LEGEND : A LANGUAGE FOR GENERIC COMPONENT LIBRARY DESCRIPTION

BY

NIKIL D. DUTT

Technical Report 89-29

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

Keywords

Hardware Description Languages, High Level Synthesis, Generator-Generator Languages, Design Component Databases.

Abstract

This paper describes a novel generator-generator language, LEGEND, for the definition, generation and maintenance of generic component libraries used in high level synthesis. Each LEGEND description generates a library generator GENUS, which is organized as a hierarchy of generic component generators, component templates, and component instances. Component instances from the GENUS library are used by high level synthesis systems to transform the abstract behavior of a design into an interconnection of generic components satisfying this behavior. Although existing hardware description languages such as VHDL are very good for describing component libraries, they lack the capability of *generating* these component libraries from a higher-level description. LEGEND complements a language such as VHDL by providing a component library generator-generator with behavioral models for simulation and subsequent synthesis. The components in a LEGEND generated library have realistic register transfer semantics, including clocking, asynchrony and data bi-directionality. LEGEND is extensible, since its simple syntax allows users to add new component types or modify existing component types easily. The LEGEND generator-generator is currently implemented on SUN3's under C/UNIX and is used by a suite of behavioral synthesis tools at U. C. Irvine.

LEGEND

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

1. Introduction

The task of high level synthesis involves the mapping of abstract behavioral design descriptions into structural designs composed of components drawn from a generic component library. The synthesized structural design must functionally implement the abstract behavior under the set of high-level constraints given by the user. Once a feasible structural design of generic components is synthesized, it is passed on to a set of logic and layout synthesizers to implement the design in a particular target technology (3-micron CMOS, for instance). With the the rapid advances in fabrication and layout technologies, it becomes increasingly important to insulate lower-level technological changes from higher-level design decisions, since these technology-specific designs become obsolete with even small changes in the technology. This creates a huge bottleneck in the design cycle, since the whole design process has to be restarted for every small change in the technology. A key to solving this design crisis in VLSI systems is *technology independence*: the concept of keeping higher level design descriptions and decisions independent of the target technology. High-level synthesis systems use components drawn from a *generic component library* to effect this technology independence; structural designs composed of components drawn from a generic library can be re-targeted to different technologies at the backend, without having to redo the task of high-level synthesis.

This paper describes a novel generator-generator language, LEGEND, for the definition, generation and maintenance of generic component libraries used in high level synthesis. LEGEND's simple syntax and strong register-transfer semantics, coupled with its extensibility, makes it a powerful language for facilitating efficient high-level synthesis.

Each generic component from a generated library is instantiated by specifying its parameters which define its structural, operational and performance 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 LEGEND generator is a template for a generic microarchitectural component; depending on the design requirements, components may be built from these templates by supplying the necessary parameters.

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

- generic components are functionality generators.

- they provide a truly "generic" view of structural elements; this makes the task of behavior-to-structure synthesis independent of technological details.

- 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; a 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.

This paper is organized as follows. Section 2 describes previous work on hardware description languages, component generators and libraries. Section 3 briefly introduces the LEGEND language, and the semantics of the generated generic component library. Section 4 illustrates how generic components and their instances are created and used. Section 5 uses a simple example to show how components derived from LEGEND are simulated in VHDL. Section 6 concludes with the status of this research.

2. Previous Work

2.1. Hardware Description Languages

Although a number of good hardware description languages have been described in the literature (DDL [DuDi67], AHPL [HiNa79], ISPS [Barb81], etc.), these have been used primarily for behavioral specification and synthesis; none of them have addressed the issue of how to describe, generate and maintain generic component libraries.

More recently, VHDL [VHDL87] was proposed as a "standard" hardware description language for the specification and maintenance of design descriptions transcending several design levels including behavior, data-flow and micro-architectural structure. Although VHDL has good constructs for describing *specific* libraries and component instances, it does not have the capability of *generating* customized component libraries. In a high-level synthesis environment, what is lacking is a generator for VHDL component libraries. VHDL is also closely tied to a simulation model of computation, hence lacks several

hardware semantics at the register transfer level.

2.2. Generic Component Characterization

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 [McPC88]. 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 and structural ports) parameterized. A hierarchical representation, using the notion of of "types", "generators", "components" and "instances" introduced in this paper, 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 paper. 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.

