On Linking RT-Component Functionality to Abstract HDL Behavior

Roger Ang and Nikil Dutt

Technical Report 92-97
revised July 1, 1993

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

rang@ics.uci.edu

Abstract

Existing High-Level Synthesis (HLS) Systems typically assume a simple representation for the functionality of RT components and for the binding of abstract behavioral (HDL) operators to RT components. Such a representation scheme simplifies synthesis but ignores the problems of representing realistic RT components that may perform several functions and generate several outputs in a single time step. In this paper, we present a novel representation scheme that links realistic RT-component behavior with abstract HDL behavior. It is useful for representing specific components in user-extendable libraries and adapting component libraries to HDL modeling styles. The representation can also be used to support interactive allocation and binding of components during HLS, as well as interactive rebinding of components once a preliminary floorplan is obtained. This allows the designer to iterate between the results of physical design and the higher-level tasks of component allocation and binding. Furthermore, the representation we describe can be used to establish formally the correctness of interactive binding using realistic RT components. We briefly describe the features of the representation and show its applicability on a HLS benchmark – the AM2901.

Contents

1 Introduction

Currently, most approaches towards high-level synthesis (HLS) of synchronous circuits from hardware description languages (HDLs) assume a simple representation and scheme for binding behavioral operators to register-transfer (RT) components. Synthesis tools often assume an abstract operator directly maps to a single RT operation which in turn can be performed by one or two components. For example, an abstract addition, such as + in an HDL, will map to an *add* operation which can be performed by a RT-component such as an Adder or ALU. This assumes a direct correspondence between behavioral operators and RT operations. Also, it is usually assumed that each RT component (even multi-functional units) can perform only one function at a time, and can produce only a single result in a time step. While these assumptions ease the mapping of abstract operators to RT functional units (FU), it also prevents users from precisely describing FU-specific behavior and prevents tools from producing efficient designs. However, a simple examination of existing RT-components from standard databooks reveals these assumptions are too restrictive: many interesting databook RT-components are multi-functional and have multiple outputs, with several functions performed simultaneously, producing several outputs in a single state. Unfortunately, this results in a mismatch with the original HDL semantics which typically has operators with single-function-single-output semantics. For example, consider the adder shown in Figure 1. At the RT level, the *add* operation performed by this adder takes three inputs and produces two outputs. However, in most standard HDLs (e.g., VHDL), the behavioral operation, +, is a two-input, one-output function. Because of the mismatch of behavioral and RT level semantics and the previous assumptions, synthesis tools often can not derive a specific component from the behavioral description of that component.

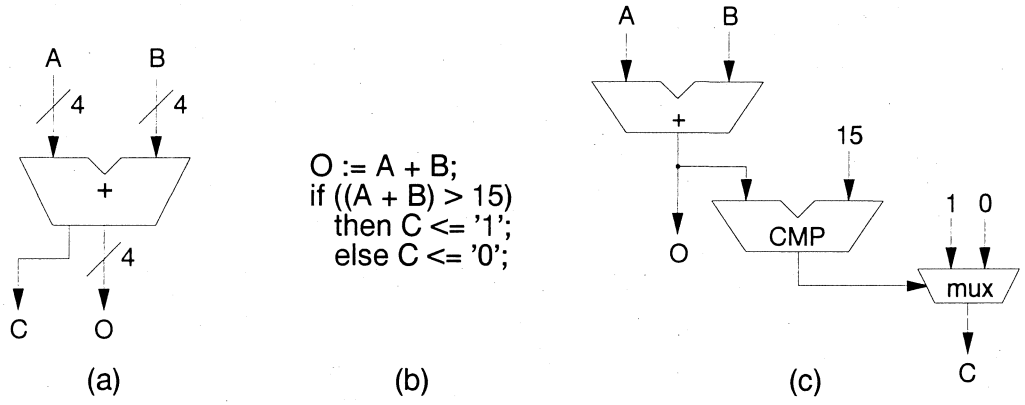


Figure 1: Adder: (a) RT component, (b) behavior in VHDL, (c) circuit synthesized from VHDL.

Current representations of HDL behavior with respect to RT component functionality are limited because they fail to recognize some basic aspects of RT components and HDLs. First, RT components can simultaneously perform multiple functions, yielding multiple outputs in a single time step (e.g., the adder). Second, RT operations are not in one-to-one correspondence with behavioral HDL operators. For example, while the RT operation *add* performed by an adder can be used to implement the behavioral operator, +, the operation *increment* performed by an incrementer can also be used to implement + when one of its behavioral operands is the constant 1. Because current representations are constructed without recognizing these facts, synthesis tools working from behavioral HDL descriptions often do not make efficient utilization of existing RT components, many of which are multi-functional and generate multiple outputs simultaneously.

This report presents a representation of RT component functionality based on the specific components available from a RT library. This representation is capable of efficiently describing the mapping of behavioral constructs to realistic RT units. There are many advantages to use of this representation. It provides a mechanism for efficient mapping of behavior to a set of available components. It is useful for representing user-extendable libraries so that synthesis systems using this representation can be customized to specific

component libraries. We will describe the necessary specifications to "construct" a component in our representation. It can also be used to help adapt a system to a particular HDL modeling style, which we will demonstrate in an example. Further, because of the type of information maintained by this representation, it is suitable not only for automatic synthesis tools, but is also useful in an interactive synthesis process, e.g., human-guided design exploration through RT unit rebinding. Finally, since our eventual goal is to provide an interactive framework for scheduling and allocation with realistic RT components, we need to ensure that the original behavior of the design is preserved; the representation we describe provides a link for formally verifying the equivalence of the realistic RT-behavior with the input HDL behavior.

