Allocation of Functional Units from Realistic Component Libraries

Roger Ang and Nikil Dutt

Technical Report 93-47
October 12, 1993

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

rang@ics.uci.edu

Abstract

Existing heuristics and algorithms used in High-Level Synthesis typically assume a direct mapping of hardware description language (HDL) operators to RT units. This assumption simplifies synthesis but requires complex technology mapping to fully utilize the functionality of complex RT components. In this paper, we present an algorithm employing a novel representation scheme to more efficiently map abstract HDL behavior to realistic RT-component behavior. It enables efficient usage of complex databook components, custom designed cells, previously synthesized RT modules, and RT module generators. This approach can be used to customize HLS tools to user-specific RT libraries.

Contents

1 Introduction

Currently, most approaches towards high-level synthesis (HLS) of synchronous circuits from hardware description languages (HDLs) assume a simple scheme for mapping 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. However, these assumptions prevent effective usage of databook components, customized cells and modules, and RT module generators. A simple examination of existing RT-components from standard databooks reveals 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 is a mismatch with HDL semantics which typically have single-function-single-output operators. Consequently, current HLS systems typically ignore some of the the functionality of components, making re-use of previously designed modules inefficient. For example, consider the adder shown in Figure 1a. The behavior of the adder is described abstractly in Figure 1b. If this description were given to a typical HLS tool, it would be likely to generate the circuit shown in Figure 1c. This is because all current approaches still continue to use a fundamental assumption: each implementation component generates a single data result. This assumption prove to be a major obstacle in using realistic models of existing RT components.

As shown in Figure 1, some commonly used RT components can generate multiple data outputs. But current algorithms work on the assumption that only a single data output is produced, and consequently, tools typically will ignore some of the data outputs on multi-output components. If a designer attempts to describe such a function, e.g., a carry output, the synthesis tool will generate additional circuitry rather than utilize the available output. In this paper, we present an algorithm employing a novel representation to more efficiently map abstract behavior to realistic RT-component behavior, to exploit more of the functionality of previously designed components and modules. This approach produces an improved allocation of functional units from a user-specified library of components, making design re-use in high-level synthesis more effective.

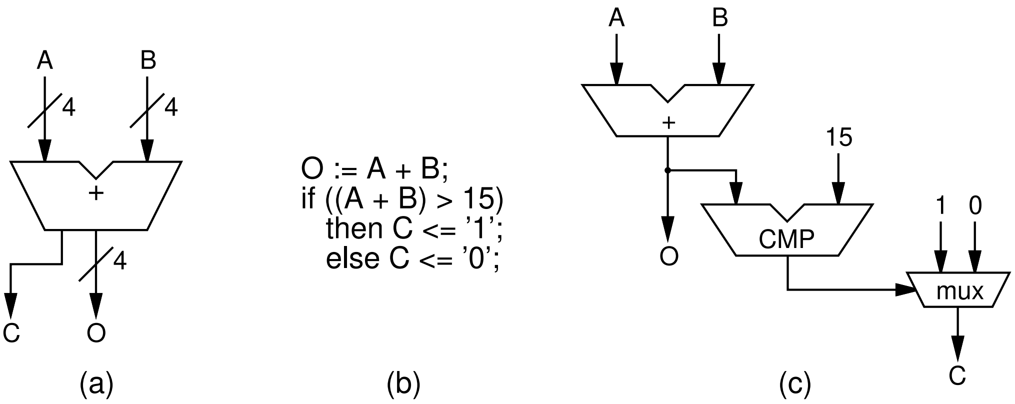

Figure 1: Adder: (a) RT component, (b) behavior in VHDL, (c) circuit synthesized from VHDL.

2 Related Work

The assumptions made in HLS about the way operators map to components are very similar to assumptions made in another form of synthesis: logic synthesis. In logic synthesis, boolean operators map directly to boolean operations which in turn map to logic gates, e.g., "&" maps to an *and* operation which is implemented as with an `and` gate [Keu87] [MaDM93]. For logic synthesis, these assumptions work well because the basic units used for implementation are single-function, single-output components, e.g., `nand` or `nor` gates. But for HLS, these assumptions are too restrictive.

