' cr
  end' cr
                                                                         " 2's complement negate on bit vector done with NOT
  d2vMSZERO=result;' cr
                                                                         and add:
'end' cr
'endfunction // function d2vMSZERO' cr
                                                                         sources 'reg_bit_vector_c'
                                                                         result 'reg_bit_vector'
exitindent cr cr
inline 30
                                                                         inline 9
                                                                         par 19'+'cr
generatefunction
                                                                         ((1 width: par1 width) width) decimal "b' (1 width: par1
'd2vMSZERO' (par1 width) decimal '(' markindent par 1
0 exitindent ')'
                                                                         width) binary
#UnaryOp mszmask
                                                                         #UnaryOp neg
" Most significant zero mask priority on single bit uses
                                                                         " Negate on single bit is removed:
NOT:
                                                                         sources 'reg_bit'
sources 'reg_bit'
                                                                         result 'reg_bit'
result 'reg_bit_c'
                                                                         forward 1
forward 1
                                                                         #UnaryOp neg
#UnaryOp mszmask
                                                                         " Negate on single bit is removed:
" Most significant zero mask priority on single bit uses
NOT:
                                                                         sources 'reg_bit_c'
                                                                         result 'reg_bit_c'
sources 'reg_bit_c'
result 'reg_bit'
                                                                         forward 1
                                                                         #UnaryOp not
forward 1
 #UnaryOp mszmask
                                                                         " On single bit values:
 Most significant zero mask priority on bit vector uses
                                                                         sources 'reg_bit'
 function:
                                                                         result 'reg_bit_c'
sources 'reg_bit_vector'
                                                                         forward 1
result 'reg_bit_vector'
                                                                         #UnaryOp not
 function
  ' markindent
                                                                         " On single bit values:
 'function [' (par1 width - 1) decimal ':0 ] d2vMSZMASK;'
                                                                         sources 'reg_bit_c'
 'input [' (par1 width - 1) decimal ':0 ] par1;' cr
                                                                         result 'reg_bit'
cr
' // Equivalent of an IDaSS most significant zero mask
                                                                         forward 1
 priority' cr
  // (mszmask) operator on a ' (par1 width) decimal '
                                                                         #UnaryOp not
 bits word.' cr
                                                                         On complete bit vectors:
 'integer index;' cr
```

```
sources 'reg_bit_vector'
                                                                        " Generate field of constant ones:
result 'reg_bit_vector_c'
forward 1
                                                                        sources 'constant'
                                                                        guard (par1 > 1)
#UnaryOp not
                                                                        result 'reg_bit_vector'
" On complete bit vectors:
                                                                        inline 30
                                                                        ((par1 ones) width) decimal "b' (par1 ones) binary
sources 'reg_bit_vector_c'
result 'reg_bit_vector'
                                                                        #UnaryOp opty
forward 1
                                                                        " Odd parity on a single bit, removed:
#UnaryOp onecnt
                                                                        sources 'reg_bit'
                                                                        result 'reg_bit'
" Count number of %1 bits in a single bit is removed:
                                                                        forward 1
sources 'reg bit'
result 'reg_bit'
                                                                        #UnaryOp opty
forward 1
                                                                        " Odd parity on a single bit, removed:
#UnaryOp onecnt
                                                                        sources 'reg_bit_c'
                                                                        result 'reg_bit_c'
" Count number of %1 bits in a single bit is removed:
                                                                        forward 1
sources 'reg_bit_c'
result 'reg_bit_c'
                                                                        #UnaryOp opty
forward 1
                                                                        " Odd parity on two bits, done with XOR:
#UnaryOp onecnt
                                                                        sources 'reg_bit_vector'
                                                                        guard (par1 width = 2)
" Count number of %1 bits in bit vector needs function:
                                                                        result 'reg_bit'
sources 'reg_bit_vector'
                                                                        inline 4
result 'reg_bit_vector'
                                                                        par 1 31 '[0] ^' cr
                                                                        par 1 31 '[1]'
function
 ' markindent
                                                                        #UnaryOp opty
'function [' (par1 width - 1) decimal ':0 ] d2vONECNT;'
                                                                        " Odd parity on > 2 bits, uses function:
'input [' (par1 width - 1) decimal ':0 ] par1;' cr
cr
                                                                        sources 'reg_bit_vector'
' // Equivalent of an IDaSS count number of ones
                                                                        guard (par1 width > 2)
(onecnt) operator' cr
                                                                        result 'reg_bit'
 // on a ' (par1 width) decimal ' bits word.' cr
                                                                        function
'integer index,count;' cr
                                                                          ' markindent
                                                                         'function [' (par1 width - 1) decimal ':0 ] d2vOPTY;' cr
 'begin' cr
  count =' ((par1 width zeroes) width) decimal "b'
                                                                        'input [' (par1 width - 1) decimal ':0 ] par1;' cr
(par1 width zeroes) binary ';' cr
                                                                        ' // Equivalent of an IDaSS odd parity (opty) operator'
  for (index = 0; index <=' (par1 width - 1) decimal ';
index = index + 1)' cr
                                                                        cr ' // on a ' (par1 width) decimal ' bits word.' cr
  if (par1[index]==1) count=count+1;' cr
  d2vONECNT=count;' cr
'end' cr
                                                                        'integer index;' cr
'endfunction // function d2vONECNT' cr
                                                                        'reg result;' cr
exitindent cr cr
                                                                        'begin' cr
                                                                           result =0;' cr
inline 30
                                                                           for (index = 0; index <=' (par1 width - 1) decimal ';
generatefunction
                                                                        index = index + 1)' cr
'd2vONECNT' (par1 width) decimal '(' markindent par 1
                                                                           result = result ^par1[index];' cr
0 exitindent ')'
                                                                           d2vOPTY=result;' cr
                                                                         'end' cr
#UnaryOp ones
                                                                         'endfunction // function d2vOPTY' cr
                                                                        exitindent cr cr
" Generate constant one bit:
                                                                        inline 30
                                                                        generatefunction
sources 'constant'
guard (par1 = 1)
                                                                        'd2vOPTY' (par1 width) decimal '(' markindent par 1 0
result 'reg_bit'
                                                                        exitindent ')'
inline 30
                                                                        #UnaryOp rev
'1'
                                                                         * Reverse all bits in a word, removed for single bit:
#UnaryOp ones
```

```
sources 'reg_bit'
                                                                       result 'reg_bit'
result 'reg_bit'
                                                                       forward 1
forward 1
#UnaryOp rev
                                                                       #UnaryOp zerocnt
" Reverse all bits in a word, removed for single bit:
                                                                       " Count number of %0 bits in bit vector needs function:
                                                                       sources 'reg_bit_vector'
sources 'reg_bit_c'
result 'reg_bit_c'
                                                                       result 'reg_bit_vector'
forward 1
                                                                       function
                                                                        ' ' markindent
#UnaryOp rev
                                                                        'function [' (par1 width - 1) decimal ':0 ] d2vZEROCNT;'
" Reverse all bits in a bit vector:
                                                                       'input [' (par1 width - 1) decimal ':0 ] par1;' cr
sources 'reg_bit_vector'
                                                                       ' // Equivalent of an IDaSS count number of zeroes
                                                                       (zerocnt) operator' cr
result 'reg_bit_vector'
                                                                        // on a ' (par1 width) decimal ' bits word.' cr
inline 3
                                                                        'integer index,count;' cr
FOR (par1 width - 1)
                                                                        'begin' cr
                                                                        ' count =' ((par1 width zeroes) width) decimal ""b'
DO
par 1 31 '[' (FORCNT - 1) decimal '], '
                                                                       (par1 width zeroes) binary ';' cr
IF (FORCNT \Lambda 3 = 0)
                                                                         for (index = 0; index <=' (par1 width - 1) decimal ';
THEN
                                                                        index = index + 1)' cr
                                                                         if (par1[index]==0) count=count+1; cr
 cr
ELSE
                                                                          d2vZEROCNT=count;' cr
ENDIF
                                                                        'endfunction // function d2vZEROCNT' cr
ENDFOR
                                                                       exitindent cr cr
par 1 31 '[' (par1 width - 1) decimal ']'
                                                                       inline 30
                                                                        generatefunction
#UnaryOp rev
                                                                        'd2vZEROCNT' (par1 width) decimal '(' markindent par
                                                                        1 0 exitindent ')'
" Reverse all bits in a bit vector:
                                                                       #UnaryOp zeroes
sources 'reg_bit_vector_c'
result 'reg_bit_vector_c'
                                                                        " Generate constant zero bit:
inline 3
                                                                       sources 'constant'
                                                                       guard (par1 = 1)
FOR (par1 width - 1)
                                                                       result 'reg_bit'
DO
par 1 31 '[' (FORCNT - 1) decimal '], '
                                                                       inline 30
IF (FORCNT \wedge 3 = 0)
                                                                        ω,
THÈN
 cr
                                                                        #UnaryOp zeroes
ELSE
                                                                        " Generate field of constant zeroes:
ENDIF
ENDFOR
                                                                       sources 'constant'
par 1 31 '[' (par1 width - 1) decimal ']'
                                                                       guard (par1 > 1)
 `}'
                                                                       result 'reg_bit_vector'
"#UnaryOp width
                                                                       inline 30
                                                                        ((par1 zeroes) width) decimal "b' (par1 zeroes) binary
" Check width of a word, never seen here...
#UnaryOp zerocnt
                                                                                       BINARY OPERATORS
" Count number of %0 bits in a single bit is done with
                                                                        #MacroFor BinaryOp
NOT:
                                                                       naming
sources 'reg_bit'
                                                                        'multiply_signed'
result 'reg_bit_c'
                                                                       contents
forward 1
                                                                        'function [' ((par1 width) + (par2 width) -1) decimal ':0]
                                                                        multiply_signed;' cr
#UnaryOp zerocnt
                                                                         "This is the signed multiply function for HDL-Verilog
" Count number of %0 bits in a single bit is done with
NOT:
                                                                         'input [' (par1 width - 1) decimal ':0 ] par1;' cr
                                                                         'input [' (par2 width - 1) decimal ':0 ] par2;' cr
sources 'reg_bit_c'
                                                                         'integer index;' cr
```