2 Related Work

Much work has been published on the representations for the binding of behavior to RTL structure (e.g., [CaTa89] [Knap89] [LGCP91] [Rose86] [Thom90]). However, most of them differ from this work in their fundamental assumptions about RT component functionality. For example, in the System Architect's Workbench [Thom90], an ISPS operator can map to multiple VT operators which in turn can map to RTL modules. However, a FU is assumed to be able to only perform one function at a time, i.e. generates a single output and is associated to a single behavioral operator for a time step. Most behavioral synthesis tools are based on this assumption, e.g., MIMOLA [Marw86], ADAM [GrKP85], and the systems described in [CaWo91] [MiLD92]. At the logic synthesis level, Takagi [Taka85] describes a representation of complex RT components using templates, and uses these templates to synthesize gate-level structures. [BeCM92] describe a similar template representation scheme using VHDL macros for sequential logic synthesis. However, neither of these approaches describes how to relate RT functionality with the abstract HDL behavior.

The traditional approach of mapping behavioral HDL operations to simple RT functions requires an additional phase of technology mapping to standard RT components [DuKi91], or requires a phase of clustering into multi-functional units corresponding to RT components [RuGB90]. Both of these options have two major drawbacks. First, they typically assume that only one function (and hence only one output) is performed in a single step. This complicates the task of high-level technology mapping to realistic RT components that exhibit multiple functions simultaneously. Second, the simple mapping scheme may not allow the exploration of certain design configurations that would otherwise have been possible if more realistic RT behavior were modeled early in the design. For instance, the add and shift behavioral operations may never be considered to execute "concurrently" on an ALU; hence with a resource constraint of a single ALU, these two operations will get scheduled into different states. A subsequent phase of clustering or technology mapping will generally be unable to detect and rebind these two operations to have them execute concurrently on the ALU.

In contrast, our representation is based on the assumption that RT components may be multi-function/multi-output. [KnWi92] take an approach similar to this work and recognizes the possibility of multi-function/multi-output components. However, their solution is quite different in that the source HDL they use, IMBSL [Knap89], is unlike other popular HDLs. In IMBSL, data operations are specified in a Lisp-like notation where functions can take multiple input values and generate multiple results. In this way, it is possible for IMBSL operators to map directly to RT structures. However, since standard HDLs, e.g., VHDL and Verilog® are grounded in imperative language semantics, simultaneous multi-functional mappings cannot be easily applied to such HDL behaviors.

3 Representation of Component Functions and Bindings

In this section we briefly describe the representation of component functions and bindings that link the RT component functionality, the original HDL behavior and the bindings between them. We use a simple example to highlight the basic representation scheme. The details of the representation are discussed in a later section.

Our representation uses three basic abstractions:

1. RT level specification of components describing the input and output pins and the input-to-output functionality of each component.

2. *RT function* sets — abstractions of the RT functionality of the components. Sets of functions are associated with each component and functions are grouped into classes, where all functions in a class can be performed simultaneously.

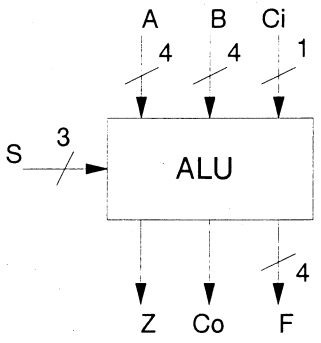
3. Intermediate behavioral representation — the internal representation used by a synthesis tool for circuit specifications, typically derived from parsing of an HDL.

For our examples, we will assume the original specification was written in a HDL with imperative programming-language like semantics (e.g., VHDL), and was parsed into a representation which uses data flow graphs (DFGs), e.g., CDFG [OrGa86]. The following abstractions are used in the primary components of our representation:

1. RT structure — describes instances of components and how input/output pins are connected.

2. RT Data Flow Graph (RTDFG) — a combination of a global data flow graph and a state transition graph. However, data transformations in this graph are based on the semantics of RT functions. From this graph there are links to the behavioral representation indicating what portions of the behavior are performed by particular RT functions. Also, there are links to the RT structure indicating which unit will perform particular RT functions.

3. Behavioral template — portions of behavior that particular RT functions can implement. These are used to determine how RT functions can "cover" the abstract behavior.



S	operation
000	F <– A+Ci, Co <– (A+Ci) > 15, Z <– (F=0)
001	F <– A+B+Ci, Co <– (A+B+Ci) > 15, Z <– (F=0)
010	F <– A+$\overline{B}$+Ci, Co <– (A+$\overline{B}$+Ci) > 15, Z <– (F=0)
011	F <– A+$\overline{Ci}$, Co <– (A+$\overline{Ci}$) > 15, Z <– (F=0)
100	F <– A and B
101	F <– A or B
110	F <– A xor B
111	F <– $\overline{A}$

Figure 2: Typical ALU.

We illustrate the semantics and relationships of these abstractions and structures through a specification of a unit and an example binding. Let us start with the ALU shown in Figure 2, a variant on an ALU described in [Mano88]. This ALU is capable of performing various operations as shown, but for our requirements, we are not concerned with how these operations are performed. We only need to know:

- how to indicate those operations, i.e., the control line values for the ALU,
- input and output pin characteristics, e.g., bit widths,
- and the relationships of inputs to outputs for each operations.

We then develop sets of RT functions for this ALU (see Figure 3). For each operation the ALU can perform there is an associated set. In each set, there is an RT function associated with each output that produces meaningful data for that operation. Each RT function has parameters which correspond to inputs of the ALU. RT functions in the same set can be performed simultaneously, while those not in the same set must be performed in separate time steps. The semantics of the RT functions at the behavioral and RT levels are determined, respectively, by how they are related to behavior, and the component they are bound to and

(S : 000)	(S : 001)	(S : 010)	(S : 011)
INC(A,Ci) -> F INC_CARRY(A,Ci) -> Co INC_ZERO(A,Ci) -> Z	ADD(A,B,Ci) -> F CARRY(A,B,Ci) -> Co ZERO(A,B,Ci) -> Z	SUB(A,B,Ci) -> F SCARRY(A,B,Ci) -> Co SZERO(A,B,Ci) -> Z	DEC(A,Ci) -> F DEC_CARRY(A,Ci) -> Co DEC_ZERO(A,Ci) -> Z
(S : 100)	(S : 101)	(S : 110)	(S : 111)
AND(A,B) -> F	OR(A,B) -> F	XOR(A,B) -> F	NOT(A) -> F

Figure 3: RT-function sets describing simultaneous operations for ALU.