For HLS, the RT components typically used for implementing designs are fundamentally different in concept from the logic gates used in logic synthesis. Many commonly used RT components have multiple *operation modes*, i.e., they can perform different operations in different time steps. Examples of such components are Adder/Subtractors, ALUs, Left/Right Shifters, etc. Consequently, one of the tasks in mapping behavior to RT units in HLS, is matching different types of operations to such multi-function RT units [GCDM93] [RaKu92] [RGB90]. However, all these approaches are still based on the assumption that each RT units generates a single data output.

At a conceptual level, the problem of mapping an HDL model to a set of RT units is very similar to mapping a programming language to a set of microprocessor instructions. Therefore, we are attempting to an approach analogous to the approach described in [GlGr78]. However, the problem we are dealing with is larger in scope. [GlGr78] described a method that worked on a fixed, single processor architecture. To (re)use realistic components in HLS, the allocation/binding problem would be analogous in programming language compilation to performing the same mapping tasks for multiple processors, where each processor may have a different instruction set. [Mar90] [LaMa93] solved a similar problem for HLS using implication rules to match HDL expressions to functions performed by a variety of RT components. But the semantics and usage of the HDL and implication rules in [LaMa93] still assumes that each RT component produces a single data result. While the **RT Functions** used in our model have many similarities to the *component activations* described in [LaMa93] RT Functions have different basic semantics. *Component activations* indicate a particular mode of operation of a particular component. RT Functions are an abstraction of component behavior. Several different components may be able to perform the same RT Function, and a single component may be able to perform multiple RT Functions concurrently. Conceptually, *component activations* and RT Functions are very different abstractions. *Component activations* still seek the traditional HLS goal of directly mapping operators to RT components. In contrast, RT Functions serve as another layer between behavior and RT structure. Operators and operands are matched to RT Functions which are then mapped to RT units.

3 Model

In [AnDu93], we presented a representation capable of describing the mapping of behavioral constructs to multi-function, multi-output RT components. These behavioral constructs are typically derived from an HDL description, and the mapping illustrated in Fig. 2 is needed for the binding of abstract behavioral operations to RT unit functionality. For the work presented in this paper, an important feature of this representation is the RT Data Flow Graph (RTDFG), which contains **RT Functions** that match to clusters of HDL behavior, and link data to the I/O pins of RT units. The RTDFG provides an additional level of abstraction between the abstract HDL behavior and the RT structure. Operators and operands are mapped to RT Functions in the RTDFG which are then matched to RT units. RT Functions correlate input data lifetimes to output data lifetimes in a manner similar to the data flow through an RT unit. The RTDFG represents an unambiguous mapping between the HDL model and the RT structure while enabling greater usage of the functionality of realistic components. Fig. 2 illustrates part of a single state in an design. The RT Functions DEC and DEC_CARRY have been bound to an ALU indicating both functions can be performed simultaneously on that unit.

4 Algorithm for Binding

4.1 Approach and Problem Definition

Our approach maps the semantics of an imperative HDL to the semantics of RT units as defined by the RT Functions. In this way, the problem of binding behavior to realistic functional units can be approached as a 2-phase task. In the first phase of binding, abstract behavior is mapped to an equivalent set of RT Functions. This set describes which RT Functions are to be performed simultaneously. In the second phase, these RT Functions are assigned to RT units. The algorithm we present here is for the second phase of this binding process.

In other words, given:

