GENUS: A GENERIC COMPONENT LIBRARY FOR HIGH LEVEL SYNTHESIS

BY

NIKIL D. DUTT

Technical Report 88-22

Information and Computer Science
University of California at Irvine
Irvine, CA 92717

Abstract

This report describes the organization of GENUS, a generic component library for high level synthesis. Generic components and instances in GENUS are organized into hierarchical classes, with the component type stored at the root of the hierarchy, and particular instances stored at the leaves. This permits a consistent representation of generic components which may be used by a variety of synthesis and analysis tools. The appendix contains the description of the GENUS generator library.

TABLE OF CONTENTS

CHAPTER

1. Introduction 1
   1.1. Advantages of Using Generic Components 2

2. Previous Work 3

3. GENUS System Overview 5
   3.1. Hierarchy 5
   3.2. Using GENUS 7
   3.3. Technology Library Restrictor 9

4. GENUS Components 10
   4.1. Port Naming Convention 10
   4.2. Port Semantics 10
   4.3. Component Control 11
   4.4. Combinatorial Components 11
   4.5. Sequential Components 13
   4.6. 14

5. Accessing GENUS 15
   5.1. Accessing Components 15

6. Format of Generator Descriptions 17
   6.1. 19
   6.2. Class 19
   6.3. Parameters 20
   6.4. Styles 20
   6.5. Ports 20
   6.6. Operations 21
   6.7. Op_classes 21
   6.8. Macro Expansion and Port Naming 21
   6.9. Estimation Functions 23

7. Acknowledgements 23

8. References 23

9. APPENDIX A: GENERIC COMPILER CALLS 25

10. APPENDIX B: GENERIC COMPONENT LIBRARY DEFINITIONS 28

LIST OF FIGURES

Figure 1. Hierarchy in GENUS 6

Figure 2. Basic GENUS Data Structures 8

Figure 3. Combinatorial Components 12

Figure 4. Sequential Components 13

Figure 5. Interface and Miscellaneous Components 14

Figure 6. Generic Component Access: General Form 15

Figure 7. Sample ALU Instance Call 15

Figure 8. List of Compiler Global Symbols 16

Figure 9. List of Compiler Global Symbols (Cont'd) 17

Figure 10. Sample GENUS Generator Description 18

Figure 11. Macro-Expand Feature 22

1. Introduction

This document describes **GENUS** [1], a library of generic microarchitectural components used by behavioral synthesis systems such as EXTEND [DuGa88] and VSS [LiGa88], under development at UC Irvine's CADLAB. The task of behavioral synthesis involves mapping the behavior of a design (often described in a language) to a structure composed of components which execute this behavior. Generic components from this library form the building blocks for this structure generated by a behavioral synthesizer. For instance, operations in the behavioral description may be mapped into instantiations of generic library elements which perform the operation, and variables may be mapped to storage elements.

This report first describes the problem and briefly discusses previous work on component modeling by representative behavioral synthesis systems. Next, the organization of GENUS is described. Conventions and semantics associated with certain entities within GENUS are explained, and access functions for the library are described. Finally, the format of the GENUS generator input is presented. This input may be used as input to a generic component library generator which loads the library into the internal data structures for use by the synthesis systems. The Appendices contain details of compiler calls and generator library definitions.

During synthesis, a component is instantiated by specifying its parameters which define its ports and attributes. Typical parameters include the component's *style* (eg. slow or fast), *functionality* (eg. add and increment for an arithmetic unit), *input-output characteristics* (ports on the component), *size* (eg. number of words for a memory), *bit-width* and *representation* (eg. two's complement). Hence each generator is a template for a generic

[1]**genus**: a class of objects divided into several subordinate species (From Webster's New Collegiate Dictionary).

microarchitectural component; depending on the design requirements, components may be built from these templates by supplying the necessary parameters.

After the structural design is synthesized (as a net list of instantiated generic components), the design is passed on to the MILO system [VaGa88]. MILO performs microarchitectural and logic optimizations on the design and invokes component generators which map the generic components to technology-specific components.

1.1. Advantages of Using Generic Components

There are several advantages in maintaining a library of generic components:

- generic components are functionality generators.

- it permits a truly "generic" view of structural elements; this makes the task of behavior-to-structure binding uniform.
- it permits efficient synthesis by generating the structure for *only* the functionality desired (for example, only the ADD and AND functions for an ALU).
- details of control encoding for components can be hidden from behavioral synthesis by requiring one control line per function; the technology mapper and logic optimizer can perform the encoding later.
- each component has associated functions that return estimates for area, power and delay based on the parameters used to invoke a component; this permits feedback of low-level information.
- it hides technology dependence of component implementation.
- it simplifies retargetting of a design to new libraries.

- it is extensible; new component types can be characterized and added to the library.

- it is general; allows modeling of buses, storage elements, functional units and finite state controllers.

2. Previous Work

Abstract component characterization is an important task in high level synthesis, since these component models determine the "goodness" of a synthesized design. Currently, most behavioral synthesis systems use a two level representation for the component data base. The parent level describes the components with their attributes and characteristics, while the lower level describes instances duplicated from these components, possibly with some limited amount of parameterization for the size or bit-width. For instance, an ALU component can be instantiated with a specified bit-width, but the functions performed by the ALU are fixed. This two-level representation is not powerful enough to handle more general types of components which have almost all of their attributes (including functionality) parameterized. A hierarchical representation, using the notion of of "types", "generators", "components" and "instances" introduced in this report, overcomes this problem.

Quite often, the component data base is embedded within the synthesis system as part of the synthesis code. This makes the task of generic library management cumbersome. Since there is no clean separation between the synthesis code and the underlying component database, modification of an existing component or addition of a new component necessitates rewriting parts of the synthesis code. Furthermore, since the models of these components are often tied to a particular technology library, a lot of effort is required to retarget the components to a new technology library. What is desired is a clean separation between the synthesis tasks and the components used for synthesis.

Another problem with existing representations is that they treat "components", "wires", "ports", "buses", etc. differently. This limits the kinds of optimizations that can be performed by the synthesis tools. For instance, the concept of "unit merging" is similar to that of "bus merging", but these tasks are treated differently since "units" and "buses" have different representations.

Although components can perform several operations simultaneously, it is a difficult task to characterize operational simultaneity in a component for the task of synthesis. Since most behavioral languages have the notion of a single assignment operation, mapping an operation to a component that performs several operations simultaneously can be messy. This requires a many-to-one mapping from the language operators to the structural component. In fact, the component may generate outputs for which there are no corresponding behavioral variables (the carry-out on an adder, for example). The other problem is the representation of costs for simultaneous operations performed by component. A carry-out on an adder component is obtained for no cost when the adder is explicitly performing an "add" operation in the language. However, if only the carry-out is required (without the sum), the cost of this operation is now that of the addition. Hence we need the notion of "operation classes" which is introduced in this report. Operation classes permit the representation of simultaneous operations and combined costs for synthesis.

Finally, many behavioral systems do not have explicit behavioral models for components in the data base. This is essential if the user wishes to perform simulation to verify the correctness of a structural design.

3. GENUS System Overview

The structure of GENUS is hierarchical. This section describes the hierarchy in GENUS, the functions used to create and access elements in GENUS, and describes how a particular technology library may be used to restrict the generators to produce only those generic components that can be feasibly realized with that library.

3.1. Hierarchy

GENUS is organized into 4 levels of hierarchy, where each level inherits attributes from its parent level. This representation closely models a hierarchical object oriented database. For implementation reasons, the current version of GENUS is not written using an object-oriented programming language, although the implementation maintains such a flavor.

Figure 1 shows a sample GENUS snapshot, where instances I1 through I5 are children of the class of 4-bit register components. The register components are generated from the class of register generators by specifying some or all of the register parameters (in this particular example, only the number of bits was specified). Finally, the register generator class belongs to the sequential type class, where all elements are activated by a clock.

Both the Design Procedure Language [BaHa80] and Fred [Wolf86] maintain similar hierarchical representation of objects at the layout level.

3.1.1. Types

The type class describes the abstract functionality of elements in GENUS. Sample type attributes include *combinatorial*, *sequential*, *interface* and *miscellaneous*.

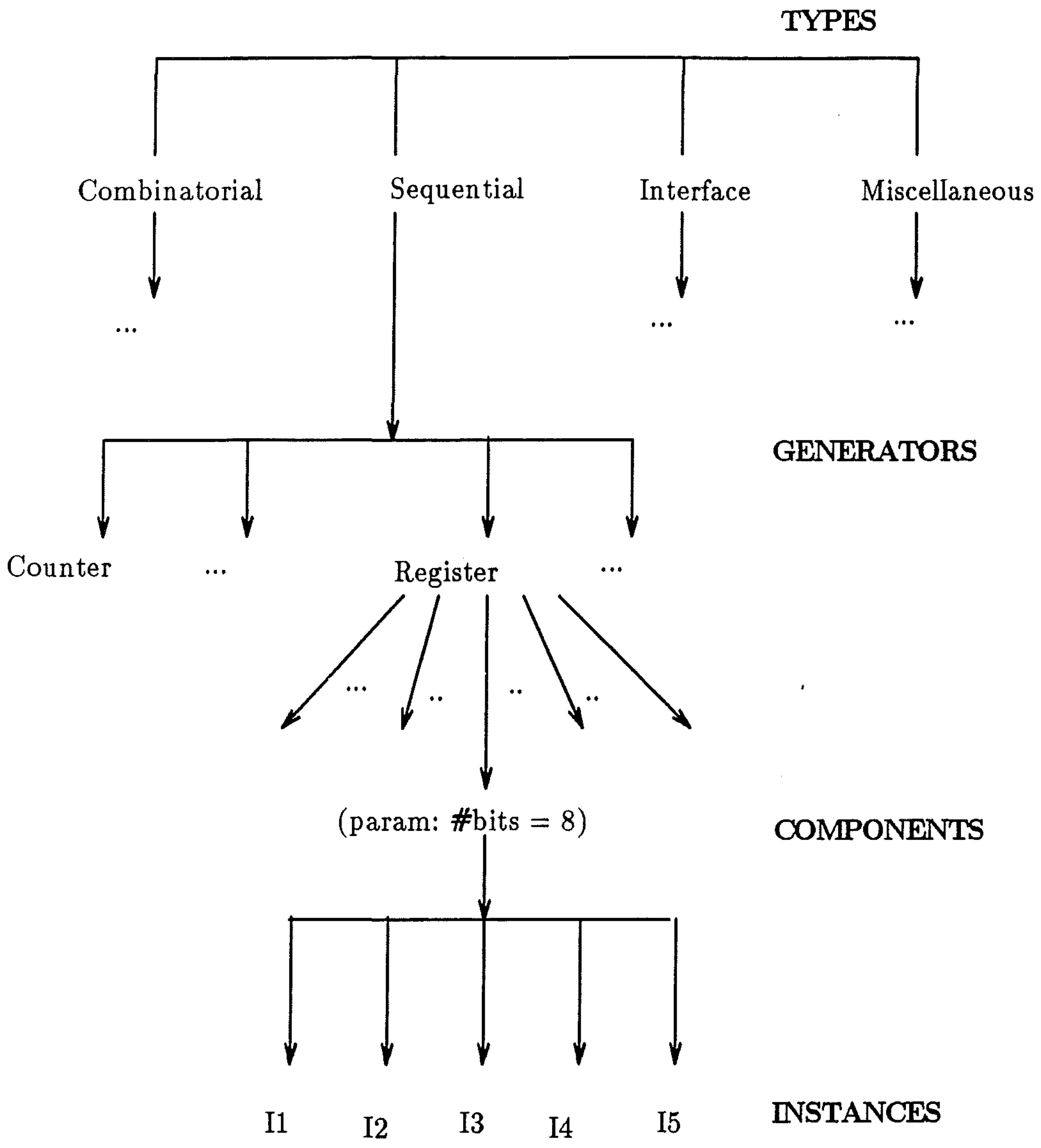

Figure 1. Hierarchy in GENUS

3.1.2. Generators

A generator class is used to generate a family of similar components and instances. Briefly, a generator descriptor maintains a list of all the possible parameters, definitions for each operation performed by a generated component. AN op-class for a generator describes those operations that may be performed in parallel and their combined costs.