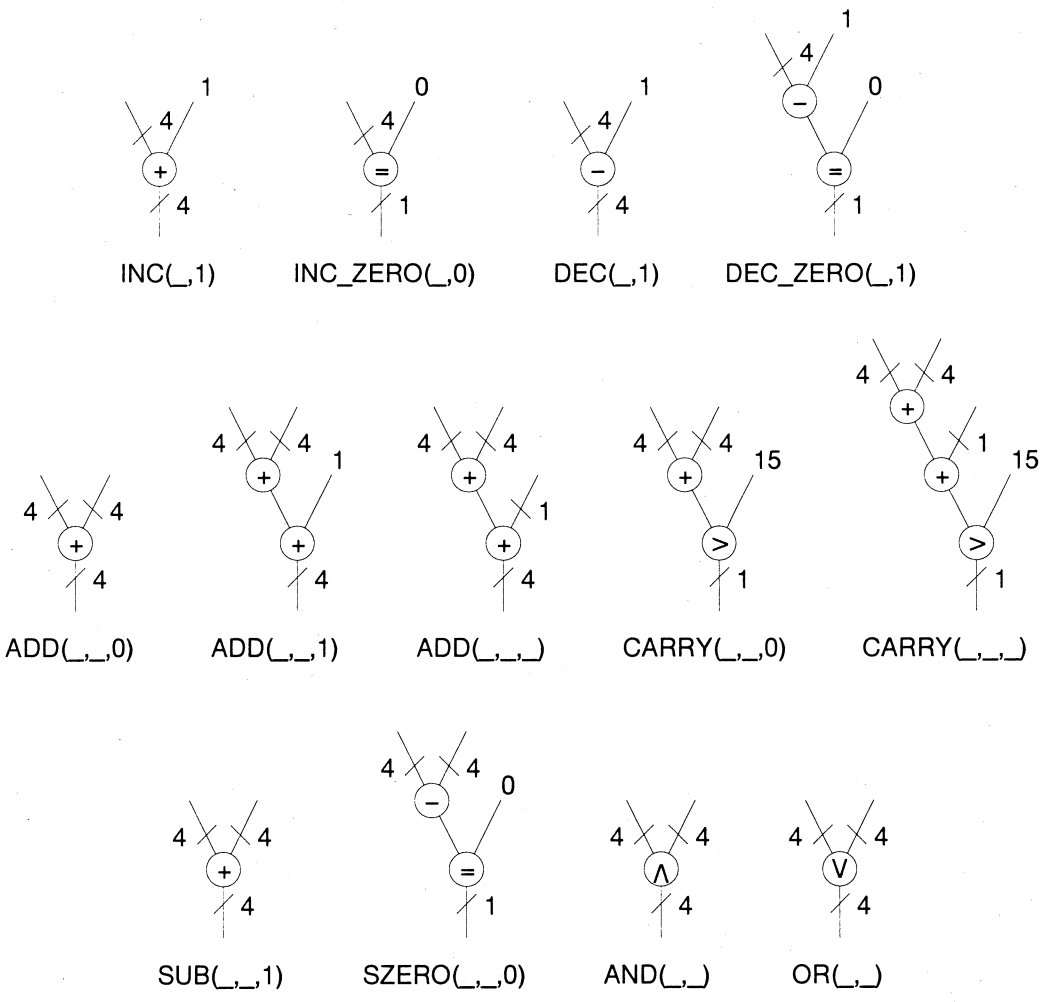


Figure 4: DFG templates for ALU RT functions.

the set, i.e., control code, they belong to. The names for the RT functions in this example are arbitrary and need not be unique to this component. A named RT function can be used by any number of components, as long as its parameters and behavioral templates are consistent and applicable to all components with that RT function. With these sets, we have an abstraction of the functionality for this specific ALU. A smaller set could be specified if the synthesis process using this component wishes to support only a subset of the ALU's functions. To use these functions for HLS, we need a mapping from behavior to these functions. Figure 4 shows some templates for the RT functions of the ALU. As stated previously, we assumed a DFG representation for behavioral assignments of data. These templates define the behavioral meaning of the RT functions and provide a mechanism to map portions of behavior to RT functions and, consequently, to RT components. Conversely, this may also provide a mechanism to verify that an existing circuit can perform a given behavior. How much of the ALU's functionality and the efficiency of ALU function usage depends on how extensive the templates are. Furthermore, since these templates are the behavioral link to the RT-components, they can form the basis of modeling guidelines for use of an HDL in a particular synthesis process.

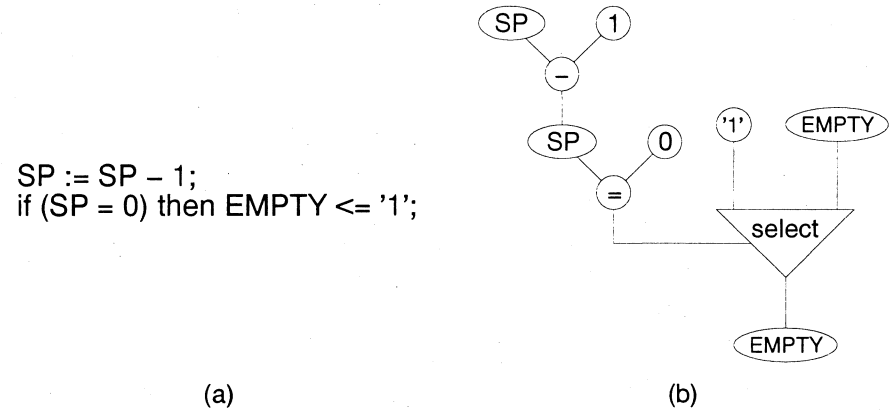


Figure 5: Example VHDL and DFG.