```
'reg [' ((par1 width) + (par2 width) -1) decimal ':0 ]
                                                                        (par1 width) decimal "b' (par2 width: par1 width) binary
result:' cr
 'reg [' (par1 width - 1) decimal ':0 ] lowbit;' cr
                                                                        #BinaryOp +
 'reg [' (par1 width + 1) decimal ':0 ] accu;' cr
                                                                        " Add bit vector and zero constant, removed:
cr
 'accu =' ((par1 width) + 1) decimal "'b' (((par1 width) +
                                                                        sources 'reg_bit_vector' 'constant'
1) zeroes ) binary ';' cr
                                                                        guard (par2 = 0)
 'for (index = 0; index <=' (par1 width - 1) decimal ';
                                                                        result 'reg_bit_vector'
index = index + 1)' cr
  begin' cr
                                                                        forward 1
   if (par2[index] ==1)' cr
                                                                        #BinaryOp +
    begin' cr
      if (index == ' (par1 width - 1) decimal ')' cr
         accu= accu - {par1 [' (par1 width - 1) decimal
                                                                        " Add bit vector and zero constant, removed:
'], par1};' cr
         else' cr
                                                                        sources 'reg_bit_vector_c' 'constant'
         accu= accu + {par1 [' (par1 width - 1) decimal
                                                                        guard (par2 = 0)
                                                                        result 'reg_bit_vector_c'
'], par1};' cr
      end' cr
    end' cr
                                                                        forward 1
    lowbit [index] = accu[0];' cr
    accu={accu[' (par1 width ) decimal '], accu[' (par1
                                                                         #BinaryOp +
width) decimal ':1]};' cr
                                                                        " Add non-zero constant and bit vector:
   end' cr
 'result={accu[' (par1 width - 1) decimal ':0], lowbit};' cr
 'multiply_signed=result;' cr
                                                                        sources 'constant' 'reg_bit_vector'
 'end' cr
                                                                        guard (par1 \sim= 0)
 'endfunction // function multiply_signed;' cr
                                                                        result 'reg_bit_vector'
                                                                        (par2 width) decimal "b' (par1 width: par2 width) binary
#MacroFor BinaryOp
                                                                         ' +' cr
naming
                                                                         par 2 10
'gentempvar1'
                                                                         #BinaryOp +
contents
                                                                         " Add zero constant and bit vector, removed:
'reg '
  IF (par1 width > 1)
  THEN
                                                                         sources 'constant' 'reg_bit_vector'
   '[' (par1 width - 1 width: 8) decimalleft ':0]'
                                                                         guard (par1 = 0)
  ENDIF
                                                                         result 'reg_bit_vector'
'd2vTEMP' (INDEX) decimal ';' tabto 54
                                                                         forward 2
#MacroFor BinaryOp
                                                                         #BinaryOp +
naming
'gentempvar2'
                                                                         " Add zero constant and bit vector, removed:
                                                                         sources 'constant' 'reg_bit_vector_c'
contents
                                                                         guard (par1 = 0)
reg
  IF (par2 width > 1)
                                                                         result 'reg_bit_vector_c'
   THEN
   '[ ' (par2 width - 1 width: 8) decimalleft ':0 ]'
                                                                         forward 2
'd2vTEMP' (INDEX) decimal ';' tabto 54
                                                                         #BinaryOp +
                                                                         " Add single bit values, converted to XOR:
#BinaryOp +
" Add bit vectors, standard work:
                                                                         sources 'reg_bit' 'reg_bit'
                                                                         result 'reg_bit'
sources 'reg_bit_vector' 'reg_bit_vector'
result 'reg_bit_vector'
                                                                         inline 4
                                                                         par 1 4 ' ^' cr
inline 9
                                                                         par 2 5
par 1 9 ' +' cr
par 2 10
                                                                         #BinaryOp +
#BinaryOp +
                                                                         " Add single bit and Constant %0, removed:
                                                                         sources 'reg_bit' 'constant'
" Add bit vector and non-zero constant:
                                                                         guard (par2 = 0)
sources 'reg_bit_vector' 'constant'
                                                                         result 'reg_bit'
guard (par2 ~= 0)
result 'reg_bit_vector'
                                                                         forward 1
inline 9
                                                                         #BinaryOp +
par 19'+'cr
```

```
" Add single bit and Constant %0, removed:
                                                                       #BinaryOp -
sources 'reg_bit_c' 'constant'
guard (par2 = 0)
                                                                       " Subtract bit vector and non-zero constant:
result 'reg_bit_c'
                                                                       sources 'reg_bit_vector' 'constant'
forward 1
                                                                       guard (par2 ~= 0)
                                                                       result 'reg_bit_vector'
#BinaryOp +
                                                                       inline 9
" Add single bit and Constant %1, converted to NOT:
                                                                       par 19'-'cr
                                                                       (par1 width) decimal "b' (par2 width: par1 width) binary
sources 'reg_bit' 'constant'
guard (par2 = 1)
                                                                       #BinaryOp -
result 'reg_bit_c'
                                                                        " Subtract bit vector and zero constant, removed:
forward 1
                                                                       sources 'reg_bit_vector' 'constant'
#BinaryOp +
                                                                       guard (par2 = 0)
                                                                       result 'reg_bit_vector'
" Add single bit and Constant %1, converted to NOT:
                                                                       forward 1
sources 'reg_bit_c' 'constant'
guard (par2 = 1)
                                                                        #BinaryOp -
result 'reg_bit'
                                                                        " Subtract bit vector and zero constant, removed:
forward 1
                                                                       sources 'reg_bit_vector_c' 'constant'
#BinaryOp +
                                                                        guard (par2 = 0)
                                                                        result 'reg_bit_vector_c'
* Add Constant %0 and single bit, removed:
                                                                       forward 1
sources 'constant' 'reg_bit'
guard (par1 = 0)
                                                                        #BinaryOp -
result 'reg_bit'
                                                                        " Subtract constant and bit vector:
forward 2
                                                                        sources 'constant' 'reg_bit_vector'
#BinaryOp +
                                                                        result 'reg_bit_vector'
" Add Constant %0 and single bit, removed:
                                                                        inline 9
                                                                        (par2 width) decimal "b' (par1 width: par2 width) binary
sources 'constant' 'reg_bit_c'
                                                                        ' -' cr
guard (par1 = 0)
                                                                       par 2 10
result 'reg_bit_c'
                                                                        #BinaryOp -
forward 2
                                                                        " Subtract single bit values, converted to XOR:
#BinaryOp +
                                                                        sources 'reg_bit' 'reg_bit'
" Add Constant %1 and single bit, converted to NOT:
                                                                        result 'reg_bit'
sources 'constant' 'reg_bit'
                                                                        inline 4
guard (par1 = 1)
                                                                        par 1 4 ' ^' cr
result 'reg_bit_c'
                                                                        par 2 5
forward 2
                                                                        #BinaryOp -
#BinaryOp +
                                                                        " Subtract single bit and Constant %0, removed:
" Add Constant %1 and single bit, converted to NOT:
                                                                        sources 'reg_bit' 'constant'
                                                                        guard (par2 = 0)
sources 'constant' 'reg_bit_c'
                                                                        result 'reg_bit'
guard (par1 = 1)
result 'reg_bit'
                                                                        forward 1
forward 2
                                                                        #BinaryOp -
#BinaryOp -
                                                                        " Subtract single bit and Constant %0, removed:
" Subtract bit vectors, standard work:
                                                                        sources 'reg_bit_c' 'constant'
                                                                        guard (par2 = 0)
sources 'reg_bit_vector' 'reg_bit_vector'
                                                                        result 'reg_bit_c'
result 'reg_bit_vector'
                                                                        forward 1
inline 9
par 1 9 ' - ' cr
                                                                        #BinaryOp -
par 2 10
```

```
" Subtract single bit and Constant %1, converted to
                                                                        sources 'reg_bit_vector' 'reg_bit_vector'
NOT:
                                                                        result 'reg_bit_vector'
sources 'reg_bit' 'constant'
                                                                        " Assume the standard library supports this:
guard (par2 = 1)
result 'reg_bit_c'
                                                                        inline 10
                                                                                  *' cr
                                                                        par 1 10 '
forward 1
                                                                        par 2 11
#BinaryOp -
                                                                        #BinaryOp *
" Subtract single bit and Constant %1, converted to
                                                                        " Multiply unsigned single bit & constant,
NOT:
                                                                        " constant value 0 would not be visible here:
sources 'reg_bit_c' 'constant'
                                                                        sources 'reg_bit' 'constant'
guard (par2 = 1)
result 'reg_bit'
                                                                        guard (par2 = 1)
                                                                        result 'reg_bit'
forward 1
                                                                        forward 1
#BinaryOp -
                                                                        #BinaryOp *
" Subtract Constant %0 and single bit, removed:
                                                                        " Multiply unsigned single bit & constant,
sources 'constant' 'reg_bit'
                                                                        " constant value 0 would not be visible here:
guard (par1 = 0)
result 'reg_bit'
                                                                        sources 'reg_bit_c' 'constant'
                                                                        guard (par2 = 1)
                                                                        result 'reg_bit_c'
forward 2
#BinaryOp -
                                                                        forward 1
" Subtract Constant %0 and single bit, removed:
                                                                        #BinaryOp *
sources 'constant' 'reg_bit_c'
                                                                        " Multiply unsigned bit vector & constant,
guard (par1 = 0)
                                                                        " constant value 0 would not be visible here:
result 'reg_bit_c'
                                                                        sources 'reg_bit_vector' 'constant'
forward 2
                                                                        guard (par2 = 1)
                                                                        result 'reg_bit_vector'
#BinaryOp -
                                                                        forward 1
" Subtract Constant %1 and single bit, converted to
NOT:
                                                                        #BinaryOp *
sources 'constant' 'reg_bit'
                                                                        " Multiply unsigned bit vector & constant,
                                                                        " constant value 0 would not be visible here:
guard (par1 = 1)
result 'reg_bit_c'
                                                                        sources 'reg_bit_vector_c' 'constant'
forward 2
                                                                        guard (par2 = 1)
                                                                        result 'reg_bit_vector_c'
#BinaryOp -
                                                                        forward 1
* Subtract Constant %1 and single bit, converted to
NOT:
                                                                        #BinaryOp *
sources 'constant' 'reg_bit_c'
                                                                        " Multiply unsigned bit vector & constant (> 1, 2**N),
guard (par1 = 1)
                                                                        " constant value 0 would not be visible here, this one
                                                                        " can be converted into a shift left:
result 'reg_bit'
forward 2
                                                                        sources 'reg_bit_vector' 'constant'
                                                                         guard (par2 > 1 \Lambda
#BinaryOp *
                                                                              ((par2 width: par1 width) onecnt = 1))
                                                                         result 'reg_bit_vector'
" Multiply unsigned on single bits, result can only be 0
or 1:
                                                                        inline 30
                                                                        par 1 31 '{[ '
sources 'reg_bit' 'reg_bit'
                                                                          (par1 width - (par2 width: par1 width) Isone - 1)
result 'reg_bit_vector'
                                                                         decimal
                                                                          IF (par1 width - (par2 width: par1 width) Isone > 1)
inline 30
                                                                          THEN ':0'
 '{0.' cr
                                                                          ENDIF
'(' markindent par 1 5 exitindent ' &' cr
                                                                          `],'cr
                                                                         IF (par2 = 2)
  markindent par 2 6 exitindent ')}'
                                                                         THEN '0}'
#BinaryOp *
                                                                         ELSE (((par2 width: par1 width) Isone asBIC zeroes)
                                                                         width) decimal "b' ((par2 width: par1 width) Isone
 " Multiply unsigned bit vectors:
                                                                         asBIC zeroes) binary '}'
```