2.3. Related Work

Similar work on hierarchical library generator-generators has been described at lower levels of the design process. At the layout level, DPL [BaHa80] provided an object-oriented hierarchical representation of layout (cell) objects. Palladio [BrTF83] describes another object-oriented representation to model designs across a number of design levels; however, it was never used in any synthesis environment. More recently, Fred [Wolf89] describes an interesting object-oriented approach for representing designs and constraints at the module and layout level. Fred uses the ETHEL language to describe a module's physical and layout characteristics. None of this work has examined the representation of generic component libraries for high level synthesis and generators for such libraries.

3. LEGEND: The Generator-Generator

LEGEND is a language used to generate a particular instance of a generic component library, **GENUS** [Dutt88], for use in a high-level synthesis system. To understand the syntax and semantics of LEGEND, it is necessary to understand the organization of a typical GENUS component library. This section will briefly outline the GENUS organization and its semantics, before describing the LEGEND language.

3.1. GENUS System Overview

Every high-level synthesis system uses an implicit or explicit generic component library. The abstract behavior of the design is implemented as a structural realization of interconnected component instances drawn from this generic library. GENUS is hierarchical generic component library used by several high-level synthesis tools at U.C. Irvine (VSS

[LiGa89], EXEL [DuGa89], MILO [VaGa88], etc.). 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 using that library.

3.1.1. GENUS 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.

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.

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

A *generator* class is used to generate a family of similar components and instances. LEGEND descriptions (described later in this section) are used to maintains lists of all the possible parameters, definitions for each operation performed by a generated component.

A *component* is generated by passing a list of parameters to the parent LEGEND generator descriptor. 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

