Template file for IDaSS to HDL-Verilog generation

Lin, X.

Award date:
1997

Eindhoven University of Technology

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 (***I***nteractive ***D***esign ***a***nd ***S***imulation ***S***ystem) 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

The ***I***nteractive ***D***esign ***a***nd ***S***imulation ***S***ystem (IDaSS) is a research project of the section ***I***nformation and ***C***ommunication ***S***ystems (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 ***R***egister ***T***ransfer ***L***evel (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 ***H***ardware ***D***escription ***L***anguages (HDLs). In this way, the template files allow translation of IDaSS designs into the languages which are suitable for commercial silicon compilers.

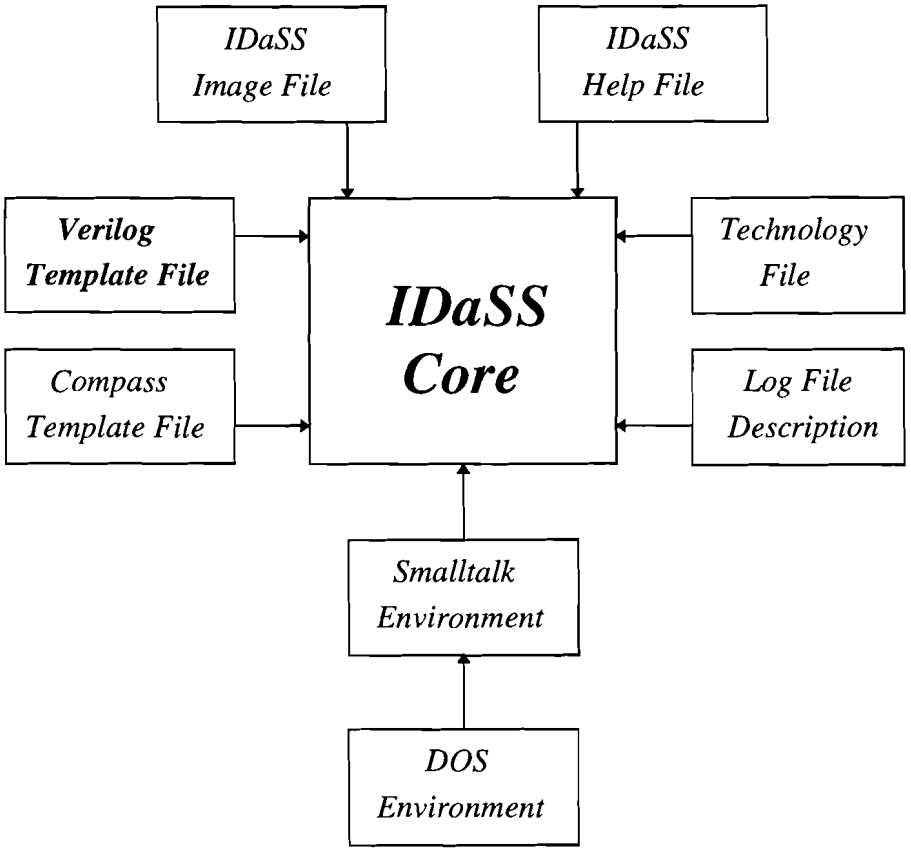

Figure 1. *The IDaSS software package*

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 ***A***lien ***F***ile ***T***emplates (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.

