

Reference Manual



Nimbus - Personal

Copyright Message

2003 Copyright © EXSEDIA.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission in writing from EXSEDIA.

www.exsedia.com

Table of Contents

- **Introduction**
Briefly outlines the organization and usage of this manual
- **The flowdiagram concept**
Introduces the general Nimbus design methodology using the flowdiagram
- **Flowdiagram objects**
Describes the flowdiagram components and their representation of the circuit design logic
- **Nimbus expression language**
Describes the expressions associated with the flowdiagram objects
- **Flowdiagram rules**
Describes the do's and dont's of designing using flowdiagrams
- **Macro functions**
Describes all the pre-defined macro functions provided in the Nimbus
- **Nimbus reserved words**
Lists all the reserve words used by VHDL, Verilog, and Nimbus
- **File formats**
Describes the file produced by Nimbus
- **Error messages**
Describes all the available error messages in Nimbus

Introduction

The **Nimbus Reference Manual** provides technical design information for designers to understand the principle of design entry using flowdiagrams - the Nimbus graphical design entry format. Designers will find this manual useful in getting them started on presenting their electronic designs in flowdiagram by familiarizing with the usage of each flowdiagram component.

The flowdiagram software package contains the following four manuals:

- User's Manual
- Reference Manual
- Tutorial
- Application Notes

The Reference Manual covers all the technical topics relating to the Nimbus design methodology and design environment.

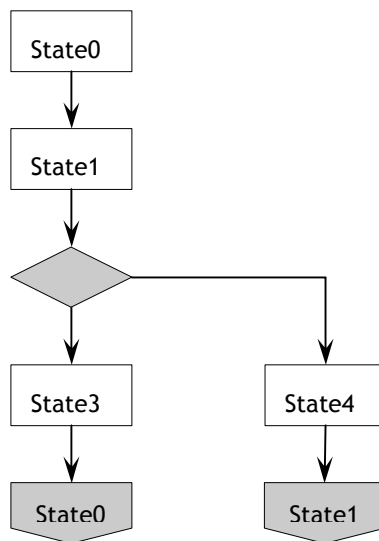
The flowdiagram concept

It is well known that a picture is worth more than a thousand words. This holds especially true in the world of digital design. Nimbus introduces the flowdiagram method of input for electronic design for generating HDL codes. This method of design entry can be easily adapted by engineers since the structure of a flowdiagram is comparable to a flowchart. The flowdiagram presents a simple and clear way to design a complex circuit logic, enhancing communication among design team members.

Flowdiagram paradigm

A flowdiagram is a graphical representation of a digital design from the standpoint of a time-based sequencing of control and data path actions that specify the behavior of a digital circuit. Though primarily based on an algorithms state machine (ASM) model, flowdiagram also allow designers to specify the sequencing of concurrent data path actions in the digital circuit.

Flowdiagrams can support both Moore and Mealy finite-state machines. A major enhancement of ASM over the Finite State Machine (FSM) model is the ability to describe combinational data paths and sequential delay elements in a flowdiagram representation of the design. Data paths in flowdiagrams are specified with expressions describing the data path operations; this capability is further augmented using combinational macro-functions. A flowdiagram consists of graphical objects, which have associated expressions describing the behavior of a design as shown in the figure below.





Modeling a digital design as a flowdiagram

To begin discussing the specific notation and constructs associated with flowdiagrams, we first discuss the point of view of a digital system represented by a flowdiagram. Flowdiagrams represent the synchronous and asynchronous sequencing of control and data path functions of a design. The digital system is modeled from the vantage point of the control circuitry, looking outward into the data path elements of the design.

Though a flowdiagram allows data path elements (including combinational, sequential and steering functions) to be modeled, they are modeled from the standpoint of sequencing and control of these functions, and not from the standpoint of the structural characteristics of these elements. This is difficult than, say, a schematic model of the circuit from a structural viewpoint. The structure of the data path elements (with the exception of specifically-defined macro-functions) are implied through their reference as “buses”.

A “bus” is a named signal path through the design that is relevant from a sequencing and control standpoint. Unlike a schematic representation, a flowdiagram doesn’t model all paths through a circuit; rather, it models those paths over which there is some explicit sequencing, gating or control behavior to be specified. In this sense, it is not a complete behavioral description of the design to be specified.

Thus, the purpose of having a flowdiagram representation of a digital system is to explicitly model the sequencing and control behavior of the system. It is not meant to replace schematic or other structural representations of the system, but to augment the available information about the design by providing a view that allows emphasis of control-oriented information. This is important because in many types of digital systems (ASICs and FPGAs) as example, a very high percentage of a designer’s time is spent in specifying and verifying the control behavior. (Specific figures published indicate that, in a 50K gate design, only 10% of the gate count is associated with the control logic, but the designer may spend 90% of their time on that small portion getting the control logic correct).



Sheet

A flowdiagram consists of one or more sheets. A sheet is a logical partition of a flowdiagram which is analogous to a sheet of paper. In a flowdiagram with more than one sheet, input labels and output labels are used to describe the control flow between sheets. The purpose of using sheets is a visual aid in partitioning the design into manageable pieces (either hierarchically or laterally) in order to reduce the complexity of larger designs. Using sheets improves readability, particularly when incorporating printouts of flowdiagrams as part of a design's documentation.

There is essentially no limit to the number of flowdiagram sheets that can be created using Nimbus, i.e. there is no limit to the size of a flowdiagram, as long as there is sufficient memory and disk space.



Subflowdiagram Definition

A flowdiagram with portions of it being several times across a design can be defined once and then reused. This portion of the flowdiagram is called a subflowdiagram definition. When a subflowdiagram definition is completed, the flowdiagram references to this subflowdiagram repeatedly.

The flowdiagram that references to a subflowdiagram definition is at level higher than that of the subflowdiagram definition itself. A subflowdiagram definition behaves just like a flowdiagram. The control flow between one level to another is indicated by using subflowdiagram references and input/output connectors.

A top level flowdiagram does not contain any input/output connectors (which are different symbols than input/output labels), since there are no any other levels making reference to this level.

Flowdiagram objects

The design of a flowdiagram is made up of flowdiagram objects interconnected by transactions, also a flowdiagram object. Generally, there are three different classes of flowdiagram objects:

- Behavioral objects
- Marker objects
- Arrows

Each of these classes of flowdiagram objects provides different modeling functions which in combination allow a flowdiagram to graphically describe the behavior of a circuit design.



Behavioral objects

Behavioral objects are flowdiagram objects that describe the behavior of a design system. These behavioral objects are put together sequentially in a meaningful way to construct a behavioral model of a digital design. There are five flowdiagram objects which fall under this class:

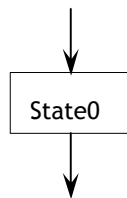
- State
- Condition
- Conditional Output
- Delay Output
- Case

These objects contain expressions that change the value of buses or reroute the transaction to the output nodes based on the evaluation of the expression.

State

A state represents the state of a digital system. At any given time, the digital system can be in only one state (except when there are concurrent controllers present in the design). A state changes synchronously in the absence of any asynchronous signals. In the presence of asynchronous signals, a state can jump to a predefined state (e.g., asynchronous reset signal resets the system by moving the control flow to a user-defined Reset State).

The period of time that a design spends in a state corresponds to the length of the defined clock cycle. The synchronous behavior of a design is implicit in the flow from one state to the next state; a state transition takes at every clock cycle.



A state in a flowdiagram is represented by a rectangle as shown above. Associated with the state is a set of expression that describes the behavior of the design in that state. The expressions are made up of concurrent operations operating on macro function data paths and user-defined buses. The expression can either be an assertion, assignment, or macro function expression. States with no expression acts as a delay or wait state.

BNF declaration

```

StateExpression      ::= { SynchronousExpression }

SynchronousExpression ::= AssertionExpression
                        | AssignmentExpression
                        | MacroExpression
                        | NULL
  
```

Examples

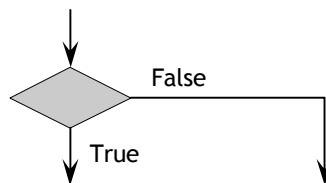
drv_ready	asserts the one-bit signal “drv_ready” for duration of 1 clock cycle, while in the current state.
outreg <- 'H5fe0'	assigns the hex value “5fe0” to the bus “outreg” (implies that the bus width is long enough to hold the assigned value).

Any number of transaction arrows can enter a state, but at most one transition arrow leaves a state. A state with no outgoing arrows implies that it is a terminal state which makes transitions to itself infinitely until an asynchronous inputs is detected. (It would be equivalent to having an output transition from a state that loops back to the same state as input).

Condition

A condition represents a decision in the control flow of a digital system. A condition changes the control flow based on certain operations on the values of certain buses. The operation yields a True or False value. Given this value the nature of the state transition is defined, in that the condition specifies the next state based on the outcome of evaluating the condition. Conditions can be cascaded, and can be interspersed with conditional outputs, so as to specify the “gating structure” associated with a particular state transition. A condition element is shown below.