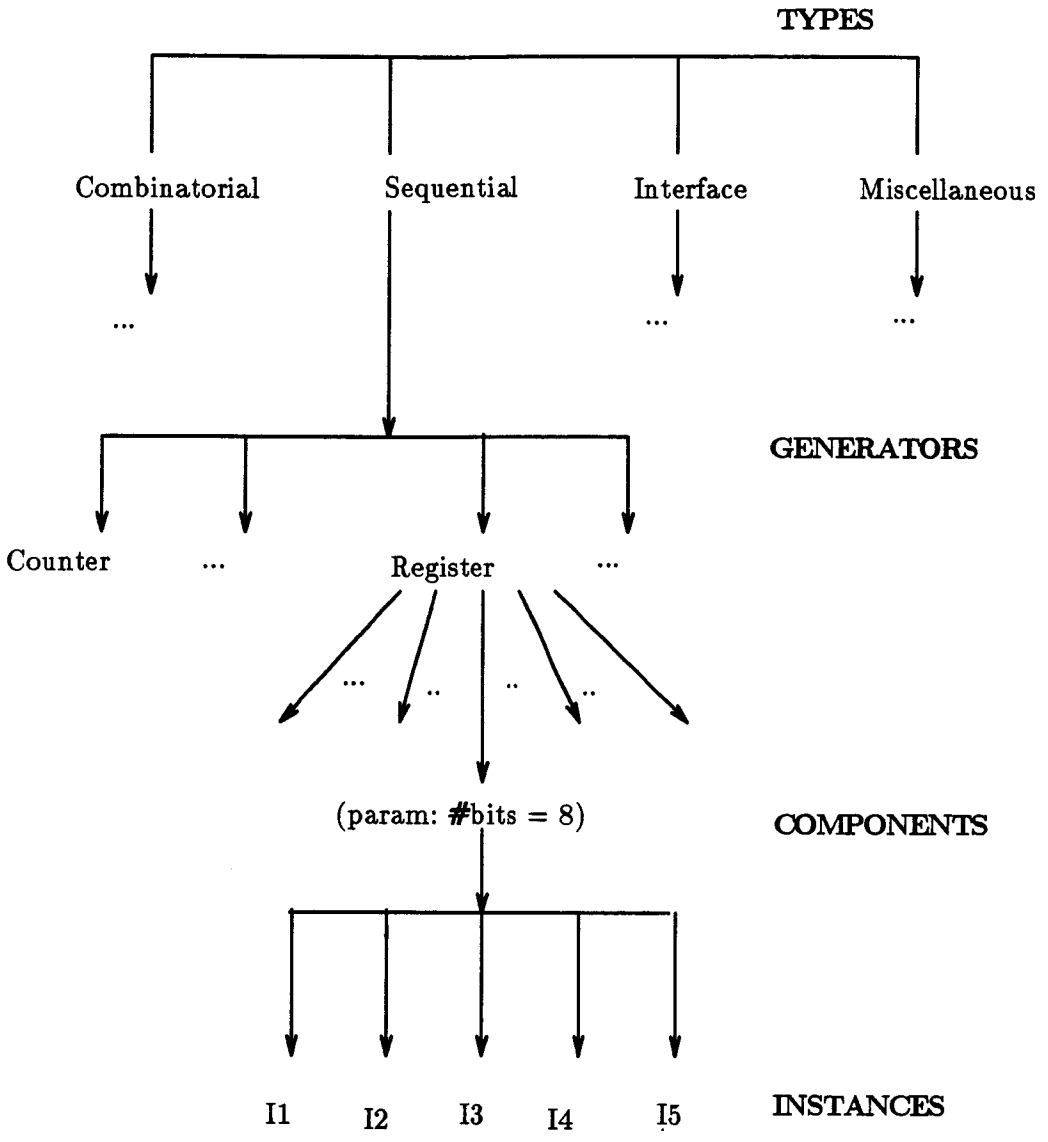

Figure 1. Hierarchy in GENUS

particular generator need not be specified; missing parameters are assigned default values.

Instances are "carbon-copies" of a generated component, with unique 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.

3.1.2. Using GENUS

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

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.

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.

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.

3.2. LEGEND Overview

A LEGEND description uses a special notation for describing individual generic component generators. Each generic component generator is characterized by a unique name and a list of attributes describing the type class, implementation styles, parameters, port information and functionality. The LEGEND description can be tailored to a particular generic component library by specifying the necessary component generator types. In addition, each component generator can produce simulatable VHDL behavioral models for the generated components; these models can be used to verify the behavior of a synthesized design. The LEGEND library description is parsed into the internal data structures used to represent the GENUS library. Appendix A contains the complete syntax of the LEGEND language; we will use Figure 2 to show the LEGEND definition for a counter generator, 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.

3.2.1. Name

This specifies a unique name for a generator.

```
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)))
VHDL_MODEL:     counter_vhdl.c
OP_CLASSES:     default
```

Figure 2. Sample LEGEND Description For a Counter Generator

3.2.2. Class

Specifies if the generator is of type class *clocked* or *combinational*. 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 0, 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 0 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 0 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.

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.

3.2.3. Parameters

The MAX_PARAMS and PARAMETERS entries indicate the number and global symbols used to describe the generic generator. For the counter in Figure 0, 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.

3.2.4. Styles

The STYLES entry indicates the list of possible implementation styles for generating instances of the component. For the counter in Figure 0, the implementation styles are SYNCHRONOUS and RIPPLE.

3.2.5. 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.

3.2.6. Operations

Each operation that can be performed by a generated component is described by its name, input, output and control port information.

3.2.7. VHDL_MODEL

The behavioral operation of a generated component is modeled in VHDL. This VHDL model is generated by the C routine indicated in this entry. The VHDL models are described further in section 5.

3.2.8. 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.

3.2.9. 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 3 shows a sample definition for a MUX component. 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 2, 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.

```
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))))
        }
VHDL_MODEL:     mux_vhdl.c
OP_CLASSES:     default
```

Figure 3. Macro-Expand Feature

3.2.10. Estimation Functions

The initial version of each generated GENUS generic component library use estimators derived from Chippe's model of function units [BrGa87]. Functions for area, speed and power return estimates based on the size, functionality and bit-width of a generated component.

4. LEGEND: Semantics and Usage

As mentioned earlier, LEGEND-generated components in the GENUS library belong one of several type classes, based on their properties and/or functions. This section describes the semantics, assumptions and naming conventions associated with these components. It then describes how components and instances can be accessed.

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

Sequential components are assumed to have a clock input; 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. This feature makes each component in a generated library truly generic, since the task of control encoding is left to a technology mapper at the time of circuit realization. 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 4 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.

4.5. Sequential Components

Figure 5 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

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 4. Combinatorial Components

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 5. Sequential Components

(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.

