EXEL:

A Language for Interactive Behavioral Synthesis

Nikil D. Dutt
Daniel D. Gajski

Technical Report 88-21

Department of Information & Computer Science
University of California at Irvine
Irvine, CA 92717.
tel: (714) 856 7063; e-mail: dutt@ics.uci.edu

Keywords: Design Languages, Behavioral Specification with Timing, High Level Synthesis.

ABSTRACT

This paper describes a new input language for behavioral synthesis called **EXEL**. EXEL is a powerful language that permits the user to specify partially designed structures in the language. It employs a mixed graphic/textual user interface to enhance user interactivity. EXEL's design model is comprehensive: it permits specification of synchronous and asynchronous behavior, and allows specification of general timing constraints. A flexible type construct permits the user to define operators and components to be used in the description. Finally, it simplifies compilation by using a small set of constructs for specifying timing and asynchronous behavior. The compiler for EXEL runs on SUN-3 workstations and is written in C and SUNVIEW.

TABLE OF CONTENTS

LIST OF FIGURES

1. Problem Description and Contributions

The task of high level synthesis spans the continuum from automatic generation of a design from a purely behavioral specification, down to compiling a completely specified structural design consisting of a set of components from a given library and their connections. In the first case, the behavior is specified as a set of assignment statements to variables, possibly with total timing constraints for input-output pairs. There is no binding of operations to time or to functional units, no binding of variables to storage elements, and the description does not have any connectivity specified between storage and functional units. At the other extreme, compilation of structure consists of mapping generic components (or components from one library) to components derived from another library. The main objective here is optimization of that mapping to satisfy technology constraints such as time, area, power, testability, etc.

Existing systems and their corresponding input languages straddle either end of this design spectrum: they either automatically synthesize structure from the abstract behavior, or require the user to perform all the behavioral synthesis and input a structural description. There is a lack of tools that fill this gap in the continuum. There are three major requirements for a synthesis tool that meets this need: *interactivity*, *general models* and *modification of compiled*

design.

An *interactive* system permits the user to participate in the iterative process of design planning, synthesis and evaluation. An input language supporting this feature must have a user-friendly interface which permits the user to specify a portion of the design, with the synthesis tools completing the rest of the design. This makes designers more willing to accept synthesis tools since they have control over the design process. The level of input description can thus be gradually elevated from fully structural (when the designer performs all of the design), to fully behavioral (when the designers feel comfortable with using automatic synthesis tools).

The underlying *models* behind the synthesis system must be general enough to handle a variety of *abstraction levels* and *applications*. The design process encompasses several *abstraction levels*: boolean equations, register transfers, algorithms, interface specifications, etc. The model must permit specification of behavior at several intermixed levels. Current synthesis systems are normally limited to one abstraction level only, for instance the algorithmic level. The model must also permit the description of a variety of *applications*, not limited to only instruction-set processors or digital signal processing designs. For this, the model must permit description of synchronous and asynchronous behavior, and allow various types of timing constraints to be specified. This

allows a broad range of designs to be described, including application specific designs.

Finally, the system must allow *modification of compiled design.* Modifying compiled design allows the user to try different alternatives on the generated design without having to rewrite and recompile the input description. With systems like MacPitts [Sout83], the user has to rewrite the description every time the resulting design did not meet the constraints. With these systems, the lack of a prediction capability requires the user to understand the assumptions and intricacies of the synthesis tool. Modification of compiled design gives the user control over the generated design; if it does not meet the requirements, the user can use his design expertise to improve on the design. Finally, it permits incremental upgrade of automatic synthesis tools: we can start with an incomplete synthesis system that performs only some of the synthesis tasks automatically, leaving the rest to the user; synthesis algorithms can be added at a later time.

This paper summarizes the features of a new input language, **EXEL**, for a system that meets these needs. This new language

(1) *is powerful*: the user can specify any level of detail, from fully bound structural designs to purely behavioral specifications;
(2) *enhances interaction*: employs a mixed graphic/textual interface for ease of use;

(3) *has a general model*: permits combined specification of synchronous and asynchronous behavior in a convenient fashion and handles general timing constraints.

(4) *simplifies compilation*: uses a small set of constructs for specifying timing and asynchronous behavior.

(5) *is extensible*: allows user-defined types, operators and components to be used.

2. Existing Input Languages

Several input languages have been used for automatic behavioral synthesis, including ISPS [Barb81], Pascal [Camp85] [PaGa87], ADA [GiBK85] and SILC [BlFR85]. These input languages exhibit a high level of input, where the behavior is described using abstract variables and operators; synthesis algorithms are used to automatically generate a structure. On the other end of the spectrum, commercial schematic capture systems require the user to perform the complete synthesis task; the input is essentially a netlist of register-transfer level modules or logic equations. Both of these types of languages are limited to a particular *level of design*: algorithms for the first kind and register transfers for the second. For describing a large range of designs, the input language must allow specification of a mixture of levels, such as algorithms with register transfers. The languages also do not permit the user to specify *partial bindings* (such as variables to registers, operations to functional units) or *partial structure* (eg. two buses and an ALU) which is input to the synthesis tools. This lack of intermediate descriptions limits user-interaction and the range of

synthesizable designs.

Most of the existing input languages are purely textual. Textual languages tend to be verbose and require the user to learn and understand the syntax of the language. Moreover, a design description written in a textual language does not convey much information to a user who is unfamiliar with the textual input language. Graphical design languages have recently begun receiving more attention [DBRI88] [OTHO88]. These input languages enhance user interaction and ease the task of design entry. However, existing graphical languages are primarily a front end for tools that are at the structural level of input.