A condition in a flowdiagram is represented by a diamond. Associated with the condition is an expression that determines the next object in the control flow. The expression is made up of a relational operation on user-defined buses, or may be an implicit evaluation of a Boolean expression (where the outcome of the expression is the result of evaluating a single bit quantity).



BNF declaration

```
ConditionExpression ::= RelationalExpression
```

Examples

Drvready means “if Drvready is set, then ...”

latchreg [3] | latchreg [2] | latchreg [1] | latchreg [0]
means “if bits 3 or 2 or 1 or 0 of latchreg are 1’s, then ...”

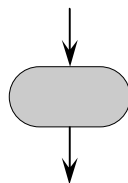
One or more transition arrows may enter a condition, but only two transition arrows leave that condition. The outgoing arrows labeled 1 and 0. If the condition expression yields a “1”, the flow proceeds to the object pointed to by the True path. Otherwise, the flow proceeds to the object pointer to by the False path. An outgoing transition arrow must make an explicit or implicit transition to a state. It cannot loop explicitly or implicitly into itself without making a transition into a state (which might be the same state that precedes the condition).

Conditional/Transitional output

A conditional/transitional output may be defined for two different modeling situations. First, a conditional output represents the placement of asynchronous output signals in a digital system. The placement depends on the state and input of the system. This generally corresponds to describing a digital system as a Mealy machine. The conditional output is defined as such, based on the output taking place on the state transition (rather than in the state) that follows a condition evaluation. In this sense, the condition and the resultant conditional output act as a “gating structure” for specifying state transitions involving complex coordination of sequential, combinational and steering logic functions in the data path.

Second, a transitional output (defined as an output on a state transition where there is no condition acting as a gating structure) may be defined so as to introduce a delay between actions that will be occurring during the same clock cycle. The transitional output is implicitly defined as having some delta delay inherent in it, occurring at some point after the leading edge of the clock cycle. This inherent delay can be used to introduce some delay between different actions, where some actions may occur on the leading edge of the clock pulse (corresponding to when the state is first entered) and other actions may occur later (for example, as when operating on the same bus for both actions).

A conditional/transitional output in a flowdiagram is represented by a capsule. Associated with the conditional/transitional output is a list of expressions that describes its behavior. The expressions are made up of concurrent operations operating expression, and may have the same form as expressions attached to state objects.



BNF declaration

```
ConditionalExpression ::= { AsynchronousExpression }

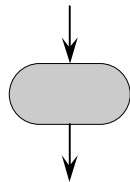
AsynchronousExpression ::= AsynAssertionExpression
                          | AsynMacroExpression
                          | AssignmentExpression
                          | NULL
```

One or more arrows may enter a conditional output, but only one arrow leaves that conditional output. An outgoing arrow must make an explicit or implicit transition to a state. It cannot loop explicitly or implicitly into itself without first making a transition into a state.

Delay output

A delay output (defined as an output on a state transition where there is no condition acting as gating structure) may be defined so as to introduce a delay between actions that will be occurring during the same clock cycle. The transitional output is implicitly defined as having some delta delay inherent in it, occurring at some point after the leading edge of the clock cycle. This inherent delay can be used to introduce some delay between different actions, where some actions may occur on the leading edge of the clock pulse (corresponding to when the state is first entered) and other actions may occur later (for example, as when operating on the same bus for both actions).

A delay output in a flowdiagram is represented by a capsule. Associated with the delay output is a list of expressions that describes its behavior. The expressions are made up of concurrent operations operating on user-defined buses. Each expression may be an assertion or expression, and may have the same form as expressions attached to state objects.



BNF declaration

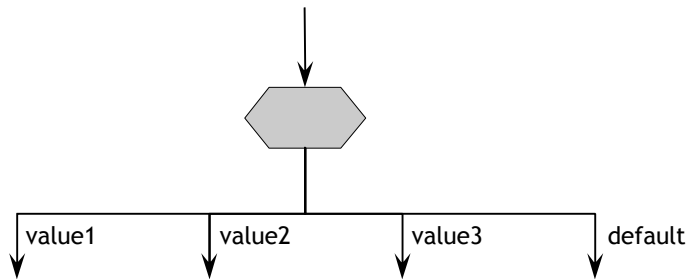
```
delay_output_expression ::= assertion_expression
                          | assignment_expression
                          | {condition_output_expression}
```

One or more transition arrows may enter a delay output, but only one transition arrow may leave that delay output. An outgoing transition arrow must make an explicit or implicit transition to a state. It cannot loop explicitly or implicitly into itself without first making a transition to a next state.

Case

A case represents a decision in the control flow of a digital system. A case changes the control based on the value of a selected input, internal or internal output bus. The bus whose value is to be checked must be more than one bit wide (i.e. no single-bit signals may be used in the expression associated with a case construct).

A case in a flowdiagram is represented by a hexagon. Associated with the case is an input, internal or internal output bus that determines the control flow of the design. The case construct, like the condition construct, is used to represent clearly the logical decision associated with state transitions and sequential control flow in a digital system.



One or more arrows may enter and leave a case. The maximum number of outgoing arrows is 2^n where n is the width of the associated bus. Each outgoing arrow is labeled as one of the possible values of the associated bus.

When using a case to enumerate specific control flow options and transitions, you do not need to completely specify all possible transition paths. There is an arrow labeled default that is used to represent values that are not specified by the other outgoing arrows.

As with the condition, an outgoing arrow must make an explicit or implicit transition to a state. It cannot loop explicitly or implicitly into itself without first making a transition into a state.



Marker objects

The marker objects are flowdiagram objects that provide visual cues as to the continuity of control flow between a top level flowdiagram and subflowdiagram definitions, between different parts of a flowdiagram, as well as between flowdiagrams in different sheets. The marker objects are necessary to make these connections since users cannot place a transition from a flowdiagram sheet to another. They act like an invisible transition.

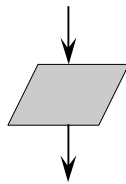
These marker objects support the sheets and subflowdiagrams features in Nimbus by allowing the control to flow between fragments of a flowdiagram. These features allow the users to decompose and partition a complex control flow and data path description into manageable pieces. There are 3 types of marker objects:

- Subflowdiagram reference
- Labels (input and output)
- Connectors (input and output)

Subflowdiagram reference

A subflowdiagram reference represents a reference to an existing subflowdiagram definition. This subflowdiagram can be referenced at multiple locations within a flowdiagram.

A subflowdiagram reference in a flowdiagram is represented by a skewed rectangle. Associated to the subflowdiagram reference is a unique identifier that identifies the subflowdiagram definition. The buses that are manipulated in the subflowdiagram definition are identical to those in flowdiagram i.e. they are global buses.



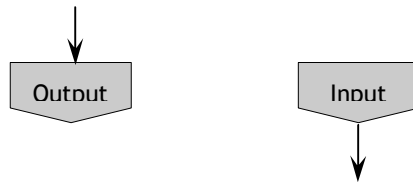
There can be zero or more arrows enter or leave a subflowdiagram reference object. However, the number of incoming and outgoing arrows must match those connectors in the subflowdiagram definition. Each outgoing arrow must have an output identifier. The identifier label on each arrow corresponds to an output connector defined as an exit for control flow to “return” from the subflowdiagram definition located on another sheet. The identifier label and output connector pair off as a control flow path which can be easily identified.



Label

A label represents a continuity of a visually discontinued arrow. There are two types of labels - output and input. An output label represents the beginning of the discontinued section of the same arrow. An output label conceptually connects to an input label of the same identifier.

A label in a flowdiagram is represented by a pentagon. Associated to each label is an identifier. In a flowdiagram, there may be several labels bearing the same identifier, however, there can be only one input label and at least one output label having the same identifier. In another word, many output labels can point to one input label. These labels may be placed in different sheets of a flowdiagram.



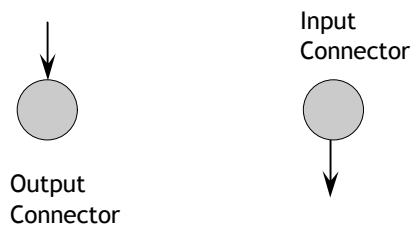
Consequently, one or more arrows may enter an output label, but no arrows leave the output label; and only one arrow may leave an input label, and no arrows enter the input label.



Connector

A connector represents the input and output connections of a subflowdiagram definition to the top level flowdiagram. There are two types of connectors - output and input. An output connector represents the outgoing control flow from the subflowdiagram definition, and an input connector represents the incoming control flow to the subflowdiagram definition.

A connector in a subflowdiagram definition is represented by a cycle. Associated with each connector is an identifier that is unique to the subflowdiagram definition. The connectors can be placed in the subflowdiagram definition only, regardless if the subflowdiagram definition is in one or multiple sheets. The number of input and output connectors of a subflowdiagram definition must match the number of incoming and outgoing arrows of the corresponding subflowdiagram reference.