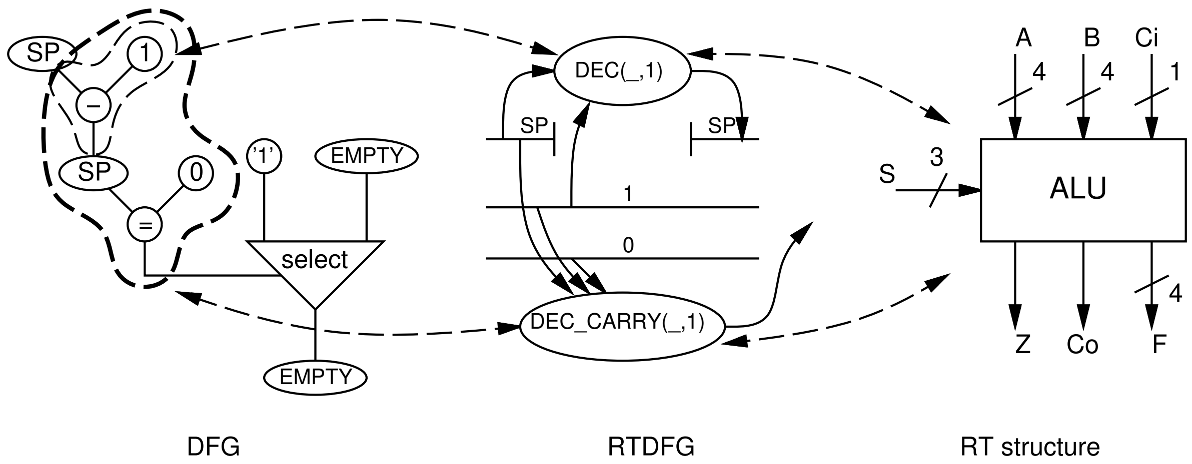

Figure 2: Links between behavior, RTDFG transformation nodes, and RT structure.

L: a user-specified component library consisting of:

> **C**: a set of RT components
>
> **O**: a set of RT operations, each component in **C** can perform one operation at any time
>
> **F**: a set of RT Functions, multiple RT Functions can be performed in a single RT operation in **O**.

This library defines the mappings between RT components, RT operations, and RT Functions. For example, $\mathbf{C} \times \mathbf{O} \rightarrow \mathbf{F}$.

I: an input graph. The first phase of binding will produce this graph which describes which RT Functions of a set, $\mathbf{IF} \subseteq \mathbf{F}$, can be performed simultaneously. We define this graph where:

- for each RT Function in **IF** there is an associated node in the graph,
- and there is an edge between two nodes if
  1. both RT Functions can be performed simultaneously on the same type of RT unit,
  2. or the RT Functions are not to be performed simultaneously and both can be performed by the same type of RT unit.

  This is determined from the information in the library **L**, and the edge between the nodes is labeled with the name of the type of RT unit that can perform the RT Functions.

The input design description can be scheduled or not scheduled. For a description that is not scheduled, the above edges can be determined from data dependencies. Fig. 3a) illustrates this graph.

This algorithm will produce a graph **B** where each node in **B** has an associated RT unit, and a set of RT Functions. That is, this algorithm performs the mapping: $\mathbf{L} \times \mathbf{I} \rightarrow \mathbf{B} \mid$ each node in **B** has

- an RT unit, $\mathbf{c} \in \mathbf{C}$.
- a set of RT Functions, $\mathbf{f} \subseteq \mathbf{IF}$. These are the functions bound to **c**.

4.2 Algorithm Overview

The input graph **I** is a compatibility graph describing how RT Functions can share RT units from the library **L**. This algorithm does a branch-and-bound search for possible solutions, recursively transforming the input graph to find possible bindings of the RT Functions **IF** to RT units **C** . The algorithm will try each node in the graph as a starting point. For each node in the graph, the algorithm will either bind the node to an already allocated RT unit, or try an RT unit allocation according to component-based cost functions. If a partial solution is larger than or redundant to an already found solution, searching of that partial solution is abandoned. Algorithm 4.3 describes the search for possible allocations/bindings of RT units.

FORM_START_LIST() orders the nodes in the graph into a list, START_LIST, so that nodes considered good starting points (or most critical) are searched first. The nodes are rated according to the cost function:

$$\text{Cost} = \text{max. operation cost} \times \text{output bitwidth}$$

Each function associated with a node in the graph can be performed by one or more units in **C**. Each component must be in a particular mode of operation, $\mathbf{o} \in \mathbf{O}$, to perform that function, e.g., shift right, increment, count up. The costs of each RT operation in **O** is given by the user in the form of a table. These costs indicate the relative difficulty of implementing various classes of functions (e.g., arithmetic functions vs. logic functions). If a function can be perform by different RT operations, the most expensive operation is considered for the cost function. These operation costs are independent of the actual bitwidth of the data used in the function, so the output bitwidth of the function is also factored in. This cost function is used to provide a measure of the potential difficulty to implement a given function using a given component library, and the START_LIST is ordered so that the most costly functions are considered first in the search.