```
ENDIF
                                                                         '],'cr
                                                                        IF (par2 at: 1)
#BinaryOp *
                                                                        THEN '0}'
                                                                        ELSE ((((par2 width: par1 width) Isone asBIC +1)
                                                                        zeroes) width) decimal "'b' (((par2 width: par1 width)
" Multiply unsigned bit vector & constant (> 1, ~= 2**N),
" constant LSB is %1:
                                                                        Isone asBIC +1) zeroes) binary '}'
                                                                        ENDIF
sources 'reg_bit_vector' 'constant'
guard (par2 > 1 A
                                                                        #BinaryOp *
     ((par2 width: par1 width) onecnt > 1) ∧
      (par2 at: 0))
                                                                        " Multiply unsigned constant & single bit,
result 'reg_bit_vector'
                                                                        " constant value 0 would not be visible here:
tempvar 1
                                                                        sources 'constant' 'reg bit'
'reg' tabto 21
'[' (par1 width +
                                                                        guard (par1 = 1)
                                                                        result 'reg_bit'
 (par2 width: par1 width) msone width: 8) decimalleft
 ':0] d2vTEMP' (INDEX) decimal '; // For unsigned
                                                                        forward 2
vector/constant multiply' cr
                                                                        #BinaryOp *
separate
 d2vTEMP' (INDEX) decimal ' =' cr
                                                                        " Multiply unsigned constant & single bit,
   'markindent par 1 10 ' *' cr
                                                                        " constant value 0 would not be visible here:
((par2 width: (par2 width: par1 width) msone asBIC + 1)
width) decimal "b' (par2 width: (par2 width: par1 width)
                                                                        sources 'constant' 'reg_bit_c'
msone asBIC + 1) binary
                                                                        guard (par1 = 1)
 '; // Active constant bit range only...' exitindent cr
                                                                        result 'reg_bit_c'
" Remove excess bits with slicing:
                                                                        forward 2
inline 30
                                                                        #BinaryOp *
generatetempvars
generateseparate
                                                                        " Multiply unsigned constant & bit vector,
'd2vTEMP' (INDEX) decimal '[' (par1 width - 1)
                                                                         constant value 0 would not be visible here:
decimal ':0]'
                                                                        sources 'constant' 'reg_bit_vector'
#BinaryOp 1
                                                                        quard (par1 = 1)
                                                                        result 'reg_bit_vector'
" Multiply unsigned bit vector & constant (> 1, ~= 2**N),
" constant contains LS zeroes:
                                                                        forward 2
sources 'reg_bit_vector' 'constant'
                                                                        #BinaryOp *
guard (par2 > 1 \Lambda
      ((par2 width: par1 width) onecnt > 1) \( \Lambda \)
                                                                        " Multiply unsigned constant & bit vector,
      (par2 at: 0) not)
                                                                         constant value 0 would not be visible here:
result "reg_bit_vector"
                                                                        sources 'constant' 'reg_bit_vector_c'
tempvar 1
                                                                        guard (par1 = 1)
'reg' tabto 21
                                                                        result 'reg_bit_vector_c'
  (par1 width + (par2 width: par1 width) msone -
                                                                        forward 2
  (par2 width: par1 width) Isone width: 8) decimalleft
  :0 ] d2vTEMP' (INDEX) decimal '; // For unsigned
                                                                        #BinaryOp *
vector/constant multiply' cr
                                                                        * Multiply unsigned constant (> 1, 2**N) & bit vector,
                                                                         constant value 0 would not be visible here, this one
separate
 d2vTEMP' (INDEX) decimal ' =' cr
                                                                        " can be converted into a shift left:
   'markindent par 1 10 ' " cr
((par2 from: (par2 width: par1 width) Isone asBIC
                                                                        sources 'constant' 'reg_bit_vector'
     to: (par2 width: par1 width) msone asBIC) width)
                                                                        guard (par1 > 1 \Lambda
decimal "b' (par2 from: (par2 width: par1 width) Isone
                                                                              ((par1 width: par2 width) onecnt = 1))
asBIC
                                                                        result 'reg_bit_vector'
         (par2 width: par1 width) msone asBIC) binary
 '; // Active constant bit range only...' exitindent cr
                                                                        inline 30
                                                                        par 1 31 '{[ '
" Remove excess bits with slicing and append constant
                                                                         (par2 width - (par1 width: par2 width) Isone - 1)
zero(es):
                                                                         IF (par2 width - (par1 width: par2 width) Isone > 1)
inline 30
                                                                         THEN ':0'
generatetempyars
                                                                         ENDIF
generateseparate
                                                                          '],'cr
                                                                        IF (par1 = 2)
'd2vTEMP' (INDEX) decimal
]}'
                                                                        THEN '0}'
 (par1 width - (par2 width: par1 width) Isone - 1)
                                                                        ELSE (((par1 width: par2 width) Isone asBIC zeroes)
decimal
                                                                        width) decimal "b' ((par1 width: par2 width) Isone
 IF (par1 width - (par2 width: par1 width) Isone ~= 1)
                                                                        asBIC zeroes) binary '}'
 THEN ':0'
                                                                        ENDIF
 ENDIF
```

```
THEN ':0'
#BinaryOp *
                                                                        ENDIF
" Multiply unsigned constant (> 1, ~= 2**N) & bit vector,
                                                                         ] ,' cr
                                                                       IF (par1 at: 1)
* constant LSB is %1:
                                                                       THEN '0}'
                                                                       ELSE ((((par1 width: par2 width) Isone asBIC +1)
sources 'constant' 'reg_bit_vector'
guard (par1 > 1 \Lambda
                                                                       zeroes) width) decimal "b' (((par1 width: par2 width)
     ((par1 width: par2 width) onecnt > 1) \Lambda
                                                                       Isone asBIC +1) zeroes) binary '}'
      (par1 at: 0))
                                                                       ENDIF
result 'reg_bit_vector'
                                                                       #BinaryOp *+
tempvar 1
'reg' tabto 21
                                                                       " Multiply right hand signed only on single bits, result
 '[' (par2 width +
                                                                       can only be
 (par1 width: par2 width) msone width: 8) decimalleft
                                                                        " 0 or -1:
 ':0] d2vTEMP' (INDEX) decimal '; // For unsigned
                                                                       sources 'reg_bit' 'reg_bit'
constant/vector multiply' cr
                                                                       result 'reg_bit_vector'
separate
 d2vTEMP' (INDEX) decimal ' =' cr
                                                                       tempvar 1
' 'markindent
                                                                       expandmacro 'gentempvar1' '; // For RHS signed
((par1 width: (par1 width: par2 width) msone asBIC + 1)
                                                                       multipy on bits' cr
width) decimal "b' (par1 width: (par1 width: par2 width)
msone asBIC + 1) binary
                                                                       separate
                                                                         d2vTEMP' (INDEX) decimal ' =' cr
par 2 11 exitindent '; // Active constant bit range only...'
                                                                          ' markindent
                                                                        par 1 5 ' &' cr
                                                                        par 2 6 exitindent ';' cr
" Remove excess bits with slicing:
                                                                       inline 3
inline 30
                                                                       generatetempvars
generatetempvars
                                                                       generateseparate
                                                                        '{ d2vTEMP' (INDEX) decimal ' ,' cr
generateseparate
'd2vTEMP' (INDEX) decimal ' [ ' (par2 width - 1)
                                                                       'd2vTEMP' (INDEX) decimal '}'
decimal ':0]'
                                                                       #BinaryOp *+
#BinaryOp *
                                                                       " Multiply RHS signed bit vectors:
" Multiply unsigned constant (> 1, ~= 2**N) & bit vector,
" constant contains LS zeroes:
                                                                       sources 'reg_bit_vector' 'reg_bit_vector'
                                                                       result 'reg_bit_vector'
sources 'constant' 'reg_bit_vector'
guard (par1 > 1 \Lambda
                                                                       tempvar 1
                                                                       'reg' tabto 21
      ((par1 width: par2 width) onecnt > 1) ∧
      (par1 at: 0) not)
                                                                         [' (par1 width + par2 width width: 8) decimalleft
                                                                        ':0 ] d2vTEMP' (INDEX) decimal '; // For RH signed
result 'reg_bit_vector
                                                                       multiply' cr
tempvar 1
                                                                       " Use signed multiply defined in 'extra_functions'
'reg' tabto 21
                                                                       package. Add
                                                                         extra '0' bit to LHS to force it positive:
  (par2 width + (par1 width: par2 width) msone -
  (par1 width: par2 width) Isone width: 8) decimalleft
 ':0] d2vTEMP' (INDEX) decimal '; // For unsigned
                                                                       separate
constant/vector multiply' cr
                                                                       setglobal %100
                                                                         d2vTEMP' (INDEX) decimal ' =' cr
separate
                                                                          multiply_signed(' markindent
 d2vTEMP' (INDEX) decimal ' =' cr
                                                                        '{0,' par 1 30 '},' cr
   ' markindent
                                                                        par 2 0 exitindent ');' cr
((par1 from: (par1 width: par2 width) Isone asBIC
     to: (par1 width: par2 width) msone asBIC) width)
                                                                       " Remove excess bits with slicing:
decimal "b' (par1 from: (par1 width: par2 width) Isone
asBIC
                                                                       inline 30
                                                                       generatetempvars
     to: (par1 width: par2 width) msone asBIC) binary
' *' cr
                                                                        generateseparate
par 1 11 exitindent '; // Active constant bit range only...'
                                                                        'd2vTEMP' (INDEX) decimal ' [ ' (par1 width + par2
                                                                        width - 1) decimal ':0 ]'
"Remove excess bits with slicing and append constant
                                                                        #BinaryOp *+
zero(es):
                                                                        " Multiply RHS signed constant & single bit,
inline 30
                                                                        " constant value 0 would not be visible here:
generatetempvars
                                                                       sources 'constant' 'reg_bit'
generateseparate
'd2vTEMP' (INDEX) decimal
                                                                        guard (par1 = 1)
]}'
                                                                       result 'reg_bit'
 (par2 width - (par1 width: par2 width) Isone - 1)
decimal
                                                                        forward 2
 IF (par2 width - (par1 width: par2 width) Isone ~= 0)
```

```
#BinaryOp *+
                                                                       " Use signed multiply defined in 'extra_functions'
                                                                       package. Add
" Multiply RHS signed constant & single bit,
                                                                         extra '0' bit to constant to make it positive:
" constant value 0 would not be visible here:
                                                                       separate
sources 'constant' 'reg_bit_c'
                                                                       setglobal %100
                                                                         d2vTEMP' (INDEX) decimal ' =' cr
quard (par1 = 1)
result 'reg_bit_c'
                                                                          multiply_signed(' markindent
                                                                       ((par1 width: (par1 width: par2 width) msone asBIC + 1)
                                                                       width +1) decimal "b' (par1 width: (par1 width: par2
forward 2
                                                                       width) msone asBIC + 1 + 1) binary
#BinaryOp *+
                                                                        par 2 0 exitindent '); // Active constant bit range only...'
" Multiply RHS signed constant & bit vector,
" constant value 0 would not be visible here:
                                                                       " Remove excess bits with slicing:
sources 'constant' 'reg_bit_vector'
guard (par1 = 1)
result 'reg_bit_vector'
                                                                       generatetempvars
                                                                       generateseparate
                                                                        'd2vTEMP' (INDEX) decimal ' [ ' (par2 width - 1)
forward 2
                                                                       decimal ':0 l'
#BinaryOp *+
                                                                        #BinaryOp *+
" Multiply RHS signed constant & bit vector.
" constant value 0 would not be visible here:
                                                                        " Multiply RHS signed constant (> 1, ~= 2**N) & bit
sources 'constant' 'reg_bit_vector_c'
                                                                        " constant contains LS zeroes:
guard (par1 = 1)
result 'reg_bit_vector_c'
                                                                       sources 'constant' 'reg_bit_vector'
                                                                       guard (par1 > 1 \Lambda
                                                                              ((par1 width: par2 width) onecnt > 1) \Lambda
forward 2
                                                                              (par1 at: 0) not)
#BinaryOp *+
                                                                        result 'reg_bit_vector'
" Multiply RHS signed constant (> 1, 2**N) & bit vector,
                                                                       tempvar 1
" constant value 0 would not be visible here, this one
                                                                        'reg [
" can be converted into a shift left:
                                                                          (par2 width + (par1 width: par2 width) msone asBIC -
                                                                          (par1 width: par2 width) Isone asBIC + 1 width: 8)
sources 'constant' 'reg_bit_vector'
                                                                        decimalleft
                                                                          :0 ] d2vTEMP' (INDEX) decimal tabto 54 '; // For RH
guard (par1 > 1 \Lambda
      ((par1 width: par2 width) onecnt = 1))
                                                                        signed constant/vector multiply' cr
result 'reg_bit_vector'
                                                                        " Use signed multiply defined in 'extra_functions'
inline 30
                                                                        package. Add
par 1 31 '{[
                                                                         extra '0' bit to constant to make it positive:
 (par2 width - (par1 width: par2 width) Isone - 1)
                                                                        separate
                                                                        setglobal %100
 IF (par2 width - (par1 width: par2 width) Isone > 1)
 THEN ':0'
                                                                         d2vTEMP' (INDEX) decimal ' =' cr
 ENDIF
                                                                           multiply_signed(' markindent
                                                                        '{0,' ((par1 from: (par1 width: par2 width) Isone asBIC
  '],'cr
IF (par1 = 2)
                                                                             to: (par1 width: par2 width) msone asBIC) width)
                                                                        decimal "b' (par1 from: (par1 width: par2 width) Isone
THEN '0}'
ELSE (((par1 width: par2 width) Isone asBIC zeroes)
                                                                        asBIC
width) decimal "b' ((par1 width: par2 width) Isone
                                                                             to: (par1 width: par2 width) msone asBIC) binary
asBIC zeroes) binary '}'
                                                                         '} ,' cr
ENDIF
                                                                        par 2 0 exitindent '); // Active constant bit range only...'
                                                                        cr
#BinaryOp *+
                                                                        inline 30
" Multiply RHS signed constant (> 1, ~= 2**N) & bit
                                                                        generatetempvars
vector,
                                                                        generateseparate
" constant LSB is %1:
                                                                        'd2vTEMP' (INDEX) decimal
                                                                        '[[ '
                                                                         (par2 width -1 - (par1 width: par2 width) Isone asBIC)
sources 'constant' 'reg_bit_vector'
guard (par1 > 1 \Lambda
                                                                        decimal
      ((par1 width: par2 width) onecnt > 1) \Lambda
                                                                         ':0],' cr
      (par1 at: 0))
                                                                        IF (par1 at: 1)
result 'reg_bit_vector'
                                                                        THEN '0}
                                                                        ELSE (((par1 width: par2 width) Isone asBIC zeroes)
tempvar 1
                                                                        width) decimal "b' ((par1 width: par2 width) Isone
reg [
                                                                        asBIC zeroes) binary '}'
  (par2 width + (par1 width: par2 width) msone +1
                                                                        ENDIF
width: 8) decimalleft
  :0 ] d2vTEMP' (INDEX) decimal tabto 54 '; // For RH
                                                                        #BinaryOp +*
signed constant/vector multiply' cr
```