The underlying model in most existing behavioral languages are limited. They are targeted to a specific application which describe instruction sets or particular algorithms. They lack a comprehensive model which includes timing specification, busing, clocking and asynchrony. Simulation languages like VHDL [VHDL87] and Verilog [Gate86] have been proposed as input vehicles for behavioral synthesis. However, the model used by these simulation languages does not have good constructs for register transfer descriptions; the behavior is written with the implicit notion of a simulation clock. Several constructs in these languages (such as events indicating signal changes) do not have feasible or efficient realizations in hardware.

3. Features of EXEL

The strengths of EXEL are in its interactivity and adequate models. These are briefly described below:

Mixed graphic/text entry. EXEL is a flowchart-like language. This provides a natural user interface and enhances designer interactivity.

Designer controlled bindings and partial structure. The user can specify bindings in the EXEL description, thereby allowing specification of partial design structure. These bindings may be redone by automatic synthesis algorithms in the design system. This allows for a gradual evolution towards automatic synthesis, where designers gain confidence in the synthesis tools and become comfortable with the idea of automatic synthesis. Hence EXEL covers a larger range of specifications, from completely user-bound (structural) to completely behavioral.

Combined specification of synchronous and asynchronous behavior. Using EXEL, the user partitions the design into processes which exhibit asynchronous, synchronous and mixed behavior.

Adequate timing model. EXEL has a succinct construct which permits specification of general timing constraints. Using this construct, the key concepts of "event", "relativity" and "duration" allow capture of synchronous and asynchronous timing constraints uniformly.

Comprehensive design model. EXEL's design model allows for specification of abstract behavior, register transfers, interface operations and logic.

4. The Language

The **EXEL** input language [DuGa88] is modeled after software flowcharts and state machine flowcharts [Clar73] [Tred81] [DrHa87]. Its intent is to provide a mixed graphic/textual interface to the user so that it facilitates the designer's thought process. A design entity in EXEL is described with the *input definitions* and the *behavior* as a set of communicating processes which operate on the defined structures and variables.

Both the definitions and the behavior of a design to be synthesized are specified in a mixed graphic/textual input form. For each design, a set of declarations specify the inputs, outputs and variables to be used. Optionally, the user may specify structural information such as the type and number of components and buses. Process behavior is specified using a graphical control flow format, along with textual expressions for operations. The control flow of the process is captured through an interconnection of graphic icons. This control flow specifies the states and their transitions for the process. Corresponding to each control flow template, data operations are expressed in a textual form. This section describes the salient features of EXEL; the reader is directed to [DuGa88] for the syntax and other details of the language.

4.1. Definitions

The definitions may be broadly categorized into four classes: *type*, *structural*, *behavioral* and *bindings*. Figure 1 shows some sample definitions using EXEL which will be used as a running example for this section.

```
type
        BOOLEAN                 = {0};
        EIGHT_BIT       = {7..0};
        MEM_1K          = [0..1023] of EIGHT_BIT;
        CMP_EIGHT       = COMPAR(8,EQ,GT,LT);

component
        REG_1   =       REGISTER(8,LOAD,RESET,ENABLE);
        COMP    :       CMP_EIGHT;

port
        DPORT       : output tristate buffered of EIGHT_BIT;
        EPORT       : input_output buffered of EIGHT_BIT;

clock
        CLK             : port;

var

        A, B        : EIGHT_BIT;
        D           : MEM_1K;

const
        SEVEN of EIGHT_BIT = 7;

operator:
        LT_GT   (inputs:
                                A of EIGHT_BIT;
                                B of EIGHT_BIT);
                (outputs:
                                LT of BOOLEAN;
                                GT of BOOLEAN;)
                (operation:
                                LT := A < B;
                                GT := A > B;)
bind
        LT_GT to CMP_EIGHT;
        A to REG_1;
```

Figure 1: SAMPLE EXEL DEFINITIONS

Type definitions allow the user to define new data types and component types for the rest of the description. Sample type definitions would include bit-type, array-type, component type, etc. For example, MEM_1K in Figure 1 is a type denoting an array of size 1024, with each element being eight bits wide. CMP_EIGHT is an eight bit comparator component type specified to have the the functions "EQ", "GT" and "LT".

Structural definitions allow the user to prespecify some or all of the structural components such as registers, function units, ports and buses to be used in the design. This creates a partial structural design on which further synthesis is performed. Each *component* is instantiated from the GENUS generic component library [Dutt88a] by specifying a call to the generic component name with its instantiation parameters. Components may be instantiated directly (eg. REG_1) with an instantiation call, or indirectly (eg. COMP) through a previously defined "type". During synthesis, additional components are automatically synthesized as and when they are required.

Port definitions specify the locations through which the process communicates with the other processes. Typical port attributes include *mode* (input/output/input-output), *gating* (tristate, wired, etc.) and storage (buffered/unbuffered).

Behavioral definitions allow the user to specify abstract data carriers such as variables, constants and special types of operators. Strong typing permits various attributes to be associated with the variables and signals. Typical attributes would include the size, type and representation for data carriers and structural units. This information is necessary during synthesis to generate a consistent structure. In Figure 1, "A", "B" and "D" are defined as variables while "SEVEN" is defined to be a constant whose value is 7.

The *operator* definition shows the capability of defining new operators in EXEL. For instance, LT_GT in Figure 1 is an operator that accepts two eight bit inputs and produces two boolean outputs simultaneously.

The process of mapping behavioral elements (such as variables) to structural components in synthesis is called **binding**. A unique feature of EXEL is that these bindings can be performed by the user in the definitions or in the description of the behavior. This allows the user to further specify some partial structure for the synthesis task. The rest of the bindings are performed by the synthesis system. The *bind* definitions specify "static" bindings which are valid through the EXEL description of the behavior. Variables and operators may be mapped to components (eg. "A" is bound to "REG_1" and "LT_GT" is bound to "CMP_EIGHT").