Chapter 6 describes the format of a generator description, while Appendix B describes the complete generator library.

3.1.3. Components

A component is generated by passing a list of parameters to the parent generator. For instance, in Figure 1, a 4-bit register component is generated by specifying the bit-width attribute to the register generator. All possible parameters for a particular generator need not be specified; missing parameters are assigned default values.

3.1.4. Instances

Instances are "carbon-copies" of a generated component, with differing names. These GENUS elements are the ones actually used for connectivity in the structural design. Since an instance inherits all of its attributes from the parent component, only the connectivity of the instance is stored in its representation.

3.1.5. Representation

Figure 2 shows the basic data structures used to represent generators, components and instances.

3.2. Using GENUS

The most common operations performed on the generic component library are creation of components and instances, and querying the library for various attributes.

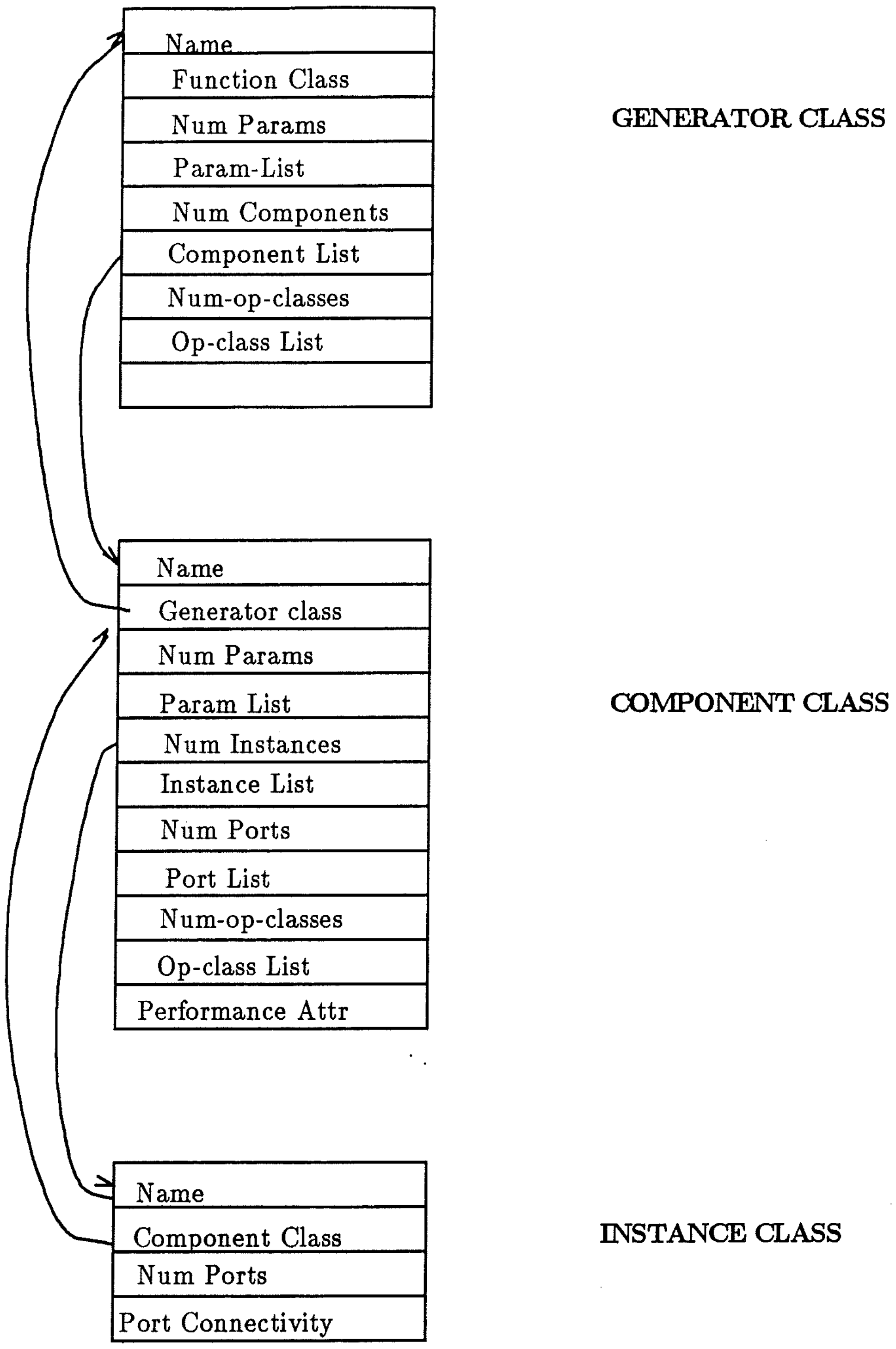

Figure 2. Basic GENUS Data Structures

3.2.1. Creating Components and Instances

Since the library is organized hierarchically, any attempt to create a new component or instance must begin at the parent generator class. Functions for creating new components are passed a parameter list; the parent generator class is searched to see if a component is already generated by matching the parameter values. Similarly, the request to create a new instance of a component is passed a parameter list to the root generator class. If a component for this parameter list does not already exist, a new one is created. Finally, the instance itself is created.

3.2.2. Query Functions

A variety of query functions access the GENUS database at each level. Queries may be initiated at the root (generator), or at a particular level of the hierarchy. For instance, a query to find the number of 4-bit registers instantiated in the database starts at the register generator (the root of the register hierarchy) with the appropriately configured parameter list. On the other hand, a query to check if instance I4 in Figure 1 has a RESET port begins at the instance level and necessitates a look-up of its parent's attribute list (the 4-bit register component) for the existence of a RESET port.

3.3. Technology Library Restrictor

When the completed structural (generic) design is to be mapped to a particular technology library, certain generic components may not exhibit a clean mapping to the corresponding technology library components. The task of performing this technology mapping can become very cumbersome unless the user provides technology specific hints to

GENUS so that a only "feasible" set of components are generated for the particular technology library.

This task is accomplished through the technology library restrictor, which prunes the parameter list for a generator so that only "well-behaved" generic components are generated.

4. GENUS Components

As mentioned earlier, components in GENUS belong one of several type classes, based on their properties and/or functions. This chapter describes the semantics, assumptions and naming conventions associated with GENUS components.

4.1. Port Naming Convention

Ports on each component are categorized into data input, data output, data input-output, control, asynchronous, enable and clock types. Input ports names begin with an "I", output port names begin with an "O", input-output port names begin with a "B" (for Bidirectional), control and enable port names begin with a "C", the clock is labeled "CLK", while async ports begin with an "A".

4.2. Port Semantics

synchronous operations are performed when the clock is high and the enable line (if any) is high. Asynchronous operations override the clocked operations. For combinatorial components, there is no port of type "CLOCK"; operations are inhibited only if the associated "ENABLE" line for the component is low. Non-sequential components do not have asynchronous ports.

4.3. Component Control

In our model of a generic component, a multi-operation component has a *separate control line* for each operation. Because of this assumption, a component which is controlled by a line wider than a single bit has this control line labeled as an input. An example is the *select* input for a SELECTOR component which is wider than a single bit for more than 2 data inputs; this line is labeled "ISEL" and is treated as an input port for consistency. Similarly, the address lines for memories and register files are treated as inputs.

4.4. Combinatorial Components

Figure 3 shows a table of combinatorial components available in the generic component library. Both primitive logic gates and bit-wise logic gates are described in the table. Except for the primitive and bit-wise logic gates, each component has an optional enable input. The logic unit (LU) performs all 16 possible logical functions of two inputs. The MUX component selects input I<i> when control line C<i> is high, and permits the generation of an inverted output. The selector component chooses the input whose guard value matches the value on the single input line ISEL. The DECODER takes an n-bit input and outputs 2^n single bit lines, where line i is 1 when the input equals the value of i. Conversely, and ENCODER component takes 2^n boolean inputs and produces n encoded outputs (where the encoding is determined by the encoder type). The COMPARATOR, SHIFTER, ADD_SUB, MULT and DIV components are self-explanatory. The ALU can perform four arithmetic, five comparison and all sixteen logical operations. At the time of instantiation, a subset of these functions may be chosen for implementation.

LIST OF COMBINATORIAL COMPONENTS					
Type	**Functions**	**Data I/O**	**Control**	**Async**	**Attributes**
Logic Gates (Single)	GAND, GOR, GNAND, GNOR GXOR, GXNOR GNOT	I0: input O0: output			#input bits
Bitwise Logic Gates	AND, OR, NAND, NOR XOR, XNOR	I0..I<n-1>:input O0: output			#inputs (n) #bits
Logic Unit	ZERO, ONE AND, NAND RINHI(xy') LNOT, LID LINHI(x'y) RID(y) XOR, OR NOR, XNOR RNOT(y') LIMPL(x+y') RIMPL(x'+y)	I0,I1: input O0: output	CZERO, CONE CAND, CNAND CRINHI CLNOT, CLID CLINHI CRID CXOR, COR CNOR, CXNOR CRNOT CLIMPL CRIMPL		#input bits, #functions, func. list
Mux	Mux input i	I0..I<n-1>: input O0: output	CI0..CI<n-1>		#bits, #inputs inv?
Selector	Select (on guard val)	ISEL,I0..I<n-1>: input O0: output			#bits, #inputs, guards, c-width, else_flag
Decoder		I0: input O0..O2^n-1			input_width(n), type, else-option
Encoder		I0..I2^n-1 O0..O<n-1>			#outputs(n), type
Comparator	EQ, NEQ GT, LT GEQ, LEQ	I0, I1: inputs OEQ, ONEQ, OGT, OLT,OGEQ,OLEQ: outputs	CEQ, CNEQ, CGT, CLT, CGEQ, CLEQ		#bits #functions func-list
Shifter	SHR0, SHR1, SHL0, SHL1, ROTR, ROTL, ASHL, ASHR	I0, ILIN, IRIN, ISHNUM: input O0: output	CSHR0, CSHR1, CSHL0, CSHL1, CROTR, CROTL CASHL, CASHR		#bits, #functions, func-list, mode, fill, maxshift
Barrel Shifter	SHR, SHR, ASHL, ASHL, ROTR, ROTL,	I0, ISHNUM, ILR, IROT,IFILL,IMODE: input O0: output	CSHR, CSHR, CASHL, CASHL, CROTR, CROTL		#bits, maxshift, #functions, func-list
Adder/ Subtractor	+, -	I0, I1, ICIN: input O0, OCOUT: output	CADD CSUB		#bits, #fns, fn-list, style, #pipe-st
ALU	{+,-,INC,DEC} {>,<,=,!=,ZRO} {16 logic fns}	I0, I1: input O0, 5-cond, OCOUT: output	1-per fn		#bits, style, #fns func-list, #pipe-st
Multiplier	*	I0, I1: input O0: output			#bits, style, #pipe-st
Divider	/	I0, I1: input O0: output			#bits, style, #pipe-st

Figure 3. Combinatorial Components

4.5. Sequential Components

Figure 4 shows the list of available sequential components. As mentioned earlier, each sequential component is assumed to have a port named "CLK". If asynchronous ports exist for the component, they override the clocked, synchronous behavior of the component. A register component may have the positive output "OQ", the negated output "OQN" or both outputs generated. Both registers and counters must have a set-value specified at instantiation time. The counter component can count up and down, besides doing a synchronous load and an asynchronous set and reset. For the register-file component, each port pair (I<i>,O<i>) has associated with it an address line A<i>, and a port-attribute which indicates if that port is of type input, output or bidirectional.