```
" Multiply left hand signed only on single bits, result can
                                                                        " Multiply LHS signed bit vector & constant,
                                                                        " constant value 0 would not be visible here:
only be
" 0 or -1:
                                                                        sources 'reg_bit_vector' 'constant'
sources 'reg_bit' 'reg_bit'
                                                                        guard (par2 = 1)
result 'reg_bit_vector'
                                                                        result 'reg_bit_vector'
                                                                        forward 1
tempvar 1
expandmacro 'gentempvar1' '; // For LHS signed
multipy on bits' cr
                                                                        #BinaryOp +*
                                                                        " Multiply LHS signed bit vector & constant,
separate
 d2vTEMP' (INDEX) decimal ' =' cr
                                                                          constant value 0 would not be visible here:
   ' markindent
 par 1 5 ' &' cr
                                                                        sources 'reg_bit_vector_c' 'constant'
 par 2 6 exitindent ';' cr
                                                                        guard (par2 = 1)
                                                                        result 'reg_bit_vector_c'
inline 3
generatetempvars
                                                                        forward 1
generateseparate
'{ d2vTEMP' (INDEX) decimal ',' cr 'd2vTEMP' (INDEX) decimal '}'
                                                                         #BinaryOp +*
                                                                         Multiply LHS signed bit vector & constant (> 1, 2**N),
#BinaryOp +*
                                                                         " constant value 0 would not be visible here, this one
                                                                          can be converted into a shift left:
" Multiply LHS signed bit vectors:
                                                                         sources 'reg_bit_vector' 'constant'
                                                                         sources 'reg_bit_vector' 'reg_bit_vector'
result 'reg_bit_vector'
                                                                               ((par2 width: par1 width) onecnt = 1))
                                                                         result 'reg_bit_vector'
tempvar 1
'reg' tabto 21
'[' (par1 width + par2 width width: 8) decimalleft
                                                                         inline 30
                                                                         par 1 31 '{[ '
 ':0] d2vTEMP' (INDEX) decimal '; // For LH signed
                                                                          (par1 width - (par2 width: par1 width) Isone - 1)
multiply' cr
                                                                         decimal
                                                                          IF (par1 width - (par2 width: par1 width) Isone > 1)
" Use signed multiply defined in 'extra_functions'
                                                                          THEN ':0'
package. Add
                                                                          ENDIF
 extra '0' bit to MS side of RHS to force it positive:
                                                                           ] , cr
                                                                         IF (par2 = 2)
separate
                                                                         THEN '0}'
setglobal %100
                                                                         ELSE (((par2 width: par1 width) Isone asBIC zeroes)
                                                                         width) decimal "'b' ((par2 width: par1 width) Isone
  d2vTEMP' (INDEX) decimal ' =' cr
   multiply_signed(' markindent par 1 0 ',' cr
                                                                         asBIC zeroes) binary '}'
 '{0,' par 1 30 '});' exitindent cr
                                                                         ENDIF
 " Remove excess bits with slicing:
                                                                         #BinaryOp +*
inline 30
                                                                         " Multiply LHS signed bit vector & constant (> 1, ~=
generatetempvars
                                                                         2**N),
                                                                         " constant LSB is %1:
generateseparate
 'd2vTEMP' (INDEX) decimal ' [ ' (par1 width + par2
width - 1) decimal ':0]'
                                                                         sources 'reg_bit_vector' 'constant'
                                                                         guard (par2 > 1 \Lambda
#BinaryOp +*
                                                                               ((par2 width: par1 width) onecnt > 1) A
                                                                               (par2 at: 0))
 " Multiply LHS signed single bit & constant,
                                                                         result 'reg_bit_vector'
 " constant value 0 would not be visible here:
                                                                         tempvar 1
 sources 'reg_bit' 'constant'
                                                                         'reg' tabto 21
 guard (par2 = 1)
                                                                           ' [ ' (par1 width +
 result 'reg_bit'
                                                                           (par2 width: par1 width) msone +1 width: 8)
                                                                         decimalleft
 forward 1
                                                                           ':0 ) d2vTEMP' (INDEX) decimal '; // For LH signed
                                                                         vector/constant multiply' cr
 #BinaryOp +*
                                                                         " Use signed multiply defined in 'extra_functions'
 " Multiply LHS signed single bit & constant,
                                                                         package. Add
 " constant value 0 would not be visible here:
                                                                          extra '0' bit to MS side of constant to make it positive:
 sources 'reg_bit_c' 'constant'
                                                                         separate
 guard (par2 = 1)
                                                                         setglobal %100
 result 'reg_bit_c'
                                                                          d2vTEMP' (INDEX) decimal ' =' cr
                                                                           multiply_signed(' markindent par 1 0 ',' cr
 forward 1
                                                                         '0' ((par2 width: (par2 width: par1 width) msone asBIC
                                                                         + 1) width) decimal "b' (par2 width: (par2 width: par1
                                                                         width) msone asBIC + 1) binary
 #BinaryOp +*
                                                                          '); // Active constant bit range only...' exitindent cr
```

```
result 'reg_bit_vector'
" Remove excess bits with slicing:
                                                                       " Uses special function defined in 'extra_functions'
inline 30
                                                                       package:
generatetempvars
generateseparate
                                                                       inline 30
'd2vTEMP' (INDEX) decimal '[' (par1 width - 1)
                                                                       setglobal %100
decimal ':0]'
                                                                       'multiply_signed(' markindent par 1 0 ',' cr
                                                                       par 2 0 exitindent ')'
#BinaryOp +1
                                                                       #BinaryOp ∧
" Multiply LHS signed bit vector & constant (> 1, ~=
                                                                       " Logical AND between single bits:
" constant contains LS zeroes:
                                                                       sources 'reg_bit' 'reg_bit'
sources 'reg_bit_vector' 'constant'
                                                                       result 'reg_bit'
guard (par2 > 1 \Lambda
      ((par2 width: par1 width) onecnt > 1) A
                                                                       inline 5
                                                                       par 1 5 ' &' cr
      (par2 at: 0) not)
result 'reg_bit_vector'
                                                                       par 2 6
tempvar 1
                                                                       #BinaryOp ∧
'reg' tabto 21
                                                                       " Logical AND between bit vectors:
  (par1 width + (par2 width: par1 width) msone asBIC -
  (par2 width: par1 width) Isone asBIC + 1 width: 8)
                                                                       sources 'reg_bit_vector' 'reg_bit_vector'
decimalleft
                                                                       result 'reg_bit_vector'
 ':0] d2vTEMP' (INDEX) decimal '; // For LH signed
vector/constant multiply' cr
                                                                       inline 5
                                                                       par 15 '&' cr
" Use signed multiply defined in 'extra_functions'
                                                                       par 2 6
package. Add
 extra '0' bit to MS side of constant to make it positive:
                                                                       #BinaryOp ∧
                                                                       "Logical AND between single bit & constant %1,
separate
setglobal %100
                                                                       removed:
  d2vTEMP' (INDEX) decimal ' =' cr
  multiply_signed(' markindent par 1 0 ',' cr
                                                                       sources 'reg_bit' 'constant'
'0' ((par2 from: (par2 width: par1 width) Isone asBIC
                                                                       guard (par2 = 1)
     to: (par2 width: par1 width) msone asBIC) width)
                                                                       result 'reg_bit'
decimal "b' (par2 from: (par2 width: par1 width) Isone
asBIC
                                                                       forward 1
     to: (par2 width: par1 width) msone asBIC) binary
'); // Active constant bit range only...' exitindent cr
                                                                       #BinaryOp ∧
inline 30
                                                                       " Logical AND between single bit & constant %1,
generatetempyars
                                                                       removed:
generateseparate
'd2vTEMP' (INDEX) decimal
                                                                       sources 'reg_bit_c' 'constant'
                                                                       guard (par2 = 1)
 (par1 width -1 - (par2 width: par1 width) Isone asBIC)
                                                                       result 'reg_bit_c'
decimal
 ':0],' cr
                                                                       forward 1
IF (par2 at: 1)
THEN '0}'
                                                                       #BinaryOp ∧
ELSE (((par2 width: par1 width) Isone asBIC zeroes)
width) decimal "b' ((par2 width: par1 width) Isone
                                                                       "Logical AND between bit vector and Constant (not
asBIC zeroes) binary '}'
                                                                       %11..11):
ENDIF
                                                                       sources 'reg bit vector' 'constant'
#BinaryOp +*+
                                                                       guard (par2 asBIC < par1 width ones asBIC)
                                                                       result 'reg_bit_vector'
" Signed multiply on single bits, result can only be 0 or
1:
                                                                       inline 5
                                                                       par 1 5 ' &' cr
                                                                       (par1 width) decimal "b' (par2 width: par1 width) binary
sources 'reg_bit' 'reg_bit'
result 'reg_bit_vector'
                                                                       #BinaryOp ∧
inline 30
'{0.' cr
                                                                       " Logical AND between bit vector and %11..11
'(' markindent par 1 5 exitindent ' &' cr
                                                                       Constant, removed:
'' markindent par 2 6 exitindent ')}'
                                                                       sources 'reg_bit_vector' 'constant'
#BinaryOp +*+
                                                                       guard (par2 asBIC = par1 width ones asBIC)
                                                                       result 'reg_bit_vector'
" Multiply signed bit vectors:
                                                                       forward 1
sources 'reg_bit_vector' 'reg_bit_vector'
```

```
#BinaryOp ∧
                                                                     par 2 6 ')' exitindent
* Logical AND between bit vector and %11..11
                                                                     #BinaryOp ~∧
Constant, removed:
                                                                     " Logical NAND between bit vectors:
sources 'reg_bit_vector_c' 'constant'
guard (par2 asBIC = par1 width ones asBIC)
                                                                     sources 'reg_bit_vector' 'reg_bit_vector'
result 'reg_bit_vector_c'
                                                                     result 'reg_bit_vector'
forward 1
                                                                     inline 11
                                                                     '~(' markindent par 1 5 ' &' cr
                                                                     par 2 6 ')' exitindent
#BinaryOp ∧
                                                                     #BinaryOp ~Λ
" Logical AND between Constant %1 & single bit,
removed:
                                                                     "Logical NAND between single bit & constant %1,
sources 'constant' 'reg_bit'
                                                                     becomes NOT:
guard (par1 = 1)
result 'reg_bit'
                                                                     sources 'reg_bit' 'constant'
                                                                     guard (par2 = 1)
forward 2
                                                                     result 'reg_bit_c'
#BinaryOp ∧
                                                                     forward 1
"Logical AND between Constant %1 & single bit,
                                                                     #BinaryOp ~ \
removed:
                                                                     "Logical NAND between single bit & constant %1,
sources 'constant' 'reg_bit_c'
                                                                     becomes NOT:
guard (par1 = 1)
result 'reg_bit_c'
                                                                     sources 'reg_bit_c' 'constant'
                                                                     guard (par2 = 1)
forward 2
                                                                     result 'reg_bit'
#BinaryOp ∧
                                                                     forward 1
"Logical AND between Constant (not %11..11) and bit
                                                                     #BinaryOp ~∧
vector:
                                                                     " Logical NAND between bit vector and Constant (not
sources 'constant' 'reg_bit_vector'
                                                                     %11..11):
guard (par1 asBIC < par2 width ones asBIC)
                                                                     sources 'reg_bit_vector' 'constant'
result 'reg_bit_vector'
                                                                     guard (par2 asBIC < par1 width ones asBIC)
                                                                     result 'reg_bit_vector'
inline 5
(par2 width) decimal "b' (par1 width: par2 width) binary
 &' cr
par 2 6
                                                                      '~(' markindent par 1 5 ' &' cr
                                                                     (par1 width) decimal "b' (par2 width: par1 width) binary
#BinaryOp ∧
                                                                     ')' exitindent
" Logical AND between %11..11 Constant and bit
                                                                     #BinaryOp ~Λ
vector, removed:
                                                                     "Logical NAND between bit vector and %11..11
sources 'constant' 'reg_bit_vector'
                                                                     Constant, becomes NOT:
guard (par1 asBIC = par2 width ones asBIC)
result 'reg_bit_vector'
                                                                     sources 'reg_bit_vector' 'constant'
                                                                     guard (par2 asBIC = par1 width ones asBIC)
forward 2
                                                                     result 'reg_bit_vector_c'
#BinaryOp ∧
                                                                     forward 1
" Logical AND between %11..11 Constant and bit
                                                                     #BinaryOp ~ \
vector, removed:
                                                                     "Logical NAND between bit vector and %11..11
sources 'constant' 'reg_bit_vector_c'
                                                                     Constant, becomes NOT:
guard (par1 asBIC = par2 width ones asBIC)
result 'reg_bit_vector_c'
                                                                     sources 'reg_bit_vector_c' 'constant'
                                                                     guard (par2 asBIC = par1 width ones asBIC)
                                                                     result 'reg_bit_vector'
forward 2
#BinaryOp ~∧
                                                                     forward 1
" Logical NAND between single bits:
                                                                     #BinaryOp ~∧
sources 'reg_bit' 'reg_bit'
                                                                     " Logical NAND between Constant %1 & single bit,
result 'reg_bit'
                                                                     becomes NOT:
inline 11
                                                                     sources 'constant' 'reg_bit'
 '~(' markindent par 1 5 ' &' cr
                                                                     guard (par1 = 1)
```