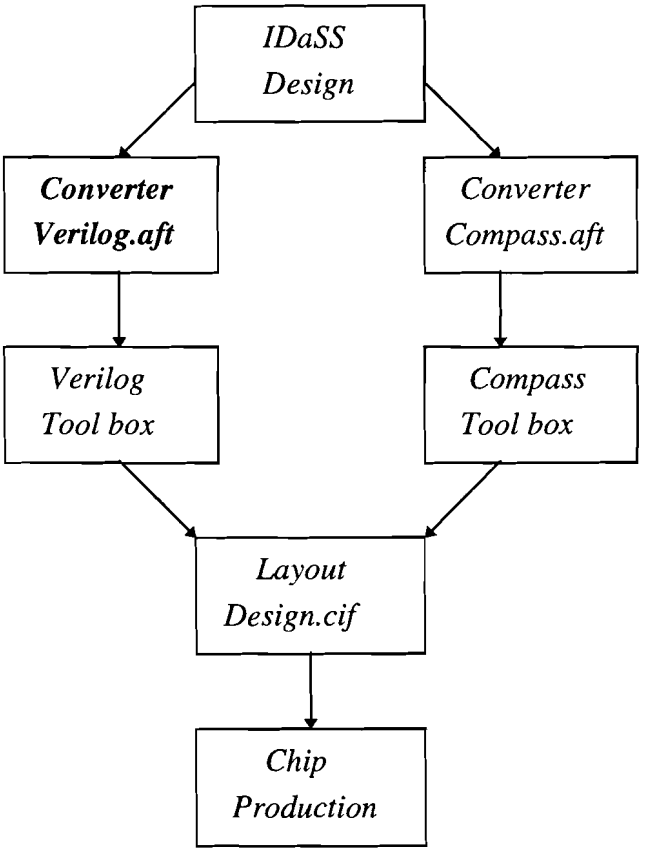

Figure 2. *Steps from the IDaSS digital system design to chip production*

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

- *IDaSS design* : the digital systems designed in IDaSS
- *Verilog.aft* : 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 (***V***HSIC-***HDL***) was started in 1981, based on the United States Department of Defense's ***V***ery ***H***igh ***S***peed ***I***ntegrated ***C***ircuit (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 (**U**nified **D**esign **L**anguage for **I**ntegrated 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
input   [ k:0 ]         Var;            // The interface specification
output  [ m:0]          Data;
inout   [ n:0 ]         Bidir;          //Integer k,m,.n specify the range in bits
reg     [ p:0 ]         Temp_var;
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.

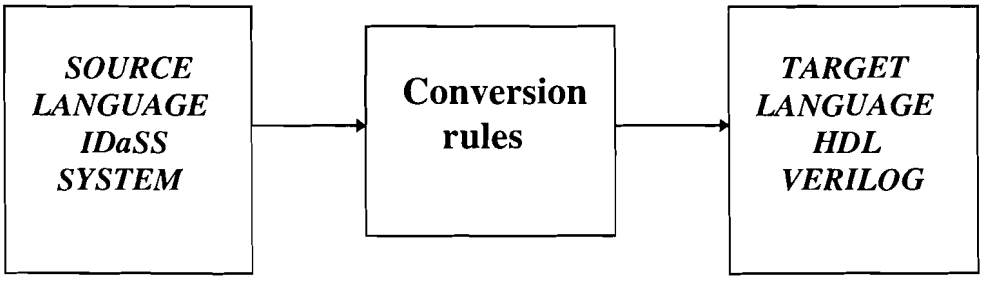

Figure 3. *The conversion rules from source to target*

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:

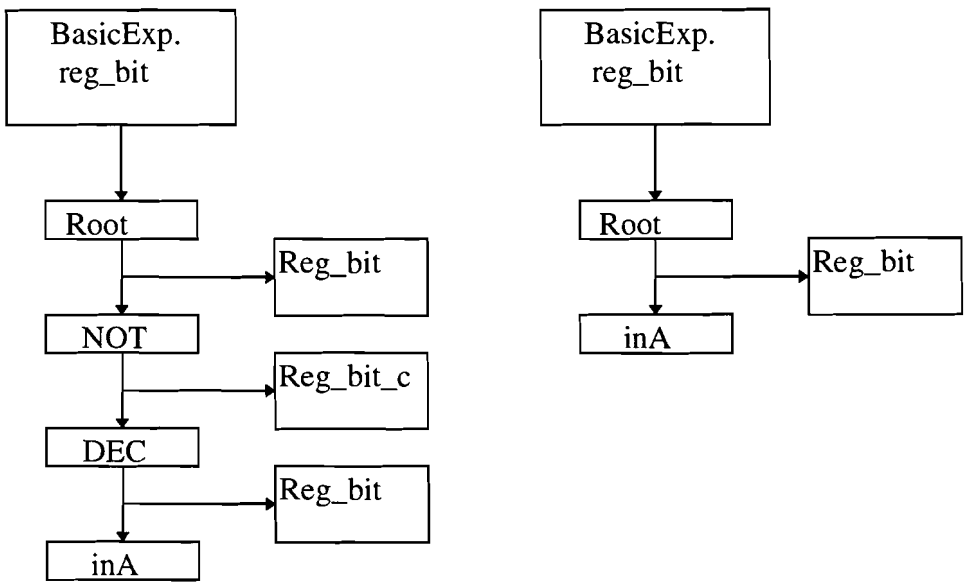

Figure 4. *The reduction of the sub-expression tree*

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 1 7.*
- 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 signficant one bit mask
lsone	least significant one bit position
lszmask	least significant zero bit mask
lszero	least significant zero bit position
maj	majority gate
msomask	most significant one bit mask
msone	most significant one bit position
mszmask	most significant zero bit mask
mszero	most significant zero bit position
neg	two's complement negative
not	complement bits (logical NOT)
onecnt	count number of ONEs in word
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;
reg      [7:0]    result;           // Test result for output
reg               found;

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
```

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
cr
'  // Equivalent of an IDaSS most significant one mask priority' cr
'  // (msomask) operator on a ' (par1 width) decimal ' bits word.' cr
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
Λ	logical AND
~Λ	logical NAND
V	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 Par1	source type2 par2	result type
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. 4th 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_lsomask

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;
reg      [7:0]    result;           // Test result for output
reg               found;

begin
result = 8'b00000000;
found = 0;
  for (index = 0; index <= 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
begin
$display ("function (d2vLSOMASK) = %b", d2vLSOMASK(8'b11101110));
                                    // Calling the function and display the result, when par1 = 11101110
end
endmodule                           // This is the end of the module fn_lsomask
```

```
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
reg               found;

begin
result = 8'b00000000;
found = 0;
for (index = 0; index <= 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_lszero;                    // 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;
reg      [7:0]    result;            // Test result for output
reg               found;

begin
result = 8'b00000000;
found = 0;
for (index = 0; index <= 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
endfunction                          // This is the end of the function d2vLSZERO

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;
reg      [7:0]    result;           // Test result for output
reg               found;

begin
result = 8'b00000000;
found = 0;
for (index = 0; index <= 7; index  = index  + 1 )
  begin
  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_lszmask
```

```
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 <= 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

input    [6:0]    par1;         // Test input
integer  index,cnt0,cnt1;
reg              result;       // Test result for output

begin
result = 0;
cnt0=0;
cnt1=0;
for (index = 0; index <= 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
reg              found;

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

input    [7:0]    par1;             // Test input
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 )
  begin
  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
endfunction                         // This is the end of the function d2vMSONE

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

input    [7:0]    par1;              // Test input
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 )
  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

input    [7:0]   par1;          // Test input
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 )
begin
  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
```

```
module fn_onecnt;                    // This is the 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
integer           index;
reg      [7:0]    cnt;               // Test result for output

begin
cnt=0;
for (index = 0; index <= 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
begin
$display ("function (d2vONECNT) = %d", d2vONECNT(8'b11110011));
                                     // Calling the function and display the result, when par1 = 11110011
end
endmodule                            // This is the end of the module fn_onecnt



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

input    [7:0]    par1;              // Test input
integer           index;
reg      [7:0]    cnt;               // Test result for output

begin
cnt=0;
for (index = 0; index <= 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
begin
$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
```

```
module fn_epty;                    // This is the module fn_epty

function  d2vEPTY;
                                   // This is the start of the function d2vEPTY
                                   // The simulation for even parity test in HDL-Verilog

input    [7:0]    par1;            // Test input
integer           index;
reg               result;          // Test result for output

begin
  result = 1;
  for (index = 0; index <= 7; index  = index  + 1 )
  result=result^par1[index];
  d2vEPTY=result;                  // This is the output of the test result
end
endfunction                        // This is the end of the function d2vEPTY

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

input    [7:0]    par1;            // Test input
integer           index;
reg               result;          // Test result for output

begin
result = 0;
for (index = 0; index <= 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
integer           index;
reg      [3:0]    result;           // Test result for output
reg      [3:0]    lowbit;
reg      [4:0]    accu;


begin
accu = 5'b00000;
for (index = 0; index <= 3; index  = index  + 1 )
  begin
  if  (r[index] ==1)
          begin
          if  (index == 3)
                    accu=accu - {l[3],l};
                    else
                    accu= accu + {l[3],l};
          end
  lowbit [index] = accu [0];
  accu={accu[4],accu[4:1]};
  end

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 l = 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
cr
  " This is the signed multiply function for HDL-Verilog
cr
  '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
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
  '  begin' cr
  '    if  (par2[index] ==1)' cr
  '    begin' cr
  '        if  (index == ' (par1 width - 1) decimal ')' cr
  '          accu= accu - {par1 [' (par1 width - 1) decimal '], par1};' cr
  '          else' 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
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;
wire     [ 7:0 ]  X_in;
wire     [ 7:0 ]  X_out;

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;
input    [ 7:0 ]  X_in;
output   [ 7:0 ]  X_out;
reg      [ 7:0 ]  d2vCONTENTS;
reg      [ 7:0 ]  d2vOUT;
reg               d2vCTRL;
reg               d2vTEST1, d2vTEST2;
reg               d2vCURRSTATE;
wire     [ 7:0 ]  X_out=d2vOUT;
```

```
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;
end
endmodule //This is the end of the module shiftregister
```