4.2. Process Behavior

EXEL is a language designed to allow the user considerable control over the synthesis process. The user decides how the behavior is to be partitioned (into synchronous/asynchronous processes), defines global "states" of each process, decides if conditionals are to be implemented in the control or data path, and performs structural bindings to specify a partial structure. Synchronous processes are described using the synchronous state chart, while asynchronous processes are described using the asynchronous event chart. Within each chart, the user can further specify state, unit, register and connection binding.

A synchronous state chart describes the behavior of a process on a global state by state basis, assuming a fixed clocking scheme (as defined in the declarations). Asynchronous operations such as sets, resets and latching controlled by any signal can also be expressed with a special construct as described later. This permits latching of a storage element in a particular state under a signal different from the synchronous clock.

The asynchronous behavior is expressed as a sequence of *event-states* in an event chart. Each event-state is triggered by a specified event (which is often a change in the value of an external signal). Within an event-state, the behavior is captured with a set of of operations on variables. A synchronous clock can also be considered to be an event, and thus may be described in an

asynchronous event chart by explicitly specifying clock events.

When there is a fair amount of both synchronous and asynchronous behavior, and if these are somewhat orthogonal, it is convenient to describe the asynchronous behavior separately from the synchronous behavior. Semantically, the asynchronous specification for a variable or structure overrides the synchronous behavior (for example, a reset-dominant flip-flop). However, there are cases when some synchronous behavior is exhibited in a particular event state. In this case, the notion of a synchronous sub-machine [Holl82] is used to describe the behavior hierarchically. Thus the user has the option of partitioning the design in a variety of ways which fit the description at hand.

4.2.1. Flow of Control

Control flow captures the sequencing of the design over time at the granularity of user-defined *global-states*. For asynchronous charts, the duration of a global-state ends when the event for the successor global-state is triggered. For synchronous state charts, the user determines the meaning of a global-state. A global-state could be at the granularity of a single clock (in which case the user has performed state binding), or could represent a block of statements that can be further scheduled into clocked states (like a VT block [McFa78]).

Both synchronous and asynchronous behavior can be described with the control flow chart by interconnecting the appropriate EXEL graphic icons.

Synchronous descriptions use clocked-state icons, while asynchronous descriptions use event-state icons. Unlike textual languages, the interconnection of these graphic icons determines the sequencing of the behavior. This allows the user to visualize the flow of control through the description and is thus a natural means of design entry, much like flowcharting.

4.2.1.1. Synchronous Icons

Four symbols are used to specify the synchronous control flow chart: the unconditional icon, the conditional test icon, the conditional output icon and the conditional join icon as shown in Figure 2. The *unconditional icon* specifies actions that are to occur unconditionally in that global state. The *conditional test icon* performs a test of some expression (written as a data flow sequence in the icon), which will determine which conditional branch is to be executed. A *conditional output icon* exists as an immediate output of a conditional test icon. It specifies the actions to be performed only when the conditional value matches that of the output branch of the conditional test icon. The *conditional join icon* indicates a merging of several conditional paths.

These symbols are connected by the user in an unambiguous manner to specify the sequencing of the intended algorithm into global states of the synchronous process.

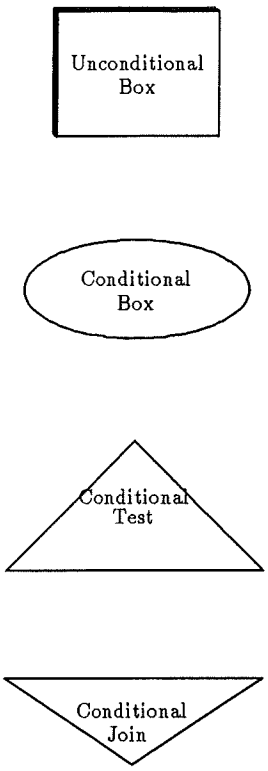

Figure 2: SYNCHRONOUS CONTROL FLOW ICONS

As mentioned before, the user decides if a conditional (such as an "if" or "case" construct) is to be implemented as control or data. A conditional specified by the user in a graphical conditional test icon is implemented in control, while conditionals expressed in the textual EXEL language constructs "if" and "case" are implemented in the data path. This gives the user the flexibility of experimenting with control/data tradeoffs.

4.2.1.2. Asynchronous Icons

A major weakness of many existing input languages is that they do not permit specification of asynchronous behavior. EXEL allows asynchronous behavior to be expressed in an asynchronous chart by specifying operations performed in each event-state.

Two concepts are of importance here: an *event* and an *event-state.* An event is defined by a change in an input signal (port), and forces the process to enter a new event-state. An event-state lasts from the time the event occurs until the occurrence of the next event. Within an event-state, the operations to be performed are described with EXEL's textual statements. Figure 3 shows a sample asynchronous event-state icon which indicates that a new state is entered on the rising edge of A. The variable B is incremented in this event state.

4.2.2. Data Operations

Data transfers and transformations in the design are performed by various types of operators. Broadly, the operators may be classified into *arithmetic* operators, *comparison* operators, *shift/rotate* operators, *logical* operators, *bit manipulation* (concatenation/selection) operators, *array references* and *assignment* operators. Since each data carrier is strongly typed, it is not necessary to have special operators to be used with variables, ports and buses of

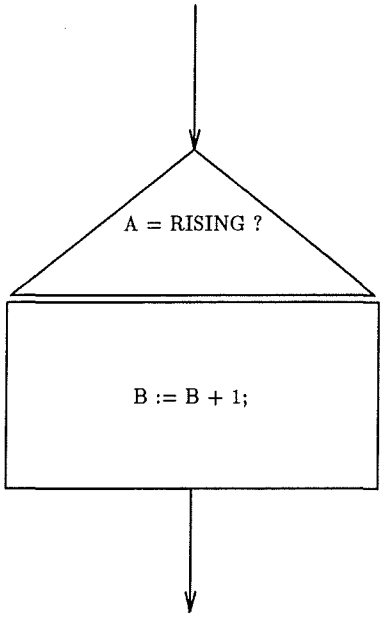

Figure 3: ASYNCHRONOUS CONTROL FLOW ICON

different types. Type mismatches are handled according to predefined rules. When a mismatch cannot be resolved, or is erroneous, the system can flag an error to inform the user. Textual conditional statement constructs such as "if" and "case" are used to perform conditional assignments.