```
result 'reg_bit_c'
                                                                     sources 'reg_bit' 'constant'
                                                                     guard (par2 = 0)
forward 2
                                                                     result 'reg_bit'
#BinaryOp ~ \
                                                                     forward 1
"Logical NAND between Constant %1 & single bit,
                                                                      #BinaryOp V
becomes NOT:
                                                                     " Logical OR between single bit & constant %0,
sources 'constant' 'req_bit_c'
                                                                      removed:
guard (par1 = 1)
result 'reg_bit'
                                                                      sources 'reg_bit_c' 'constant'
                                                                      guard (par2 = 0)
                                                                     result 'reg_bit_c'
forward 2
#BinaryOp ~ \
                                                                      forward 1
" Logical NAND between Constant (not %11..11) and
                                                                      #BinaryOp V
bit vector:
                                                                      " Logical OR between bit vector and Constant (not
sources 'constant' 'reg_bit_vector'
                                                                      %00..00):
guard (par1 asBIC < par2 width ones asBIC)
result 'reg_bit_vector'
                                                                      sources 'reg_bit_vector' 'constant'
                                                                     guard (par2 ~= 0)
                                                                      result 'reg_bit_vector'
'~(' markindent (par2 width) decimal ""b' (par1 width:
par2 width) binary '&' cr
                                                                      inline 3
par 2 6 ')' exitindent
                                                                      par 1 3 ' l' cr
                                                                      (par1 width) decimal "b' (par2 width: par1 width) binary
#BinaryOp ~∧
                                                                      #BinaryOp V
" Logical NAND between %11..11 Constant and bit
                                                                      " Logical OR between bit vector and %00..00 Constant,
vector, becomes NOT:
                                                                      removed:
sources 'constant' 'reg_bit_vector'
guard (par1 asBIC = par2 width ones asBIC)
                                                                      sources 'reg_bit_vector' 'constant'
result 'reg_bit_vector_c'
                                                                      guard (par2 = 0)
                                                                      result 'reg_bit_vector'
forward 2
                                                                      forward 1
#BinaryOp ~ \
                                                                      #BinaryOp V
* Logical NAND between %11..11 Constant and bit
                                                                      " Logical OR between bit vector and %00..00 Constant,
vector, becomes NOT:
                                                                      removed:
sources 'constant' 'reg_bit_vector_c'
guard (par1 asBIC = par2 width ones asBIC)
                                                                      sources 'reg_bit_vector_c' 'constant'
result 'reg_bit_vector'
                                                                      guard (par2 = 0)
                                                                      result 'reg_bit_vector_c'
forward 2
                                                                      forward 1
#BinaryOp V
                                                                      #BinaryOp V
" Logical OR between single bits:
                                                                      " Logical OR between Constant %0 & single bit,
sources 'reg_bit' 'reg_bit'
                                                                      removed:
result 'reg_bit'
                                                                      sources 'constant' 'reg_bit'
inline 3
                                                                      guard (par1 = 0)
par 1 3 ' l' cr
                                                                      result 'reg_bit'
par 2 4
                                                                      forward 2
#BinaryOp V
                                                                      #BinaryOp V
* Logical OR between bit vectors:
                                                                      "Logical OR between Constant %0 & single bit,
sources 'reg_bit_vector' 'reg_bit_vector'
                                                                      removed:
result 'reg_bit_vector'
                                                                      sources 'constant' 'reg_bit_c'
inline 3
                                                                      guard (par1 = 0)
par 13' l' cr
                                                                      result 'reg_bit_c'
par 2 4
                                                                      forward 2
#BinaryOp V
                                                                      #BinaryOp V
" Logical OR between single bit & constant %0,
removed:
                                                                      Logical OR between Constant (not %00..00) and bit
                                                                      vector:
```

```
" Logical NOR between bit vector and Constant (not
sources 'constant' 'reg_bit_vector'
                                                                      %00..00):
guard (par1 ~= 0)
                                                                      sources 'reg_bit_vector' 'constant'
result 'reg_bit_vector'
                                                                      guard (par2 ~= 0)
                                                                      result 'reg_bit_vector'
inline 3
(par2 width) decimal "b' (par1 width: par2 width) binary
                                                                      inline 11
par 2 4
                                                                       ~(' markindent par 1 3 ' l' cr
                                                                      (par1 width) decimal "b' (par2 width: par1 width) binary
#BinaryOp V
                                                                      ')' exitindent
" Logical OR between %00..00 Constant and bit vector,
                                                                      #BinaryOp ~V
removed:
                                                                      "Logical NOR between bit vector and %00..00
                                                                      Constant, becomes NOT:
sources 'constant' 'reg_bit_vector'
guard (par1 = 0)
result 'reg_bit_vector'
                                                                      sources 'reg_bit_vector' 'constant'
                                                                      guard (par2 = 0)
forward 2
                                                                      result 'reg_bit_vector_c'
#BinaryOp V
                                                                      forward 1
"Logical OR between %00..00 Constant and bit vector,
                                                                      #BinaryOp ~V
removed:
                                                                      " Logical NOR between bit vector and %00..00
sources 'constant' 'reg_bit_vector_c'
                                                                      Constant, becomes NOT:
guard (par1 = 0)
result 'reg_bit_vector_c'
                                                                      sources 'reg_bit_vector_c' 'constant'
                                                                      guard (par2 = 0)
                                                                      result 'reg_bit_vector'
forward 2
#BinaryOp ~V
                                                                      forward 1
" Logical NOR between single bits:
                                                                      #BinaryOp ~V
sources 'reg_bit' 'reg_bit'
                                                                      "Logical NOR between Constant %0 & single bit,
result 'reg_bit'
                                                                      becomes NOT:
inline 11
                                                                      sources 'constant' 'reg_bit'
 '~(' markindent par 1 3 ' l' cr
                                                                      guard (par1 = 0)
                                                                      result 'reg_bit_c'
par 2 4 ')' exitindent
 #BinaryOp ~V
                                                                      forward 2
 " Logical NOR between bit vectors:
                                                                      #BinaryOp ~V
                                                                      " Logical NOR between Constant %0 & single bit,
 sources 'reg_bit_vector' 'reg_bit_vector'
result 'reg_bit_vector'
                                                                      becomes NOT:
                                                                      sources 'constant' 'reg_bit_c'
 inline 11
 '~(' markindent par 1 3 ' l' cr
                                                                      guard (par1 = 0)
 par 2 4 ')' exitindent
                                                                      result 'reg_bit'
 #BinaryOp ~V
                                                                      forward 2
" Logical NOR between single bit & constant %0,
                                                                      #BinaryOp ~V
 becomes NOT:
                                                                      " Logical NOR between Constant (not %00..00) and bit
 sources 'reg_bit' 'constant'
 guard (par2 = 0)
                                                                      sources 'constant' 'reg_bit_vector'
 result 'reg_bit_c'
                                                                      guard (par1 \sim= 0)
                                                                      result 'reg_bit_vector'
 forward 1
 #BinaryOp ~V
                                                                      inline 11
                                                                       '~(' markindent (par2 width) decimal "b' (par1 width:
 "Logical NOR between single bit & constant %0,
                                                                      par2 width) binary ' I' cr
 becomes NOT:
                                                                      par 2 4 ')' exitindent
 sources 'reg_bit_c' 'constant'
                                                                      #BinaryOp ~V
 guard (par2 = 0)
result 'reg_bit'
                                                                      "Logical NOR between %00..00 Constant and bit
                                                                      vector, becomes NOT:
 forward 1
                                                                      sources 'constant' 'reg_bit_vector'
 #BinaryOp ~V
                                                                      guard (par1 = 0)
                                                                       result 'reg_bit_vector_c'
```

```
forward 1
forward 2
                                                                     #BinaryOp ><
#BinaryOp ~V
                                                                     " Logical XOR between bit vector and Constant (not
" Logical NOR between %00..00 Constant and bit
                                                                     %00..00
                                                                     " or %11..11):
vector, becomes NOT:
sources 'constant' 'reg_bit_vector_c'
                                                                     sources 'reg_bit_vector' 'constant'
                                                                     guard (par2 \sim= 0 \Lambda
guard (par1 = 0)
result 'reg_bit_vector'
                                                                           (par2 asBIC ~= par1 width ones asBIC))
                                                                     result 'reg_bit_vector'
forward 2
                                                                     inline 4
                                                                     par 1 4 ' ^' cr
#BinaryOp ><
                                                                     (par1 width) decimal "b' (par2 width: par1 width) binary
" Logical XOR between single bits:
                                                                     #BinaryOp ><
sources 'reg_bit' 'reg_bit'
result 'reg_bit'
                                                                     " Logical XOR between bit vector and %00..00
                                                                     Constant, removed:
inline 4
par 1 4 ' ^' cr
                                                                     sources 'reg_bit_vector' 'constant'
par 2 5
                                                                     guard (par2 = 0)
                                                                     result 'reg_bit_vector'
#BinaryOp ><
                                                                     forward 1
" Logical XOR between bit vectors:
                                                                     #BinaryOp ><
sources 'reg_bit_vector' 'reg_bit_vector'
result 'reg_bit_vector'
                                                                     "Logical XOR between bit vector and %00..00
                                                                     Constant, removed:
inline 4
par 1 4 ' ^' cr
                                                                     sources 'reg_bit_vector_c' 'constant'
par 2 5
                                                                     guard (par2 = 0)
                                                                     result 'reg_bit_vector_c'
#BinaryOp ><
                                                                     forward 1
"Logical XOR between single bit & constant %0,
removed:
                                                                     #BinaryOp ><
sources 'reg_bit' 'constant'
                                                                     " Logical XOR between bit vector and %11..11
guard (par2 = 0)
                                                                     Constant, becomes NOT:
result 'reg_bit'
                                                                     sources 'reg_bit_vector' 'constant'
forward 1
                                                                     guard (par2 = 0)
                                                                     result 'reg_bit_vector_c'
#BinaryOp ><
                                                                     forward 1
" Logical XOR between single bit & constant %0,
removed:
                                                                     #BinaryOp ><
sources 'reg_bit_c' 'constant'
                                                                      Logical XOR between bit vector and %11..11
guard (par2 = 0)
                                                                     Constant, becomes NOT:
result 'reg_bit_c'
                                                                     sources 'reg_bit_vector_c' 'constant'
forward 1
                                                                     guard (par2 = 0)
                                                                     result 'reg_bit_vector
#BinaryOp ><
                                                                     forward 1
"Logical XOR between single bit & constant %1,
becomes NOT:
                                                                      #BinaryOp ><
sources 'reg_bit' 'constant'
                                                                      "Logical XOR between Constant %0 & single bit,
guard (par2 = 1)
                                                                      removed:
result 'reg_bit_c'
                                                                     sources 'constant' 'reg_bit'
forward 1
                                                                     guard (par1 = 0)
                                                                     result 'reg_bit'
#BinaryOp ><
                                                                      forward 2
"Logical XOR between single bit & constant %1,
becomes NOT:
                                                                      #BinaryOp ><
sources 'reg_bit_c' 'constant'
                                                                      " Logical XOR between Constant %0 & single bit,
guard (par2 = 1)
                                                                      removed:
result 'reg_bit'
                                                                      sources 'constant' 'reg_bit_c'
```