BRANCH_ON_BIND() is the main routine that recursively builds possible solutions. RT_FUNCTION(n) refers to the RT Function in **IF** that is associated with node n. ALLOCATE_AND_BIND_NEW_UNIT(n) transforms the graph to allocate an instance of an RT unit in **C** and bind RT_FUNCTION(n) to it. BIND_RT_FUNCTION_TO_UNIT(n) binds the RT Function of node n to an RT unit already allocated to the graph. AREA(`solution`) returns the estimated area cost for the RT units allocated in `solution`. REDUNDANT_SOLUTION(`solution`) determines if the partial solution `solution` is a subset of a possible solution already found, or a partial solution already abandoned. SELECT_BINDABLE_RT_FUNCTION() returns a node from the graph whose RT Function can bind to an already allocated unit.

4.3 Algorithm

Initially, all RT Functions are unbound and no RT units have been allocated, so all nodes in the graph have an associated RT Function (see Fig. 3a). The algorithm begins on a selected node and recursively applies the routine BRANCH_ON_BIND(). The algorithm will try each node as a potential starting point to find alternative solutions. In BRANCH_ON_BIND(), if the RT Function for the selected node cannot bind to an available RT unit, or no RT units are allocated yet, a list of units that can be bound to that function is made. This list is ordered by *RT unit ratings*, which rates how many of the remaining functions in the graph a unit can cover.

$$\text{Unit rating} = \frac{\text{sum of operation costs of nodes that can bind to unit}}{\text{area of unit}}$$

ALLOCATE_AND_BIND_NEW_UNIT() is applied to each of these possible bindings to produce a partial solution (see Fig. 3b). However, if the RT Function for the selected node can bind to an available RT unit, BIND_RT_FUNCTION_TO_UNIT() is applied (see Fig. 3c). In either case, the partial solution produced is abandoned if it is larger than the best solution found so far, or is redundant to another solution.

If no suitable binding or allocation is found, the search of the current partial solution is abandoned. If all the RT Functions have been bound, then the solution is saved if it is smaller than the best solution found. Otherwise, FIND_BINDABLE_RT_FUNCTION() selects which, if any, RT Function should be bound to an available unit. If an RT Function can bind to an available unit, BRANCH_ON_BIND() is applied to the node for the selected function. Else, BRANCH_ON_BIND() is applied to each of the remaining function nodes in the graph.

Algorithm 4.3

```
a: START_LIST = FORM_START_LIST();
b: for all n ∈ START_LIST do BRANCH_ON_BIND(n);

   BRANCH_ON_BIND(n) {
      if RT_FUNCTION(n) can't bind to an available RT unit then
         find possible bindings and order by RT unit rating
c:       for each possible binding do
            possible_solution = ALLOCATE_AND_BIND_NEW_UNIT(n);
            if AREA(possible_solution) > AREA(best_solution) then go to 1; /* try next */
            if REDUNDANT_SOLUTION(possible_solution) then go to 1; /* try next */
            elseif no RT Functions left then
               if AREA(possible_solution) < AREA(best_solution) then
                  save possible_solution as best_solution;
               endif
            else there are RT Functions left,
               n = FIND_BINDABLE_FUNCTION();
               if an RT Function can bind to an RT unit then BRANCH_ON_BIND(n);
               else no RT Function can bind to an RT unit
d:                for each remaining RT Function, n do BRANCH_ON_BIND(n);
               endif
            endif
         endfor
      else RT_FUNCTION(n) can bind to an available RT unit m
e:       possible_solution = BIND_RT_FUNCTION_TO_UNIT(n, m);
         if AREA(possible_solution) > AREA(best_solution) then return; /* stop search */
         if REDUNDANT_SOLUTION(possible_solution) then return; /* stop search */
         elseif no RT Functions left then
            if AREA(possible_solution) < AREA(best_solution) then
               save possible_solution as best_solution;
            endif
         else there are RT Functions left,
            n = FIND_BINDABLE_FUNCTION();
            if an RT Function can bind to an RT unit then BRANCH_ON_BIND(n);
            else no RT Function can bind to an RT unit
               for each remaining RT Function, n do BRANCH_ON_BIND(n);
            endif
         endif
      endif
   }
```