4.2.3. Binding Operators and Variables

Operators and variables in EXEL may be bound to components defined by the user. This gives the user control over selective binding of certain parts of

the behavioral description. For instance, if the user has identified the critical path, he can bind components and connections along the path to meet the critical constraints, while leaving the rest of the bindings to the synthesis system.

The pair "{" and "}" is used for specifying bindings in the textual expressions. For instance, if ALU_1 is defined to be an ALU component, REG_1 is defined to be a register, and A, B are defined to be variables, the following statement:

A{REG_1} := B +{ALU_1} 1;

binds the variable A to REG_1, and the "+" operation to the component ALU_1.

4.2.4. Timing Specification

Several issues relating to specification of timing for synthesis have not been resolved in previous synthesis systems. [NeTh86] and [BoKa87] describe two attempts which try to rectify this situation, but the timing models are not satisfactory. EXEL has a single, concise construct for expressing general timing constraints. This section describes EXEL's model of timing and its specification.

Two types of timing specification are supported by EXEL. The first is a *path-relative delay* which expresses the delay from one point in the structural implementation to another. This delay is the sum of the transmission delays on

the wires, the operation times for components and the set-up and hold times for registers that exist on the physical path between the two points.

The second is *event-relative delay* which expresses the delay for one event with respect to another. An *event* corresponds to the change in the value of a particular signal (or of a set of signals). Since two events may be logically unrelated, event-relative timing specifies the exact sequencing of the two signal waveforms. This is often used in describing protocols which involve two or more signals that are not data dependent, but which must follow a particular sequence over time to ensure correct behavior.

In synchronous systems, the major event is the system clock. All actions are performed on a state-by-state basis where the rising or falling edge of the system clock initiates a new state. Hence, event-relative delays in the synchronous case refer to delays specified with respect to the rising or falling edge of the system clock.

The three notions of *relativity*, *duration* and *event-cause* permit general timing specifications. *Relativity* specifies the change of one signal with respect to another. If the two signals are data dependent, we call this delay specification a *path constraint* and use the keyword **from** to indicate the relativity of the delay. If the two signals are data independent, we call this delay an *event constraint* delay and use the keywords **before** or **after** to specify

event-relativity.

Duration specifies the length of the delay to be **minimum**, **maximum** or **nominal**. A nominal duration is an "average" value with a certain tolerance. A delay not specified to be of a particular duration type defaults to maximum for path-relative delays, and minimum for event-relative delays.

Event-cause specifies the characteristics of the event as being of type **rising**, **falling** or **changing**.

A general form of the assignment statement permits the user to express both kinds of event and path constraints in a concise notation. The general form is:

carrier | event-constraint := expression | path-constraint;

where:

expression is a standard expression using the EXEL operators;

path-constraint is a delay specified from some input (of the expression or port) to the carrier on the left hand side;

and

event-constraint is a set of delays which specify when the signal on the left hand side should receive the computed value on the right hand side, with respect to the specified event.

The versatility of this assignment construct permits the designer to specify timing at various levels: combinatorial delays, delays relative to clock phases, delays relative to latching signals, and asynchronous assignments. The following examples describe different types of timing.

A general *path constraint* can specify delays from inputs to outputs over several assignments, encompassing one or several states. However, if state binding has already been performed, this delay is used to capture the combinatorial delay on the path from the input to the output of the expression. The syntax of each path constraint is:

delay *delay-value* **from** *input*

For instance, if A and B are registers and INPORT is an input port, the statement

B := INPORT + A, delay 80 ns from INPORT, 40 - 60 ns from A;

specifies a delay of 80 ns maximum (by default) from the input port INPORT to the register B, and a delay of 40 ns minimum, 60 ns maximum from the output of register A to the register B. These delay constraints may be passed on to the generator for the component which performs the '+' operation. Note that simulation languages such as VHDL do not allow specification of delay from specific inputs -- they lump all the inputs together and permit specification of only one delay value for the output. This ambiguity makes it hard to synthesize for timing with respect to each input.

The syntax of an *event-constraint* is:

delay *delay-value* {**after** or **before**} *event-cause*

where delay, as before, is a **minimum, maximum** or **nominal** delay, and event-cause is a signal **rising, falling** or **changing**. This type of timing constraint is most often used to capture timing chains from a timing chart, which specifies the change of one signal with respect to another over time.

For example, if A is an output port and B is an input port, the statement

A | (delay 100-1500 ns after B = rising) := X + 1;

specifies that the port A be assigned the value "X + 1" with a delay of 100 to 1500 ns after the value on port B rises.

Clock phase assignment and *signal latching* is also achieved with this construct. For example, if R and Q are registers, and the system clock is 2-phase (with names phase-1 and phase-2), the statement:

R | (after phase-2 = rising) := Q;

assigns the value in register Q to register R in phase 2 of the system clock.

Asynchronous assignments to variables are described using the special assignment operator '<='. Semantically, the asynchronous assignment implies the use of an asynchronous input on the structure bound to the variable, to achieve the assignment. Most often, this type of assignment is used to clear or set a register asynchronously. For instance, if R is a variable bound to a

register and RESET is defined to be an input port, the statement

$$R \mid (\text{RESET} = \text{rising}) <:= 0;$$

ties the RESET line to the 'clear' input of the register R.

For further details on the exact syntax and semantics of the EXEL language, the reader is directed to [DuGa88].