LIST OF SEQUENTIAL COMPONENTS					
Type	**Functions**	**Data-i/o**	**control**	**async**	**attributes**
Register	load, shl, shr,	I0, LIN, RIN: input, OQ, OQN: output	CLOAD, CSHL, CSHR, CEN	ACLEAR, ASET	#bits, #fns, type, set-val, en OQ?, OQN?
Counter	load, up, down, clear, set	I0: input O0: output	CLOAD, CUP, CDOWN, CEN	ACLEAR ASET	#bits, #fns, set-val, style, type, enable
Register File		I0,..,I<n-1> IA0,..,IA<n-1>	CR0,CW0,.. CR<n-1>,CW<n-1>		#bits, #words #ports, port_attr, en
Stack/ FIFO	push, pop	I0: input, O0: output	CPUSH, CPOP, CEN		#bits, #words, type, enable
Memory	read, write	I0, IADDR, IA_VALID: input OD_READY, O0: output	CWRITE,CREAD, CEN		#bits, #words, enable

Figure 4. Sequential Components

4.6. Interface and Miscellaneous Components

Figure 5 shows the list of interface, bus, switchbox, clock and delay components. An interface component has several attributes that describes its function (buffer/clock_driver/...), mode (input/output/...), level (CMOS/TTL/...), output_type(inverting/non-inverting) and drive (L/M/H). The port component models ports on a design, with the attributes number_of_bits and port_mode. The port component is useful in constructing a hierarchy of designs. The BUS and WIRED-OR components are similar, except that the the BUS component has tristate drivers at each input to the bus. CONCAT and EXTRACT components simply model switchbox operations for merging streams of data and extracting substreams of data. At present, the clock generator

LIST OF INTERFACE, BUS, SWITCHBOX AND MISC. COMPONENTS					
Type	**Functions**	**Data I/O**	**Control**	**Async**	**Attributes**

Interface Units	Buffer
	Clock Driver
	
	Schmidt Trigger
	Tristate
		I0: input
		O0: output
			CEN		#bits,
					function
					mode:(i, o, i/o),
					level:(CMOS,TTL, ..)
					output:(inv/non-inv)
					drive:(l, m, h)

Port		I0: input O0: output			#bits, mode:(i, o, i/o)
BUS		I0..I<n-1>: input O0: output	C0..C<n-1>		#bits, n-in, fan-out
WIRED-OR		n-inputs 1-output			#bits, n-in, fan-out
Switchbox Concat	O0 = I0@..@I<n-1>	I0,..,I<n-1>: input O0: output			#inputs, width0,..,width<n-1>
Switchbox Extract	O1 = I0{i:j}	I0: input O1: output			inp-width, l,r index
Clock Generator		O0: output	CEN		clock-period, duration-high
Delay	Delay δ	I0:in, O0:out			delay-value (δ)

Figure 5. Interface and Miscellaneous Components

component is used for modeling a very simple system clock, using the attributes clock-period and duration-high. The DELAY component is used to model a delay element on a logic path.

5. Accessing GENUS

5.1. Accessing Components

Library generators, components and instances are accessed using the appropriate access function with the generator name and a variable number of arguments. Figure 6 shows the general form of an access function. This call specifies the name of the library component and a list of attributes, with the list being terminated by a "0". The call to a generic_component_routine returns an object of the appropriate type (generator, component or instance). A set of standard query routines can be applied to the object to extract any attribute or characteristic for it. Figure 7 shows a sample call used to generate an instance of an ALU. The arguments in the call consist of pairs of reserved global

```
<generic_component_routine>(GC_COMPILER_NAME, <name>, <attribute_list>, 0)
```

Figure 6. Generic Component Access: General Form

```
get_gc_instance(        GC_COMPILER_NAME, ALU,
                        GC_BIT_WIDTH, 16,
                        GC_NUM_FUNCTIONS, 8,
                        GC_FUNCTION_LIST, +, -, INC, DEC, >, <, =, AND,
                        GC_ENABLE_FLAG, FALSE,
                        GC_STYLE, CLA,
                0)
```

Figure 7. Sample ALU Instance Call

symbols (which begin with the letters "GC_") and the appropriate value or list. The size of a list must **always** precede the list itself. For instance, in Figure 7, GC_NUM_FUNCTIONS is assigned the value "8" before specifying the GC_FUNCTION_LIST which consists of 8 operations that the ALU instance will perform. Figure 8 and Figure 9 show the list of global symbols reserved for indicating the type of argument specified in a call, together with their possible values. Appendix A has a complete list of generator calls for all the generic components.

GENERIC COMPONENT GLOBALS		
Name	**Possible Values**	**Default Value**
GC_COMPILER_NAME	*<component-name>*	
GC_NUM_FUNCTIONS	*<integer>*	
GC_FUNCTION_LIST	*<list-of-character-strings>*	
GC_NUM_PORTS	*<integer>*	
GC_PORT_ATTRIBUTE_LIST	*<list-of-character-strings>*	
GC_NUM_GUARDS	*<integer>*	
GC_GUARD_LIST	*<list-of-guard-values>*	
GC_INPUT_WIDTH_LIST	*<list-of-integers>*	
GC_NUM_WORDS	*<integer>*	
GC_NUM_INPUTS	*<integer>*	
GC_NUM_OUTPUTS	*<integer>*	
GC_INPUT_WIDTH	*<integer>*	
GC_CONTROL_WIDTH	*<integer>*	
GC_ADDER_STYLE	GC_RIPPLE_CARRY, GC_CARRY_LOOKAHEAD	GC_RIPPLE_CARRY
GC_ALU_STYLE	GC_RIPPLE_CARRY, GC_CARRY_LOOKAHEAD	GC_RIPPLE_CARRY
GC_MULT_STYLE	GC_ARRAY, GC_WALLACE_DADDA, GC_ITERATIVE	GC_ARRAY
GC_DIV_STYLE	GC_RESTORING, GC_NON_RESTORING, GC_MULTIPLICATIVE	GC_RESTORING
GC_COUNTER_STYLE	GC_RIPPLE_CARRY, GC_CARRY_LOOKAHEAD	GC_RIPPLE_CARRY
GC_ENABLE_FLAG	TRUE, FALSE	FALSE
GC_INVERT_FLAG	TRUE, FALSE	FALSE
GC_ELSE_FLAG	TRUE, FALSE	FALSE
GC_SET_FLAG	TRUE, FALSE	FALSE
GC_RESET_FLAG	TRUE, FALSE	FALSE
GC_PIPELINE_FLAG	TRUE, FALSE	FALSE
GC_PIPELINE_STAGES	*<integer>*	
GC_PIPELINE_DELAY	*<integer>*	

Figure 8. List of Compiler Global Symbols

GENERIC COMPONENT GLOBALS		
Name	Possible Values	Default Value
GC_DECODER_TYPE	GC_BINARY, GC_BCD	GC_BINARY
GC_ENCODER_TYPE	GC_BINARY, GC_BCD	GC_BINARY
GC_REGISTER_TYPE	GC_LATCH, GC_D_FF	GC_D_FF
GC_COUNTER_TYPE	GC_BINARY, GC_BCD, GC_JOHNSON, GC_GRAY	GC_BINARY
GC_STACK_TYPE	GC_STACK, GC_FIFO	
GC_SHIFT_MODE	GC_FILL, GC_EXTEND	GC_FILL
GC_FILL_INPUT	0, 1	0
GC_SHIFT_DISTANCE	<*integer*>	
GC_CLOCK_PERIOD	<*integer*>	
GC_CLOCK_HIGH	<*integer*>	
GC_DELAY_VALUE	<*integer*>	
GC_LEFT_INDEX	<*integer*>	
GC_RIGHT_INDEX	<*integer*>	
GC_INTERFACE_FUNCTION	GC_BUFFER, GC_CLOCK_DRIVER, GC_SCHMIDT, GC_TRISTATE	
GC_INTERFACE_MODE	GC_INPUT, GC_OUTPUT, GC_BIDIRECTIONAL	
GC_INTERFACE_LEVEL	GC_CMOS, GC_TTL, GC_ECL	
GC_INTERFACE_DRIVE	GC_LOW, GC_MEDIUM, GC_HIGH	
GC_FAN_OUT	<*integer*>	
GC_SET_VALUE	<*integer*>	
GC_COUNTER_MODE	GC_SYNCHRONOUS, GC_RIPPLE	GC_SYNCHRONOUS
GC_REG_POS_OUT	TRUE, FALSE	TRUE
GC_REG_INVERT_OUT	TRUE, FALSE	FALSE

Figure 9. List of Compiler Global Symbols (Cont'd)

6. Format of Generator Descriptions

Each generic component generator is described using a special notation which allows it to be characterized by a unique name and a list of attributes describing the type class, implementation styles, parameters, port information and functionality. This table (described in a textual form) can be updated whenever a new generator is to be added to the library. The library description can then be parsed into the internal data structures used to represent the GENUS library. Hence this table may be considered to be an input to the GENUS generic library generator. Appendix B contains the complete description of the generator library.