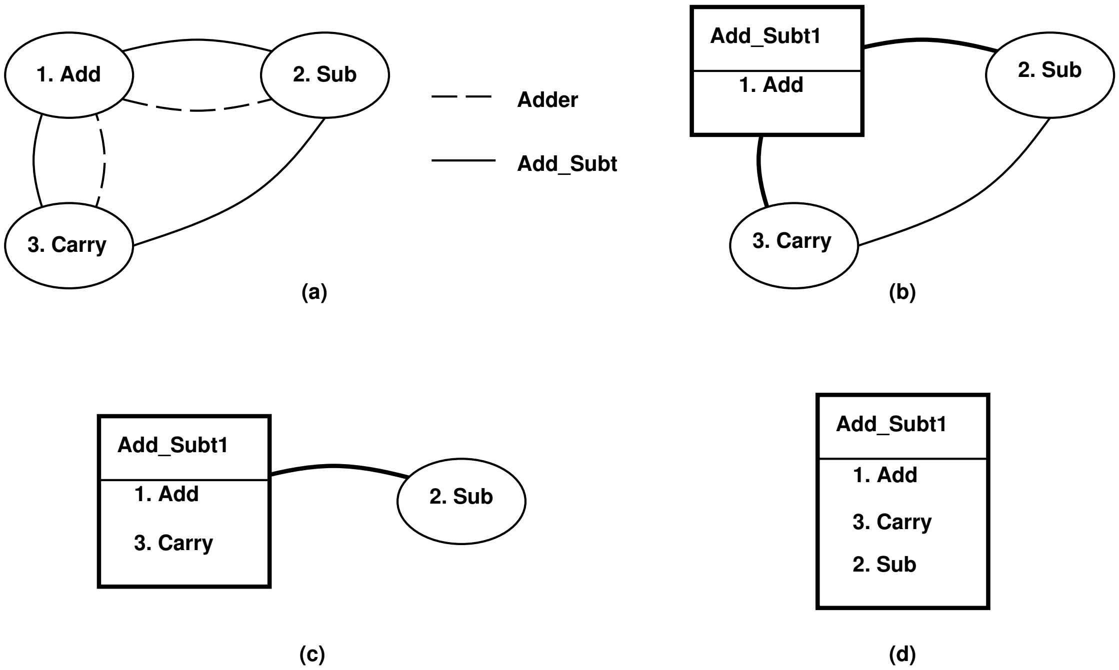

Figure 3: Example bindings. (a) Initial Graph. (b) First possible binding of node 1. (c) Binding of node 3 to Add_Subt1. (d) Binding of node 2 to Add_Subt1 to form solution.

Fig. 3 illustrates the execution Algorithm 4.3. From the initial graph, Fig. 3a, the first iteration of line **b** of the algorithm will apply BRANCH_ON_BIND() on node 1. Since there are RT units allocated yet, BRANCH_ON_BIND() will form a list of possible bindings for node 1. In this case, node 1 can bind to either an Adder/Subtractor or Adder. The first iteration of line **c** of the algorithm will apply ALLOCATE_AND_BIND_NEW_UNIT() to allocate an Adder/Subtractor and bind node 1 to it, producing the graph in Fig. 3b. Then, FIND_BINDABLE_FUNCTION() will select node 3 for the next recursion of BRANCH_ON_BIND(). When BRANCH_ON_BIND() is applied to node 3, node 3 will be bound to the Adder/Subtractor by BIND_RT_FUNCTION_TO_UNIT(), producing the graph in Fig. 3c. Then, FIND_BINDABLE_FUNCTION() will select node 2 for the next recursion of BRANCH_ON_BIND() which will bind node 2 to the Adder/Subtractor (see Fig. 3d).

4.4 Algorithm Complexity

For the following discussion, let n be the number of nodes, or RT functions, in the graph and m be the number of components in the user-specified library.

The fastest case for BRANCH_ON_BIND() is when all the nodes in the graph can only map to a single RT unit. During the call to BRANCH_ON_BIND() in the first iteration of line **b**, there will only one iteration of line **c** for the first node. Subsequent nodes will be recursively bound to the same unit at line **e**. So for the first iteration of line **b**, there will be n calls to BRANCH_ON_BIND(). Following iterations of line **b**, will produce redundant solutions which will be abandoned. That is, the remaining $n-1$ iterations of line **b** will make only $n-1$ calls to BRANCH_ON_BIND(). So, for this case, the algorithm will make $2n-1$ calls to BRANCH_ON_BIND(), i.e., the algorithm runs in $\Omega(n)$ time.