```
guard (par1 = 0)
                                                                     "Logical XOR between %11..11 Constant and bit
result 'reg_bit_c'
                                                                     vector, becomes NOT:
                                                                     sources 'constant' 'reg_bit_vector_c'
forward 2
                                                                     guard (par1 asBIC = par2 width ones asBIC)
#BinaryOp ><
                                                                     result 'reg_bit_vector'
" Logical XOR between Constant %1 & single bit,
                                                                     forward 2
becomes NOT:
                                                                     #BinaryOp <>
sources 'constant' 'reg_bit'
guard (par1 = 1)
                                                                     "Logical XNOR between single bits (done with NOT
result 'reg_bit_c'
                                                                     XOR):
                                                                     sources 'reg_bit' 'reg_bit'
forward 2
                                                                     result 'reg_bit_c'
#BinaryOp ><
                                                                     inline 4
                                                                     par 1 4 ' ^' cr
* Logical XOR between Constant %1 & single bit,
becomes NOT:
                                                                     par 2 5
sources 'constant' 'reg_bit_c'
                                                                     #BinaryOp <>
guard (par1 = 1)
                                                                     " Logical XNOR between bit vectors (done with NOT
result 'reg_bit'
                                                                     XOR):
forward 2
                                                                     sources 'reg_bit_vector' 'reg_bit_vector'
#BinaryOp ><
                                                                     result 'reg_bit_vector_c'
" Logical XOR between Constant (not %00..00 or
%11..11) and
                                                                     par 1 4 ' ^' cr
" bit vector:
                                                                     par 2 5
sources 'constant' 'reg_bit_vector'
                                                                     #BinaryOp <>
guard (par1 \sim= 0 \Lambda
      (par1 asBIC ~= par2 width ones asBIC))
                                                                      "Logical XNOR between single bit & constant %0,
result 'reg_bit_vector'
                                                                     becomes NOT:
inline 4
                                                                     sources 'reg_bit' 'constant'
(par2 width) decimal "b' (par1 width: par2 width) binary
                                                                     guard (par2 = 0)
                                                                     result 'reg_bit_c'
par 2 5
                                                                     forward 1
#BinaryOp ><
                                                                     #BinaryOp <>
" Logical XOR between %00..00 Constant and bit
vector, removed:
                                                                     "Logical XNOR between single bit & constant %0,
                                                                     becomes NOT:
sources 'constant' 'reg_bit_vector'
guard (par1 = 0)
                                                                     sources 'reg_bit_c' 'constant'
                                                                     guard (par2 = 0)
result 'reg_bit_vector'
                                                                     result 'reg_bit'
forward 2
                                                                     forward 1
#BinaryOp ><
                                                                     #BinaryOp <>
" Logical XOR between %00..00 Constant and bit
vector, removed:
                                                                     " Logical XNOR between single bit & constant %1,
                                                                     removed:
sources 'constant' 'reg_bit_vector_c
guard (par1 = 0)
                                                                     sources 'reg_bit' 'constant'
result 'reg_bit_vector_c'
                                                                     guard (par2 = 1)
                                                                     result 'reg_bit'
forward 2
                                                                     forward 1
#BinaryOp ><
                                                                     #BinaryOp <>
" Logical XOR between %11..11 Constant and bit
vector, becomes NOT:
                                                                     "Logical XNOR between single bit & constant %1,
                                                                     removed:
sources 'constant' 'reg_bit_vector'
guard (par1 asBIC = par2 width ones asBIC)
                                                                     sources 'reg_bit_c' 'constant'
result 'reg_bit_vector_c'
                                                                     guard (par2 = 1)
                                                                     result 'reg_bit_c'
forward 2
                                                                     forward 1
 #BinaryOp ><
                                                                     #BinaryOp <>
```

n e e e e e e e e e e e e e e e e e e e	u .
" Logical XNOR between bit vector and Constant (not %0000	forward 2
" or %1111), uses XOR with complement of Constant:	#BinaryOp <>
sources 'reg_bit_vector' 'constant' guard (par2 ~= 0 ∧	"Logical XNOR between Constant %1 & single bit, removed:
(par2 asBIC -= par1 width ones asBIC)) result 'reg_bit_vector'	sources 'constant' 'reg_bit' guard (par1 = 1)
inline 4 par 1 4 ' ^' cr	result 'reg_bit' "
(par1 width) decimal "b' ((par2 width: par1 width)not) binary	forward 2
#BinaryOp <>	#BinaryOp <>
* Logical XNOR between bit vector and %0000 Constant, becomes NOT:	"Logical XNOR between Constant %1 & single bit, removed:
sources 'reg_bit_vector' 'constant' guard (par2 = 0)	sources 'constant' 'reg_bit_c' guard (par1 = 1) result 'reg_bit_c'
result 'reg_bit_vector_c'	forward 2
forward 1	# #BinaryOp <>
#BinaryOp <>	" Logical XNOR between Constant (not %0000 or
" Logical XNOR between bit vector and %0000 Constant, becomes NOT:	%1111) and " bit vector (complements constant and uses XOR): "
sources 'reg_bit_vector_c' 'constant' guard (par2 = 0)	sources 'constant' 'reg_bit_vector' guard (par1 ~= 0 ∧
result 'reg_bit_vector'	(par1 asBIC -= par2 width ones asBIC)) result 'reg_bit_vector'
forward 1	inline 4
#BinaryOp <>	(par2 width) decimal "'b' ((par1 width: par2 width) not) binary " ^' cr
" Logical XNOR between bit vector and %1111 Constant, removed:	par 2 5
sources 'reg_bit_vector' 'constant'	#BinaryOp <>
guard (par2 asBIC = par1 width ones asBIC) result 'reg_bit_vector' "	" Logical XNOR between %0000 Constant and bit vector, becomes NOT: "
forward 1	sources 'constant' 'reg_bit_vector' guard (par1 = 0)
#BinaryOp <>	result 'reg_bit_vector_c'
" Logical XNOR between bit vector and %1111 Constant, removed:	forward 2
" sources 'reg_bit_vector_c' 'constant'	#BinaryOp <>
guard (par2 asBIC = par1 width ones asBIC) result 'reg_bit_vector_c' "	" Logical XNOR between %0000 Constant and bit vector, becomes NOT:
forward 1	sources 'constant' 'reg_bit_vector_c' guard (par1 = 0)
#BinaryOp <>	result 'reg_bit_vector'
"Logical XNOR between Constant %0 & single bit, becomes NOT:	forward 2
sources 'constant' 'reg_bit'	#BinaryOp <>
guard (par1 = 0) result 'reg_bit_c' "	" Logical XNOR between %1111 Constant and bit vector, removed:
forward 2	sources 'constant' 'reg_bit_vector' guard (par1 asBIC = par2 width ones asBIC)
#BinaryOp <>	result 'reg_bit_vector'
" Logical XNOR between Constant %0 & single bit, becomes NOT:	forward 2
sources 'constant' 'reg_bit_c'	#BinaryOp <>
guard (par1 = 0) result 'reg_bit'	"Logical XNOR between %1111 Constant and bit vector, removed:

```
sources 'constant' 'reg_bit'
sources 'constant' 'reg_bit_vector_c'
                                                                     guard (par1 = 0)
guard (par1 asBIC = par2 width ones asBIC)
                                                                     result 'reg_bit_c'
result 'reg_bit_vector_c'
                                                                     forward 2
forward 2
                                                                     #BinaryOp =
#BinaryOp =
                                                                     * Unsigned compare equal constant %0 & single bit,
" Unsigned compare equal single bit & constant %1,
                                                                     made into NOT:
removed:
                                                                     sources 'constant' 'req_bit_c'
sources 'reg_bit' 'constant'
                                                                     guard (par1 = 0)
guard (par2 = 1)
                                                                     result 'reg_bit'
result 'reg_bit'
                                                                     forward 2
forward 1
                                                                     #BinaryOp ~=
#BinaryOp =
                                                                     "Unsigned compare not equal single bit & constant
" Unsigned compare equal single bit & constant %1,
                                                                      %1, made into NOT:
removed:
                                                                     sources 'reg_bit' 'constant'
sources 'reg_bit_c' 'constant'
                                                                     guard (par2 = 1)
guard (par2 = 1)
                                                                     result 'reg_bit_c'
result 'reg_bit_c'
                                                                     forward 1
forward 1
                                                                     #BinaryOp ~=
#BinaryOp =
                                                                     " Unsigned compare not equal single bit & constant
" Unsigned compare equal single bit & constant %0,
                                                                      %1, made into NOT:
made into NOT:
                                                                     sources 'reg_bit_c' 'constant'
sources 'reg_bit' 'constant'
                                                                     guard (par2 = 1)
guard (par2 = 0)
                                                                     result 'reg_bit'
result 'reg_bit_c'
                                                                     forward 1
forward 1
                                                                      #BinaryOp ~=
#BinaryOp =
                                                                      " Unsigned compare not equal single bit & constant
" Unsigned compare equal single bit & constant %0,
                                                                      %0, removed:
made into NOT:
                                                                      sources 'reg_bit' 'constant'
sources 'reg_bit_c' 'constant'
                                                                      guard (par2 = 0)
guard (par2 = 0)
                                                                      result 'reg_bit'
result 'reg_bit'
                                                                      forward 1
forward 1
                                                                      #BinaryOp ~=
#BinaryOp =
                                                                      " Unsigned compare not equal single bit & constant
" Unsigned compare equal constant %1 & single bit,
                                                                      %0, removed:
removed:
                                                                      sources 'reg_bit_c' 'constant'
sources 'constant' 'reg_bit'
                                                                      guard (par2 = 0)
guard (par1 = 1)
                                                                      result 'reg_bit_c'
result 'reg_bit'
                                                                      forward 1
forward 2
                                                                      #BinaryOp ~=
#BinaryOp =
                                                                      " Unsigned compare not equal constant %1 & single
" Unsigned compare equal constant %1 & single bit,
                                                                      bit, made into NOT:
removed:
                                                                      sources 'constant' 'reg_bit'
sources 'constant' 'reg_bit_c'
                                                                      guard (par1 = 1)
guard (par1 = 1)
                                                                      result 'reg_bit_c'
result 'reg_bit_c'
                                                                      forward 2
forward 2
                                                                      #BinaryOp ~=
#BinaryOp =
                                                                      " Unsigned compare not equal constant %1 & single
 " Unsigned compare equal constant %0 & single bit,
                                                                      bit, made into NOT:
made into NOT:
                                                                      sources 'constant' 'reg_bit_c'
```