Appendix 5 The template file

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

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

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

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

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

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

```
sources 'reg_bit_c'
result  'reg_bit'
"
forward 1
"
#UnaryOp epty
"
" Even parity on a single bit, done with NOT:
"
sources 'reg_bit'
result  'reg_bit_c'
"
forward 1
"
#UnaryOp epty
"
" Even parity on a single bit, done with NOT:
"
sources 'reg_bit_c'
result  'reg_bit'
"
forward 1
"
#UnaryOp epty
"
" Even parity on two bits, done with NOT XOR:
"
sources 'reg_bit_vector'
guard   (par1 width = 2)
result  'reg_bit_c'
"
inline 4
par 1 31 '[0] ^' cr
par 1 31 '[1]'
"
#UnaryOp epty
"
" Even parity on > 2 bits, uses function:
"
sources 'reg_bit_vector'
guard   (par1 width > 2)
result  'reg_bit'
"
function
'  ' markindent
'function [' (par1 width - 1) decimal ':0 ] d2vEPTY;' cr
'input [' (par1 width - 1) decimal ':0 ] par1;' cr
cr
'  // Equivalent of IDaSS even parity (epty) operator' cr
'  // on a ' (par1 width) decimal ' bits word.' cr
cr
'integer  index;' cr
'reg  result;' cr
'begin' cr
'   result =1;' cr
'   for (index = 0; index <=' (par1 width - 1) decimal ';
index  = index  + 1 )' cr
'    result = result ^par1[index];' cr
'   d2vEPTY=result;' cr
'end' cr
'endfunction // function d2vEPTY' cr
exitindent cr cr
"
inline 30
generatefunction
'd2vEPTY' (par1 width) decimal '(' markindent par 1 0
exitindent ')'
"
#UnaryOp inc
"
" Increment on bit vector done with add:
"
sources 'reg_bit_vector'
result  'reg_bit_vector'
"
inline 9
par 1 9 ' +' cr
((1 width: par1 width) width) decimal "'b' (1 width: par1
width) binary
"
#UnaryOp inc
"
" Increment on single bit equals NOT operator:
"
sources 'reg_bit'
result  'reg_bit_c'
"
forward 1
"
#UnaryOp inc
"
" Increment on single bit equals NOT operator:
"
sources 'reg_bit_c'
result  'reg_bit'
"
forward 1
"
"#UnaryOp log2
"
" Log2 operator, never seen here - always returns
constant...
"
#UnaryOp lsomask
"
" Least significant one mask priority on single bit is
removed:
"
sources 'reg_bit'
result  'reg_bit'
"
forward 1
"
#UnaryOp lsomask
"
" Least significant one mask priority on single bit is
removed:
"
sources 'reg_bit_c'
result  'reg_bit_c'
"
forward 1
"
#UnaryOp lsomask
"
" Least 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 ] d2vLSOMASK;'
cr
'input [' (par1 width - 1) decimal ':0 ] par1;' cr
cr
'  // Equivalent of an IDaSS least significant one mask
priority' cr
'  // (lsomask) operator on a ' (par1 width) decimal ' bits
word.' cr
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 = 0; index <=' (par1 width - 1) decimal ';
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
'  d2vLSOMASK=result;' cr
'end' cr
'endfunction // function d2vLSOMASK' cr