4.6. Interface and Miscellaneous Components

Figure 6 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 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

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 6. Interface and Miscellaneous Components

path.

4.7. 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 7 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,

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

Figure 7. Generic Component Access: General Form

component or instance). A set of standard query routines can be applied to the object to extract any attribute or characteristic for it. Figure 8 shows a sample call used to generate an instance of an ALU. The arguments in the call consist of pairs of reserved global 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 8, 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 9 and Figure 10 show the list of global symbols reserved for indicating the type of argument specified in a call, together with their possible values. Appendix A in [Dutt88] has a complete list of generator calls for all the generic components.

5. VHDL Models for LEGEND Descriptions

LEGEND generates VHDL models for specifying the behavior of generated components. LEGEND thus complements and overcomes a deficiency in VHDL by providing ai generator-generator language for VHDL component libraries. These VHDL models can be used for functional simulation of synthesized register-transfer designs, and can also be used for lower-level synthesis of individual components at the logic and gate levels.

```
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 8. Sample ALU Instance Call

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

A typical VHDL model generated for a 4-bit up/down counter is shown in Figure 11. These VHDL models are currently simulated on the Vantage VHDL simulator [Vant89].

6. Summary

This paper described the features of LEGEND, a novel language used to define, generate, maintain, and upgrade generic component libraries used in high level synthesis. LEGEND provides a powerful generator-generator environment with a consistent hierarchical organization of generic components and instances. LEGEND complements VHDL, a standard hardware description language, by providing a library generator facility. Each generated component has a simulatable VHDL model generated for it. The semantics of LEGEND model register-transfer behavior such as asynchrony and clocking realistically.

```
use work.defs.all;
entity counter is
port (In1 : in bit_vector(3 downto 0);
     CLK, Cen : in bit;
     CLOAD, CUP, CDOWN : in bit;
     Out1 : out bit_vector(3 downto 0);
     Aset, Areset : in bit
     );
end counter;

architecture counter_behavior of counter is
begin
    process
variable temp : bit_vector(3 downto 0);
        begin
            if (Aset or Areset) = '0' then
                if (CLK and Cen) = '1' then
                    if Cload = '1' then
                        temp := In1;
                    else
                    if Cup = '1' then
                        temp := inc(temp);
                    else
                    if Cdown = '1' then
                        temp := decr(temp);
                    end if;
                    end if;
                    end if;
                end if;
            else
                if Aset = '1' then
                    temp := "1111";
                else
                    if Areset = '1' then
                        temp := "0000";
                    end if;
                end if;
            end if;
            Out1 <= temp;
            wait on CLK, Aset, Areset;
        end process;
end counter_behavior;

configuration counter_config of counter is
    for counter_behavior
    end for;
end counter_config;
```

Figure 11. Generated VHDL Model for 4-Bit Up/Down Counter

The LEGEND generator-generator is implemented on SUN3's under C/UNIX, and is used by a suite of behavioral synthesis tools at U.C. Irvine. Future work will address the modeling of better estimators for generic components in the generated libraries.

7. Acknowledgements

I'd like to thank William Carrasco for helping me with the generation of VHDL models for components in the GENUS library.

8. References

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

[Barb81] M. R. Barbacci, "Instruction Set Processor Specification (ISPS)," *IEEE Transactions on Computers*, vol. c-30, no. 1, January 1981.

[BrGa87] Forrest D. Brewer, Daniel D. Gajski, "Knowledge Based Control in Micro-Architecture Design" *24th IEEE Design Automation Conference* Miami, Fl (July, 1987).

[DuDi68] J.R. Duley and D.L. Dietmeyer, "A Digital System Design Language (DDL)," *IEEE Trans. Computers*, Vol C-17, Sept. 1968.

[DuGa89] N.D. Dutt and D. Gajski, "EXEL: A Language for Interactive Behavioral Synthesis," *Proc. Ninth International Symposium on Computer Hardware Description Languages*, Washington D.C., June 1989.

[Dutt88] N. D. Dutt, "GENUS: A Generic Component Library for High Level Synthesis," *Tech Rep* 88-22, U.C. Irvine, Sept. 1988.

[HiNa79] F.J. Hill and Z. Navabi, "Extending Second Generation AHPL," *Proc. Fourth International Symposium on Hardware Description Languages*, Palo Alto, CA, Oct. 1979.

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

[McPC88] M.C. McFarland, A.C. Parker and R. Camposano, "Tutorial on High Level Synthesis," *25th Design Automation Conference*, July 1988.

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

[VHDL87] *VHDL Tutorial for IEEE Standard* 1076 *VHDL*, CAD Language Systems Inc., June 1987.

[Vant89] *Vantage VHDL Simulator*, Vantage Analysis Systems Inc., 1989.

[Wolf89] Wayne Wolf, "How to Build a Hardware Description and Measurement System on an Object-Oriented Programming Langauge," *IEEE Trans. Computer Aided-Design*, Vol. 8, No. 3, March 1989.

9. APPENDIX A: LEGEND SYNTAX

LEGEND SYNTAX