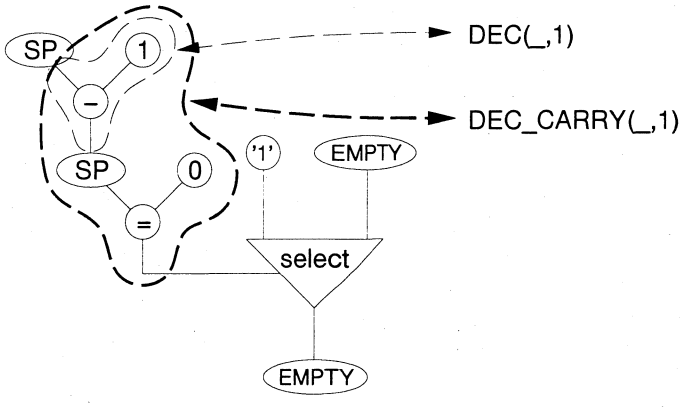


Figure 6: Mapping DFG to RT functions.

With this specification of the ALU, we now show how to represent the binding of behavior to the ALU.

Since the primary focus of this paper is to describe a representation for the binding of "real" RT-components to abstract HDL behavior, we will not describe algorithms for RT unit allocation or binding, or scheduling of HDL descriptions. Rather, we will present only the representation of bindings from the behavior to the synthesized RT structure. We illustrate this process by describing the binding of a segment of VHDL behavior to an RT-structure corresponding to a databook component, as shown in Figure 2.

We start with the two lines of VHDL shown in Figure 5a. These can be parsed into a DFG show in Figure 5b. Allocation using the previously described templates for the ALU will find the DFG can be "covered" as shown in Figure 6, indicating the ALU can be used in this instance. Scheduling based on this covering and allocation will find that these two behavioral operations can be performed in a single time step since the RT functions they are bound to belong to the same set. At this point, the RT Data Flow Graph comes into use.

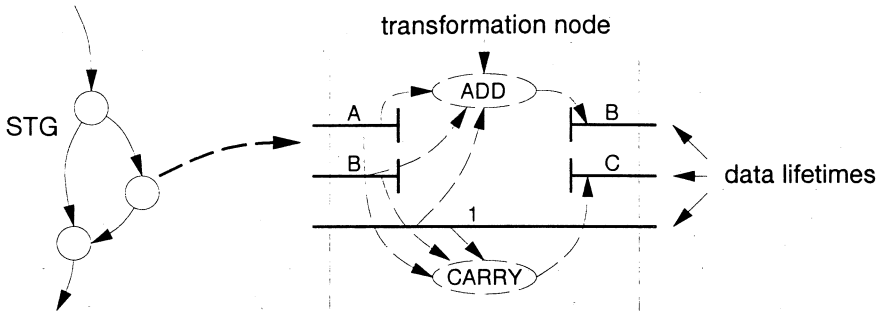


Figure 7: RTDFG

To construct the RTDFG, global data flow analysis is needed as well as construction of a state-transition graph. However, these tasks are already necessary for scheduling. The key features of the RTDFG, illustrated in Figure 7, are:

- State transition graph — provides a "timeline" through the control paths of the design.
- Data lifetimes — based on the standard compiler concept of *use-definition chains* [AhSU86]. For every data carrier, i.e., variable, each *use*, i.e., variable as an operand, is linked to every applicable *definition*, i.e., assignment to that variable. A definition is "live" in a state if a control path to an applicable use goes through that state.
- Transformation nodes — combined with the data lifetimes, these form a representation of global data flow for the specification, and provide *explicit functional mappings* of inputs to outputs. These nodes indicate how data is transformed and the type of transformation done. For each node, there is a set of data lifetimes that are inputs of the transformation, a data lifetime that is the output of the transformation, and a mapping to an RT function specifying the type of function and relating how the inputs and output correspond to RT function parameters. In addition, these nodes maintain links to the behavior and RT structure to indicate RT unit bindings to behavior. The link to behavior specifies the portion of behavior covered by the RT function, and the link to the RT structure indicate the FU that will perform the RT function.

Figure 8 shows how the RTDFG data lifetimes relate to the DFG of our example. The use of SP ends a lifetime, while the assignment to SP begins another lifetime. Figure 9 illustrates the use of the RTDFG transformation nodes. The solid arcs to the nodes indicate input data lifetimes while the arcs away from the nodes indicate output data lifetimes. Figure 9 also shows how links to the RTDFG transformation nodes are used to represent binding of the behavior to the RT structure.

In [AnDu92], we described equivalence-preserving transformations for interactive rescheduling, by ensuring that behavior of the original CDFG from the HDL specification was equivalent to the scheduled SFSM.

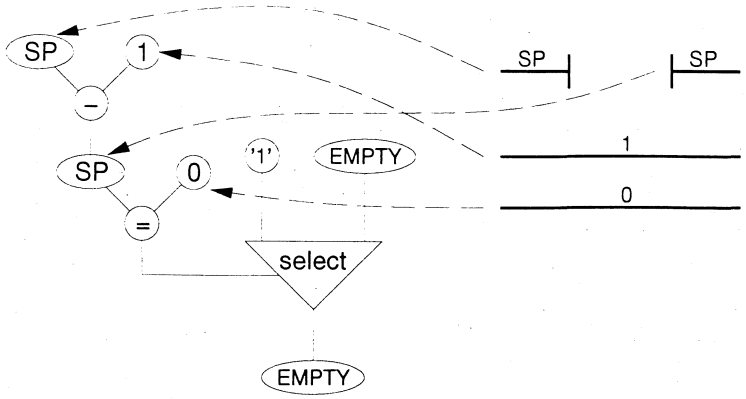


Figure 8: Relationship between behavior and RTDFG data lifetimes.