The worst case for BRANCH_ON_BIND() is when each node in the graph can map to every type of RT unit but cannot share a unit with any other node. In other words, there are no edges in the graph though each node can bind to any unit. This problem can be reformulated as having n nodes which can be colored with

m different colors. Obviously, there are m^n different solutions. The algorithm would find all these solutions during the first iteration of line **b** if no "pruning", abandoning search of a solution, occurs. That means all solutions have the same area. In such a case, there would be m^n calls of BRANCH_ON_BIND(), since each call "colors" a node. Following iterations of line **b**, will produce redundant solutions which will be abandoned. That is, the remaining $n-1$ iterations of line **b** will make only $n-1$ calls to BRANCH_ON_BIND(). So theoretically, the algorithm runs in $O(m^n + n)$ time. However, the assumptions made for this scenario are unrealistic.

Intuitively, we can see that the above worst case situation is unrealistic for two reasons:

1. All RT components do not have the same functionality, bit width, area, etc. We expect that a designer will wish to explore the possible use of a variety of RT components with varied functionality. Consequently, it is a more reasonable assumption that each RT function can map to only a small subset of the components in a library.

2. If all allocations for a model were of identical area, there would be no need to apply an algorithm such as this. Consequently, for applicable problems, we expect that the search for some of these solutions can be "pruned".

From these expectations, we can make some assumptions about the expected performance of this algorithm. First, regardless of the size of the component library, each RT function can typically only bind to a small number of components. For example, a 4-bit addition RT function can only map to the set of 4-bit adders, ALUs, etc., regardless of how many different components of different bitwidths are in the library. So for a realistic library, the worst case performance is $O(c^n + n)$ where c is a constant. In addition, while the affects of the pruning in the search are unpredictable, they have a great affect on the amount of search done. Table 1 illustrates the amount of search done on the experimental examples. For these cases, there were less than n^3 calls to BRANCH_ON_BIND(). We believe further experiments will show this algorithm run in n^c time, where c is a constant, for the average case.

		Component Set	
		Synopsys ($m = 90$)	GENUS ($m = 80$)
Example	nodes (n)	calls to BRANCH_ON_BIND()	
facet+	8	34	42
FPA	36	458	364
GCD	4	16	9
2901	23	748	216

Table 1: Number of times BRANCH_ON_BIND() executed on experimental examples.

5 Results

Table 2 shows results comparing our approach against an approach assuming a direct mapping of HDL operators to RT units. Facet+ is a modified version of an example description from [TsSi83], a carry input was added for the additions in the description. FPA is a model of a IEEE standard 32-bit floating point adder/subtractor [HLSWB]. GCD is a description of an 8-bit greatest common divisor circuit [HLSWB]. The AM2901 is a model of a 4-bit ALU [HLSWB].

Each approach allocated units to an unscheduled description from a given library of components. Unit sharing was determined by data dependencies in the model and mutual exclusiveness of conditional branches. These allocations will satisfy an as-soon-as-possible or as-late-as-possible schedule of the description. The results show the total estimated gate counts for the allocated RT units. For the first set, gate counts for the components in the library were derived using Synopsys® 3.0 design tools targeting LSI 10k CMOS technology. The components were either generated from modules available with the Synopsys tools or were synthesized from VHDL models [Synop]. The figures are for the number of 2-input nand/nor gates to implement the allocated units. For the second set, gate counts for the components in the library were derived either using gate estimators in GENUS, a generic component database [JHDu93], or by hand optimized and counted schematics. These figures are for the number of technology independent, equivalent 2-input gates to implement the allocated units.

	Synopsys			GENUS		
Example	Multi-I/O	Single Func.	% diff.	Multi-I/O	Single Func.	% diff.
facet+	514	598	14.0	745	831	10.3
FPA	2865	3366	14.9	3453	4462	22.6
GCD	247	420	41.2	265	423	37.4
AM2901	142	209	32.1	130	202	35.6

Table 2: Allocation results of two approaches and percentage differences.