Figure 10 shows parts of the generator definition for a counter, which will be used as a running example in this section. The ports for a component are categorized into data inputs (INPUTS) outputs (OUTPUTS), while control-specific ports are listed under CLOCK, ENABLE, CONTROL and ASYNC entries.

```
NAME:           COUNTER
CLASS:          Clocked
MAX_PARAMS:     7
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_FUNCTIONS,
                GC_FUNCTION_LIST, GC_SET_VALUE, GC_STYLE, GC_ENABLE_FLAG
NUM_STYLES:     2
STYLES:         SYNCHRONOUS, RIPPLE
NUM_INPUTS:     1
INPUTS:         I0[%w]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
CLOCK:          CLK
NUM_ENABLE:     1
ENABLE:                 CEN
NUM_CONTROL:    3
CONTROL:        CLOAD, CUP, CDOWN
NUM_ASYNC:      2
ASYNC:          ASET, ARESET
NUM_OPERATIONS:         3
OPERATIONS:
        (       (LOAD)
                (INPUTS:        I0)
                (OUTPUTS:       O0)
                (CONTROL:       CLOAD)
                (OPS:   (LOAD:  O0 = I0)))
        (       (COUNT_UP)
                (OUTPUTS:       O0)
                (CONTROL:       CUP)
                (OPS:   (COUNT_UP: O0 = O0 + 1)))
        (       (COUNT_DOWN)
                (OUTPUTS:       O0)
                (CONTROL:       CDOWN)
                (OPS:   (COUNT_DOWN: O0 = O0 - 1)))
OP_CLASSES:     default
```

Figure 10. Sample GENUS Generator Description

6.1. Name

This specifies a unique name for a generator.

6.2. Class

Specifies if the generator is of type class *clocked* or *combinational.*

6.2.1. Clocked Class When a component is *clocked*, certain semantics are associated with the ports on the component.

The CLOCK entry specifies the name of the clock line(s) for the component (currently only one clock line is assumed). For edge-triggered components, the attribute "RISING_EDGE" or "FALLING_EDGE" indicates when the clock is active.

The ENABLE attribute, when assigned a port name, activates the component for clocked behavior. For instance, in Figure 9, the counter exhibits synchronous operation only when CEN is high. If no port is specified for the ENABLE entry, a clocked component is assumed to be enabled all the time.

The CONTROL attributes specify the clocked control with one line per function. For instance, the counter in Figure 9 has separate lines for the synchronous operations LOAD, COUNT_UP and COUNT_DOWN.

The ASYNC ports specify control lines that invoke asynchronous behavior: they override any clocked control that may be simultaneously active. For example, the ASET port in Figure 9 is an asynchronous set line for the counter.

The semantics of the CLOCK, ENABLE, CONTROL and ASYNC lines are implicit in the definition of a component.

6.2.2. Combinational Class

For combinational generators, there are no entries under CLOCK and ASYNC in the table. The ENABLE entry is optional; if it is specified, a component is generated with an enable line. For combinational generators exhibiting multi-function behavior, each function is assigned a unique CONTROL line.

6.3. Parameters

The MAX_PARAMS and PARAMETERS entries indicate the number and global symbols used to describe the generic generator. For the counter in Figure 9, the parameterized input width is represented by the variable "%w"; this variable is used in the rest of the component description as a parameterized variable.

6.4. Styles

The STYLES entry indicates the list of possible implementation styles for generating instances of the component. For the counter in Figure 9, the implementation styles are SYNCHRONOUS and RIPPLE.

6.5. Ports

Ports are specified under the INPUTS, OUTPUTS, INPUT_OUTPUTS, CONTROL, CLOCK, ASYNC and ENABLE entries. Ports specified under CONTROL, CLOCK, ASYNC and ENABLE are assumed to be one bit wide by default. For the INPUTS and OUTPUTS, each port has a bit-width specified within the "[" and "]" pair. A parameterized variable (which starts with the character "%") may be used when necessary.

6.6. Operations

Each operation that can be performed by a generated component is described by its name, input, output and control port information. The operation is itself modeled with a piece of C code so that it can be easily simulated.

6.7. Op_classes

Each entry here describes the list of possible operations that may be performed in parallel for the generated component. We can associate cost functions for implementing any combination of these operations in each OP_CLASS. This permits realistic modeling of structural components. A "default" op_class indicates that each operation is mutually exclusive and cannot be performed simultaneously with any other operation.

6.8. Macro Expansion and Port Naming

For generated components that have a parameterizable number of ports (or operations), the list of port names are generated by calling special functions that return a name or a list of names. These function names start with the special symbol "&" to distinguish them from other names in the table. Similarly, the operations performed by a component may depend on some arguments of the parameter list. Hence the "macro-expand" feature is used to describe this functionality. Figure 11 shows a sample definition for a MUX com-

```
NAME:            MUX
CLASS:           Combinatorial
MAX_PARAMS:      5
PARAMETERS:      GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_INPUTS (%n),
                 GC_ENABLE_FLAG, GC_INVERT_FLAG
NUM_INPUTS:      %n
INPUTS:          &get_component_pin_name_list(MUX, INPUT, %n, %w)
NUM_OUTPUTS:     1
OUTPUTS:         O0[%w]
NUM_CONTROL:     %n
CONTROL:         &get_component_pin_name_list(MUX, CONTROL, %n, 1)
NUM_ENABLE:      1
ENABLE:                 CEN
NUM_OPERATIONS:         %n
OPERATIONS:
        macro_expand ($i = 0 to %n-1)
        {
    ( (&get_component_function(MUX, $i))
                (INPUTS:        &get_component_pin_name(MUX, INPUT, $i))
                (OUTPUTS:       O0)
                (CONTROL:       &get_component_pin_name(MUX, CONTROL, $i))
                (OPS:    (      O0 = &get_component_pin_name(MUX, INPUT, $i))))
        }
OP_CLASSES:      default
```

Figure 11. Macro-Expand Feature

ponent. The index of the macro-expand loop is a variable whose name begins with a "$". Note that in the parameter list, the input width and the number of inputs are parameterized (and represented by %w and %n respectively). Since the input port names depend on the number of inputs, we use the function "&get_component_pin_name_list" to generate the list of pin-names for the MUX inputs.

Further, in Figure 10, the operation of the MUX component is dependent on the number of inputs and the input and control port names, all of which are parameterized. Hence we use the macro-expand feature to describe the functionality by looping through every input and control pair.

6.9. Estimation Functions

The first version of the generic component library will use estimators derived from Chippe's model of function units [Brew88]. Functions for area, speed and power return estimates based on the size, functionality and bit-width of a generated component.

7. Acknowledgements

The definition of the library went through several iterations; thanks are due to Prof. Gajski, Tedd Hadley, Joe Lis and Nels Vander Zanden for making suggestions and correcting errors in the document.

8. References

[BaHa80] J. Batali and A. Hartheimer, "The Design Procedure Language Manual," A.I. Memo No. 598, MIT A.I. Laboratory, Sept. 1980.

[Brew88] Forrest D. Brewer, "Constraint Driven Behavioral Synthesis," *PhD Dissertation*, *University of Illinois*, *Urbana-Champaign* (May, 1988).

[DuGa88] N.D. Dutt and D. Gajski, "EXEL: An Input Language for Extensible Register Transfer Compilation," Technical Report #88-11, University of California at Irvine, April 1988.

[LiGa88] J. S. Lis and D. D. Gajski, "VSS: A VHDL Synthesis System," Technical Report (in preparation), University of California at Irvine, April 1988.

[VaGa88] N. Vander Zanden and D. D. Gajski, "MILO: A Microarchitecture and Logic Optimizer," *Proc. 25th Design Automation Conference*, Anaheim, CA, June 1988.

[Wolf86] Wayne Wolf, "An Object Oriented, Procedural Database for VLSI Chip Planning" *23rd Design Automation Conference* IEEE, pp. 744-751, Las Vegas, NV, (July, 1986).

9. APPENDIX A: GENERIC COMPILER CALLS

COMBINATORIAL COMPONENTS

LOGIC GATES

(GC_COMPILER_NAME, <name>, GC_INPUT_WIDTH, <inp-width>, 0)

BITWISE LOGIC GATES

(GC_COMPILER_NAME, <name>, GC_NUM_INPUTS, <n>, GC_INPUT_WIDTH, <inp-width>, 0)

LOGIC UNIT

(GC_COMPILER_NAME, LU, GC_INPUT_WIDTH, <num-inputs>, GC_NUM_FUNCTIONS, <num-fus>, GC_FUNCTION_LIST, <fn-list>, 0)

MUX

(GC_COMPILER_NAME, MUX, GC_INPUT_WIDTH, <inp-width>, GC_NUM_INPUTS, <n>, GC_INVERT_FLAG, <T/F>, 0)

SELECTOR

(GC_COMPILER_NAME, SELECTOR, GC_INPUT_WIDTH, <inp-width>, GC_NUM_GUARDS, <n>, GC_GUARD_LIST, <g-list>, GC_ELSE_FLAG, <T/F>, GC_CONTROL_WIDTH, <c-width>, 0)

ENCODER

(GC_COMPILER_NAME, ENCODER, GC_NUM_OUTPUTS, <n>, GC_ENCODER_TYPE, <gc-type>, 0)

DECODER

(GC_COMPILER_NAME, DECODER, GC_INPUT_WIDTH, <inp-width>, GC_DECODER_TYPE, <gc-type>, GC_ELSE_FLAG, <T/F>, 0)

COMPARATOR

(GC_COMPILER_NAME, COMPARATOR, GC_INPUT_WIDTH, <inp-width>, GC_NUM_FUNCTIONS, <num-fus>, GC_FUNCTION_LIST, <fn-list>, 0)

SHIFTER

(GC_COMPILER_NAME, SHIFTER, GC_INPUT_WIDTH, <inp-width>, GC_SHIFT_DISTANCE, <dist>, GC_NUM_FUNCTIONS, <num-fus>, GC_FUNCTION_LIST, <fn-list>, GC_SHIFT_MODE, <mode>, GC_FILL_INPUT, <0/1>, 0)

BARREL SHIFTER

(GC_COMPILER_NAME, BARREL_SHIFTER, GC_INPUT_WIDTH, <inp-width>, GC_SHIFT_DISTANCE, <dist>, GC_NUM_FUNCTIONS, <num-fus>, GC_FUNCTION_LIST, <fn-list>, 0)

ADDER_SUBTRACTOR

(GC_COMPILER_NAME, ADD_SUB, GC_INPUT_WIDTH, <inp-width>, GC_NUM_FUNCTIONS, <num-fus>, GC_FUNCTION_LIST, <fn-list>, GC_ADDER_STYLE, <style>, 0)

ALU

(GC_COMPILER_NAME, ALU, GC_INPUT_WIDTH, <inp-width>, GC_ALU_STYLE, <style>, GC_NUM_FUNCTIONS, <num-fus>, GC_FUNCTION_LIST, <fn-list>, 0)

MULTIPLIER

(GC_COMPILER_NAME, MULT, GC_INPUT_WIDTH, <inp-width>, GC_MULT_STYLE, <style>, 0)

DIVIDER

(GC_COMPILER_NAME, DIV, GC_INPUT_WIDTH, <inp-width>, GC_DIV_STYLE, <style>, 0)

SEQUENTIAL COMPONENTS

REGISTER

(GC_COMPILER_NAME, REGISTER, GC_INPUT_WIDTH, <inp-width>, GC_NUM_FUNCTIONS, <num-fus>, GC_FUNCTION_LIST, <fn-list>, GC_REGISTER_TYPE, <type>, GC_SET_VALUE, <value>, GC_REGISTER_POS_OUT, <T>, GC_REGISTER_INVERT_OUT, <T>, GC_ENABLE_FLAG, <T/F>, 0)