exitindent cr cr
"
inline 30
generatefunction
'd2vLSOMASK' (par1 width) decimal '(' markindent par
1 0 exitindent ')'
"
#UnaryOp lsone
"
" Least significant one bit pos priority on single bit is
done with NOT:
"
sources 'reg_bit'
result  'reg_bit_c'
"
forward 1
"
#UnaryOp lsone
"
" Least significant one bit pos priority on single bit is
done with NOT:
"
sources 'reg_bit_c'
result  'reg_bit'
"
forward 1
"
#UnaryOp lsone
"
" Least significant one bit pos priority on bit vector uses
function:
"
sources 'reg_bit_vector'
result  'reg_bit_vector'
"
function
' ' markindent
'function [' (par1 width - 1) decimal ':0 ] d2vLSONE;' cr
'input [' (par1 width - 1) decimal ':0 ] par1;' cr
cr
'  // Equivalent of an IDaSS least significant one bit
position' cr
'  // priority (lsone) operator on a ' (par1 width) decimal '
bits word.' cr
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 = 0; index <=' (par1 width - 1) decimal ';
index  = index  + 1 )' cr
'  begin' cr
'    if (~found &&  par1[index]==1)' cr
'    begin' cr
'    result=index;' cr
'    found=1;' cr
'    end' cr
'  if (~found) result =' (par1 width ) decimal ';' cr
'  end' cr
'  d2vLSONE=result;' cr
'end' cr
'endfunction // function d2vLSONE' cr
exitindent cr cr
"
inline 30
generatefunction
'd2vLSONE' (par1 width) decimal '(' markindent par 1 0
exitindent ')'
"
#UnaryOp lszero
"
" Least significant zero bit pos priority on single bit is
removed:
"
sources 'reg_bit'
result  'reg_bit'
"
forward 1
"
#UnaryOp lszero
"
" Least significant zero bit pos priority on single bit is
removed:
"
sources 'reg_bit_c'
result  'reg_bit_c'
"
forward 1
"
#UnaryOp lszero
"
" Least significant zero bit pos priority on bit vector
uses function:
"
sources 'reg_bit_vector'
result  'reg_bit_vector'
"
function
' ' markindent
'function [' (par1 width - 1) decimal ':0 ] d2vLSZERO;'
cr
'input [' (par1 width - 1) decimal ':0 ] par1;' cr
cr
'  // Equivalent of an IDaSS least significant zero bit
position' cr
'  // priority (lszero) operator on a ' (par1 width) decimal
' bits word.' cr
cr
'integer  index;' cr
'reg  found;' cr
'reg [' (par1 width - 1) decimal ':0 ] result;' cr
'begin' cr
'  result =' ((par1 width zeroes) width) decimal '''b'
(par1 width zeroes) binary ';' cr
'  found = 0;' cr
'  for (index = 0; index <=' (par1 width - 1) decimal ';
index  = index  + 1 )' cr
'  begin' cr
'    if (~found &&  par1[index]==0)' cr
'    begin' cr
'    result=index;' cr
'    found=1;' cr
'    end' cr
'  if (~found) result =' (par1 width ) decimal ';' cr
'  end' cr
'  d2vLSZERO=result;' cr
'end' cr
'endfunction // function d2vLSZERO' cr
exitindent cr cr
"
inline 30
generatefunction
'd2vLSZERO' (par1 width) decimal '(' markindent par 1
0 exitindent ')'
"
#UnaryOp lszmask
"
" Least significant zero mask priority on single bit uses
NOT:
"
sources 'reg_bit'
result  'reg_bit_c'
"
forward 1
"
#UnaryOp lszmask
"
```