5. Compiler

A compiler for EXEL has been developed on SUN-3 workstations running UNIX. The EXEL compiler is written in 'C', with the graphic routines implemented in SUNVIEW. EXEL is input to **EXTEND**, the synthesis environment. EXTEND currently performs connectivity binding and some register and unit binding, given a description where the user has performed state binding. The output of the compiler consists of symbolic control (for both synchronous and asynchronous parts), and a netlist of generic RTL components [Dutt88a]. This may be passed as input to existing technology mappers such as MILO [VaGa88] or MIS [BrRu87] (for combinatorial logic), where further optimizations may be performed. For further details on the compiler, the reader is directed to [Dutt88b].

6. Example

We illustrate the use of EXEL to describe a simple application which exhibits both synchronous and asynchronous behavior. Figure 4 shows the block diagram of a process that we will call a controlled counter, adapted from [Arms89]. Although this example does not illustrate the complete power of EXEL language, it highlights several of its salient features.

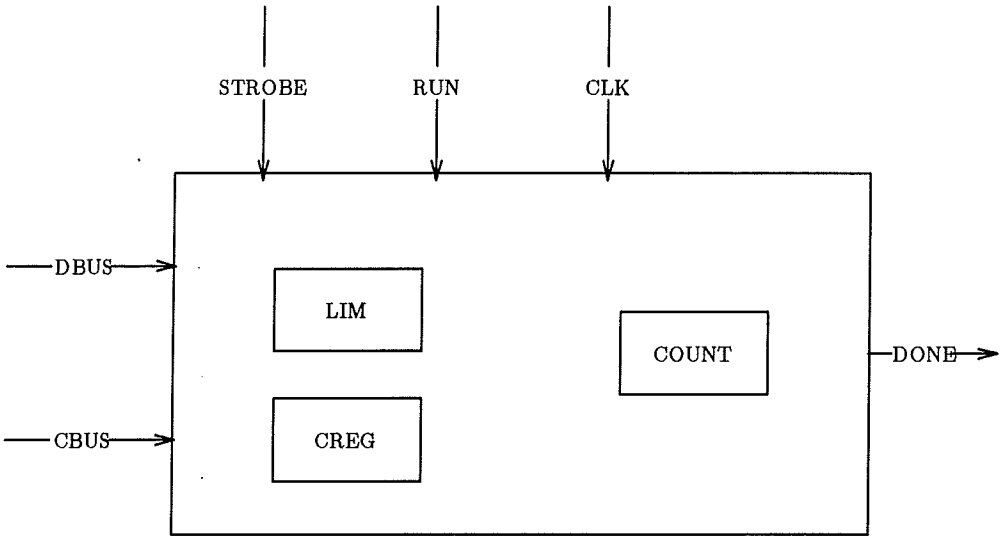

Figure 4: BLOCK DIAGRAM OF CONTROLLED COUNTER

6.1. Controlled Counter

6.1.1. Principles of Operation

The basic operation of the controlled counter is sketched in Figure 5. On the rising edge of the signal STROBE, an internal control register is loaded with the value on CBUS. The value in the internal control register is decoded to perform one of four functions: clear the counter, load a limit register, count up till limit, or count down till limit. The counter runs synchronously under the input clock, and the counting functions are performed only when RUN is high.

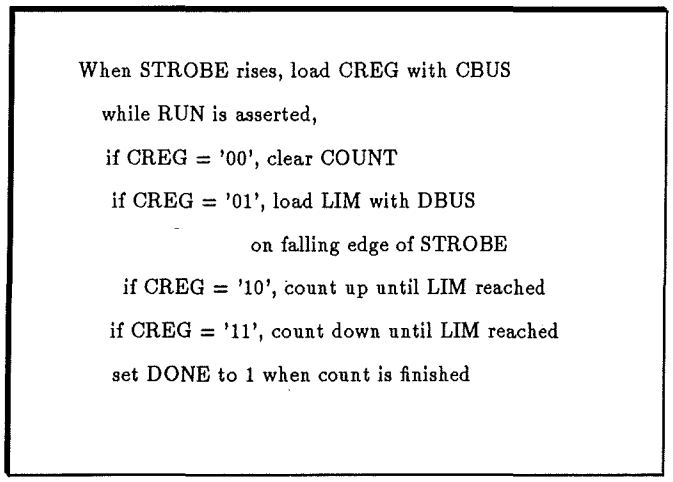

Figure 5: CONTROLLED COUNTER OPERATIONAL PRINCIPLES

6.1.1.1. Declarations

Figure 6 shows the definitions for the process. Two registers, LIMIT and CREG, are defined. The input ports consist of STROBE, RUN, CLK, DBUS and CBUS, while the output port is DONE. The port definitions specify the width, type and direction of the ports for the process. For synchronous operation, the clock has to be defined explicitly. In this example, the system

```
type
        BOOLEAN         = {0};
        TWO_BIT         = {1..0};
        FOUR_BIT        = {3..0};
        REG_TWO                 = REGISTER(2,LOAD,,,,RESET,ENABLE);
        REG_FOUR        = REGISTER(4,LOAD,,,,RESET,ENABLE);
        DEC_TWO                 = DEC(2,4);
        CMP_FOUR        = CMP(4,,GT,LT);
        CNT_FOUR        = UP_DWN_CNT(4,UP,DOWN,LOAD,RESET,SET,ENABLE);

component
        CREG            : REG_TWO;
        LIMIT           : REG_FOUR;
        COUNT           : CNT_FOUR;
        COMP            : CMP_FOUR;
        DECODER                 : DEC_TWO;

port
        CBUS            : input of TWO_BIT;
        STROBE, RUN     : input of BOOLEAN;
        DBUS            : input of FOUR_BIT;
        DONE            : output of BOOLEAN;

clock
        CLK             : port;

var
        LOAD_LIM, UP, DOWN      : BOOLEAN;

const
        ZERO of FOUR_BIT = 0;
        B_ONE of BOOLEAN = 1;
        ONE of  FOUR_BIT = 1;
```

Figure 6: CONTROLLED COUNTER DEFINITIONS

clock is defined to be 1 phase, with the source being the input port CLK. Two variables, COUNT and LIM_EN, are also defined.