```
guard (par1 = 1)
                                                                     guard (par1 = 0)
result 'reg_bit'
                                                                     result 'reg_bit_c'
forward 2
                                                                     forward 2
#BinaryOp ~=
                                                                     #BinaryOp <=
                                                                     " Unsigned compare below or equal between single
" Unsigned compare not equal constant %0 & single
bit, removed:
                                                                     bits, made into NAND/NOT:
                                                                     sources 'reg_bit' 'reg_bit_c'
sources 'constant' 'reg_bit'
guard (par1 = 0)
                                                                     result 'reg_bit'
result 'reg_bit'
                                                                     inline 11
forward 2
                                                                     '~(' markindent par 1 5 ' &' cr
                                                                     par 2 6 ')' exitindent
#BinaryOp ~=
                                                                     #BinaryOp <=
" Unsigned compare not equal constant %0 & single
                                                                     " Unsigned compare below or equal single bit &
bit, removed:
                                                                     constant %0, made into NOT:
sources 'constant' 'reg_bit_c'
                                                                     sources 'reg_bit' 'constant'
guard (par1 = 0)
result 'reg_bit_c'
                                                                     guard (par2 = 0)
                                                                     result 'reg_bit_c'
forward 2
                                                                     forward 1
#BinaryOp <
                                                                     #BinaryOp <=
" Unsigned compare below between single bits, made
into AND/NOT:
                                                                     " Unsigned compare below or equal single bit &
                                                                     constant %0, made into NOT:
sources 'reg_bit_c' 'reg_bit'
result 'reg_bit'
                                                                     sources 'reg_bit_c' 'constant'
                                                                     guard (par2 = 0)
inline 5
                                                                     result 'reg_bit'
par 1 5 ' &' cr
par 2 6
                                                                     forward 1
#BinaryOp <
                                                                     #BinaryOp <=
" Unsigned compare below single bit & constant %1,
                                                                     " Unsigned compare below or equal constant %1 &
made into NOT:
                                                                     single bit, removed:
sources 'reg_bit' 'constant'
                                                                     sources 'constant' 'reg_bit'
guard (par2 = 1)
                                                                     guard (par1 = 1)
result 'reg_bit_c'
                                                                     result 'reg_bit'
forward 1
                                                                     forward 2
#BinaryOp <
                                                                     #BinaryOp <=
" Unsigned compare below single bit & constant %1,
                                                                     " Unsigned compare below or equal constant %1 &
made into NOT:
                                                                     single bit, removed:
sources 'reg_bit_c' 'constant'
                                                                     sources 'constant' 'reg_bit_c'
guard (par2 = 1)
                                                                     guard (par1 = 1)
result 'reg_bit'
                                                                     result 'reg_bit_c'
forward 1
                                                                     forward 2
#BinaryOp <
                                                                     #BinaryOp =<
" Unsigned compare below constant %0 & single bit,
                                                                     "Unsigned compare below or equal between single
                                                                     bits, made into NAND/NOT:
removed:
sources 'constant' 'reg_bit'
                                                                     sources 'reg_bit' 'reg_bit_c'
guard (par1 = 0)
                                                                     result 'reg_bit'
result 'reg_bit'
                                                                     inline 11
forward 2
                                                                      '~(' markindent par 15 ' &' cr
                                                                     par 2 6 ')' exitindent
#BinaryOp <
                                                                     #BinaryOp =<
" Unsigned compare below constant %0 & single bit,
removed:
                                                                     " Unsigned compare below or equal single bit &
                                                                     constant %0, made into NOT:
sources 'constant' 'reg_bit_c'
```

```
sources 'reg_bit' 'constant'
                                                                     sources 'constant' 'reg_bit'
guard (par2 = 0)
                                                                     guard (par1 = 1)
result 'reg_bit_c'
                                                                     result 'reg_bit_c'
forward 1
                                                                     forward 2
#BinaryOp =<
                                                                     #BinaryOp >
" Unsigned compare below or equal single bit &
                                                                     " Unsigned compare above constant %1 & single bit,
constant %0, made into NOT:
                                                                     made into NOT:
sources 'reg_bit_c' 'constant'
                                                                     sources 'constant' 'reg_bit_c'
guard (par2 = 0)
                                                                     guard (par1 = 1)
result 'reg_bit'
                                                                     result 'reg_bit'
                                                                     forward 2
forward 1
#BinaryOp =<
                                                                     #BinaryOp >=
" Unsigned compare below or equal constant %1 &
                                                                     * Unsigned compare above or equal between single
                                                                     bits, made into NAND/NOT:
single bit, removed:
sources 'constant' 'reg_bit'
                                                                     sources 'reg_bit_c' 'reg_bit'
guard (par1 = 1)
                                                                     result 'reg_bit'
result 'reg_bit'
                                                                     inline 11
                                                                     '~(' markindent par 1 5 ' &' cr
forward 2
                                                                     par 2 6 ')' exitindent
#BinaryOp =<
                                                                     #BinaryOp >=
" Unsigned compare below or equal constant %1 &
single bit, removed:
                                                                     " Unsigned compare above or equal single bit &
                                                                     constant %1, removed:
sources 'constant' 'reg_bit_c'
guard (par1 = 1)
                                                                     sources 'reg_bit' 'constant'
result 'reg_bit_c'
                                                                     guard (par2 = 1)
                                                                     result 'reg_bit'
forward 2
                                                                     forward 1
#BinaryOp >
                                                                     #BinaryOp >=
* Unsigned compare above between single bits, made
into AND/NOT:
                                                                     "Unsigned compare above or equal single bit &
                                                                     constant %1, removed:
sources 'reg_bit' 'reg_bit_c'
result 'reg_bit'
                                                                     sources 'reg_bit_c' 'constant'
                                                                     guard (par2 = 1)
inline 5
                                                                      result 'reg_bit_c'
par 1 5 ' &' cr
par 2 6
                                                                     forward 1
#BinaryOp >
                                                                      #BinaryOp >=
" Unsigned compare above single bit & constant %0,
                                                                      " Unsigned compare above or equal constant %0 &
removed:
                                                                      single bit, made into NOT:
sources 'reg_bit' 'constant'
                                                                     sources 'constant' 'reg_bit'
guard (par2 = 0)
                                                                      guard (par1 = 0)
result 'reg_bit'
                                                                      result 'reg_bit_c'
forward 1
                                                                     forward 2
#BinaryOp >
                                                                      #BinaryOp >=
" Unsigned compare above single bit & constant %0,
                                                                      " Unsigned compare above or equal constant %0 &
removed:
                                                                      single bit, made into NOT:
sources 'reg_bit_c' 'constant'
                                                                      sources 'constant' 'reg_bit_c'
guard (par2 = 0)
                                                                      guard (par1 = 0)
result 'reg_bit_c'
                                                                      result 'reg_bit'
forward 1
                                                                      forward 2
#BinaryOp >
                                                                      #BinaryOp =>
" Unsigned compare above constant %1 & single bit,
                                                                      " Unsigned compare above or equal between single
made into NOT:
                                                                      bits, made into NAND/NOT:
```

```
sources 'reg_bit' 'constant'
sources 'reg_bit_c' 'reg_bit'
result 'reg_bit'
                                                                     guard (par2 = 0)
                                                                     result 'reg_bit_c'
inline 11
'~(' markindent par 1 5 ' &' cr
                                                                      forward 1
par 2 6 ')' exitindent
                                                                      #BinaryOp +=+
#BinaryOp =>
                                                                      " Signed compare equal single bit & constant %0,
" Unsigned compare above or equal single bit &
                                                                      made into NOT:
constant %1, removed:
                                                                      sources 'reg_bit_c' 'constant'
sources 'reg_bit' 'constant'
                                                                      guard (par2 = 0)
guard (par2 = 1)
                                                                      result 'reg_bit'
result 'reg_bit'
                                                                      forward 1
forward 1
                                                                      #BinaryOp +=+
#BinaryOp =>
                                                                      " Signed compare equal constant %1 & single bit,
" Unsigned compare above or equal single bit &
                                                                      removed:
constant %1, removed:
                                                                      sources 'constant' 'reg_bit'
sources 'reg_bit_c' 'constant'
                                                                      guard (par1 = 1)
                                                                      result 'reg_bit'
guard (par2 = 1)
result 'reg_bit_c'
                                                                      forward 2
forward 1
                                                                      #BinaryOp +=+
#BinaryOp =>
                                                                      * Signed compare equal constant %1 & single bit,
" Unsigned compare above or equal constant %0 &
                                                                      removed:
single bit, made into NOT:
                                                                      sources 'constant' 'reg_bit_c'
sources 'constant' 'reg_bit'
                                                                      guard (par1 = 1)
guard (par1 = 0)
                                                                      result 'reg_bit_c'
result 'reg_bit_c'
                                                                      forward 2
forward 2
                                                                      #BinaryOp +=+
#BinaryOp =>
                                                                      " Signed compare equal constant %0 & single bit,
" Unsigned compare above or equal constant %0 &
                                                                      made into NOT:
single bit, made into NOT:
                                                                      sources 'constant' 'reg_bit'
sources 'constant' 'reg_bit_c'
                                                                      guard (par1 = 0)
guard (par1 = 0)
                                                                      result 'reg_bit_c'
result 'reg_bit'
                                                                      forward 2
forward 2
                                                                      #BinaryOp +=+
#BinaryOp +=+
                                                                      " Signed compare equal constant %0 & single bit,
" Signed compare equal single bit & constant %1,
                                                                      made into NOT:
removed:
                                                                      sources 'constant' 'reg_bit_c'
sources 'reg_bit' 'constant'
                                                                      guard (par1 = 0)
guard (par2 = 1)
                                                                      result 'reg_bit'
result 'reg_bit'
                                                                      forward 2
forward 1
                                                                      #BinaryOp +~=+
 #BinaryOp +=+
                                                                      " Signed compare not equal single bit & constant %1,
 " Signed compare equal single bit & constant %1,
                                                                      made into NOT:
 removed:
                                                                      sources 'reg_bit' 'constant'
 sources 'reg_bit_c' 'constant'
                                                                      guard (par2 = 1)
guard (par2 = 1)
                                                                      result 'reg_bit_c'
result 'reg_bit_c'
                                                                      forward 1
 forward 1
                                                                      #BinaryOp +~=+
 #BinaryOp +=+
                                                                      " Signed compare not equal single bit & constant %1,
 * Signed compare equal single bit & constant %0,
                                                                      made into NOT:
 made into NOT:
                                                                      sources 'reg_bit_c' 'constant'
```

```
guard (par2 = 1)
result 'reg_bit'
                                                                     inline 5
                                                                     par 15 ' &' cr
                                                                     par 2 6
forward 1
#BinaryOp +~=+
                                                                     #BinaryOp +<+
" Signed compare not equal single bit & constant %0,

    Signed compare less than single bit & constant %0,

removed:
                                                                      removed:
                                                                      sources 'reg_bit' 'constant'
sources 'reg_bit' 'constant'
guard (par2 = 0)
                                                                      guard (par2 = 0)
result 'reg_bit'
                                                                      result 'reg_bit'
forward 1
                                                                      forward 1
#BinaryOp +~=+
                                                                      #BinaryOp +<+
                                                                      " Signed compare less than single bit & constant %0,
" Signed compare not equal single bit & constant %0,
                                                                      removed:
removed:
sources 'reg_bit_c' 'constant'
                                                                      sources 'reg_bit_c' 'constant'
guard (par2 = 0)
                                                                      guard (par2 = 0)
result 'reg_bit_c'
                                                                      result 'reg_bit_c'
forward 1
                                                                      forward 1
#BinaryOp +~=+
                                                                      #BinaryOp +<+
" Signed compare not equal constant %1 & single bit,
                                                                      " Signed compare less than constant %1 & single bit,
made into NOT:
                                                                      made into NOT:
                                                                      sources 'constant' 'reg_bit'
sources 'constant' 'reg_bit'
guard (par1 = 1)
                                                                      guard (par1 = 1)
result 'reg_bit_c'
                                                                      result 'reg_bit_c'
forward 2
                                                                      forward 2
#BinaryOp +~=+
                                                                      #BinaryOp +<+
" Signed compare not equal constant %1 & single bit,
                                                                      " Signed compare less than constant %1 & single bit,
made into NOT:
                                                                      made into NOT:
sources 'constant' 'reg_bit_c'
                                                                      sources 'constant' 'reg_bit_c'
guard (par1 = 1)
                                                                      guard (par1 = 1)
                                                                      result 'reg_bit'
result 'reg_bit'
forward 2
                                                                      forward 2
                                                                      #BinaryOp +<=+
#BinaryOp +~=+
                                                                      " Signed compare less than or equal between single
" Signed compare not equal constant %0 & single bit,
                                                                      bits.
removed:
                                                                      " made into NAND/NOT:
sources 'constant' 'reg_bit'
                                                                      sources 'reg_bit_c' 'reg_bit'
guard (par1 = 0)
                                                                      result 'reg_bit'
result 'reg_bit'
                                                                      inline 11
                                                                      '~(' markindent par 15 ' &' cr
forward 2
                                                                      par 2 6 ')' exitindent
 #BinaryOp +~=+
                                                                      #BinaryOp +<=+
" Signed compare not equal constant %0 & single bit,
 removed:
                                                                      " Signed compare less than or equal single bit &
                                                                      constant %1,
sources 'constant' 'reg_bit_c'
                                                                      " removed:
 guard (par1 = 0)
 result 'reg_bit_c'
                                                                      sources 'reg_bit' 'constant'
                                                                      guard (par2 = 1)
 forward 2
                                                                      result 'reg_bit'
 #BinaryOp +<+
                                                                      forward 1
 Signed compare less than between single bits, made
                                                                      #BinaryOp +<=+
 into AND/NOT:
                                                                      Signed compare less than or equal single bit &
 sources 'reg_bit' 'reg_bit_c'
                                                                      constant %1,
 result 'reg_bit'
                                                                       removed:
```