```
legend_description :    list_of_generators
                        ;

list_of_generators :    list_of_generators generator_spec
                        |       list_of_generators
                        ;

generator_spec          :       gen_name
                                gen_class
                                gen_style_list
                                gen_param_list
                                gen_port_def_list
                                gen_op_list
                                gen_vhdl
                                gen_op_classes
                        ;

gen_name        :       TNAME COLON identifier
                        ;

gen_class       :       TCLASS COLON class_type
                        ;

class_type              :       TCLOCKED
                        |       TCOMBINATORIAL
                        ;

gen_style_list          :       TSTYLE COLON style_list
                        |       empty
                        ;

style_list      :       style_list comma gen_style
                        |       gen_style
                        ;

gen_style       :       identifier
                        ;

gen_param_list          :       TPARAM COLON param_list
                        |       empty
                        ;

param_list              :       param_list comma  param_item
                        |       param_item
                        ;

param_item              :       identifier
                        ;

gen_port_def_list :     gen_port_def_list gen_port_def
                        |       gen_port_def
                        ;

gen_port_def            :       gen_port_type COLON gen_port_list
                        ;

gen_port_type           :       TINPUTS
```

```
                        |       TCLOCK
                        |       TENABLE
                        |       TCONTROL
                        |       TASYNC
                        |       TOUTPUTS
                        |       TINPUT_OUTPUTS
                        ;

gen_port_list           :       gen_port_list comma gen_port_name
                        |       gen_port_name
                        ;

gen_port_name           :       identifier
                        ;

gen_op_list             :       TOPS COLON op_list
                        ;

op_list                 :       op_list   op_list_item
                        |       op_list_item
                        ;

op_list_item            :       gen_op_name LPAREN op_port_map_list commutative_indicator RPAREN
                        ;

gen_op_name             :       gen_op
                        ;

commutative_indicator   :       semicolon TCOMMUTATIVE
                        |       empty
                        ;
                        ;

op_port_map_list  :     op_port_map_list semicolon op_port_map
                        |       op_port_map
                        ;

op_port_map             :       op_port_type COLON op_map_list
                        ;


op_port_type            :       TINPUTS
                        |       TCONTROL
                        |       TASYNC
                        |       TENABLE
                        |       TOUTPUTS
                        |       TINPUT_OUTPUTS
                        |       TCLOCK
                        ;

op_map_list             :       op_map_list comma op_map
                        |       op_map
                        ;

op_map                  :       DIGSEQ

                        ;

gen_op                  :       concatop
                        |       genlop
                        |       mulop
                        |       addop
                        |       shiftop_unary
```

```
                        |       shiftop_binary
                        |       relop
                        |       relop_two
                        |       band
                        |       bxor
                        |       bor
                        |       and
                        |       xor
                        |       or
                        |       not
                        |       TINC
                        |       TDEC
                        |       TCLEAR
                        |       TSET
                        |       TUSHL
                        |       CASE
                        |       IF
                        |       identifier
                        ;

gen_vhdl        :       identifier
                        |       identifier DOT identifier
                        ;

gen_op_classes          :       TOP_CLASSES COLON gen_op_class_list
                        |       TOP_CLASSES COLON DEFAULT
                        |       empty
                        ;

gen_op_class_list   :   gen_op_class_list comma op_class
                        |       op_class
                        ;

op_class        :       LPAREN class_list RPAREN
                        ;

class_list      :       class_list comma class_list_item
                        |       class_list_item
                        ;

class_list_item         :       gen_op
                        ;
```