In this case, one or more arrows may enter an output connector, but no arrows leave it; and only one arrow may leave an input connector with no arrows entering it.



Transition arrow object

A transition arrow traces the control flow from one behavioral or marker object to another. It provides visual continuity between these objects and shows the transition of information.

A transition arrow in a flowdiagram is represented by an unbroken horizontal line or vertical line or combination of both lines. The transition arrow originates from a source object and terminates at a targeted object. The direction of the transition arrow is marked by an arrow head.



Nimbus expression language

The flowdiagram expression language is a system used in writing expressions that are associated with flowdiagram objects. There are different types of expressions for different flowdiagram objects, due to the modeling functions performed by each flowdiagram object. These expressions are written by combining the data types, operators, and macro functions supported by Nimbus.



Data Types

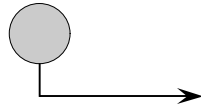
Data types provide values that can be used to determine a further action. Some of these data types carry values that can be manipulated to give specific meanings in the flowdiagram design, e.g. counter which keep track of the current count. There are three data types supported by Nimbus:

- Asynchronous Signal
- Bus
- Constant



Asynchronous signal

An asynchronous signal is an input signal that represents an asynchronous reset and is external to the designed system. One asynchronous signal is used in a flowdiagram. In a flowdiagram, an asynchronous signal is represented by a circle with a curved arrow coming out of it, as shown below.



The asynchronous transition occurs on a low value. An asynchronous signal is associated to a state in the flowdiagram. When the signal is low, the associated state becomes the current state and the flow continues from this state regardless of where the flow was in the last state.

Bus

A bus is collection of signals, which has a binary value. Nimbus uses several user-defined buses in the expressions associated to flowdiagram objects. There are 4 types of buses, which defer by the way they function to handle values:

- Input
- Internal
- Internal Output
- Output
- Bidirectional

An input bus is a collection of external signals entering the designed system. An output bus is collection of signals that leaves the designed system. An internal bus is used within the system. An internal output bus is used internally and which leaves the system. Nimbus allows usage of bus slices in defining expressions.

BNF declaration

WriteBus	::= InternalBus InternalOutputBus OutputBus BidirectionalBus	(* declare internal *) (* declare internal output *) (* declare output *) (* declare bidirectional *)
ReadBus	::= InputBus InternalBus InternalOutputBus BidirectionalBus Constant	(* declare input *) (* declare internal *) (* declare internal output *) (* declare bidirectional *)
ReadWriteBus	::= InternalBus InternalOutputBus BidirectionalBus	(* declare internal *) (* declare internal output *) (* declare bidirectional *)
AsynchronousBus	::= WriteBus ReadWriteBus	(* declare asynchronous *) (* declare asynchronous *)
SynchronousBus	::= WriteBus ReadWriteBus	(* declare synchronous *) (* declare synchronous *)
InputBus	::= BusName BusName ([Integer : Integer])	(* declare multi-bit *)
InternalBus	::= BusName BusName ([Integer : Integer])	(* declare multi-bit *)
InternalOutputBus	::= BusName BusName ([Integer : Integer])	(* declare multi-bit *)
OutputBus	::= BusName BusName ([Integer : Integer])	(* declare multi-bit *)
BidirectionalBus	::= BusName BusName ([Integer : Integer])	(* declare multi-bit *)

WriteBinary	::= InternalBinary (* declare internal *) InternalOutputBinary (* declare internal output *) OutputBinary (* declare output *) BidirectionalBinary (* declare bidirectional *)
ReadBinary	::= InputBinary (* declare input *) InternalBinary (* declare internal *) InternalOutputBinary (* declare internal output *) BidirectionalBinary (* declare bidirectional *) BinaryConstant
ReadWriteBinary	::= InternalBinary (* declare internal *) InternalOutputBinary (* declare internal output *) BidirectionalBinary (* declare bidirectional *)
AsynchronousBinary	::= WriteBinary (* declare asynchronous *) ReadWriteBinary (* declare asynchronous *)
SynchronousBinary	::= WriteBinary (* declare synchronous *) ReadWriteBinary (* declare synchronous *)
InputBinary	::= BusName (* declare 1-bit wide*) BusName ([Integer])
InternalBinary	::= BusName (* declare 1-bit wide*) BusName ([Integer])
InternalOutputBinary	::= BusName (* declare 1-bit wide*) BusName ([Integer])
OutputBinary	::= BusName (* declare 1-bit wide*) BusName ([Integer])
BidirectionalBinary	::= BusName (* declare 1-bit wide*) BusName ([Integer])
Integer	::= 0 ... 32
BusName	::= Alpha { Alphanumeric }
Alpha	::= a ... z A ... Z _
Alphanumeric	::= Alpha Numeric
Numeric	::= 0 ... 9

Examples

Abus	all bits are used
Abus [2]	the second bit of Abus is used
Abus [4:2]	the fourth, third, and second bits of Abus are used. Note: ordering is important
Abus [2:4]	illegal usage of a bus slice. The first slice must be greater or equal to the second slice.

Constant

Constant in Nimbus can be represented only by positive integers. These integers can be expressed in binary, octal, decimal, and hexadecimal.

BNF declaration

```

Constant      ::= ' ( B | b ) BinaryDigit { BinaryDigit } '
               | ' ( O | o ) OctalDigit { OctalDigit } '
               | ' [ ( D | d ) ] DecimalDigit { DecimalDigit } '
               | ' ( H | h ) HexadecimalDigit { HexadecimalDigit } '

BinaryConstant ::= ' ( B | b ) BinaryDigit '
               | ' ( O | o ) OctalDigit '
               | ' [ ( D | d ) ] DecimalDigit '
               | ' ( H | h ) HexadecimalDigit '

BinaryDigit   ::= 0 | 1

OctalDigit    ::= 0 | ... | 7

DecimalDigit  ::= 0 | ... | 9

HexadecimalDigit ::= 0 | ... | 9 | a | ... | f | ... | A | ... | F

```

Examples

'D33'	decimal 33
'H21'	hexadecimal representation of decimal 33
'O41'	octal representation of decimal 33
'B00100001'	binary representation of decimal 33

Operators

Nimbus uses a set of operators that can be used to change or modify values of buses. These operators are divided into different groups:

- Deassertion operator
- Logical not operator
- Logical operator
- Relational operator
- Assignment operator
- Associative operator
- Macro operator

Deassertion operator

The deassertion operator asserts low the value of the operand to its right during a clock cycle. The operator is represented by an exclamation mark, !.

BNF declaration

```
DeassertionOperator ::= !
```

Logical NOT operator

The logical NOT inverts a one-bit value of a bus. It is used in Boolean expressions to evaluate the inverse value of a bit. The logical operator is represented by the caret character, ^.

BNF declaration

```
LogicalNOT ::= ^
```

Assignment operator

An assignment puts the resultant value of an expression to a bus, the operator is represented by the “<-“ sign. In an expression, the operand (bus) on the left hand side of this operator is assigned the value of the evaluation of the operands on the right hand side.

Examples

```
Abus <- '1'           Abus is assigned the value of 1
```

Logical operators

The logical operators evaluate on one bit slice of a bus only, and return a bit of '0' or '1' as result.

BNF declaration:

LogicalOperator	::= LogicalAND LogicalOR LogicalNAND LogicalNOR LogicalXOR LogicalXNOR	
LogicalAND	::= &	(* logical AND *)
LogicalOR	::=	(* logical OR *)
LogicalNAND	::= ^&	(* logical NAND *)
LogicalNOR	::= ^	(* logical NOR *)
LogicalXOR	::= @	(* logical XOR *)
LogicalXNOR	::= ^@	(* logical XNOR *)

Relational operators

Nimbus uses relational operators to compare two values, yielding a result based on whether the comparison is 1 or 0.

BNF declaration:

RelationalOperator	::= EqualOperator NotEqualOperator GreaterOperator GreaterEqualOperator LessOperator LessEqualOperator	
EqualOperator	::= =	(* equal *)
NotEqualOperator	::= !=	(* not equal *)
GreaterOperator	::= >	(* greater than *)
GreaterEqualOperator	::= >=	(* greater than or equal to *)
LessOperator	::= <	(* less than *)
LessEqualOperator	::= <=	(* less than or equal to *)

Associative operators

The associative operators are used to group selected operands and operators together so that they are evaluated as a group. The operators are represented by the parenthesis, (). The left and right parenthesis bound the selected expressions as a group. The result of the evaluation of these expressions inside these operators is applied to the expressions outside the group.

Macro operators

The macro operators in flowdiagrams are used to operate on data buses. Flowdiagrams have a set of predefined macros. Appendix A provides a listing of the predefined macros.

BNF declaration:

MacroFunction	::= MacroName (MacroArgument { , MacroArgument })
MacroArgument	::= ReadBus ReadBinary MacroFunction

Precedence of operators

In an expression, the operators are evaluated sequentially based on the ranking of the operator on the list. Operators with the highest precedence are evaluated first, then the second highest, and all the way to the operators with the lowest precedence until all the operators is evaluated. The following is a summary of the precedence of operators that is used in flowdiagrams evaluated from ‘top-to-bottom’. Operators of the same precedence are evaluated from ‘left-to-right’ of the expression.