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

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

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

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

```
sources 'reg_bit'
result  'reg_bit'
"
forward 1
"
#UnaryOp rev
"
" Reverse all bits in a word, removed for single bit:
"
sources 'reg_bit_c'
result  'reg_bit_c'
"
forward 1
"
#UnaryOp rev
"
" Reverse all bits in a bit vector:
"
sources 'reg_bit_vector'
result  'reg_bit_vector'
"
inline 3
'{ '
FOR (par1 width - 1)
DO
par 1 31 '[' (FORCNT - 1) decimal '] ,'
IF (FORCNT /\ 3 = 0)
THEN
  cr
ELSE
  ' '
ENDIF
ENDFOR
par 1 31 '[' (par1 width - 1) decimal ']'
' }'
"
#UnaryOp rev
"
" Reverse all bits in a bit vector:
"
sources 'reg_bit_vector_c'
result  'reg_bit_vector_c'
"
inline 3
'{ '
FOR (par1 width - 1)
DO
par 1 31 '[' (FORCNT - 1) decimal '] ,'
IF (FORCNT /\ 3 = 0)
THEN
  cr
ELSE
  ' '
ENDIF
ENDFOR
par 1 31 '[' (par1 width - 1) decimal ']'
' }'
"
"#UnaryOp width
"
" Check width of a word, never seen here...
"
#UnaryOp zerocnt
"
" Count number of %0 bits in a single bit is done with
NOT:
"
sources 'reg_bit'
result  'reg_bit_c'
"
forward 1
"
#UnaryOp zerocnt
"
" Count number of %0 bits in a single bit is done with
NOT:
"
sources 'reg_bit_c'
result  'reg_bit'
"
forward 1
"

#UnaryOp zerocnt
"
" Count number of %0 bits in bit vector needs function:
"
sources 'reg_bit_vector'
result  'reg_bit_vector'
"
function
'  ' markindent
'function [' (par1 width - 1) decimal ':0 ] d2vZEROCNT;'
cr
'input [' (par1 width - 1) decimal ':0 ] par1;' cr
cr
'  // Equivalent of an IDaSS count number of zeroes
(zerocnt) operator' cr
'  // on a ' (par1 width) decimal ' bits word.' cr
cr
' integer  index,count;' cr
 'begin' cr
'   count =' ((par1 width zeroes) width) decimal "'b'
(par1 width zeroes) binary ';' cr
'   for (index = 0; index <=' (par1 width - 1) decimal ';
index  = index  + 1 )' cr
'   if (par1[index]==0) count=count+1;' cr
'   d2vZEROCNT=count;' cr
'end' cr
'endfunction // function d2vZEROCNT' cr
exitindent cr cr
"
inline 30
generatefunction
'd2vZEROCNT' (par1 width) decimal '(' markindent par
1 0 exitindent ')'
"
#UnaryOp zeroes
"
" Generate constant zero bit:
"
sources 'constant'
guard   (par1 = 1)
result  'reg_bit'
"
inline 30
'0'
"
#UnaryOp zeroes
"
" Generate field of constant zeroes:
"
sources 'constant'
guard   (par1 > 1)
result  'reg_bit_vector'
"
inline 30
((par1 zeroes) width) decimal "'b' (par1 zeroes) binary
"
"=======================================
"                BINARY OPERATORS
"=======================================
#MacroFor BinaryOp
"
naming
'multiply_signed'
"
contents
'function [' ((par1 width) + (par2 width) -1) decimal ':0 ]
multiply_signed;' cr
cr
  " This is the signed multiply function for HDL-Verilog
cr