COUNTER

(GC_COMPILER_NAME, COUNTER, GC_INPUT_WIDTH, <inp-width>, GC_NUM_FUNCTIONS, <num-fus>, GC_FUNCTION_LIST, <fn-list>, GC_COUNTER_TYPE, <type>, GC_SET_VALUE, <value>, GC_COUNTER_STYLE, <style>, GC_ENABLE_FLAG, <T/F>, 0)

REGISTER FILE

(GC_COMPILER_NAME, REG_FILE, GC_INPUT_WIDTH, <inp-width>, GC_NUM_WORDS, <num-words>, GC_NUM_PORTS, <num-ports>, GC_PORT_ATTRIBUTE_LIST, <a-list>, GC_ENABLE_FLAG, <T/F>, 0)

STACK or FIFO

(GC_COMPILER_NAME, <STACK or FIFO>, GC_INPUT_WIDTH, <inp-width>, GC_NUM_WORDS, <num-words>, GC_ENABLE_FLAG, <T/F>, 0)

MEMORY

(GC_COMPILER_NAME, MEMORY, GC_INPUT_WIDTH, <inp-width>, GC_NUM_WORDS, <num-words>, GC_NUM_PORTS, <num-ports>, GC_PORT_ATTRIBUTE_LIST, <a-list>, GC_ENABLE_FLAG, <T/F>, 0)

INTERFACE AND MISCELLANEOUS COMPONENTS

INTERFACE

(GC_COMPILER_NAME, INTERFACE, GC_INPUT_WIDTH, <inp-width>, GC_INTERFACE_FUNCTION, <fn-name>, GC_INTERFACE_MODE, <mode>, GC_INTERFACE_LEVEL, <level>, GC_INVERT_FLAG, <T/F>, GC_INTERFACE_DRIVE, <drive>, 0)

PORT

(GC_COMPILER_NAME, PORT, GC_INPUT_WIDTH, <inp-width>, GC_PORT_MODE,

<mode>, 0)

BUS

(GC_COMPILER_NAME, BUS, GC_INPUT_WIDTH, <inp-width>, GC_NUM_INPUTS, <num-inp>, GC_FAN_OUT, <fan-out>, 0)

WIRED-OR

(GC_COMPILER_NAME, WIRED-OR, GC_INPUT_WIDTH, <inp-width>, GC_NUM_INPUTS, <num-inp>, GC_FAN_OUT, <fan-out>, 0)

CONCAT

(GC_COMPILER_NAME, CONCAT, GC_NUM_INPUTS, <num-inp>, GC_INPUT_WIDTH_LIST, <w-list>, 0)

EXTRACT

(GC_COMPILER_NAME, EXTRACT, GC_INPUT_WIDTH, <inp-width>, GC_LEFT_INDEX, <l-index>, GC_RIGHT_INDEX, <r-index>, 0)

CLOCK GENERATOR

(GC_COMPILER_NAME, CLOCK, GC_CLOCK_PERIOD, <period>, GC_CLOCK_HIGH, <high>, 0)

DELAY ELEMENT

(GC_COMPILER_NAME, DELAY, GC_DELAY_VALUE, <delay>, 0)

10. APPENDIX B: GENERIC COMPONENT LIBRARY DEFINITIONS

GENERIC COMPONENTS

```
                    /* COMBINATORIAL COMPONENTS */


                    /* Primitive Logic Gates with Boolean outputs */
               /* Similar entries for primitive GOR, GNAND, GNOR, GXOR, GXNOR and GNOT */

NAME:          GAND
CLASS:         Combinatorial
MAX_PARAMS:    2
PARAMETERS:    GC_COMPILER_NAME, GC_INPUT_WIDTH (%w)
NUM_INPUTS:    1
INPUTS:        I0[%w]
NUM_OUTPUTS:   1
OUTPUTS:       O0[1]
NUM_OPERATIONS:     1
OPERATIONS:
     ((BAND)
               (INPUTS:       I0)
               (OUTPUTS:      O0)
               (OPS:    (     O0 = 1;
                              macro_expand ($i = 0 to %w-1)
                                { O0 = O0 AND I0[$i]; }  )))
OP_CLASSES:    default


NAME:          GOR
CLASS:         Combinatorial
MAX_PARAMS:    2
PARAMETERS:    GC_COMPILER_NAME, GC_INPUT_WIDTH (%w)
NUM_INPUTS:    1
INPUTS:        I0[%w]
NUM_OUTPUTS:   1
OUTPUTS:       O0[1]
NUM_OPERATIONS:     1
OPERATIONS:
     ((BOR)
               (INPUTS:       I0)
               (OUTPUTS:      O0)
               (OPS:    (     O0 = 1;
                              macro_expand ($i = 0 to %w-1)
                                { O0 = O0 OR I0[$i]; }    )))
OP_CLASSES:    default


NAME:          GNAND
CLASS:         Combinatorial
MAX_PARAMS:    2
PARAMETERS:    GC_COMPILER_NAME, GC_INPUT_WIDTH (%w)
NUM_INPUTS:    1
INPUTS:        I0[%w]
NUM_OUTPUTS:   1
OUTPUTS:       O0[1]
NUM_OPERATIONS:     1
OPERATIONS:
     ((BNAND)
               (INPUTS:       I0)
               (OUTPUTS:      O0)
```

```
                    (OPS:    (          O0 = 1;
                                        macro_expand ($i = 0 to %w-1)
                                          { O0 = O0 NAND I0[$i]; } )))
OP_CLASSES:         default


NAME:               GNOR
CLASS:              Combinatorial
MAX_PARAMS:         2
PARAMETERS:         GC_COMPILER_NAME, GC_INPUT_WIDTH (%w)
NUM_INPUTS:         1
INPUTS:             I0[%w]
NUM_OUTPUTS:        1
OUTPUTS:            O0[1]
NUM_OPERATIONS:            1
OPERATIONS:
          ((BNOR)
                    (INPUTS:            I0)
                    (OUTPUTS:           O0)
                    (OPS:    (          O0 = 1;
                                        macro_expand ($i = 0 to %w-1)
                                          { O0 = O0 NOR I0[$i]; }   )))
OP_CLASSES:         default


NAME:               GXOR
CLASS:              Combinatorial
MAX_PARAMS:         2
PARAMETERS:         GC_COMPILER_NAME, GC_INPUT_WIDTH (%w)
NUM_INPUTS:         1
INPUTS:             I0[%w]
NUM_OUTPUTS:        1
OUTPUTS:            O0[1]
NUM_OPERATIONS:            1
OPERATIONS:
          ((BXOR)
                    (INPUTS:            I0)
                    (OUTPUTS:           O0)
                    (OPS:    (          O0 = 1;
                                        macro_expand ($i = 0 to %w-1)
                                          { O0 = O0 XOR I0[$i]; }   )))
OP_CLASSES:         default


NAME:               GXNOR
CLASS:              Combinatorial
MAX_PARAMS:         2
PARAMETERS:         GC_COMPILER_NAME, GC_INPUT_WIDTH (%w)
NUM_INPUTS:         1
INPUTS:             I0[%w]
NUM_OUTPUTS:        1
OUTPUTS:            O0[1]
NUM_OPERATIONS:            1
OPERATIONS:
          ((BXNOR)
                    (INPUTS:            I0)
                    (OUTPUTS:           O0)
                    (OPS:    (          O0 = 1;
                                        macro_expand ($i = 0 to %w-1)
                                          { O0 = O0 XNOR I0[$i]; } )))
OP_CLASSES:         default
```

```
NAME:           GNOT
CLASS:          Combinatorial
MAX_PARAMS:     1
PARAMETERS:     GC_COMPILER_NAME  /* boolean input */
NUM_INPUTS:     1
INPUTS:         I0[1]
NUM_OUTPUTS:    1
OUTPUTS:        O0[1]
NUM_OPERATIONS:         1
OPERATIONS:
        ((BNOT)
                (INPUTS:        I0)
                (OUTPUTS:       O0)
                (OPS:     (O0 = NOT I0; )))
OP_CLASSES:     default
```