6.1.1.2. Behavior

The behavior of the controlled counter can be expressed in many ways, depending on how the user partitions it. For illustration, we choose to describe the behavior by partitioning it into an asynchronous process (driven by the STROBE signal), and a synchronous process (clocked by CLK). Figure 7 shows the asynchronous chart. Two main events can be recognized in this example: STROBE rising and STROBE falling. We therefore describe the behavior in each of these event-states.

When STROBE rises, CREG is asynchronously loaded with the value on CBUS. Next, based on the value in CREG, either COUNT is cleared, the limit register is enabled, or the count-up/count-down sequence is initiated by setting the signals UP or DOWN high.

When STROBE falls, if the LIM_EN signal is high, the LIMIT register is loaded asynchronously with the value on DBUS.

The synchronous behavior is described by the synchronous chart, shown in Figure 8. "COUNT_UP" is a one state loop with several synchronous control icons. First, based on the concatenated value of the signals RUN, UP and DOWN, one of three branches is taken: if RUN and UP are high, the counter

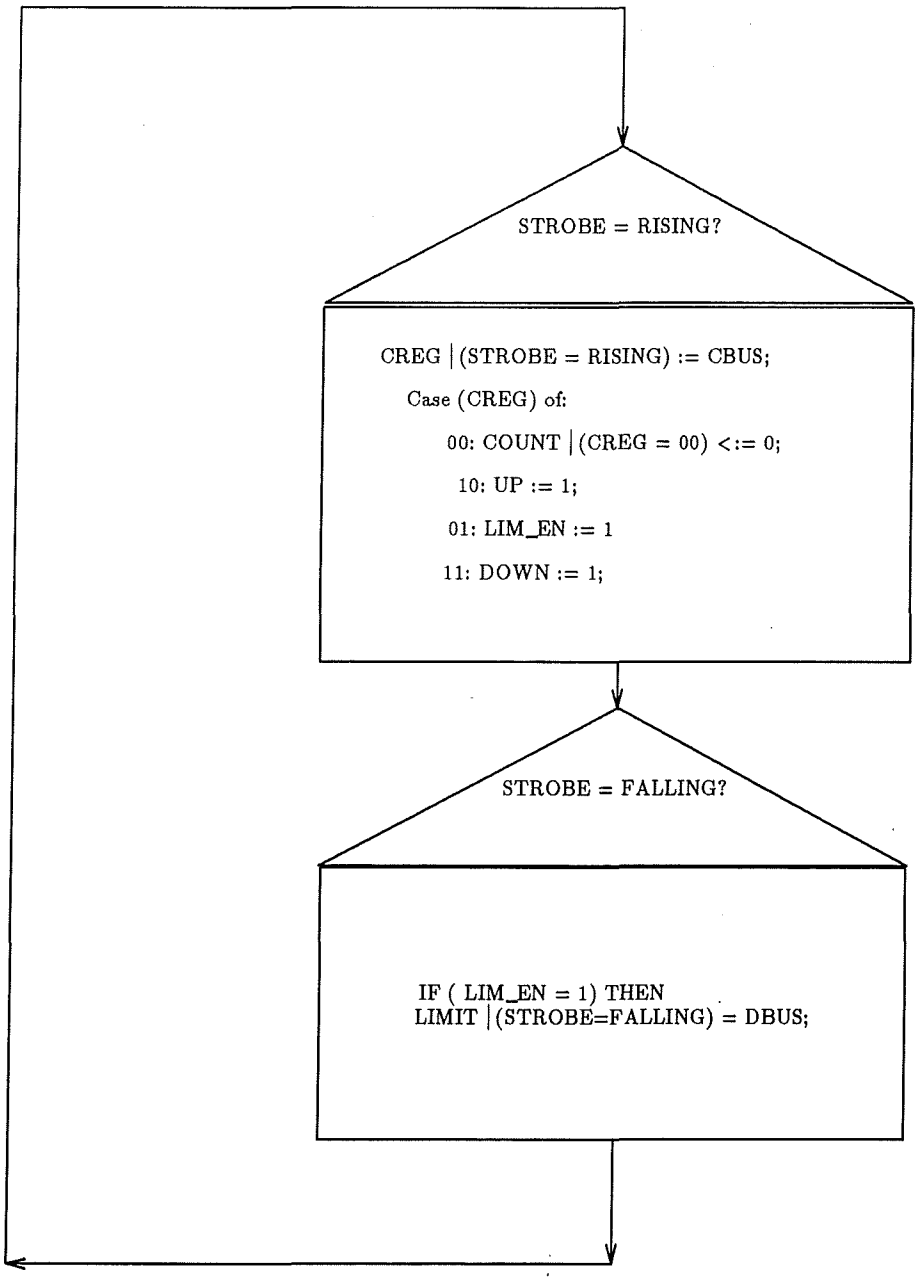

Figure 7: ASYNCHRONOUS CHART FOR CONTROLLED COUNTER

counts up; if RUN and DOWN are high, the counter counts down; in all other cases, the counter busy-waits. When either the count-up or count-down sequence is completed, the DONE signal is set to 1 to indicate completion of the counting task.