  '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
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
 '  begin' cr
 '    if  (par2[index] ==1)' cr
 '    begin' cr
 '       if  (index == ' (par1 width - 1) decimal ')' cr
 '          accu= accu - {par1 [' (par1 width - 1) decimal
'], par1};' cr
 '          else' 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
cr
"
#MacroFor BinaryOp
"
naming
'gentempvar1'
"
contents
'reg '
   IF (par1 width > 1)
   THEN
   '[ ' (par1 width - 1 width: 8) decimalleft ':0 ]'
   ENDIF
'd2vTEMP' (INDEX) decimal ';' tabto 54
"
#MacroFor BinaryOp
"
naming
'gentempvar2'
"
contents
'reg '
   IF (par2 width > 1)
   THEN
   '[ ' (par2 width - 1 width: 8) decimalleft ':0 ]'
   ENDIF
'd2vTEMP' (INDEX) decimal ';' tabto 54
"
#BinaryOp +
"
" Add bit vectors, standard work:
"
sources 'reg_bit_vector' 'reg_bit_vector'
result  'reg_bit_vector'
"
inline 9
par 1 9 ' +' cr
par 2 10
"
#BinaryOp +
"
" Add bit vector and non-zero constant:
"
sources 'reg_bit_vector' 'constant'
guard   (par2 ~= 0)
result  'reg_bit_vector'
"
inline 9
par 1 9 ' +' cr
(par1 width) decimal "'b' (par2 width: par1 width) binary
"
#BinaryOp +
"
" Add bit vector and zero constant, removed:
"
sources 'reg_bit_vector' 'constant'
guard   (par2 = 0)
result  'reg_bit_vector'
"
forward 1
"
#BinaryOp +
"
" Add bit vector and zero constant, removed:
"
sources 'reg_bit_vector_c' 'constant'
guard   (par2 = 0)
result  'reg_bit_vector_c'
"
forward 1
"
#BinaryOp +
"
" Add non-zero constant and bit vector:
"
sources 'constant' 'reg_bit_vector'
guard   (par1 ~= 0)
result  'reg_bit_vector'
"
inline 9
(par2 width) decimal "'b' (par1 width: par2 width) binary
' +' cr
par 2 10
"
#BinaryOp +
"
" Add zero constant and bit vector, removed:
"
sources 'constant' 'reg_bit_vector'
guard   (par1 = 0)
result  'reg_bit_vector'
"
forward 2
"
#BinaryOp +
"
" Add zero constant and bit vector, removed:
"
sources 'constant' 'reg_bit_vector_c'
guard   (par1 = 0)
result  'reg_bit_vector_c'
"
forward 2
"
#BinaryOp +
"
" Add single bit values, converted to XOR:
"
sources 'reg_bit' 'reg_bit'
result  'reg_bit'
"
inline 4
par 1 4 ' ^' cr
par 2 5
"
#BinaryOp +
"
" Add single bit and Constant %0, removed:
"
sources 'reg_bit' 'constant'
guard   (par2 = 0)
result  'reg_bit'
"
forward 1
"
#BinaryOp +
"
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

#BinaryOp ,
"
" Concatenation between vectors:
"
sources 'concat_mltbit' 'reg_bit_vector'
result 'concat_mltbit'
"
inline 30
par 1 0 ',' par 2 0
"
#BinaryOp ,
"
" Concatenation between vectors:
"
sources 'reg_bit_vector' 'concat_mltbit'
result 'concat_mltbit'
"
inline 30
par 1 0 ',' par 2 0
"
#BinaryOp ,
"
" Concatenation between vectors:
"
sources 'reg_bit_vector' 'reg_bit_vector'
result 'concat_mltbit'
```

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