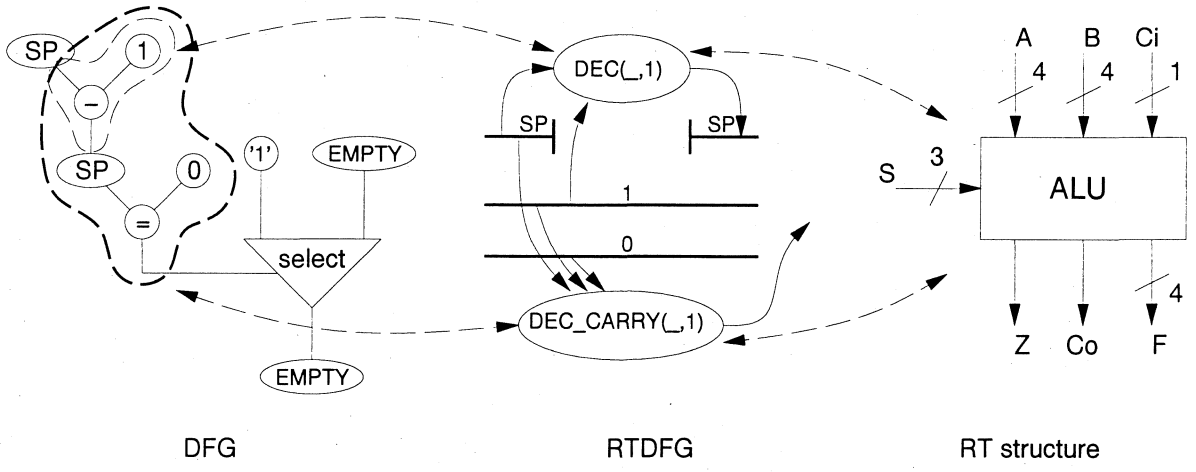


Figure 9: Relationship between behavior, RTDFG transformation nodes, and links to RT structure.

In a similar fashion, the representation presented here links the behavior of the scheduled CDFG (left side of Figure 9) with the more general FSMD behavior of the RT structure (right side of Figure 9). Hence we can use this representation to establish formally the correctness of interactive rebinding transformations used during HLS.

4 Example Application

To demonstrate one of the advantages to using our representation, we examined a behavioral description of the AM2901 microprocessor-slice present in the 1992 High-Level Synthesis Workshop Benchmark set [HLSW92]. The AM2901 is a four-bit, cascadable microprocessor slice. The description is a behavioral model, written in VHDL for simulation. The design was modeled as a single VHDL process. For this example, we will examine portions of the model describing two critical features of the AM2901:

- A 4-bit ALU, capable of performing arithmetic and logical functions on selected source words.
- A destination selector which decides:
  1. whether to load the ALU output (with or without shifting) into the RAM,
  2. whether to load the ALU output (with or without shifting) or the Q register contents (with shifting) into the Q register, and
  3. whether to forward the ALU output or the Port A contents to the external data output.

The partial VHDL derived from the actual HLSW92 benchmark for these portions of the design are shown in Figure 10 and Figure 11.

```
-- SELECT THE FUNCTION FOR ALU.
       case I(5 downto 3) is
          when "000" =>
                       R_ext := '0' & RE;
                       S_ext := '0' & S;
                       result := R_ext + S_ext + ("0000" & C0);
                       C4 <= result(4);
                       F := result(3 downto 0);
          when "001" =>
                       R_ext := '0' & not(RE);
                       S_ext := '0' & S;
                       result := R_ext + S_ext + ("0000" & C0);
                       C4 <= result(4);
                       F := result(3 downto 0);
          when "010" =>
                       R_ext := '0' & RE;
                       S_ext := '0' & not(S);
                       result := R_ext + S_ext + ("0000" & C0);
                       C4 <= result(4);
                       F := result(3 downto 0);

                                    :
```

Figure 10: VHDL for ALU functions.

Because of the semantics for VHDL according to IEEE standard 1076 [IEEE87], the writer of this description used an unusual modeling style to produce a simulatable model of the 2901. In order to generate a carry bit for the addition operations, the operands are extended with an additional high-order bit to be 5 bits wide, so that the extended bit can be used as the carry output (see Figure 10). To add in the carry in input, `C0` , four additional bits were required. Since there are no shift operators in VHDL, the shifting of data is accomplished by concatenation of selected bits shown in Figure 11. Obviously, current synthesis approaches would have difficulty with this style of description. Most current synthesis processes would only examine operator types and widths of operands. It would not be possible, in the general case, to recognize

```
-- WRITE TO DESTINATION(S) WITH/WITHOUT SHIFTING.
  case I(8 downto 6) is
    when "000" =>
      dout := F;    -- INTERMEDIATE OUTPUT
      Q := F;       -- WRITE TO DESTINATION
      Q0 <= 'Z';
      Q3 <= 'Z';
      RAM0 <= 'Z';
      RAM3 <= 'Z';
    when "001" =>
      dout := F;
      Q0 <= 'Z';
      Q3 <= 'Z';
      RAM0 <= 'Z';
      RAM3 <= 'Z';
    when "010" =>
      dout := A;
      RAM(Badd) := F;
      Q0 <= 'Z';
      Q3 <= 'Z';
      RAM0 <= 'Z';
      RAM3 <= 'Z';
    when "011" =>
      dout := F;
      RAM(Badd) := F;
      Q0 <= 'Z';
      Q3 <= 'Z';
      RAM0 <= 'Z';
      RAM3 <= 'Z';
    when "100" =>
      dout := F;
      RAM(Badd) := RAM3 & F(3 downto 1);
      Q := Q3 & Q(3 downto 1);
      Q3 <= 'Z';
      RAM3 <= 'Z';
      RAM0 <= F(0);  -- SHIFTER SIGNALS
      Q0 <= Q(0) ;
    when "101" =>
      dout := F;
      RAM(Badd) := RAM3 & F(3 downto 1);
      Q0 <= 'Z';
      Q3 <= 'Z';
      RAM3 <= 'Z';
      RAM0 <= F(0) ;
    when "110" =>
      dout := F;
      RAM(Badd) := F(2 downto 0) & RAM0;
      Q := Q(2 downto 0) & Q0;
      Q0 <= 'Z';
      RAM0 <= 'Z';
      RAM3 <= F(3) ;
      Q3 <= Q(3) ;
    when "111" =>
      dout := F;
      RAM(Badd) := F(2 downto 0) & RAM0;
      Q0 <= 'Z';
      Q3 <= 'Z';
      RAM0 <= 'Z';
      RAM3 <= F(3) ;
    when others =>
  end case;
```