The savings shown were obtained by exploiting the special functionality of various RT components. For facet+, there were several additions of three variable where the third variable was a carry input. Traditional synthesis would require two adders to accomplish this addition in a single time step. However, our approach recognizes this as a single addition of two inputs with a carry input, resulting in an allocation with two less 4-bit adders. For the FPA, the description included simultaneous *less than* and *equal to* comparisons of the same two 8-bit variables, and an *equal to zero* comparison of a 28-bit variable. Our algorithm recognized that the comparisons of the 8-bit variables could be generated simultaneous by a single comparator, while the direct operator mapping approach allocated a comparator for each comparison. Also, our algorithm recognized that the *equal to zero* comparison could be perform by an ALU in the library which could also perform the 28-bit add and subtract required elsewhere in the design. In contrast, the direct operator mapping approach allocated a 28-bit comparator and a 28-bit adder/subtractor for the same operations. The GCD circuit also had simultaneous *less than* and *not equal* comparisons of the same variables, and once again, our approach found that one less comparator was needed. For the AM2901, the writer of the description need to generate a carry output from the addition of two inputs and a carry input. Again, for the carry input, the addition was written as an addition of three variables. Also, the writer extended the addition variables to be 5-bits wide so that the 5th bit of the output could be used as a carry output. By recognizing these expressions as a special, stylized description of a 4-bit addition with carry input and output, our algorithm allocated a single 4-bit ALU for the addition and subtraction operations. In contrast, the direct operator mapping approach allocated two 5-bit wide Adder/Subtractors for the same operations.

6 Summary and Future Directions

We have presented an algorithm for allocating realistic RT components for the abstract operations within a behavioral description of a digital circuit, enabling effective re-use of previously designed custom components, and customization of synthesis tools to user-specific RT libraries. This algorithm employs a representation of RT units that recognizes their multi-output capability. This enables usage of technology specific RT unit information in HLS, and reduces the need in HLS for complex algorithms for estimation and technology mapping. We have shown that this approach makes a significant improvement, about an average of 25%, over conventional allocation methods when applied to sets of custom RT components.

We intend to continue this work for the mapping of HDL syntax/constructs to RT Functions, and the allocation of interconnection units, registers, and memory units. We also intend to examine how scheduling can be performed with our new representation, and how scheduling concerns can be correlated with allocation and binding and vice versa.

7 Acknowledgments

We gratefully acknowledge support for this work by SRC contract #93-DJ-146 and the U. C. Irvine Graduate and Professional Opportunity Program.

References

[AnDu93] R. Ang and N. Dutt, "A Representation for the Binding of RT-Component Functionality to HDL Behavior," *Proceedings of the Conference on Hardware Description Languages*, pp. 251–266, April 1993.

[GlGr78] R. Glanville, S. Graham "A New Method for Compiler Code Generation," *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages*, pp. 231–240, 1978.

[GCDM93] W. Guerts, F. Catthoor, H. De Man, "Heuristic Techniques for the Synthesis of Complex Functional Units," *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 552–556, 1993.

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

[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.

[Keu87] K. Keutzer "DAGON: Technology Binding and Local Optimization by DAG Matching," *Proceedings of the Design Automation Conference (DAC)*, pp. 617–623, 1987.

[LaMa93] B. Landwehr and P. Marwedel, "Intelligent Library Component Selection and Management in an IP-model based High Level Synthesis System," *submitted to the International Workshop on Logic and Architecture Synthesis* , September 1993.

[MaDM93] F. Mailhot and G. De Micheli, "Algorithms for Technology Mapping Based on Binary Decision Diagrams and on Boolean Operations," *IEEE Transactions on CAD*, pp. 599–620, Vol. 12, May 1993.

[Mar90] P. Marwedel, "Matching System and Component Behaviour in MIMOLA Synthesis Tools," *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 146–156, 1990.

[RaKu92] D. S. Rao and F. J. Kurdahi, "System Modeling for High-Level Synthesis Using Regularity Extraction," *Proceeding of* 6^{th} *Workshop on High Level Synthesis*, pp. 267–272, 1992.

[RGB90] E. Rundensteiner, D. Gajski, L. Bic, "The Component Synthesis Algorithm: Technology Mapping for Register Transfer Descriptions," *Proceedings of ICCAD*, pp. 208–211, 1990.

[Synop] *Synopsys DesignWare Databook, Version 3.0*, Synopsys, Inc., Dec. 1992.

[TsSi83] C. Tseng and D. Siewiorek "Facet: A Procedure for Automated Synthesis of Digital Systems" *Proceedings of the Design Automation Conference (DAC)*, pp. 490–496, 1983.