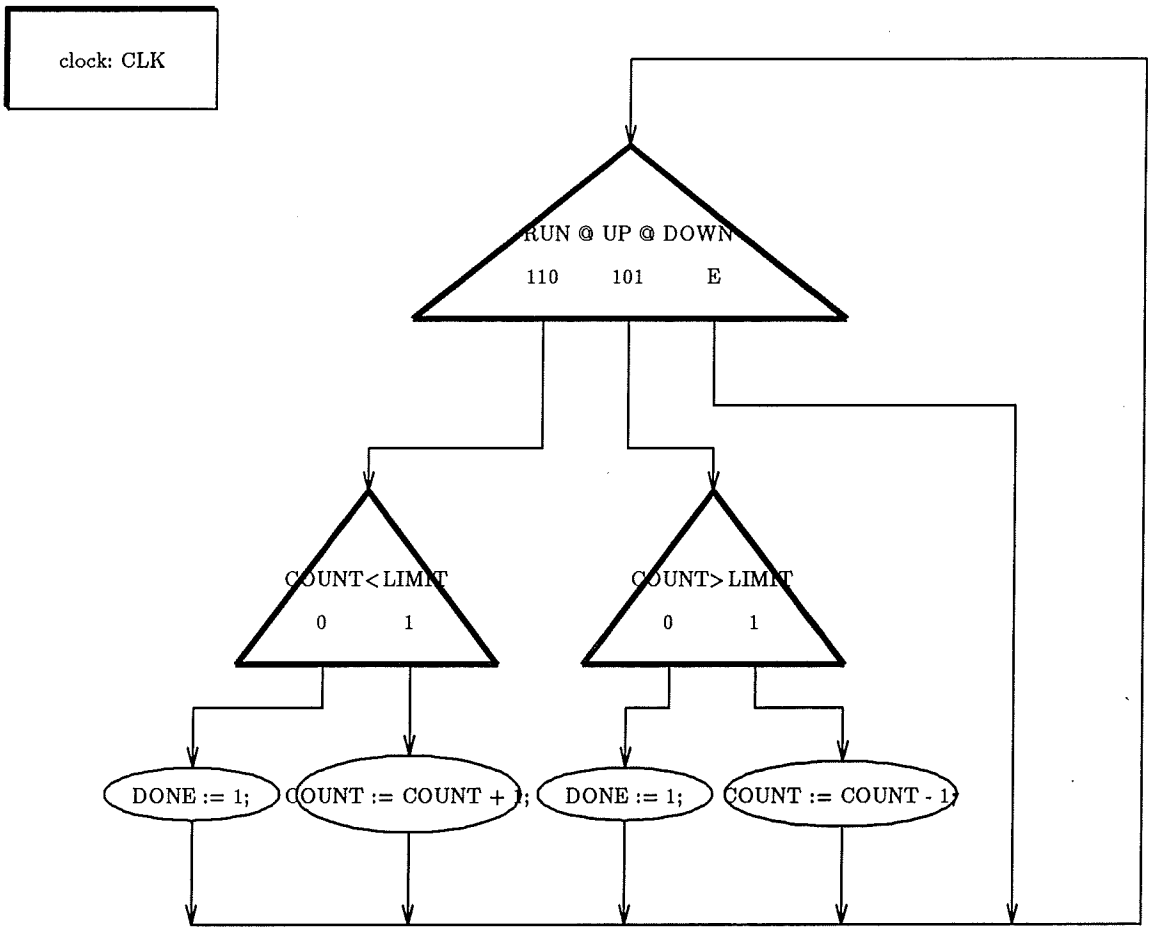

Figure 8: CONTROLLED COUNTER SYNCHRONOUS CHART

6.1.1.3. Structure Generated

Figure 9 shows the structure that is generated after synthesis from the description. Note how the EXEL asynchronous and synchronous charts operated on a common set of defined variables and structures to produce a final design.

This example shows the power of the input description: synchronous and asynchronous behavior is described together in a natural fashion; the resulting description is quite concise and easy to compile. The user can see the behavior at a glance and make any modifications easily.

In contrast, the same example would require several pages of VHDL text to describe [Arms89]. The VHDL description is bulky and cumbersome. The user does not have an immediate feel for the design just by looking at the VHDL description. Most of the existing high-level synthesis languages are not capable of describing this kind of mixed behavior (synchronous and asynchronous), and do not permit mixed behavioral and structural descriptions with partial bindings.

7. Conclusions

In this paper, we introduced EXEL, a language designed for interactive and extensible behavioral synthesis. EXEL has several features that enhance user-interaction: the graphic/textual input, user defined types and operations and