Precedence

Deassertion	!
Assignment	<-
Relational	= != > >= < <=
Parenthesis	()
Logical NOT	^
Logical	& ^& ^ @ ^@

Expressions

An expression defines different behavior that involves buses and constants. There are several types of expression that are associated to some of the flowdiagram objects. The behavior of the expression contributes to the functional behavior of the flowdiagram object it is associated with.

Expressions are associated to states, conditional outputs, and conditions. Expressions in a state and conditional output are executed concurrently. Therefore, the buses used in one expression must not depend on the results of other expressions in the same flowdiagram object.

Boolean expression

A Boolean expression is used to determine the result of logical operations on input, internal and internal output buses. The parenthesis, (and), are used to override the precedence of the logical operators. They have the highest precedence.

BNF declaration

```
BooleanExpression ::= [ LogicalNOT ] ReadBinary
                  | [ LogicalNOT ] ( ReadBinary )
                  | ReadBus RelationalOperator ReadBus
                  | BooleanExpression RelationalOperator BooleanExpression
                  | BooleanExpression { LogicalOperator BooleanExpression }
                  | [ LogicalNOT ] ( BooleanExpression )
```

Assertion expression

An assertion expression sets the value of one bit of an output bus to either high or low when the expression is activated. Assertion expressions are associated to states and conditional outputs only.

When a bus is asserted, the value of that bus becomes high, but it is low at other times. When a bus is deasserted, the value of that bus becomes low, but it is high at other times. A bus cannot be asserted and deasserted within a flowdiagram.

BNF declaration

```
AssertionExpression ::= [ DeassertionOperator ] ReadBinary
```

Examples

Abus	Abus, a one bit output bus, is asserted. At the next state, it is low. Nowhere in the flowdiagram can !Abus be used.
!Bbus [4]	The fourth bit of Bbus, a multi-bit output bus, is deasserted. At the next state, it is high, despite the values of the other bits in Bbus. Nowhere in the flowdiagram can be Bbus [4] be used.

When a bit of an output bus is used in an assertion expression, it cannot be used in other expressions.

Assignment expression

An assignment expression implies that an internal, internal output, or output bus is assigned a value. The bus will hold this value until it is reassigned. The assigned bus cannot be used in an assertion expression.

Assignment expressions are associated with states and conditional outputs. The output bus can be assigned the value of the result of a Boolean expression or macro function.

BNF declaration

```
AssignmentExpression ::= WriteBinary <- BooleanExpression
```

Examples

Abus <- A & B	Abus is assigned the value of the result of a logically ANDed with B. Abus, A and B must be one bit of buses. Abus must also be an output bus, while A and B cannot be an output bus.
Bbus [3] <- '1'	The third bit of Bbus is assigned '1'. Bbus must be an output bus.

Condition expression

A condition expression compares two values and yields a 1 or 0 result. If one bit of buses is involved, the two values compared can be derived from a Boolean expression. If multi-bit buses are involved, the two values compared must directly be values of buses or a constant. An output bus cannot be used in a condition expression. In the absence of a relational operator, the expression assumes that the comparison is between a one-bit of a bus or the result of a Boolean expression with the value 1.

Condition expressions are associated with conditions only.

BNF declaration

```
RelationalExpression ::= BooleanExpression
```

Examples

Abus = A & B	Abus is compared to the value of the result of A logically ANDed with B. Abus, A and B must be one bit of buses and cannot be output buses.
Bbus ='H20'	Bbus is compared to hexadecimal 20. Bbus is a multi-bit bus and cannot be an output bus.
Abus	Abus is implicitly compared to 1. Abus must be a one bit bus and cannot be an output bus.

Macro expressions

A macro expression operates on buses, macros, and constants. Macros return values, which can be assigned to a bus. The bus will hold this value until it is reassigned. Buses that are assigned values in macro expressions cannot be used in assertion and assign expressions. Input buses cannot be assigned. Some macros accept arguments. Output buses cannot be used as an argument in a macro expression. A macro can be used as an argument of another macro provided it is used properly.

Macro expressions are associated to states. See the section on Macros for more information.

BNF declaration

```
MacroExpression ::= WriteBus <- MacroFunction
```

Examples

```
C <= MAC1 (A,B)
```

A and B are arguments for MAC1. C is assigned the result of MAC1. C, A and B can be one bit of buses or multi-bits buses. C cannot be an input bus, and A and B cannot be output buses.

```
A <= MAC2 (MAC3(B,C))
```

This is a nested macro expression. The result of MACRO3 becomes an argument to MACRO2. A can be a one bit or multi-bits, but cannot be an input bus.

Flowdiagram rules

There is a finite set of rules which derives the designing of a flowdiagram. The rules are categorized in the following sections according to where the rules are applied to in a flowdiagram.



Flowdiagram objects

- A flowdiagram must consist of at least one sheet
- A flowdiagram must consist of at least one state
- A flowdiagram must consist of one asynchronous reset state
- A state can have any number of incoming arrows
- A state must have one or no outgoing arrows
- A state must make a transition to another state directly or indirectly
- A state with no outgoing arrow is assumed to make a transition into itself
- The expression in a state must be either an assertion or assignment
- A state may not be associated with an expression
- A condition must have at least one incoming arrow
- A condition must have two outgoing arrows
- Outgoing arrows from a condition cannot loop back directly or indirectly into the same condition without first going into a state
- A condition must have a condition expression
- A conditional output must have at least one incoming arrow
- A conditional output must have one outgoing arrow
- Outgoing arrow from a conditional output cannot loop back directly or indirectly into the same conditional output without first going into a state
- A conditional output must have at least one expression associated with it
- The expression associated to a conditional output must be either an assertion and/or assignment expression
- A case must have at least one incoming arrow
- A case must have at least two outgoing arrows
- A case must be associated with an input, internal, or internal output bus
- The number of outgoing arrows from a case must be less than or equal to the value of 2 raised to the power of the width of the associated bus
- Outgoing arrow from a case cannot loop back directly or indirectly into the same case without first going into a state
- A subflowdiagram representation must represent an existing subflowdiagram definition
- A subflowdiagram representation must have one incoming arrow
- The number of outgoing arrows from a subflowdiagram must be equal to the number of output connectors in the corresponding subflowdiagram definition

- An input label must not have an incoming arrow
- An input label must have only one outgoing arrow
- An input label must have an identifier associated with it
- No two input labels can have the same identifier
- An output label must have at least one incoming arrow
- An output label must have at least one incoming arrow
- An output label must have an identifier associated with it
- An output label's identifier must be the same as one other input label's identifier, which is not residing in a subflowdiagram definition
- An input connector must not have an incoming arrow
- An input connector must have only one outgoing arrow
- An input connector must have a unique identifier associated with it
- An input connector must have only reside in a subflowdiagram definition
- An output connector must not have an outgoing arrow
- An output connector must have at least one incoming arrow
- An output connector must have a unique identifier associated with it
- An output connector must only reside in a subflowdiagram definition
- An arrow must originate from a flowdiagram object and end at another



Data types

- An asynchronous reset must be one bit wide
- An asynchronous reset must be associated with a state
- An asynchronous reset must be used only once
- A bus cannot be zero bits wide
- A bus must have a unique identifier associated with it
- A bit of an output bus used in an assertion expression cannot be used in other type of expressions
- A bit of an output bus used as the left operand in an assignment expression cannot be used in other type of expression
- A constant must only be expressed in positive integers
- A constant must only be expressed in binary, octal, decimal, or hexadecimal



Operators

- An assignment operator must have a left operand and a right expression
- A deassert operator must have a right operand
- A logical operator must be either a NOT, AND, OR, NAND, NOR, XOR, or XNOR
- A set of parenthesis can be used to override the logical operator's precedence
- A relational operator must be either an =, !=, >, >=, <, or <=



Expressions

- A Boolean expression must only operate on a bit of an input, internal or internal output bus
- An assertion expression must only operate on a bit of an output bus
- An assertion expression must either assert or deassert a bit of a bus
- An assignment expression must have an output bus as the left operand
- An assignment expression may have either a bit of an input, internal, or internal output bus at the right expression
- A condition expression must have at least one operand
- A condition expression with a bit of input, internal, and internal output buses can use Boolean expressions
- A condition expression with a bit of input, internal, and internal output buses cannot use multi-bit buses in the same expression
- A condition expression with multi-bit input, internal, and internal output buses can use =, !=, >, >=, <, or <= as the relational operators
- A condition expression with a multi-bit input, internal, and internal output bus can only be compared with another multi-bit input, internal, and internal output bus, or a constant

Macro functions

Nimbus provides macro functions are used in a flowdiagram. A listing of all the predefined macro functions is shown below.