Figure 11: VHDL for destination selector.

the stylized description of an addition in Figure 10 as an operation that can be performed by a 4-bit ALU with carry input and output. This is a result of a modeling style that seems counter-intuitive. However such counter-intuitive modeling styles are often forced by HDLs where lower-level RT constructs (e.g., carry, overflow, and reset) are not easily represented within the standard behavioral constructs. For example, in Figure 11, the lack of a behavioral construct for the RT operation, *shift in*, forces the use of a concatenation operator. This makes the `&` operator ambiguous to conventional synthesis since it is usually viewed as only indicating a grouping of signals rather than an actual function to be performed by an RT unit, in this case bit shifts done by a shifter. As a consequence of this stylized modeling, current synthesis systems would not produce RT structures resembling the actual implementation of the 2901, even with complex function and expression analysis similar to that described in [RuGB90]. For the VHDL in Figure 10, current synthesis tools will at best arrive at a 5-bit ALU where the carry out bit is derived from the high-order bit of the data output, i.e., the carry out of the ALU is not utilized. For the VHDL in Figure 11, instead of placing a shifter as input to the register for `Q`, a 3-input, 4-bit wide multiplexor would be requested.

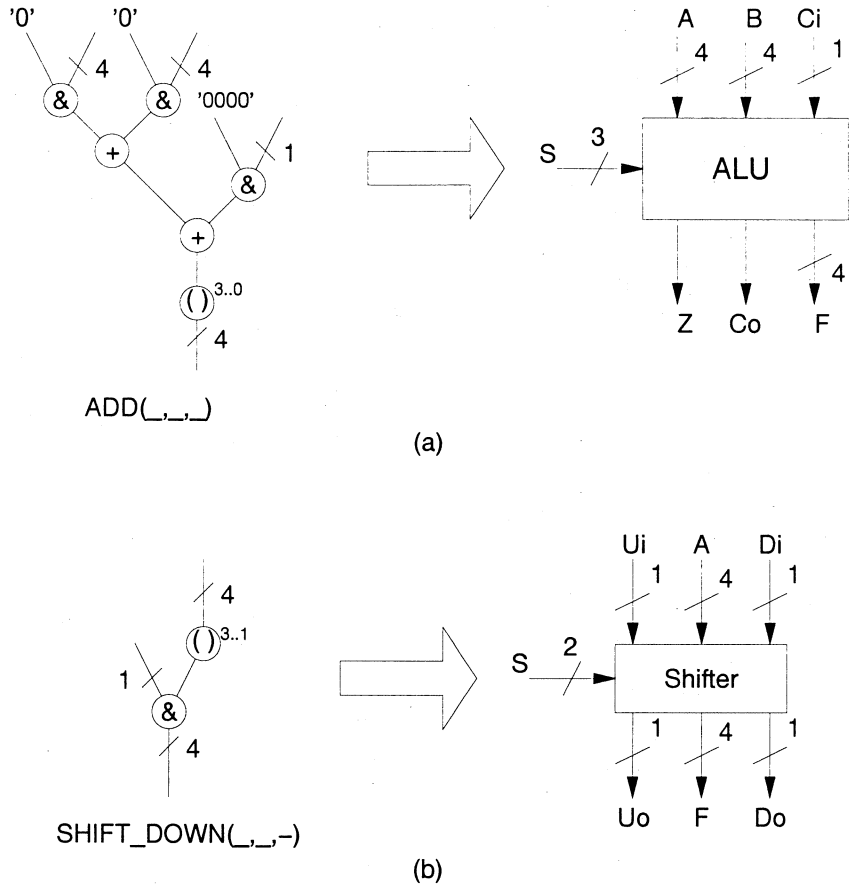


Figure 12: Templates and RT components for modeling style: (a) extended addition for ALU, (b) shift down with input for Up/Down Shifter. `&` indicates bit concatenation. `()` indicates bit selection with the bit range beside the node.

Our representation would overcome this difficulty with the modeling style by enabling specification of mappings that account for the stylized approach through behavioral templates. Figure 12 shows two templates that can be applied to the VHDL examples. Figure 12a illustrates a template for the "extended" addition used in this modeling style and the RT function it can map to, the `ADD` for our example ALU.

Figure 12b shows how behavioral templates can be used to recognize "work-arounds" used in a modeling style, in this case, the combined usage of a bit selection and bit concatenation to emulate a bit shift that can be performed by an Up/Down Shifter. Using additional templates similar these, a synthesis tool using our representation should be able to determine that the VHDL in our example can be implemented with a 4-bit ALU and a 4-bit shifter similar to the actual 2901 component. For a synthesis system using our representation, the user is allowed to specify behavioral templates and RT function mappings; thus the user is able to customize the system to both a specific library of components and to a particular HDL modeling style to facilitate efficient synthesis.

5 Representation Structures

As previously discussed, our binding representation scheme requires representations for RT level components and behavior. This binding representation is initially targeted to be used in the BIF design environment using the GENUS componentlibrary [DuHG90] [JHDu93]. Consequently, the BIF data structures will be used as the behavioral representation and GENUS for RT components. For this scheme, there are actually two separate concerns that must be addressed by the data structures:

1. Representing the current bindings of behavior to RT components.
2. Representing how behavior should map to RT components (i.e., how to specify and represent the behavioral templates and their mappings).