/* **Bitwise Logic Gates** */

```
NAME:           AND
CLASS:          Combinatorial
MAX_PARAMS:     3
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_INPUTS (%n)
NUM_INPUTS:     %n
INPUTS:         &get_component_pin_name_list(AND, INPUT, %n, %w)
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
NUM_OPERATIONS:         1
OPERATIONS:
        ((AND)
                (COMMUTATIVE)
                (INPUTS:        &get_component_pin_name_list(AND, INPUT, %n, %w)
                (OUTPUTS:       O0)
                (OPS:     (     O0 = 1;
                                macro_expand ($i = 0 to %n-1)
                                  { O0 = O0 AND I0[$i]; }  )))
OP_CLASSES:     default

NAME:           NAND
CLASS:          Combinatorial
MAX_PARAMS:     3
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_INPUTS (%n)
NUM_INPUTS:     %n
INPUTS:         &get_component_pin_name_list(NAND, INPUT, %n, %w)
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
NUM_OPERATIONS:         1
OPERATIONS:
        ((NAND)
                (COMMUTATIVE)
                (INPUTS:        &get_component_pin_name_list(NAND, INPUT, %n, %w)
                (OUTPUTS:       O0)
                (OPS:     (     O0 = 1;
                                macro_expand ($i = 0 to %n-1)
                                  { O0 = O0 NAND I0[$i]; })))
OP_CLASSES:     default

NAME:           OR
CLASS:          Combinatorial
MAX_PARAMS:     3
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_INPUTS (%n)
```

```
NUM_INPUTS:     %n
INPUTS:         &get_component_pin_name_list(OR , INPUT, %n, %w)
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
NUM_OPERATIONS:         1
OPERATIONS:
        ((OR )
                (COMMUTATIVE)
                (INPUTS:        &get_component_pin_name_list(OR , INPUT, %n, %w)
                (OUTPUTS:       O0)
                (OPS:    (      O0 = 1;
                                macro_expand ($i = 0 to %n-1)
                                   { O0 = O0 OR  I0[$i]; }   )))


NAME:           NOR
CLASS:          Combinatorial
MAX_PARAMS:     3
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_INPUTS (%n)
NUM_INPUTS:     %n
INPUTS:         &get_component_pin_name_list(NOR, INPUT, %n, %w)
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
NUM_OPERATIONS:         1
OPERATIONS:
        ((NOR)
                (COMMUTATIVE)
                (INPUTS:        &get_component_pin_name_list(NOR, INPUT, %n, %w)
                (OUTPUTS:       O0)
                (OPS:    (      O0 = 1;
                                macro_expand ($i = 0 to %n-1)
                                   { O0 = O0 NOR I0[$i]; }  )))


NAME:           XOR
CLASS:          Combinatorial
MAX_PARAMS:     3
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_INPUTS (%n)
NUM_INPUTS:     %n
INPUTS:         &get_component_pin_name_list(XOR, INPUT, %n, %w)
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
NUM_OPERATIONS:         1
OPERATIONS:
        ((XOR)
                (COMMUTATIVE)
                (INPUTS:        &get_component_pin_name_list(XOR, INPUT, %n, %w)
                (OUTPUTS:       O0)
                (OPS:    (      O0 = 1;
                                macro_expand ($i = 0 to %n-1)
                                   { O0 = O0 XOR I0[$i]; }  )))


NAME:           XNOR
CLASS:          Combinatorial
MAX_PARAMS:     3
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_INPUTS (%n)
NUM_INPUTS:     %n
INPUTS:         &get_component_pin_name_list(XNOR, INPUT, %n, %w)
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
NUM_OPERATIONS:         1
OPERATIONS:
        ((XNOR)
                (COMMUTATIVE)
                (INPUTS:        &get_component_pin_name_list(XNOR, INPUT, %n, %w)
                (OUTPUTS:       O0)
                (OPS:    (      O0 = 1;
                                macro_expand ($i = 0 to %n-1)
                                   { O0 = O0 XNOR I0[$i]; } )))


                /* Logic Unit */

NAME:           LU
CLASS:          Combinatorial
MAX_PARAMS:     4
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_FUNCTIONS,
                GC_FUNCTION_LIST
NUM_INPUTS:     2
INPUTS:         I0[%w], I1[%w]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
NUM_CONTROL:    16
CONTROL:        CZERO, CONE, CAND, CNAND, COR, CNOR, CXOR, CXNOR,
                CLID, CRID, CLNOT, CRNOT, CRINHI, CLINHI, CLIMPL, CRIMPL
NUM_OPERATIONS:         16
OPERATIONS:
        ((ZERO)
                (OUTPUTS:       O0)
                (CONTROL:       CZERO)
                (OPS:    (      O0 = 0)))
        ((ONE)
                (OUTPUTS:       O0)
                (CONTROL:       CONE)
                (OPS:    (      O0 = 1)))
        ((AND)
                (COMMUTATIVE)
                (INPUTS:        I0, I1)
                (OUTPUTS:       O0)
                (CONTROL:       CAND)
                (OPS:    (      O0 = I0 AND I1)))
        ((NAND)
                (COMMUTATIVE)
                (INPUTS:        I0, I1)
                (OUTPUTS:       O0)
                (CONTROL:       CNAND)
                (OPS:    (      O0 = I0 NAND I1)))
        ((OR)
                (COMMUTATIVE)
                (INPUTS:        I0, I1)
                (OUTPUTS:       O0)
                (CONTROL:       COR)
                (OPS:    (      O0 = I0 OR I1)))
        ((NOR)
                (COMMUTATIVE)
                (INPUTS:        I0, I1)
                (OUTPUTS:       O0)
                (CONTROL:       CNOR)
                (OPS:    (      O0 = I0 NOR I1)))
        ((XOR)
                (COMMUTATIVE)
                (INPUTS:        I0, I1)
                (OUTPUTS:       O0)
                (CONTROL:       CXOR)
                (OPS:    (      O0 = I0 XOR I1)))
```

```
          ((XNOR)
                    (COMMUTATIVE)
                    (INPUTS:           I0, I1)
                    (OUTPUTS:          O0)
                    (CONTROL:          CXNOR)
                    (OPS:     (        O0 = I0 XNOR I1)))
          ((LID)
                    (INPUTS:           I0)
                    (OUTPUTS:          O0)
                    (CONTROL:          CLID)
                    (OPS:     (        O0 = I0 )))
          ((RID)
                    (INPUTS:           I1)
                    (OUTPUTS:          O0)
                    (CONTROL:          CRID)
                    (OPS:     (        O0 = I1 )))
          ((LNOT)
                    (INPUTS:           I0)
                    (OUTPUTS:          O0)
                    (CONTROL:          CLNOT)
                    (OPS:     (        O0 = NOT I0 )))
          ((RNOT)
                    (INPUTS:           I1)
                    (OUTPUTS:          O0)
                    (CONTROL:          CRNOT)
                    (OPS:     (        O0 = NOT I1 )))
          ((RINHI)
                    (INPUTS:           I0, I1)
                    (OUTPUTS:          O0)
                    (CONTROL:          CRINHI)
                    (OPS:     (        O0 = I0 RINHI I1)))
          ((LINHI)
                    (INPUTS:           I0, I1)
                    (OUTPUTS:          O0)
                    (CONTROL:          CLINHI)
                    (OPS:     (        O0 = I0 LINHI I1)))
          ((RIMPL)
                    (INPUTS:           I0, I1)
                    (OUTPUTS:          O0)
                    (CONTROL:          CRIMPL)
                    (OPS:     (        O0 = I0 RIMPL I1)))
          ((LIMPL)
                    (INPUTS:           I0, I1)
                    (OUTPUTS:          O0)
                    (CONTROL:          CLIMPL)
                    (OPS:     (        O0 = I0 LIMPL I1)))
OP_CLASSES:         default


                    /* Mux */

NAME:               MUX
CLASS:              Combinatorial
MAX_PARAMS:         4
PARAMETERS:         GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_INPUTS (%n),
                    GC_INVERT_FLAG
NUM_INPUTS:         %n
INPUTS:             &get_component_pin_name_list(MUX, INPUT, %n, %w)
NUM_OUTPUTS:        1
OUTPUTS:            O0[%w]
NUM_CONTROL:        %n
CONTROL:            &get_component_pin_name_list(MUX, CONTROL, %n, 1)
NUM_OPERATIONS:           %n
```

```
OPERATIONS:
          macro_expand ($i = 0 to %n-1)
          {
     ( (&get_component_function(MUX, $i))
                    (INPUTS:        &get_component_pin_name(MUX, INPUT, $i))
                    (OUTPUTS:       O0)
                    (CONTROL:       &get_component_pin_name(MUX, CONTROL, $i))
                    (OPS:     (     O0 = &get_component_pin_name(MUX, INPUT, $i))))
          }
OP_CLASSES:         default


                    /* Selector (Select on guard value) */

  /* Note: the select line(ISEL) is treated as an input since it is > 1 bit wide */

NAME:               SELECTOR
CLASS:              Combinatorial
MAX_PARAMS:         6
PARAMETERS:         GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
                    GC_NUM_GUARDS (%n), GC_GUARD_LIST,
                    GC_ELSE_FLAG, GC_CONTROL_WIDTH (%cw)
NUM_INPUTS:         %n + 1
INPUTS:             ISEL[%cw], &get_component_pin_name_list(SELECTOR, INPUT, %n, %w)
NUM_OUTPUTS:        1
OUTPUTS:            O0[%w]
NUM_OPERATIONS:           1
OPERATIONS:
         ((SELECT)
                    (INPUTS:        ISEL, &get_component_pin_name(SELECTOR, INPUT, $i))
                    (OUTPUTS:       O0)
                    (OPS:
            macro_expand ($i = 0 to %n-1)
            {
                    if (&get_component_pin_name(SELECTOR, INPUT,, $i)
                                                  == &get_selector_guard_val($i))
                              O0 = &get_component_pin_name(SELECTOR, INPUT,, $i);
            }
         ))
OP_CLASSES:         default


                    /* Decoder */

NAME:               DECODE
CLASS:              Combinatorial
MAX_PARAMS:         5
PARAMETERS:         GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_OUTPUTS (%n),
                    GC_DECODER_TYPE, GC_ELSE_FLAG
NUM_INPUTS:         1
INPUTS:             I0[%w]
NUM_OUTPUTS:        %n
OUTPUTS:            &get_component_pin_name_list(DECODER, OUTPUT, %n, 1)
NUM_OPERATIONS:           1
OPERATIONS:
         ((DECODE)
            (INPUTS:        I0         )
            (OUTPUTS:       &get_component_pin_name_list(DECODER, OUTPUT, %m,1))
            (OPS:
            macro_expand ($i = 0 to %n-1)
            {
                   if ($i == I0)
                      &get_component_pin_name(DECODER, OUTPUT, $i) = 1;
```

```
                else
                   &get_component_pin_name(DECODER, OUTPUT,$i) = 0;
          }
        ))
OP_CLASSES:     default


                /* Encoder */

NAME:           ENCODE
CLASS:          Combinatorial
MAX_PARAMS:     5
PARAMETERS:     GC_COMPILER_NAME, GC_NUM_INPUTS (%n), GC_NUM_OUTPUTS (%m),
                GC_ENCODER_TYPE, GC_ELSE_FLAG
NUM_INPUTS:     1
INPUTS:         &get_component_pin_name_list(ENCODER, INPUT, %n, 1)
NUM_OUTPUTS:    %n
OUTPUTS:        &get_component_pin_name_list(ENCODER, OUTPUT, %m, 1)
NUM_OPERATIONS:        1
OPERATIONS:
        ((ENCODE)
           (INPUTS:       &get_component_pin_name_list(ENCODER, INPUT, %n, 1))
           (OUTPUTS:      &get_component_pin_name_list(ENCODER, OUTPUT, %m,1))
           (OPS: (ENCODE))
        )


                /* Comparator */

NAME:           COMPARATOR
CLASS:          Combinatorial
MAX_PARAMS:     4
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
                GC_NUM_FUNCTIONS, GC_FUNCTION_LIST
NUM_INPUTS:     2
INPUTS:         I0[%w], I1[%w]
NUM_OUTPUTS:    6
OUTPUTS:        OEQ[1], ONEQ[1], OGT[1], OLT[1], OGEQ[1], OLEQ[1]
NUM_CONTROL:    5
CONTROL:        CEQ, CNEQ, CGT, CLT, CGEQ, CLEQ
NUM_OPERATIONS:        5
OPERATIONS:
        ((EQ)
                (COMMUTATIVE)
                (INPUTS:        I0, I1)
                (OUTPUTS:       OEQ)
                (CONTROL:       CEQ)
                (OPS:    (if (I0 == I1)
                                OEQ = 1; )))
        ((NEQ)
                (COMMUTATIVE)
                (INPUTS:        I0, I1)
                (OUTPUTS:       OEQ)
                (CONTROL:       CNEQ)
                (OPS:    (if (I0 == I1)
                                ONEQ = 1; )))
        ((GT)
                (INPUTS:        I0, I1)
                (OUTPUTS:       OGT)
                (CONTROL:       CGT)
                (OPS:    (if (I0 > I1)
                                OGT = 1; )))
        ((LT)
                (INPUTS:        I0, I1)
                (OUTPUTS:       OLT)
                (CONTROL:       CLT)
                (OPS:    (if (I0 < I1)
                                OLT = 1; )))
        ((GEQ)
                (INPUTS:        I0, I1)
                (OUTPUTS:       OGEQ)
                (CONTROL:       CGEQ)
                (OPS:    (if (I0 >= I1)
                                OGEQ = 1; )))
        ((LEQ)
                (INPUTS:        I0, I1)
                (OUTPUTS:       OLEQ)
                (CONTROL:       CLEQ)
                (OPS:    (if (I0 <= I1)
                                OLEQ = 1; )))
OP_CLASSES:     (EQ, NEQ, GT, LT, GEQ, LEQ)
```