ADD	AND	BOR	BXOR	CAT
CMP	CMPL2	DECO	DECR	DMUX
INCR	MUL	MUX	NAND	NMUX
NOR	OR	NOT	REFL	ROL
ROR	SHL0	SHL1	SHLIN	SHR0
SHR1	SHRIN	SUB	XNOR	XOR

Interpreting the macro function

Each of the predefined macro functions are described in this chapter with examples of the usage. The conventions used in presenting the macro functions are generalized in this section.

The following rules apply to all macro functions descriptions.

- All bus widths are between 1 and 32.
- Constant arguments must be positive integers
- When an argument is a constant
argument width = minimum possible width of the constant
otherwise argument width = 1
- Two-valued logic is implied (i.e. 0 or 1)
- All macro-functions return values
- All buses are assumed to be of the type Abus [MSB : LSB]

All sections for each macro functions uses the following symbols to describe the syntax:

- { } 0 or more. Enclosed operand can occur zero or more times
- [] 0 or 1. Enclosed operand can occur zero or one times

The Description section for each macro function uses the following symbols:

	Operator	Description
+	Addition	Sums two operands
-	Subtract	Subtracts two operands
÷	Division	Divides two operands
x	Multiply	Multiplies two operands
[]	Bus slice	Enclosed operands can only have natural integer values (greater than or equal to 0). Bus slice ranges can be specified as Abus [x:y] where $x \geq y$. A single bit bus slice can specified as Abus [x]
=	Assignment	Assigns RHS operand to LHS operand
=	Logical Equality	Compares two values for equality
!=	Logical Inequality	Compares two values for inequality
<	Less than	Determines relative value
>	Greater than	Determines relative value

ADD

Purpose

Adds two values with an optional carry bit. An overflow bit is assigned to the MSB of the resultant if the width of the resultant bus is wider than the operand bus.

Syntax

`Xbus <- ADD (Abus, Bbus [,Carry])`

Description

If Carry is specified

$$Xbus = Abus + Bbus + Carry$$

Otherwise

$$Xbus = Abus + Bbus$$

Constant width conversion

Argument(s)	Condition	Width Conversion
Abus, Bbus	unconditional	Abus = max_width (all arguments) Bbus = Abus
Abus	Abus < Bbus	Abus = Bbus
Bbus	Bbus < Abus	Bbus = Abus

Bus width criteria

$$Xbus.width = Abus.width = Xbus.width - 1$$

$$Bbus.width = Abus.width$$

$$Carry.width = 1$$

$$1 = Xbus.width = 32$$

VHDL translation

Condition	Translation
<code>Xbus.width = Abus.width</code>	<code>Xbus <= ADDNC (Abus, Bbus [,Carry]);</code>
<code>Xbus.width = Abus.width + 1</code>	<code>Xbus <= ADD (Abus, Bbus [,Carry]);</code>

Verilog translation

`Xbus = ADDx (Abus, Bbus [,Carry]);`

AND

Purpose

Performs bitwise AND on all inputs and stores the result in Xbus

Syntax

```
Xbus <- AND (Abus,Bbus [,Nbus])
```

Description

Xbus = Abus & Bbus {and Nbus}

Constant width conversion

Argument(s)	Condition	Width Conversion
All	unconditional	Abus = max_width (all arguments) Bbus = Abus Nbus = Abus
Any	Any < non-constant	Any = non-constant

Bus width criteria

1 = Abus.width = 32

Bbus.width = Abus.width

Nbus.width = Abus.width

Xbus.width = Abus.width

VHDL translation

```
Xbus <= FAND (Abus, Bbus [,Nbus]);
```

Verilog translation

```
Xbus = ANDx (Abus, Bbus [,Nbus]);
```



BAND

Purpose

Performs bitwise AND on all the bits in the input bus

Syntax

```
Xbus <- BAND (Abus)
```

Description

```
Xbus = Abus [0] & Abus [1] & ... & Abus [MSB]
```

Bus width criteria

```
2 = Abus.width =32
```

```
Xbus.width = 1
```

VHDL translation

```
Xbus <= BAND (Abus);
```

Verilog translation

```
Xbus =BANDx (Abus);
```



BOR

Purpose

Performs bitwise OR on all the bits in the input bus

Syntax

```
Xbus <- BOR(Abus)
```

Description

```
Xbus = Abus [0] | Abus [1] | ... | Abus [MSB]
```

Bus width criteria

```
2 = Abus.width = 32
```

```
Xbus.width = 1
```

VHDL translation

```
Xbus <= BOR (Abus);
```

Verilog translation

```
Xbus = BORx (Abus);
```



BXOR

Purpose

Performs bitwise XOR on all the bits in the input bus.

Syntax

```
Xbus <- BXOR(Abus)
```

Description

```
Xbus = Abus [0] xor Abus [1] xor ...xor Abus [MSB]
```

Bus width criteria

```
2 = Abus.width =32
```

```
Xbus.width = 1
```

VHDL translation

```
Xbus <= BXOR (Abus);
```

Verilog translation

```
Xbus = BXORx (Abus);
```



CAT

Purpose

Concatenates all input values in order of appearance and stores the result in Xbus

Syntax

```
Xbus <- CAT (Abus,Bbus)
```

Description

Xbus [MSB to Bbus.width + 1] = Abus

Xbus [Bbus.width to 0] = Bbus

Bus width criteria

1 = Abus.width = 31

1 = Bbus.width = 32 - Abus.width

Abus.width + Bbus.width = Xbus.width = Abus.width + Bbus.width

VHDL translation

```
Xbus <= CAT (Abus, Bbus);
```

Verilog translation

```
Xbus = CATx (Abus, Bbus);
```



CMP

Purpose

Compares the values of two buses or signals and stores the comparison result in Xbus

Syntax

```
Xbus <- CMP(Abus,Bbus)
```

Description

```
If Abus = Bbus
    Xbus = 0
else if Abus < Bbus
    Xbus = 1
else
    Xbus = 2
```

Constant width conversion

Argument(s)	Condition	Width Conversion
All	unconditional	Bbus = Abus
Any	Any < non-constant	Any = non-constant

Bus width criteria

```
1 = Abus;width = 32
Bbus.width = Abus.width
Xbus.width = 2
```

VHDL translation

```
Xbus <= CMP (Abus, Bbus);
```

Verilog translation

```
Xbus = CMPx (Abus, Bbus);
```



CMPL2

Purpose

Performs a two's complement on the input. Discards the overflow bit if the width of the result is equal to the width of the input; otherwise stores the overflow bit in the MSB of the result.

Syntax

```
Xbus <- CMPL2 (Abus)
```

Description

```
Xbus = Abus + 1
```

Bus width criteria

```
Xbus.width - 1 = Abus.width = Xbus.width  
1 = Xbus.width = 32
```

VHDL translation

```
Xbus <= CMPL2 (Abus);
```

Verilog translation

```
Xbus = CMPL2x (Abus);
```



DECO

Purpose

Decodes the value on a bus

Syntax

```
Xbus <- DECO (Abus)
```

Description

Xbus = 0

Xbus [Abus] = 1

Bus width criteria

1 = Abus.width = 5

Xbus.width = 2^Abus.width

VHDL translation

```
Xbus <= DECO (Abus);
```

Verilog translation

```
Xbus = DECOx (Abus);
```



DECR

Purpose

Decrements the value on a bus. Discards the MSB of the result if the width of the result is more than that of the input

Syntax

```
Xbus <- DECR (Abus)
```

Description

```
Xbus = Abus - 1
```

Bus width criteria

```
1 = Abus.width = Xbus.width - 1
```

```
1 = Xbus.width = 32
```

VHDL translation

```
Xbus <= DECR (Abus);
```

Verilog translation

```
Xbus = DECRx (Abus);
```



DMUX

Purpose

Demultiplexes input data based on control signals

Syntax

```
Xbus <- DMUX (Abus,Bbus)
```

Description

Xbus = 0

Xbus [Bbus] = Abus

Bus width criteria

Abus.width = 1

1 = Bbus.width = 32

$2^{Bbus.width} = Xbus.width = 32$

VHDL translation

```
Xbus <= DMUX (Abus, Bbus);
```

Verilog translation

```
Xbus = DMUXx (Abus, Bbus);
```



INCR

Purpose

Increments the value on a bus. Discards the MSB of the result if the width of result exceeds that of the input

Syntax

```
Xbus <- INCR (Abus)
```

Description

```
Xbus = Abus + 1
```

Bus width criteria

```
Xbus.width - 1 = Abus.width = Xbus.width  
1 = Xbus.width = 32
```

VHDL translation

```
Xbus <= INCR (Abus);
```

Verilog translation

```
Xbus = INCRx (Abus);
```



MUL

Purpose

Multiple two values

Syntax

Xbus <- MUL (Abus,Bbus)

Description

Xbus = Abus x Bbus

Constant width conversion

Argument(s)	Condition	Width Conversion
All	unconditional	Abus = max_width (all arguments) Bbus = Abus
Any	Any < non-constant	Any = non-constant