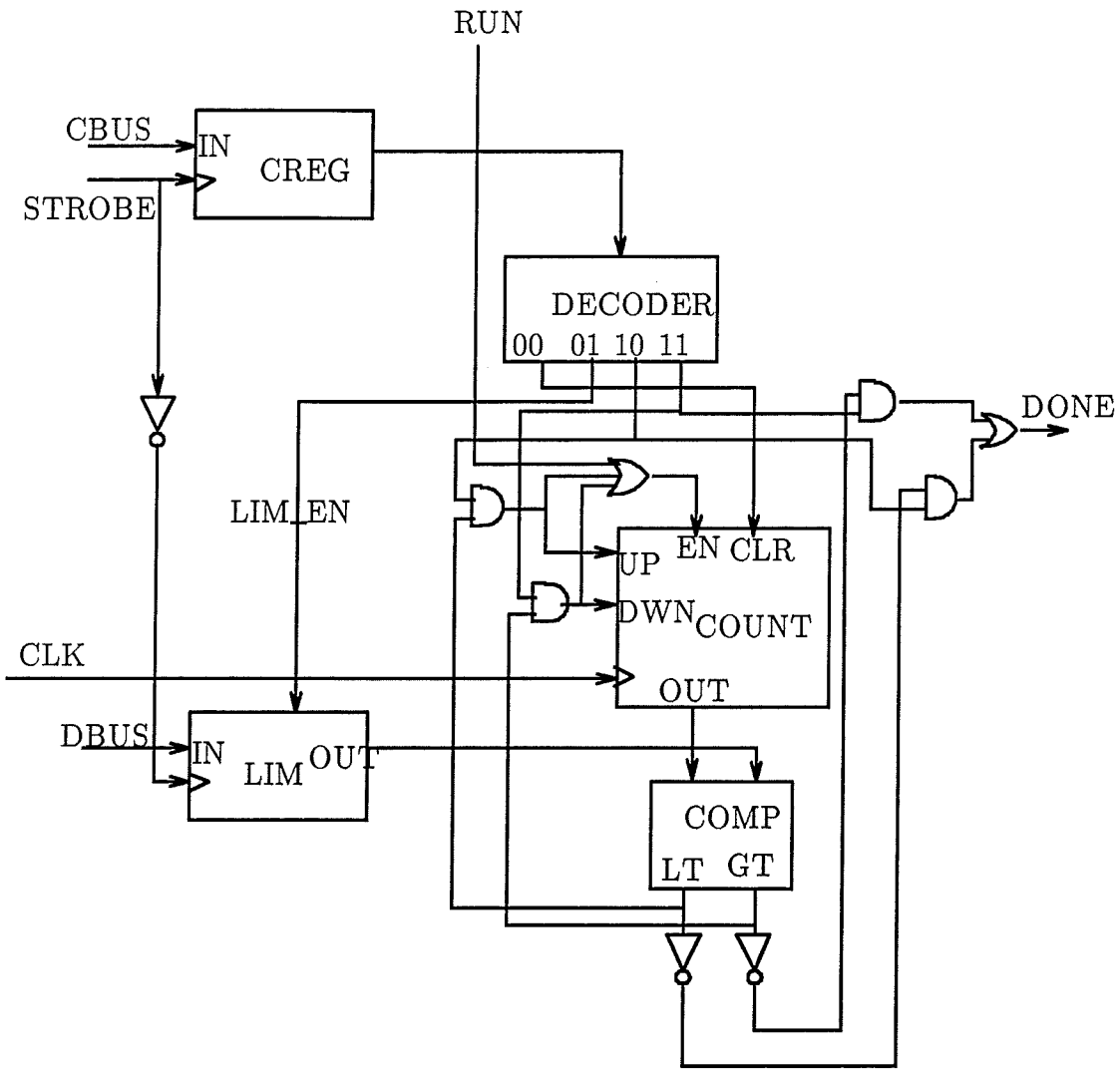

Figure 9: CONTROLLED COUNTER: GENERATED STRUCTURE

partial structural bindings by the user. Its general model permits the specification of both synchronous and asynchronous behavior. EXEL has a small but comprehensive set of constructs for specifying various kinds of timing and assignments. It fills the gap between behavioral and structural representations,

and can be used as input to a VHDL synthesis system, since it is easily translatable to VHDL. EXEL has a working compiler which is input to EXTEND, a behavioral synthesis system.

8. REFERENCES

[Arms89] J. R. Armstrong, "Chip Level Modeling with VHDL," *Prentice Hall*, 1988.

[BRSW87] R. K. Brayton et al., "MIS: A Multiple Level Logic Optimization System," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, Number 6, Nov. 1987.

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

[BlFR85] T. Blackman, J. Fox, and C. Rosebrugh, "The SILC Silicon Compiler: Language and Features," *Proc. 22nd Design Automation Conf.*, June 1985.

[BoKa87] G. Borriello and R. H. Katz, "Synthesis and Optimization of Interface Transducer Logic," *Proc. ICCAD*, Nov. 1987.

[Clar73] C. R. Clare, "Designing Logic Systems using State Machines," *McGraw-Hill Inc.*, 1973.

[Camp85] R. Camposano, "Synthesis Techniques for Digital System Design," *Proc. 22nd Design Automation Conf.*, June, 1985.

[DBRI88] P. J. Drongowski et al., "A Graphical Hardware Design Language," *Proc. 25th ACM/IEEE Design Automation Conference*, Anaheim, CA, June 1988.

[DrHa87] D. Druzinsky and D. Harel, "Using Statecharts for Hardware Description," Proc. ICCAD, Nov. 1987.

[DuGa88] N. D. Dutt and D. D. Gajski, "EXEL: An Input Language for Extensible Register Transfer Compilation," *Technical Report* 88-11, U.C. Irvine, April, 1988.

[Dutt88a] N. D. Dutt, "GENUS: A Generic Component Library for High Level Synthesis," *Technical Report* 88-22, University of California at Irvine, September, 1988.

[Dutt88b] N. D. Dutt, "Behavioral Synthesis from Partial Descriptions," *Ph.D. Dissertation*, University of Illinois at Urbana-Champaign, December,

1988.

[Gate86] Gateway Design Automation Corporation, "Verilog: A Digital Logic Design Language and Simulator," *Gateway Design Automation Corporation*, Westford, MA, 1986.

[GiBK85] E. F. Girczyk, R. J. Buhr, and J. P. Knight, "Applicability of a Subset of Ada as an Algorithmic Hardware Description Language for Graph-Based Hardware Compilation," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-4, No. 2, April 1985.

[Holl82] L. A. Hollaar, "Direct Implementation of Asynchronous Control Units," *IEEE Transactions on Computers*, Vol. c-31, Number 12, December 1982.

[McFa78] M. C. MacFarland, S. J., "The Value Trace: A Data Base for Automated Digital Design," *Technical Report DRC*-01-04-80, Carnegie Mellon University, Dec. 1978.

[NeTh86] J. A. Nestor and D. E. Thomas, "Behavioral Synthesis with Interfaces," *Proc. ICCAD*, Nov. 1986.

[OTHO88] G. Odawara et al., "A Human Machine Interface for Silicon Compilation," *25th Design Automation Conference*, (June 1988).

[PaGa87] B. M. Pangrle and D. D. Gajski, "Design Tools for Intelligent Silicon Compilation," *IEEE Transactions on CAD*, Vol. CAD-6, Number 6, Nov. 1987.

[Sout83] J. R. Southard, "MacPitts: An Approach to Silicon Compilation," *IEEE Computer*, vol. 16, no. 12, (Dec, 1983).

[Tred81] N. Tredennick, "How to Flowchart for Hardware," *IEEE Computer*, December 1981.

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

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