5.1 Structures for Bindings

A BIF table represents a finite state machine. To represent a state transition graph, additional structures are created and linked to the BIF structures. The state transition graph consists of a set of linked nodes. Each node has

- the name of the state the node is for and if it is the first (initial) state of the table.
- a link to the BIF entry structure describing the state.
- a set of links to predecessor nodes (i.e., states that transition to this state).
- a set of links to successor nodes (i.e., states that can be transitioned to from this state).
- a set of live variables (i.e., variable definitions valid through this state).

As can be noted from the above, the state transition graph is used to maintain variable lifetimes. The BIF data structures use a form of data flow graph to represent individual data assignments. In BIF, data assignments are parsed into a data flow graph representation, with each assignment creating an individual flow graph. In this representation, *read* and *write* nodes indicate the use and definition, respectively, of variable or port values. To correlate these individual uses and definitions across the entire table, additional structures are created and linked to the BIF flow graph structures. These structures are centered around "dependency nodes" which maintain data dependencies between variable uses and definitions. These nodes contain

- links to various BIF data structures, indicating the exact location of the variable *instance* this node is for.
- link to a symbol table entry for the variable.
- a set of links to "parents" of this node. If this node is for a use of a variable, this is the list of nodes for definitions that can apply to it.
- a set of links to "children" of this node. If this node is for a definition of a variable, this is the list of nodes for uses that it can apply to.

As previously stated, we use transformation nodes to explicitly represent how these data lifetimes are used by data transformations (RT functions) performed by various RT units. These transformation nodes maintain

- the RT function being performed.
- a set of dependency nodes that are the values being used by the RT function, and how they relate to the parameters of the RT function.
- a dependency node that is the value being defined by the RT function.
- a set of data flow nodes which is the behavior being "covered" by the RT function.
- a named RT unit that can perform the RT function (i.e., the unit bound to).

With these structures, it is possible to maintain the binding information, full and partial, for behavior described in a BIF table.

5.2 Structures for Mappings

A novel feature of the representation we present is that the mapping of abstract behavior to RT functionality is explicitly specified and represented. To accomplish this, data structures are needed that can represent abstractions of RT unit functionality, patterns for recognizing abstract behavior, and mappings of abstract behavior to RT functions. Figure 13 illustrates the organization of the structures used to represent the mappings.

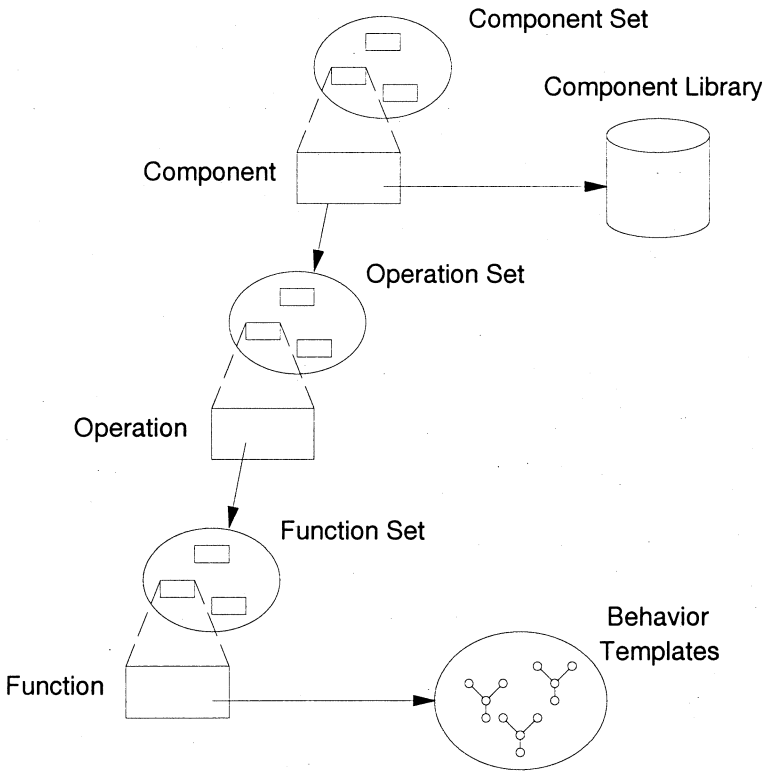


Figure 13: Representation hierarchy for component to behavior mappings.

The representation is a three level hierarchy of sets: a component set, operation sets, and function sets. The component set provides an abstraction for and link to each element of an RT component library. Each component in a component set has the following:

- a unique component name.
- a set of operations.
- a link to RT library component specific information.

For each component there is a unique set of operations. Each operation represents a distinct mode of operation that an RT component can be configured for, i.e., the control line values for each function an RT unit can perform. Each operation in the set has the following:

- a link to the component this operation is for.
- a name. Not necessarily unique, but different from the other operations for the component.
- a set of functions.
- a set of parameters. These specify how the inputs and outputs of the functions map to the RT component inputs and outputs.
- a link to RT operation information.

Each operation has a set of functions. However, each function does not have to be unique to each set, i.e., operation. That is, a function may belong to several function sets. Intuitively, this means it is possible for several different components to perform the same abstract function. Each function has the following:

- a unique name.
- a set of *behavioral templates*. These are patterns for matching the function to representations of behavior.

The specification and usage of the behavioral templates is heavily dependent on the representation used for the abstract behavior. For now, let it suffice, that DFG-like structures will be used as the behavioral templates. It is easy to realize that it is possible to use a simple graph matching algorithm to find possible bindings.

It can be noted that the described structures are very general as far as the representations of behavior and components are concerned. This was intended so that the semantics of the structures are independent of such representations. But for examples of integration of this work with an actual library and behavioral representation please see Appendix A.

References

[AhSU86] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[AnDu92] R. Ang and N. Dutt, "Equivalent Design Representations and Transformations for Interactive Rescheduling," *Proceedings of ICCAD-92*, 1992.