Bus width criteria

1 = Abus.width = 16

Bbus.width = Abus.width

Xbus.width = 2 x Abus.width

VHDL translation

Xbus <= MUL (Abus, Bbus);

Verilog translation

Xbus = MULx (Abus, Bbus);



MUX

Purpose

Multiplexes input data based on control signals

Syntax

Xbus <- MUX (Abus,Bbus)

Description

Xbus = Abus [Bbus]

Bus width criteria

1 = Abus.width = $2^{Bbus.width}$

1 = Bbus.width = 5

Xbu.widths = 2 x Abus.width

VHDL translation

Xbus <= MUX (Abus, Bbus);

Verilog translation

Xbus = MUXx (Abus, Bbus);

NAND

Purpose

Performs bitwise NAND on all the values

Syntax

```
Xbus <- NAND (Abus,Bbus [,Nbus])
```

Description

Xbus = Abus and Bbus {and Nbus}

Constant width conversion

Argument(s)	Condition	Width Conversion
All	unconditional	Abus = max_width (all arguments) Bbus = Abus Nbus = Abus
Any	Any < non-constant	Any = non-constant

Bus width criteria

1 = Abus.width = 32

Bbus.width = Nbus.width = Xbus.width = Abus.width

VHDL translation

```
Xbus <= FAND (Abus, Bbus {,Nbus});
```

Verilog translation

```
Xbus = ANDx (Abus, Bbus {,Nbus});
```

NMUX

Purpose

Selects one of the two values depending on the select value

Syntax

```
Xbus <- NMUX (Abus,Bbus, Select)
```

Description

If Select = 0

Xbus = Abus

else

Xbus = Bbus

Constant width conversion

Argument(s)	Condition	Width Conversion
All	unconditional	Abus = max_width (all arguments) Bbus = Abus Nbus = Abus
Any	Any < non-constant	Any = non-constant

Bus width criteria

1 = Abus.width = 32

Xbus.width = Bbus.width = Abus.width

Select.width = 1

VHDL translation

```
Xbus <= NMUX (Abus, Bbus, Select);
```

Verilog translation

```
Xbus = NMUXx (Abus, Bbus,Select);
```

NOR

Purpose

Performs bitwise NOR on all the values

Syntax

```
Xbus <- NOR(Abus,Bbus {,Nbus})
```

Description

Xbus = Abus and Bbus {and Nbus}

Constant width conversion

Argument(s)	Condition	Width Conversion
All	unconditional	Abus = max_width (all arguments) Bbus = Abus Nbus = Abus
Any	Any < non-constant	Any = non-constant

Bus width criteria

1 = Abus.width = 32

Bbus.width = Nbus.width = Xbus.width = Abus.width

VHDL translation

```
Xbus <= FNOR (Abus, Bbus {,Nbus});
```

Verilog translation

```
Xbus = NORx (Abus, Bbus {,Nbus});
```



NOT

Purpose

Inverts all the bits of the bus

Syntax

Xbus <- NOT (Abus)

Description

Xbus = !Abus

Bus width criteria

1 = Abus.width = 32

Xbus.width = Abus.width

VHDL translation

Xbus <= FNOT (Abus);

Verilog translation

Xbus = NOTx (Abus);



OR

Purpose

Performs bitwise OR on all the values

Syntax

Xbus <- OR (Abus,Bbus {,Nbus})

Description

Xbus = Abus or Bbus {or Nbus}

Constant width conversion

Argument(s)	Condition	Width Conversion
All	unconditional	Abus = max_width (all arguments) Bbus = Abus Nbus = Abus
Any	Any < non-constant	Any = non-constant

Bus width criteria

1 = Abus.width = 32

Bbus.width = Nbus.width = Xbus.width = Abus.width

VHDL translation

Xbus <= FFOR (Abus, Bbus {,Nbus});

Verilog translation

Xbus = ORx (Abus, Bbus {,Nbus});



REFL

Purpose

Transposes the bits of the input

Syntax

```
Xbus <- REFL (Abus)
```

Description

```
for l = MSB to LSB loop
    Xbus [MSB -i] = Abus [i]
end loop
```

Bus width criteria

```
1 = Abus.width = 32
Xbus.width = Abus.width
```

VHDL translation

```
Xbus <= REFL (Abus);
```

Verilog translation

```
Xbus = REFLx (Abus);
```



ROL

Purpose

Rotates left all bits to the next higher significant bit, with the MSB rotating to the LSB

Syntax

```
Xbus <- ROL (Abus)
```

Description

```
Xbus = Abus
```

```
Xbus [LSB] = Xbus[MSB]
```

```
for j = MSB to LSB + 1 loop
```

```
    Xbus [j] = Xbus [j - 1]
```

```
end loop
```

Bus width criteria

```
1 = Abus.width = 32
```

```
Xbus.width = Abus.width
```

VHDL translation

```
Xbus <= ROL (Abus);
```

Verilog translation

```
Xbus = ROLx (Abus);
```



ROR

Purpose

Rotates right all the bits of the input. Shift all bits to the next lower significant bit respectively, with the LSB rotating to the MSB

Syntax

```
Xbus <- ROR(Abus)
```

Description

```
Xbus = Abus
```

```
Xbus [MSB] = Xbus [LSB]
```

```
for j = MSB to LSB + 1 loop
```

```
    Xbus [j - 1] = Xbus [j]
```

```
end loop
```

Bus width criteria

```
1 = Abus.width = 32
```

```
Xbus.width = Abus.width
```

VHDL translation

```
Xbus <= ROR (Abus);
```

Verilog translation

```
Xbus = RORx (Abus);
```



SHLO

Purpose

Shifts left all the bits to the next higher significant bit respectively, with the LSB assigned to 0.

Syntax

```
Xbus <- SHLO(Abus)
```

Description

```
Xbus = Abus
```

```
Xbus [LSB] = 0
```

```
for j = MSB to LSB + 1 loop
```

```
    Xbus [j] = Xbus [j - 1]
```

```
end loop
```

Bus width criteria

```
1 = Abus.width = 32
```

```
Xbus.width = Abus.width
```

VHDL translation

```
Xbus <= SHLO (Abus);
```

Verilog translation

```
Xbus = SHLOx (Abus);
```



SHL1

Purpose

Shifts left all the bits to the next higher significant bit respectively, with the LSB assigned to 1.

Syntax

```
Xbus <- SHL1(Abus)
```

Description

```
Xbus = Abus
```

```
Xbus [LSB] = 1
```

```
for j = MSB to LSB + 1 loop
```

```
    Xbus [j] = Xbus [j - 1]
```

```
end loop
```

Bus width criteria

```
1 = Abus.width = 32
```

```
Xbus.width = Abus.width
```

VHDL translation

```
Xbus <= SHL1 (Abus);
```

Verilog translation

```
Xbus = SHL1x (Abus);
```



SHLIN

Purpose

Shifts left all the bits, with the LSB assigned the value of the second input.

Syntax

```
Xbus <- SHLIN (Abus,Bbus)
```

Description

```
Xbus = Abus
```

```
Xbus [LSB] = Bbus
```

```
for j = MSB to LSB + 1 loop
```

```
    Xbus [j] = Xbus [j - 1]
```

```
end loop
```

Bus width criteria

```
1 = Abus.width = 32
```

```
Bbus.width = 1
```

```
Xbus.width = Abus.width
```

VHDL translation

```
Xbus <= SHLIN (Abus,Bbus);
```

Verilog translation

```
Xbus = SHLINx (Abus,Bbus);
```



SHR0

Purpose

Shifts right all the bits to the next lower significant bit respectively, with the MSB assigned to 0.

Syntax

```
Xbus <- SHR0(Abus)
```

Description

```
Xbus = Abus
```

```
Xbus [MSB] = 0
```

```
for j = MSB to LSB + 1 loop
```

```
    Xbus [j - 1] = Xbus [j]
```

```
end loop
```

Bus width criteria

```
1 = Abus.width = 32
```

```
Xbus.width = Abus.width
```

VHDL translation

```
Xbus <= SHR0 (Abus);
```

Verilog translation

```
Xbus = SHR0x (Abus);
```



SHR1

Purpose

Shifts right all the bits to the next lower significant bit respectively, with the MSB assigned to 1.

Syntax

```
Xbus <- SHR1(Abus)
```

Description

```
Xbus = Abus
```

```
Xbus [MSB] = 0
```

```
for j = MSB to LSB + 1 loop
```

```
    Xbus [j - 1] = Xbus [j]
```

```
end loop
```

Bus width criteria

```
1 = Abus.width = 32
```

```
Xbus.width = Abus.width
```

VHDL translation

```
Xbus <= SHR1 (Abus);
```

Verilog translation

```
Xbus = SHR1x (Abus);
```



SHRIN

Purpose

Shifts right all the bits, with the MSB assigned the value of the second input.