/* Shifter (fixed distance) */

```
NAME:           SHIFTER
CLASS:          Combinatorial
MAX_PARAMS:     7
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
                GC_NUM_FUNCTIONS, GC_FUNCTION_LIST,
                GC_SHIFT_DISTANCE, GC_SHIFT_MODE, GC_FILL_INPUT
NUM_INPUTS:     3
INPUTS:         I0[%w], ILIN[1], IRIN[1]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
NUM_CONTROL:    8
CONTROL:        CSHR0, CHSR1, CSHL0, CSHL1, CROTR, CROTL, CASHL, CASHR
NUM_OPERATIONS:         8
OPERATIONS:
        ((SHL0)  (INPUTS: I0) (OUTPUTS: O0) (CONTROL: CSHL0)
                 (OPS: /* code modeling SHL0 */))
        ((SHR0)  (INPUTS: I0) (OUTPUTS: O0) (CONTROL: CSHR0)
                 (OPS: /* code modeling SHR0 */))
        ((SHL1)  (INPUTS: I0) (OUTPUTS: O0) (CONTROL: CSHL1)
                 (OPS: /* code modeling SHL1 */))
        ((SHR1)  (INPUTS: I0) (OUTPUTS: O0) (CONTROL: CSHR1)
                 (OPS: /* code modeling SHR1 */))
        ((ROTR)  (INPUTS: I0) (OUTPUTS: O0) (CONTROL: CROTR)
                 (OPS: /* code modeling ROTR */))
        ((ROTL)  (INPUTS: I0) (OUTPUTS: O0) (CONTROL: CROTL)
                 (OPS: /* code modeling ROTL */))
        ((ASHL)  (INPUTS: I0, IRIN) (OUTPUTS: O0) (CONTROL: CASHL)
                 (OPS: /* code modeling ASHL */))
        ((ASHR)  (INPUTS: ILIN, I0) (OUTPUTS: O0) (CONTROL: CASHR)
                 (OPS: /* code modeling ASHR */))
OP_CLASSES:     default
```

/* Barrel Shifter */

```
NAME:           BARREL_SHIFTER
CLASS:          Combinatorial
MAX_PARAMS:     5
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
                GC_NUM_FUNCTIONS, GC_FUNCTION_LIST,
                GC_SHIFT_DISTANCE
```

```
NUM_INPUTS:      6
INPUTS:          I0[%w], ISHNUM[%w], ILR[1], IROT[1], IFILL[1], IMODE[1]
NUM_OUTPUTS:  1
OUTPUTS:         O0[%w]
NUM_CONTROL:  6
CONTROL:         CSHR, CSHL, CROTR, CROTL, ASHR, ASHL
NUM_OPERATIONS:       6
OPERATIONS:
          ((SHL)   (INPUTS: I0, ISHNUM, ILR, IROT, IFILL, IMODE) (OUTPUTS: O0) (CONTROL: CSHL)
                   (OPS: /* code modeling SHL */))
          ((SHR)   (INPUTS: I0, ISHNUM, ILR, IROT, IFILL, IMODE) (OUTPUTS: O0) (CONTROL: CSHR)
                   (OPS: /* code modeling SHR */))
          ((ROTR)  (INPUTS: I0, ISHNUM, ILR, IROT, IFILL, IMODE) (OUTPUTS: O0) (CONTROL: CROTR)
                   (OPS: /* code modeling ROTR */))
          ((ROTL)  (INPUTS: I0, ISHNUM, ILR, IROT, IFILL, IMODE) (OUTPUTS: O0) (CONTROL: CROTL)
                   (OPS: /* code modeling ROTL */))
          ((ASHL)  (INPUTS: I0, ISHNUM, ILR, IROT, IFILL, IMODE) (OUTPUTS: O0) (CONTROL: CASHL)
                   (OPS: /* code modeling ASHL */))
          ((ASHR)  (INPUTS: I0, ISHNUM, ILR, IROT, IFILL, IMODE) (OUTPUTS: O0) (CONTROL: CASHR)
                   (OPS: /* code modeling ASHR */))
```

/* Adder/Subtractor */

```
NAME:            ADD_SUB
CLASS:           Combinatorial
MAX_PARAMS:      5
PARAMETERS:      GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_FUNCTIONS,
                 GC_FUNCTION_LIST, GC_STYLE
NUM_STYLES:      2
STYLES:          RIPPLE_CARRY, CLA
NUM_INPUTS:      2
INPUTS:          I0[%w], I1[%w], ICIN[1]
NUM_OUTPUTS:  2
OUTPUTS:         O0[%w], OCOUT[1]
NUM_CONTROL:  2
CONTROL:         CADD, CSUB
NUM_OPERATIONS:       2
OPERATIONS:
          ((ADD)
                 (COMMUTATIVE)
                 (INPUTS:          I0, I1, CIN)
                 (OUTPUTS:         O0, COUT)
                 (CONTROL:         CADD)
                 (OPS:     (       O0 = I0 + I1 + CIN;
                                   COUT = $carry_out(O0 = I0 + I1 + CIN);
                        )))
          ((SUB)
                 (INPUTS:          I0, I1, CIN)
                 (OUTPUTS:         O0, COUT)
                 (CONTROL:         CSUB)
                 (OPS:     (       O0 = I0 - I1 + CIN;
                                   COUT = $borrow_in(O0 = I0 - I1 + CIN);
                        )))
OP_CLASSES:      default
```

/* ALU */

```
NAME:            ALU
CLASS:           Combinatorial
MAX_PARAMS:      5
PARAMETERS:      GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_FUNCTIONS,
                GC_FUNCTION_LIST, GC_STYLE
NUM_INPUTS:     2
INPUTS:         I0[%w], I1[%w]
NUM_OUTPUTS:    5
OUTPUTS:        O0[%w], OCOUT[1], OZERO[1], OSIGN[1], OVFLOW[1]
NUM_CONTROL:    25
CONTROL:        CADD, CSUB, CINC, CDEC, CEQ, CGT, CLT, CEQ, CNEQ, CZERO,
                CONE, CAND, CNAND, COR, CNOR, CXOR, CXNOR,
                CLID, CRID, CLNOT, CRNOT, CRINHI, CLINHI, CLIMPL, CRIMPL
NUM_OPERATIONS:         25
OPERATIONS:
        ((ADD)
                (COMMUTATIVE)
                (INPUTS:        I0, I1)
                (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:       CADD)
                (OPS:    (      O0 = I0 + I1;
                                /* code for other outputs */
                         )))
        ((SUB)
                (INPUTS:        I0, I1)
                (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:       CSUB)
                (OPS:    (      O0 = I0 - I1;
                                /* code for other outputs */
                         )))
        ((INC)
                (INPUTS:        I0)
                (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:       CINC)
                (OPS:    (      O0 = I0 + 1;
                                /* code for other outputs */
                         )))
        ((DEC)
                (INPUTS:        I0)
                (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:       CDEC)
                (OPS:    (      O0 = I0 - 1;
                                /* code for other outputs */
                         )))
        ((EQ)
                (COMMUTATIVE)
                (INPUTS:        I0, I1)
                (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:       CEQ)
                (OPS:    (if (I0 == I1)
                                O0 = 1; )))
        ((NEQ)
                (COMMUTATIVE)
                (INPUTS:        I0, I1)
                (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:       CNEQ)
                (OPS:    (if (I0 == I1)
                                O0 = 1; )))
        ((GT)
                (INPUTS:        I0, I1)
                (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:       CGT)
                (OPS:    (if (I0 > I1)
                                O0 = 1; )))
        ((LT)
                (INPUTS:        I0, I1)
                (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
```

```
            (CONTROL:       CLT)
            (OPS:    (if (I0 < I1)
                            O0 = 1; )))
    ((GEQ)
            (INPUTS:        I0, I1)
            (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
            (CONTROL:       CGEQ)
            (OPS:    (if (I0 > = I1)
                            O0 = 1; )))
    ((LEQ)
            (INPUTS:        I0, I1)
            (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
            (CONTROL:       CLEQ)
            (OPS:    (if (I0 < = I1)
                            O0 = 1; )))
    ((ZERO)
            (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
            (CONTROL:       CZERO)
            (OPS:    (      O0 = 0)))
    ((ONE)
            (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
            (CONTROL:       CONE)
            (OPS:    (      O0 = 1)))
    ((AND)
            (COMMUTATIVE)
            (INPUTS:        I0, I1)
            (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
            (CONTROL:       CAND)
            (OPS:    (      O0 = I0 AND I1)))
    ((NAND)
            (COMMUTATIVE)
            (INPUTS:        I0, I1)
            (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
            (CONTROL:       CNAND)
            (OPS:    (      O0 = I0 NAND I1)))
    ((OR)
            (COMMUTATIVE)
            (INPUTS:        I0, I1)
            (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
            (CONTROL:       COR)
            (OPS:    (      O0 = I0 OR I1)))
    ((NOR)
            (COMMUTATIVE)
            (INPUTS:        I0, I1)
            (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
            (CONTROL:       CNOR)
            (OPS:    (      O0 = I0 NOR I1)))
    ((XOR)
            (COMMUTATIVE)
            (INPUTS:        I0, I1)
            (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
            (CONTROL:       CXOR)
            (OPS:    (      O0 = I0 XOR I1)))
    ((XNOR)
            (COMMUTATIVE)
            (INPUTS:        I0, I1)
            (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
            (CONTROL:       CXNOR)
            (OPS:    (      O0 = I0 XNOR I1)))
    ((LID)
            (INPUTS:        I0)
            (OUTPUTS:       O0, OCOUT, OZERO, OSIGN, OVFLOW)
            (CONTROL:       CLID)
```

```
                (OPS:     (           O0 = I0 )))
        ((RID)
                (INPUTS:              I1)
                (OUTPUTS:             O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:             CRID)
                (OPS:     (           O0 = I1 )))
        ((LNOT)
                (INPUTS:              I0)
                (OUTPUTS:             O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:             CLNOT)
                (OPS:     (           O0 = NOT I0 )))
        ((RNOT)
                (INPUTS:              I1)
                (OUTPUTS:             O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:             CRNOT)
                (OPS:     (           O0 = NOT I1 )))
        ((RINHI)
                (INPUTS:              I0, I1)
                (OUTPUTS:             O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:             CRINHI)
                (OPS:     (           O0 = I0 RINHI I1)))
        ((LINHI)
                (INPUTS:              I0, I1)
                (OUTPUTS:             O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:             CLINHI)
                (OPS:     (           O0 = I0 LINHI I1)))
        ((RIMPL)
                (INPUTS:              I0, I1)
                (OUTPUTS:             O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:             CRIMPL)
                (OPS:     (           O0 = I0 RIMPL I1)))
        ((LIMPL)
                (INPUTS:              I0, I1)
                (OUTPUTS:             O0, OCOUT, OZERO, OSIGN, OVFLOW)
                (CONTROL:             CLIMPL)

OP_CLASSES:     default               /* need to have separate control lines for each output ? */


                /* Multiplier */

NAME:           MULT
CLASS:          Combinatorial
MAX_PARAMS:     3
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
                GC_STYLE
NUM_STYLES:     3
STYLES:         ARRAY, WALLACE, ITERATIVE
NUM_INPUTS:     2
INPUTS:         I0[%w], I1[%w]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
NUM_OPERATIONS:         1
OPERATIONS:
        ((MULT)
                (COMMUTATIVE)
                (INPUTS:              I0, I1)
                (OUTPUTS:             O0)
                (OPS:     (           O0 = I0 * I1;)))
OP_CLASSES:     default


                /* Divider */
```

```
NAME:          DIV
CLASS:         Combinatorial
MAX_PARAMS:    3
PARAMETERS:    GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
               GC_STYLE
NUM_STYLES:    3
STYLES:        RESTORING, NON_RESTORING, MULTIPLICATIVE
NUM_INPUTS:    2
INPUTS:        I0[%w], I1[%w]
NUM_OUTPUTS:   1
OUTPUTS:       O0[%w]
NUM_OPERATIONS:       1
OPERATIONS:
        ((DIV)
               (INPUTS:        I0, I1)
               (OUTPUTS:       O0)
               (OPS:    (      O0 = I0 / I1;)))
OP_CLASSES:    default
```