[BeCM92] M-C Bertrand, M. Crastes and A. Mignotte, "VHDL Subset for Control Dominated Synthesis using Macro Block," *Proceedings IFIP Workshop on Control-Dominated Synthesis from a Register-Transfer Level Description*, Grenoble, Sept. 1992.

[CaTa89] R. Camposano and R.M. Tabet, "Design Representation for the Synthesis of Behavioral VHDL Models," in J.A. Darringer, F.J. Rammig, Editors, *Proceedings of the 9th International Symposium on Computer Hardware Description Languages and their Applications*, 1989.

[CaWo91] R. Camposano and W. Wolf, Editors, *High-Level VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1991.

[DuHG90] N. Dutt, T. Hadley, and D. Gajski, "An Intermediate Representation for Behavioral Synthesis," *Proceedings of the 27^{th} Design Automation Conference*, pp. 14–19, 1990.

[DuKi91] N.D. Dutt and J.R. Kipps, "Bridging High-Level Synthesis to RTL Technology Libraries," *Proceedings of the 28th Design Automation Conference*, pp. 526–529, 1991.

[GrKP85] J. Granacki, D. Knapp, and A. Parker, "The ADAM Advanced Design Automation System: Overview, Planner and Natural Language Interface," *Proceedings of DAC*, pp. 727–730, 1985.

[HLSW92] The 1992 High-Level Synthesis Workshop Benchmarks, *available for anonymous ftp from ics.uci.edu* .

[IEEE87] *IEEE Standard VHDL Language Reference Manual*, Institute of Electrical and Electronic Engineers, Inc., 1988.

[JHDu93] Pradip K. Jha, Tedd Hadley, and Nikil D. Dutt, "The GENUS User Manual and C Programming Library" Tech. Report 93-32 University of California at Irvine, April 1993.

[Knap89] D. Knapp, "Synthesis from partial structure," in *Design Methodologies for VLSI and Computer Architecture*, D. Edwards, Ed., Elsevier, pp. 35-51, 1989.

[KnWi92] D. Knapp and M. Winslett, "A Prescriptive Formal Model for Data-Path Hardware," *IEEE Transactions on CAD*, pp. 158–184, Feb., 1992.

[LGCP91] D. Lanneer, G. Goossens, F. Catthoor, M. Panwels, H. Deman, "An Object-oriented Framework Supporting the Full High-level Synthesis Trajectory," *Proceedings of the 10th International Symposium on Computer Hardware Description Languages and their Applications*, pp. 281-301, 1991.

[Mano88] M. Mano, *Computer Engineering: Hardware Design*, Prentice Hall, 1988.

[Marw86] P. Marwedel, "A New Synthesis Algorithm for the MIMOLA Software System," *Proceedings of DAC*, pp. 271–277, 1986.

[MiLD92] P. Michel, U. Lauther and P. Duzy, *The Synthesis Approach to Digital System Design*, Kluwer, 1992.

[OrGa86] A. Orailoglu and D. Gajski, "Flow Graph Representation," *Proceedings of DAC*, pp. 503–509, 1986.

[Rose86] W. Rosenstiel, "Optimizations in High Level Synthesis," *Microprocessing and Microprogramming (18)*, pp. 543–549, 1986.

[RuGB90] E. Rundensteiner, D. Gajski, and L. Bic, "The Component Synthesis Algorithm: Technology Mapping for Register Transfer Descriptions," *Proceedings of ICCAD*, pp. 208–211, 1990.

[Taka85] S. Takagi, "Design Method Based Logic Synthesis," in C.J. Koomen and T. Moto-oka, Editors, *Proceedings of the 7th International Symposium on Computer Hardware Description Languages and their Applications*, 1985.

[Thom90] D. Thomas et al., *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Kluwer, 1990.

A Detailed Description of Data Structures

This appendix describes in detail the actual C data structures used to implement the binding representations described in this report.

A.1 Structures for Mappings

As previously described the representation for the mapping of behavior to RT components is a three level hierarchy of sets: a component set, operation sets, and function sets. The file *BRM.h* contains the declarations of the elements for these sets:

```
typedef struct _RTComp
{
  char *Name;
  g_list *Operations;  /* generic list of BRM_RTOps */
  generic_ptr Comp;
  struct _RTComp *next;
  int flag;  /* general flag */
} BRM_RTComp;

typedef struct _RTOperation
{
  struct _RTComp *RTComp;
  char *Name;
  g_list *Params;
  g_list *RTFunctions;
  generic_ptr info;
} BRM_RTOp;

typedef struct _RTFunction
{
  char *Name;
  g_list *Behav_Templates;
  int flag;  /* general flag */
} BRM_RTFunction;
```

The structure `_RTComp` is used to represent an RT component :

- `Name` is a string of the unique component name.
- `Operations` is a set of operations.
- `Comp` is a link to RT library component specific information. For the GENUS library, this link will be to the data structures for a GENUS component, i.e. a `GENUS_component*`.

For each component there is a unique set of operations. Each operation in the set has the following:

- `RTComp` is a link to the component this operation is for.
- `Name` is a name for this operation.
- `RTFunctions` a set of functions.
- `Params` is a set of parameters. For this implementation, this is a list of name pairs. Each pair matches a variable used in the templates with a GENUS pin.
- `info` is a link to RT operation information. For this implementation, this point to a string with the name of a GENUS function.

Each operation has a set of functions. However, each function does not have to be unique to each set, i.e., operation. That is, a function may belong to several function sets. Intuitively, this means it is possible for several different components to perform the same abstract function. Each function has the following:

- a unique name.
- a set of *behavioral templates*. These are patterns for matching the function to representations of behavior. For this implementation, we intend to use an intermediate format that resembles a data flow graph. So this field is a list of pointers to graph data structures.

The specification and usage of the behavioral templates is heavily dependent on the representation used for the abstract behavior. For now, let it suffice, that DFG-like structures will be used as the behavioral templates. It is easy to realize that it is possible to use a simple graph matching algorithm to find possible bindings.