Syntax

```
Xbus <- SHRIN(Abus,Bbus)
```

Description

```
Xbus = Abus
for i = 1 to Count loop
    Xbus [MSB] = Bbus
    for j = MSB to LSB + 1 loop
        Xbus [j - 1] = Xbus [j]
    end loop
end loop
```

Bus width criteria

```
1 = Abus.width = 32
Bbus.width = 1
Xbus.width = Abus.width
```

VHDL translation

```
Xbus <= SHRIN (Abus,Bbus);
```

Verilog translation

```
Xbus = SHRINx (Abus,Bbus);
```

SUB

Purpose

Subtract one value from another with a borrow (optional). A borrow occurs in the MSB if needed, but it is not reflected in the result

Syntax

`Xbus <- SUB (Abus, Bbus [,Borrow])`

Description

if Borrow is specified

`Xbus = Abus - Bbus - Borrow`

else

`Xbus = Abus - Bbus`

Constant width conversion

Argument(s)	Condition	Width Conversion
All	unconditional	Bbus = Abus
Any	Any < non-constant	Any = non-constant

Bus width criteria

`Xbus.width - 1 = Abus.width = Xbus.width`

`Bbus.width = Abus.width`

`Borrow.width = 1`

`1 = Xbus.width = 32`

VHDL translation

Condition	Translation
<code>Xbus = Bbus</code>	<code>Xbus <= SUBNC (Abus, Bbus [,Borrow]);</code>
<code>Xbus = Abus + 1</code>	<code>Xbus <= SUB (Abus, Bbus, [,Borrow]);</code>

Verilog translation

`Xbus = SUBx (Abus, Bbus [,Borrow]);`

XNOR

Purpose

Performs bitwise XNOR on all values

Syntax

Xbus <- XNOR (Abus, Bbus [,Nbus])

Description

Xbus = !(Abus xor Bbus {xor Nbus})

Constant width conversion

Argument(s)	Condition	Width Conversion
All	unconditional	Abus = max_width (all arguments) Bbus = Abus Nbus = Abus
Any	Any < non-constant	Any = non-constant

Bus width criteria

1 = Abus.width = 32

Xbus.width = Nbus.width = Bbus.width = Abus.width

VHDL translation

Xbus <= FXNOR (Abus, Bbus{,Nbus});

Verilog translation

Xbus = XNORx (Abus, Bbus {,Nbus});

XOR

Purpose

Performs bitwise XOR on all the values

Syntax

```
Xbus <- XOR (Abus, Bbus [,Nbus])
```

Description

```
Xbus = Abus xor Bbus {xor Nbus}
```

Constant width conversion

Argument(s)	Condition	Width Conversion
All	unconditional	Abus = max_width (all arguments) Bbus = Abus Nbus = Abus
Any	Any < non-constant	Any = non-constant

Bus width criteria

```
1 = Abus.width = 32
```

```
Xbus.width = Nbus.width = Bbus.width = Abus.width
```

VHDL translation

```
Xbus <= FXOR (Abus, Bbus{,Nbus});
```

Verilog translation

```
Xbus = XORx (Abus, Bbus {,Nbus});
```

Nimbus reserved words

There are many specific word strings that are considered “reserved” by Nimbus. Generally, a specific word is reserved from use in a user’s flowdiagram design because it coincides with reserved words defined by the VHDL or Verilog output.

VHDL IEEE Standard 1076-1987 reserve words

Source: IEEE Standard VHDL Language Reference Manual

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



Verilog reserve words

Source: Verilog HDL Language Reference Manual (Open Verilog International)

always	assign	begin	but	bufif0
bufif1	case	casez	cmos	deassign
default	defparam	disable	edge	endcase
endmodule	endfunction	endprimitive	endspecify	endtable
endtask	event	force	forever	fork
highz0	highz1	initial	input	integer
join	large	Macromodule	negedge	nmos
notify0	notify1	output	parameter	pmos
posedge	primitive	pull0	pull1	pullup
pulldown	rcmos	reg	release	repeat
rnmos	rpmos	rtran	rtranif0	rtranif1
scalared	small	specify	specparam	strength
strong0	strong1	supply0	supply1	table
task	time	tran	transif0	transif1
tri	tri0	tri1	triand	trior
triereg	vectored	wand	weak0	weak1
wire	wor	xnor		



Nimbus reserved words

res	clk	clk1	clk2	test_bench
next_state	dp_assertion	cp_assertion	data_path	current_state
transition	control_path			

File formats

Several file types (below) are used by Nimbus, or shared between Nimbus and third-party HDL synthesis and simulation products.

Data Type	File Type	Description
Binary	*.nim	File created when a design is saved.
Binary	*.wav	Waveform file created from a simulation run.
ASCII	*.bat	Batch file created from a simulation session.
ASCII	*.err	Error file optionally generated after compilation and checking.
ASCII	*.his	History file optionally generated from a Nimbus simulation.
ASCII	*.vhd	VHDL design file created as a result of HDL translation.
ASCII	*.v	Verilog design file created as a result of HDL translation.
ASCII	*.test.vhdl	VHDL testbench file created from batch file.
ASCII	*.test.v	Verilog testbench file created from batch file.
ASCII	*.prn	PostScript® file created when printing.
Binary	*.mem	Memory file.

Batch file

A batch file stores user-generated test vectors and timing attributes for use with the Nimbus simulator. It contains the sequence of commands to the simulator for setting breakpoints, setting the signal values, and executing a design simulation. Hexadecimal, octal, binary, and decimal values are specified using the prefixes "h," "o," "b," and "d." Values with no prefix are assumed to be in decimal format. A batch file is created as a result of an interactive simulation session, by clicking the batch save button in the simulator control panel after the simulation and specifying an output file name.

```
break at Wait_ME
current state Wait_ME
set Rxddata_1 1
set DCD_2 1
step
set ME_1 1
set ACC_1 b10100000
step
step
set MRWE_1 1
step
set MRWE_1 0
step
step
step
set ACC_1 b00110011
set MRWE_1 1
set RS_1 1
set CTS_2 1
step
set MRWE_1 0
step
step
step
```

Once a batch file has been saved, it can be reloaded and used as the basis for a new simulation run. You can modify a batch file by hand in any text editor to vary the set-up or execution of the simulation run. A sample Batch file is shown above.

The batch file consists of a list of commands to the simulation engine which are interpreted sequentially. The lists of valid commands are described below.

- **break at <state name>**
Sets a breakpoint at the specified state name in a batch file.
- **clear at <state name>**
Clears the breakpoint from the specified state.
- **clear all**
Clears all breakpoints that have been entered.
- **current state <state name>**
Indicates the start of the simulation. This command should come after any breakpoint commands. Can be specified using the state name of any state

with an asynchronous reset. If no reset event is available, any enable event can be used.

- `set <bus name> <value>`
Sets the value of a bus signal to a specified value. You may only set values for input and bi-directional buses.
- `step [n]`
Steps through n simulation cycles.
- `step`
Steps one cycle through the simulation.
- `continue`
Specifies that the simulation is to be restarted.
- `stop`
Stops the simulation on the current cycle at the current simulation time.

To bunch multiple batch runs into a single file, follow your break at command with a current state `<state name>` command.

Compiler/Checker error file

```

The following errors were found in Food_Vending_Machine
-----
SYS1010
1.  There are error(s) in the Flowdiagram
*****
Flowdiagram Sheets
*****
SHEET: pretzel
Error free
SHEET: chips
Error free
SHEET: vend_machine
State (initime1)
FDC30
1.  Semantic error in the expression
*****
Bus Table
*****
Bus: exit
BTC4000
Bus name is a reserved HDL name
*****
Local Macro Function Table
*****
Error free

```

You can save all error and warning codes and messages in an output file called <design>.err. The error output file lists all errors and descriptions in three categories: Flowdiagram Sheets, Bus, and User-defined Macros. The flowdiagram sheet errors are listed under the flowdiagram sheet in which they are found. A sample error output file is shown above.



VHDL output file

Output VHDL code is partitioned into control process (the state machine model), synchronous data path process, and asynchronous data path process. This basic style varies depending on the elements of the design model depicted in the flowdiagram.

If a flowdiagram has only one state, the resultant VHDL model will have one or two processes. Having a single state implies that the design is a data path design, which does not require a state machine controller. The synchronous and asynchronous data path behaviors are included in the individual VHDL processes, and depend on the bus element information for each bus defined in the design.

If buses are registers, operations involving them will be in the clocked data path process in the VHDL. If the buses are defined as wire or latch, the resultant operations on these buses will be contained in the asynchronous data path process. Buses defined as asynchronous register elements will have aspects of their description located in both the clocked and the asynchronous data path processes.

If a flowdiagram is comprised of a sequence of states and conditions without specified actions, the resultant VHDL code will be a state machine control process only.

If a flowdiagram represents a purely synchronous design, the resultant VHDL code will contain a two-process model, with control and synchronous data path elements.

If a flowdiagram represents a partially asynchronous design, the resultant VHDL model will be a two-process model comprised of a control process and an asynchronous data path process.