```
forward 2
sources 'reg_bit_c' 'constant'
guard (par2 = 1)
                                                                       #BinaryOp +=<+
result 'reg_bit_c'
                                                                       " Signed compare less than or equal constant %0 &
forward 1
                                                                       single bit,
                                                                        made into NOT:
#BinaryOp +<=+
                                                                       sources 'constant' 'req_bit_c'
" Signed compare less than or equal constant %0 &
                                                                       guard (par1 = 0)
                                                                       result 'reg_bit'
single bit,
" made into NOT:
                                                                       forward 2
sources 'constant' 'reg_bit'
                                                                       #BinaryOp +>+
guard (par1 = 0)
result 'reg_bit_c'
                                                                       " Signed compare more than between single bits, made
                                                                       into AND/NOT:
forward 2
                                                                       sources 'reg_bit_c' 'reg_bit'
#BinaryOp +<=+
                                                                       result 'reg_bit'
" Signed compare less than or equal constant %0 &
single bit,
                                                                       inline 5
" made into NOT:
                                                                       par 1 5 ' &' cr
                                                                       par 2 6
sources 'constant' 'reg_bit_c'
guard (par1 = 0)
                                                                       #BinaryOp +>+
result 'reg_bit'
                                                                       " Signed compare more than single bit & constant %1,
forward 2
                                                                       made into NOT:
#BinaryOp +=<+
                                                                       sources 'reg_bit' 'constant'
                                                                       guard (par2 = 1)
" Signed compare less than or equal between single
                                                                       result 'reg_bit_c'
 " made into NAND/NOT:
                                                                       forward 1
sources 'reg_bit_c' 'reg_bit'
                                                                       #BinaryOp +>+
result 'reg_bit'
                                                                       " Signed compare more than single bit & constant %1,
                                                                       made into NOT:
inline 11
'~(' markindent par 1 5 ' &' cr
par 2 6 ')' exitindent
                                                                       sources 'reg_bit_c' 'constant'
                                                                       guard (par2 = 1)
                                                                       result 'reg_bit'
#BinaryOp +=<+
" Signed compare less than or equal single bit &
                                                                       forward 1
constant %1,
 " removed:
                                                                       #BinaryOp +>+
sources 'reg_bit' 'constant'
                                                                       " Signed compare more than constant %0 & single bit,
guard (par2 = 1)
result 'reg_bit'
                                                                       removed:
                                                                       sources 'constant' 'reg_bit'
forward 1
                                                                       guard (par1 = 0)
                                                                       result 'reg_bit'
#BinaryOp +=<+
                                                                       forward 2
" Signed compare less than or equal single bit &
constant %1,
                                                                       #BinaryOp +>+
 " removed:
                                                                       " Signed compare more than constant %0 & single bit,
                                                                       removed:
sources 'reg_bit_c' 'constant'
guard (par2 = 1)
result 'reg_bit_c'
                                                                       sources 'constant' 'reg_bit_c'
                                                                       guard (par1 = 0)
                                                                       result 'reg_bit_c'
forward 1
#BinaryOp +=<+
                                                                       forward 2
" Signed compare less than or equal constant %0 &
                                                                       #BinaryOp +>=+
single bit,
 " made into NOT:
                                                                       " Signed compare more than or equal between single
                                                                       bits
sources 'constant' 'reg_bit'
                                                                        " made into NAND/NOT:
guard (par1 = 0)
result 'reg_bit_c'
                                                                       sources 'reg_bit' 'reg_bit_c'
                                                                       result 'reg_bit'
```

```
#BinaryOp +=>+
inline 11
                                                                      " Signed compare more than or equal single bit &
'~(' markindent par 1 5 ' &' cr
par 2 6 ')' exitindent
                                                                      constant %0,
                                                                      " made into NOT:
#BinaryOp +>=+
                                                                      sources 'reg_bit_c' 'constant'
" Signed compare more than or equal single bit &
                                                                      guard (par2 = 0)
constant %0,
                                                                      result 'reg_bit'
" made into NOT:
                                                                      forward 1
sources 'reg_bit' 'constant'
guard (par2 = 0)
                                                                      #BinaryOp +=>+
result 'reg_bit_c'
                                                                      " Signed compare more than or equal constant %1 &
forward 1
                                                                      single bit,
                                                                      " removed:
#BinaryOp +>=+
                                                                      sources 'constant' 'reg_bit'
                                                                      guard (par1 = 0)
" Signed compare more than or equal single bit &
constant %0
                                                                      result 'reg_bit'
" made into NOT:
                                                                      forward 2
sources 'reg_bit_c' 'constant'
guard (par2 = 0)
                                                                      #BinaryOp +=>+
result 'reg_bit'
                                                                      " Signed compare more than or equal constant %1 &
forward 1
                                                                      single bit,
                                                                      " removed:
#BinaryOp +>=+
                                                                      sources 'constant' 'reg_bit_c'
" Signed compare more than or equal constant %1 &
                                                                      guard (par1 = 0)
single bit,
                                                                      result 'reg_bit_c'
" removed:
                                                                      forward 2
sources 'constant' 'reg_bit'
guard (par1 = 0)
                                                                      "======
                                                                                                     ______
result 'reg_bit'
                                                                                 Concatenation Operator
forward 2
                                                                       ----begin (vector, vector)-> vector----
#BinaryOp +>=+
                                                                      #BinaryOp,
" Signed compare more than or equal constant %1 &
                                                                      " Concatenation between vectors:
single bit,
 " removed:
                                                                      sources 'concat_mltbit' 'concat_mltbit'
                                                                      result 'concat_mltbit'
sources 'constant' 'reg_bit_c'
guard (par1 = 0)
                                                                      inline 30
result 'reg_bit_c'
                                                                      par 1 0 ',' par 2 0
forward 2
                                                                      #BinaryOp,
#BinaryOp +=>+
                                                                      " Concatenation between vectors:
" Signed compare more than or equal between single
bits,
                                                                      sources 'concat_mltbit' 'reg_bit_vector'
 " made into NAND/NOT:
                                                                      result 'concat_mltbit'
sources 'reg_bit' 'reg_bit_c'
                                                                      inline 30
result 'reg_bit'
                                                                      par 1 0 ',' par 2 0
inline 11
                                                                      #BinaryOp,
 '~(' markindent par 1 5 ' &' cr
par 2 6 ')' exitindent
                                                                      " Concatenation between vectors:
 #BinaryOp +=>+
                                                                      sources 'reg_bit_vector' 'concat_mltbit'
                                                                      result 'concat_mltbit'
 " Signed compare more than or equal single bit &
constant %0,
                                                                      inline 30
 " made into NOT:
                                                                      par 1 0 ',' par 2 0
sources 'reg_bit' 'constant'
                                                                      #BinaryOp,
 guard (par2 = 0)
 result 'reg_bit_c'
                                                                      " Concatenation between vectors:
 forward 1
                                                                      sources 'reg_bit_vector' 'reg_bit_vector'
                                                                      result 'concat_mltbit'
```

```
* Concatenation between bit and vector:
inline 30
par 1 0 ',' par 2 0
                                                                        sources 'reg_bit_c' 'concat_mltbit_c'
                                                                        result 'concat_mltbit_c'
"----end-(vector,vector)-> vector----
                                                                        inline 30
" -----begin (vector_c,vector_c)-> vector_c -----
                                                                        par 1 0 ',' par 2 0
#BinaryOp,
                                                                        #BinaryOp,
* Concatenation between vectors:
                                                                        " Concatenation between bit and vector:
                                                                        sources 'reg_bit_c' 'reg_bit_vector_c'
sources 'concat_mltbit_c' 'concat_mltbit_c'
                                                                        result 'concat_mltbit_c'
result 'concat_mltbit_c'
inline 30
                                                                        inline 30
par 1 0 ',' par 2 0
                                                                        par 1 0 ',' par 2 0
#BinaryOp,
                                                                        "----end-(bit_c,vector_c)-> vector_c----
" Concatenation between vectors:
                                                                        "----begin-(vector,bit)-> vector----
sources 'concat_mltbit_c' 'reg_bit_vector_c'
                                                                        #BinaryOp,
result 'concat_mltbit_c'
                                                                        " Concatenation between bit and vector:
inline 30
par 1 0 ',' par 2 0
                                                                        sources 'concat_mltbit' 'reg_bit'
                                                                        result 'concat_mltbit'
#BinaryOp,
                                                                        inline 30
"Concatenation between vectors:
                                                                        par 1 0 ',' par 2 0
sources 'reg_bit_vector_c' 'concat_mltbit_c'
                                                                        #BinaryOp,
result 'concat_mltbit_c'
                                                                        " Concatenation between bit and vector:
inline 30
par 1 0 ',' par 2 0
                                                                        sources 'reg_bit_vector' 'reg_bit'
                                                                        result 'concat_mltbit'
#BinaryOp,
                                                                        inline 30
" Concatenation between vectors:
                                                                        par 1 0 ',' par 2 0
sources 'reg_bit_vector_c' 'reg_bit_vector_c'
                                                                        "----end-(vector,bit)-> vector----
result 'concat_mltbit_c'
                                                                        "----begin-(vector_c,bit_c)-> vector_c ----
inline 30
par 1 0 ',' par 2 0
                                                                        #BinaryOp,
"----end-(vector_c,vector_c)-> vector_c ----
                                                                        " Concatenation between bit and vector:
"----begin-(bit,vector)-> vector----
                                                                        sources 'concat_mltbit_c' 'reg_bit_c'
                                                                        result 'concat_mltbit_c'
#BinaryOp,
                                                                        inline 30
* Concatenation between bit and vector:
                                                                        par 1 0 ',' par 2 0
sources 'reg_bit' 'concat_mltbit'
                                                                        #BinaryOp,
result 'concat_mltbit'
                                                                        " Concatenation between bit and vector:
inline 30
par 1 0 ',' par 2 0
                                                                        sources 'reg_bit_vector_c' 'reg_bit_c'
                                                                        result 'concat_mltbit_c'
#BinaryOp,
                                                                        inline 30
                                                                        par 1 0 ',' par 2 0
" Concatenation between bit and vector:
sources 'reg_bit' 'reg_bit_vector'
                                                                        "----end-(vector_c,bit_c)-> vector_c -----
result 'concat_mltbit'
                                                                        "-----begin-(bit,bit)-> vector-----
inline 30
par 1 0 ',' par 2 0
                                                                        #BinaryOp,
"----end-(bit,vector)-> vector----
                                                                        Concatenation between bit and bit:
"-----begin-(bit_c,vector_c)-> vector_c -----
                                                                        sources 'reg_bit' 'reg_bit'
                                                                        result 'concat_mltbit'
#BinaryOp,
                                                                        inline 30
```

```
par 1 0 ',' par 2 0

"----end-(bit,bit)-> vector----"

"----begin-(bit_c,bit_c)-> vector_c -----

#BinaryOp ,

" Concatenation between bit and bit:

sources 'reg_bit_c' 'reg_bit_c' result 'concat_mltbit_c'

inline 30
par 1 0 ',' par 2 0

"----end-(bit_c,bit_c)-> vector_c -----"

"END OF FILE:
. THIS IS THE END
```