/* SEQUENTIAL COMPONENTS */

```
NAME:          REGISTER
CLASS:         Clocked
MAX_PARAMS:    9
PARAMETERS:    GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_FUNCTIONS,
               GC_FUNCTION_LIST, GC_REGISTER_TYPE,
               GC_REGISTER_POS_OUT, GC_REGISTER_INVERT_OUT,
               GC_SET_VALUE, GC_ENABLE_FLAG
NUM_INPUTS:    3
INPUTS:        I0[%w], ILIN[1], IRIN[1]
NUM_OUTPUTS:   1
OUTPUTS:       O0[%w]
CLOCK:         CLK
NUM_ENABLE:    1
ENABLE:               CEN
NUM_CONTROL:   3
CONTROL:       CLOAD, CSHL, CSHR
NUM_ASYNC:     2
ASYNC:         ASET, ARESET
NUM_OPERATIONS:       3
OPERATIONS:
        ((LOAD)
               (INPUTS:        I0)
               (OUTPUTS:       O0)
               (CONTROL:       CLOAD)
               (OPS:    (      O0 = I0;)))
        ((SHL)
               (INPUTS:        I0, IRIN)
               (OUTPUTS:       O0)
               (CONTROL:       CSHL)
               (OPS:    (      O0 = O0 << CLIN)))
        ((SHR)
               (INPUTS:        ILIN, I0)
               (OUTPUTS:       O0)
               (CONTROL:       CSHR)
               (OPS:    (      O0 = CRIN >> O0)))
OP_CLASSES:    default

NAME:          COUNTER
CLASS:         Clocked
MAX_PARAMS:    8
PARAMETERS:    GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_FUNCTIONS,
                GC_FUNCTION_LIST, GC_COUNTER_TYPE, GC_STYLE,
                GC_SET_VALUE, GC_ENABLE_FLAG
NUM_STYLES:     2
STYLES:         SYNCHRONOUS, RIPPLE
NUM_INPUTS:     1
INPUTS:         I0[%w]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
CLOCK:          CLK
NUM_ENABLE:     1
ENABLE:                 CEN
NUM_CONTROL:    3
CONTROL:        CLOAD, CUP, CDOWN
NUM_ASYNC:      2
ASYNC:          ASET, ARESET
NUM_OPERATIONS:         3
OPERATIONS:
        ((LOAD)
                (INPUTS:        I0)
                (OUTPUTS:       O0)
                (CONTROL:       CLOAD)
                (OPS:     (     O0 = I0)))
        ((COUNT_UP)
                (OUTPUTS:       O0)
                (CONTROL:       CUP)
                (OPS:     (     O0 = O0 + 1)))
        ((COUNT_DOWN)
                (OUTPUTS:       O0)
                (CONTROL:       CDOWN)
                (OPS:     (     O0 = O0 - 1)))
OP_CLASSES:     default


                /* Register File */

NAME:           REG_FILE
CLASS:          Clocked
MAX_PARAMS:     6
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_NUM_WORDS (%n)
                GC_NUM_PORTS (%p), GC_PORT_ATTRIBUTE_LIST, GC_ENABLE_FLAG
NUM_INPUTS:     2 * %p
INPUTS:         &get_component_pin_name_list(REG_FILE, INPUT, %p, %w)
NUM_OUTPUTS:    %p
OUTPUTS:        &get_component_pin_name_list(REG_FILE, OUTPUTS, %p, %w)
CLOCK:          CLK
NUM_CONTROL:    2 * %p
CONTROL:        &get_component_pin_name_list(REG_FILE, CONTROL, %p, %w)
NUM_OPERATIONS:         %p
OPERATIONS:
        /* read and write for each reg_file port */
        macro_expand ($i = 0 to %p-1)
        {
        ( (&get_component_function(READ, $i))
                (INPUTS:        &get_component_pin_name(REG_FILE, INPUT, 2*$i + 1))
                (OUTPUTS:       &get_component_pin_name(REG_FILE, OUTPUT, $i))
                (CONTROL:       &get_component_pin_name(REG_FILE, CONTROL, 2*$i))
                (OPS:     /* code for read op */))
        ( (&get_component_function(WRITE, $i))
                (INPUTS:        &get_component_pin_name(REG_FILE, INPUT, 2*$i),
                                &get_component_pin_name(REG_FILE, INPUT, 2*$i + 1))
                (CONTROL:       &get_component_pin_name(REG_FILE, CONTROL, 2*$i + 1))
                (OPS:     /* code for write op */))
        }
```

```
OP_CLASSES:     default

                /* Stack */

NAME:           STACK
CLASS:          Clocked
MAX_PARAMS:     4
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
                GC_NUM_WORDS, GC_ENABLE_FLAG
NUM_INPUTS:     1
INPUTS:         I0[%w]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
CLOCK:          CLK
NUM_ENABLE:     1
ENABLE:                 CEN
NUM_CONTROL:    2
CONTROL:        CPUSH, CPOP
NUM_OPERATIONS:         2
OPERATIONS:
        ((PUSH)
                (INPUTS:        I0)
                (OUTPUTS:       O0)
                (CONTROL:       CPUSH)
                (OPS:    (      O0 = $push(I0))))
        ((POP)
                (INPUTS:        I0)
                (OUTPUTS:       O0)
                (CONTROL:       CPOP)
                (OPS:    (      O0 = $pop(I0))))
OP_CLASSES:     default

                /* FIFO */

NAME:           FIFO
CLASS:          Clocked
MAX_PARAMS:     4
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
                GC_NUM_WORDS, GC_ENABLE_FLAG
NUM_INPUTS:     1
INPUTS:         I0[%w]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
CLOCK:          CLK
NUM_ENABLE:     1
ENABLE:                 CEN
NUM_CONTROL:    2
CONTROL:        CPUSH, CPOP
NUM_OPERATIONS:         2
OPERATIONS:
        ((PUSH)
                (INPUTS:        I0)
                (OUTPUTS:       O0)
                (CONTROL:       CPUSH)
                (OPS:    (      O0 = $push(I0))))
        ((POP)
                (INPUTS:        I0)
                (OUTPUTS:       O0)
                (CONTROL:       CPOP)
                (OPS:    (      O0 = $pop(I0))))
OP_CLASSES:     default
```

```
                /* Memory */

NAME:           MEMORY
CLASS:          Clocked
MAX_PARAMS:     6
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
                GC_NUM_WORDS (%n), GC_ENABLE_FLAG
NUM_INPUTS:     3
INPUTS:         I0[%w], IA0[$log(%n)], IA_VALID[1]
NUM_OUTPUTS:    2
OUTPUTS:        O0[%w], OD_READY[1]
CLOCK:          CLK
NUM_CONTROL:    2
CONTROL:        CREAD, CWRITE
NUM_OPERATIONS:      2
OPERATIONS:
                /* read and write operations for the memory */
OP_CLASSES:     default


                /* Interface and Miscellaneous Components */


                /* Interface Unit */

NAME:           INTERFACE
CLASS:          Combinatorial
MAX_PARAMS:     7
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
                GC_INTERFACE_FUNCTION, GC_INTERFACE_MODE,
                GC_INTERFACE_LEVEL, GC_INTERFACE_DRIVE, GC_INVERT_FLAG
NUM_INPUTS:     1
INPUTS:         I0[%w]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]


                /* Port */

NAME:           PORT
CLASS:          Combinatorial
MAX_PARAMS:     3
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w), GC_PORT_MODE
NUM_INPUTS:     1
INPUTS:         I0[%w]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]


                /* Bus */

NAME:           BUS
CLASS:          Combinatorial
MAX_PARAMS:     4
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
                GC_NUM_INPUTS (%n), GC_FAN_OUT
NUM_INPUTS:     %n
INPUTS:         $get_component_pin_name_list(BUS, INPUT, %n, %w)
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
NUM_CONTROL:    %n
CONTROL:        $get_component_pin_name_list(BUS, CONTROL, %n, %w)
NUM_OPERATIONS:      %n + 1
```

```
OPERATIONS:     /* ONE READ AND %n WRITE OPS */


                /* Wired-Or */

NAME:           WIRED_OR
CLASS:          Combinatorial
MAX_PARAMS:     4
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH (%w),
                GC_NUM_INPUTS (%n), GC_FAN_OUT
NUM_INPUTS:     1
INPUTS:         I0[%w]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]


                /* Concat Switchbox */

NAME:           CONCAT
CLASS:          Combinatorial
MAX_PARAMS:     4
PARAMETERS:     GC_COMPILER_NAME, GC_NUM_INPUTS (%n),
                GC_INPUT_WIDTH_LIST, GC_OUTPUT_WIDTH (%o)
NUM_INPUTS:     %n
INPUTS:         $get_component_pin_name_list(CONCAT, INPUT, %n, 0)
NUM_OUTPUTS:    1
OUTPUTS:        O0[%o]
NUM_OPERATIONS:        1
OPERATIONS:     ((CONCAT))


                /* Extract Switchbox */

NAME:           EXTRACT
CLASS:          Combinatorial
MAX_PARAMS:     4
PARAMETERS:     GC_COMPILER_NAME, GC_INPUT_WIDTH(%w),
                GC_LEFT_INDEX(%l), GC_RIGHT_INDEX (%r)
NUM_INPUTS:     1
INPUTS:         I0[%w]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%r - %l + 1]
NUM_OPERATIONS:        1
OPERATIONS:     ((EXTRACT))     /* left index to right index */


                /* Clock Generator */

NAME:           CLOCK_GEN
CLASS:          Combinatorial
MAX_PARAMS:     3
PARAMETERS:     GC_COMPILER_NAME, GC_CLOCK_PERIOD, GC_CLOCK_HIGH
NUM_OUTPUTS:    1
OUTPUTS:        O0[1]
OPERATIONS:     ((EXTRACT))     /* left index to right index */


                /* Delay Element */

NAME:           DELAY
CLASS:          Combinatorial
MAX_PARAMS:     3
PARAMETERS:     GC_COMPILER_NAME, GC_DELAY_VALUE, GC_INPUT_WIDTH (%w)
```

```
NUM_INPUTS:     1
INPUTS:         I0[%w]
NUM_OUTPUTS:    1
OUTPUTS:        O0[%w]
NUM_OPERATIONS:         DELAY
OPERATIONS:     ((DELAY))
```