The HDL translation style affects the code style of the model written in VHDL. Coding style preferences are discussed in Section 12.2.4 of the Users Manual.



Verilog® output file

The Verilog output code style consists of a basic three-process model, delineating control path, clocked data path and asynchronous data path elements into different processes.

The organization of the Verilog files is similar to that of the VHDL files generated by Nimbus, as described in Section 7.7. of the Users Manual.

Error messages

This lists all error codes in alphanumeric order and all error messages as they appear in the error message dialog boxes.

Code	Message	Possible Solution
BTC0100	Illegal bus type.	
BTC0200	Illegal bus width.	
BTC0300	Illegal bus default value.	
BTC0400	Illegal bus value.	
BTC0500	Illegal bus enumeration type.	
BTC1000	Duplicate state name.	
BTC2000	Illegal bus name.	
BTC3000	Bus name is a reserved Nimbus name.	
BTC4000	Bus name is a reserved HDL name.	
BTC5000	Duplicate bus name.	
BTE0700	At least one bus in this thread type must be of a non internal mode; otherwise an empty list of ports will result during HDL generation.	
BTE6000U	No bus name is specified.	Enter the bus name.
BTE6001U	Bus name defined is not a unique identifier.	Use a different bus name.
BTE6002U	Illegal bus name.	Use alphanumeric characters for the bus name, starting with a letter.
BTE6003U	Illegal bus type.	Use input, internal, internal output, or output only.
BTE6004U	Illegal bus width.	Enter a bus width between 1 and 32 inclusive.

Code	Message	Possible Solution
BTE6005U	Illegal bus value.	Enter a value that conforms to the bus width, logic type and bus data format.
BTE6006U	Illegal bus default value.	Enter a default value that conforms to the bus width, logic type and bus data format.
BTE6007U	This bus is attached to a flowdiagram case/state object.	Delete the flowdiagram case or event to delete this bus.
BTE6008U	There is no bus selected.	Select/Highlight a bus in the Bus List Box.
BTE6009U	This bus is not an input or a bidirectional bus.	Select input or bidirectional buses only.
BTE600AU	Illegal enumerated bus type.	Select a type from the enumerated type list.
BTE600BU	Illegal wire bus default value.	Enter a binary/Z default value that conforms to the bus width.
BTE600CU	Illegal bidirectional bus default value.	Enter a Z default value that conforms to the bus width.
BTE600DU	This bus is defined as a system clock.	Remove reference from system clock table first to delete/modify this bus.
BTE600EU	Cannot delete a bus used in expression(s).	Remove its reference in expressions to delete this bus.
BTE600FU	This bus is used in a CASE object that uses don't-care transitions.	Only MVL9 buses can be used with don't-care transitions.
BTE6010U	No enumeration type name is specified.	Enter the enumeration type name.
BTE6011U	Enumeration type name is not a unique identifier.	Use a different enumeration type name.
BTE6012U	Illegal enumeration type name.	Use alphanumeric characters for the enumeration type name, starting with a letter.
BTE6013U	No enumerated type element list specified.	Enter the enumeration type element list.
BTE6014U	Illegal enumeration expression.	Enter a value that conforms to the enumeration expression syntax.
BTE6015U	This enumeration type variable is referenced to a flowdiagram object.	Delete the flowdiagram object to delete/modify this enumeration type.
BTE6016U	There is no enumeration type name selected.	Select/Highlight a name in the Enumeration Table List Box.
BTE6017U	One of the enumeration element names is not a unique identifier.	Use a different enumeration element name.
BTE6018U	Duplicate enumeration element names used.	Use unique enumeration element names.
CLE00U	No clock selected for this operation.	Select a clock from the Clock Table.

Code	Message	Possible Solution
CLE01U	No clock/clock-expression specified.	Select a clock from the list or specify a clock expression.
CLE02U	No candidate clock(s) available from the bus table.	Define input, Binary, 1 bit bus (in Bus Table) to use as system clock.
CLE03U	This system clock is referred by other system clocks.	Remove reference(s) to this system clock before you delete/modify.
CLE04U	No reset/reset-expression specified.	Select a reset from the list or specify a reset expression.
CLE05U	Clock cycle range selected is out of range.	Select a range that encloses cycle periods of clocks already specified.
CTEF000U	No constant name is specified.	Enter the constant name.
CTEF001U	Constant name is not an unique identifier.	Use a different constant name.
CTEF002U	Illegal constant name.	Use alphanumeric characters for the constant name, starting with an alphabet.
CTEF003U	No constant value is specified.	Enter the constant value.
CTEF004U	Illegal constant value.	Enter a value (width less than 32) that conforms to the constant value definition.
CTEF005U	This constant is referenced to a flowdiagram object.	Delete the flowdiagram object to delete the constant.
CTEF006U	There is no constant selected.	Select/Highlight a constant in the Constant Table List Box.
FDC0001	Duplicate Object name.	
FDC0002	Illegal usage of Object name.	
FDC0003	Object name is an HDL reserved word.	
FDC0004	Object name is a Nimbus reserved word.	
FDC0005	This Object needs a name.	
FDC0006	The bus associated to this case does not exist.	
FDC0007	Illegal association of bus and case.	
FDC0008	The corresponding input label does not exist.	
FDC0009	The subflowdiagram sheet does not exist.	
FDC000A	The sub-flowdiagram sheet associated is not a subflowdiagram.	
FDC000B	The state name is the same as a bus name.	
FDC000C	This sub-flowdiagram cannot be shared with other flowdiagrams.	
FDC000D	Flowdiagram specification will result in creation of HDL code with an empty signal port list.	
FDC000E	Flowdiagram specification will result in creation of HDL code with an empty	

Code	Message	Possible Solution
	sensitivity list.	
FDC000F	This state name is the same as a thread name.	
FDC0010	An unknown symbol is detected in the expression.	
FDC0020	Syntax error in the expression.	
FDC0030	Semantic error in the expression.	
FDC0040	Asynchronous feedback.	
FDC0050	Clock signal illegal mode; use readable signals that are not asynchronously driven.	
FDC0060	Clock signal user-defined enumeration type.	
FDC0070	Clock signal illegal width; Use single bit signals.	
FDC0080	Clock signal closed loop; state machines are driving each other's clock signals.	
FDC0090	Cannot mix event and clock using the same signal.	
FDC00A0	There is a conflict in the clock that triggers the flow.	
FDC00B0	This conditional output needs an expression.	
FDC00C0	This condition needs an expression.	
FDC00D0	This case needs an associated bus.	
FDC00E0	There is a conflict in the asynchronous signal in the same flow.	
FDC00F0	There is a conflict in the asynchronous signal in another flow.	
FDC0200	This object needs a transition.	
FDC0300	This condition needs a true(1) transition.	
FDC0400	This condition needs a false(0) transition.	
FDC0500	This case needs a default transition to account for remaining binary or MVL values.	
FDC0600	This sub-flowdiagram needs matching transitions.	
FDC0700	This object needs to have an asynchronous input in the flow.	
FDC0800	This output label has no corresponding input label/state.	
FDC0900	This case does not need a default transition.	
FDC0A00	This Object may loop into itself.	
FDC0B00	This condition may loop into itself.	
FDC0C00	This case may loop into itself.	
FDC0D00	This conditional output may loop into itself.	
FDC0E00	This sub-flowdiagram may loop into itself.	

Code	Message	Possible Solution
FDC0F00	The null state machine is not valid.	
FDC1000	There is a multiple memory object being accessed by multiple threads (reading/writing to the same summary).	
FDC2000	There is a multiple drive bus error in the object (multiple threads sharing a bus).	
FDC2000000	This sub-flowdiagram output connector is not connected.	
FDC5000	There is an inconsistent usage of an Assertion.	
FDC600000	There are errors in the Event expression(s).	
FDC80000	This thread contains no objects.	
FDC90000	This thread contains isolated objects.	
FDCA0000	This object can never be reached.	
FDCB0000	This thread contains clocks/resets without valid expressions.	
FDC00000	This object contains an event without a valid expression.	
FDCD0000	Test harness type threads cannot use external buses.	
FDCE0000	The design must contain at least one model type thread.	
FDCF0000	Clock/reset/event expressions cannot contain input register buses.	
FDE2000U	This sub-flowdiagram is in use.	Delete references to this sub-flowdiagram first.
FDE2001U	Unique objects cannot be copied to the buffer.	Refer to the Users Manual.
FDE2002U	Some labels and connectors cannot be cleared from the sheet.	Refer to the Users Manual.
FDE2003U	Some objects cannot be deleted.	Delete references to these objects first.
FDE2004U	Illegal transition source.	Do not select an output label as a transition source.
FDE2005U	Illegal transition source.	Do not select an output connector as a transition source.
FDE2006U	Illegal transition target.	Do not select an input label as a transition target.
FDE2007U	Illegal transition target.	Do not select an input connector as a transition target.
FDE2008U	Object cannot transit into itself.	Select a different transition target.
FDE2009U	Overlapping of Objects and Transition violation.	Select a different location to paste.