A Pertinent VHDL portions of examples

A.1 Facet+

```
entity FACET is
   port (V6,V4,V10,V1, V2: in BIT_VECTOR(3 downto 0);
         cin: in BIT
        );
end FACET;

-- VSS: design_style BEHAVIORAL
```

```
architecture EX of FACET is

begin

process

   variable R1,R2,R3,R4,R5,R6,R7: BIT_VECTOR(3 downto 0);
   variable cout : BOOLEAN;
begin

     R1 := V1;
     R2 := V2;
     R3 := V4;
     R4 := V6;
     R5 := V10;
     R7 := R1 + R2 + ("000" & cin);
     R6 := R4 - R3;
     R2 := R4 * R7;
     R2 := R2 + R1 + ("000" & cin);
     R7 := R7 + R6 + ("000" & cin);
     R6 := R5 / R6;
     R2 := R2 or R1;
     R1 := R7 and R6;

end process;

end EX;
```

A.2 Floating Point Adder

Simultaneous 8-bit comparisons:

```
-- Step 3: Compare exponents.  Swap the operands if exp1 < exp2
--         or if (exp1 = exp2 AND mnt1 < mnt2)
swapped  := false;
IF (exp1 < exp2 OR (exp1 = exp2 AND mnt1 < mnt2)) THEN
    swap_exponents(exp1, exp2);
    swap_mantissas(mnt1, mnt2);
    swap_bits(sign1, sign2);
    swapped := true;
END IF;
exp_diff := exp1 - exp2;
exp_result := exp1;
sign_result := sign1;
```

28-bit addition or subtraction, and equal to zero comparison:

```
--  Step 5: Add or subtract the mantissas.
IF (operation = subtract) THEN
    IF swapped THEN
        sign_result := NOT(sign_result);
    END IF;
    IF (sign1 = sign2) THEN
        true_subtract := true;
    ELSE
        true_subtract := false;
    END IF;
ELSIF (sign1 /= sign2 and operation = add) THEN
```

```
            true_subtract := true;
        ELSE
            true_subtract := false;
        END IF;


        IF (special_flag = false) THEN
            IF true_subtract THEN
                prelim_result := addend1 - addend2; --**
                overflow := ((addend1(27) and
                              (addend2(27) nor prelim_result(27))) or
                             (addend1(27) nor
                              (addend2(27) nand prelim_result(27))));
            ELSE
                prelim_result := addend1 + addend2;  --**
                overflow := (((addend1(27) nor addend2(27)) and
                             prelim_result(27)) or
                             ((addend1(27) and addend2(27)) and
                             not(prelim_result(27))));
            END IF;
        END IF;


        -- Step 6: Normalize the result.
        IF (special_flag = false) THEN
            IF (prelim_result = "0000000000000000000000000000") AND
                (overflow = false) THEN  --**
                sign_result := '0';
                exp_result:= 0;
                zero_res <= '1';
            ELSIF overflow THEN
                IF exp_result >= (MAX_EXP - 1) THEN
                    prelim_result := "0000000000000000000000000000";
                    exp_result := MAX_EXP;
                    pos_inf_res <= '1';
                ELSE
                    exp_result := exp_result + 1;
                    right_shift(reg_val => prelim_result, shift_amt => 1);
                END IF;
            ELSE
            -- Shift left until normalized.  Normalized when the value to the
            -- left of the decimal point is 1.

                normalize(prelim_result((ADDEND_LENGTH-2) downto 0),
                          exp_result);
            END IF;
        END IF;
```

A.3 Greatest Common Divisor

```
entity gcd is
 port (xi, yi : in bit16;
       rst    : in bit;
       ou     : out bit16);
end gcd;

architecture behavior of gcd is
begin
```

```
  process
   variable x,y: integer;

  begin
    wait until (rst = '0');

    x := xi;
    y := yi;

    while (x /= y) loop
      if (x < y)
        then y:=y-x;
        else x:=x-y;
      end if;
    end loop;

    ou <= x;
  end process;

end behavior;
```

A.4 AM2901

```
--   TO FACILITATE COMPUTATION OF CARRY-OUT "C4", WE EXTEND THE CHOSEN
--   ALU OPERANDS "RE" AND "S" (4 BIT OPERANDS) BY 1 BIT IN THE MSB POSITION.

--   THUS THE EXTENDED OPERANDS "R_EXT" AND "S_EXT" (5 BIT OPERANDS) ARE
--   FORMED AND ARE USED IN THE ALU OPERATION. THE EXTRA BIT IS SET TO '0'
--   INITIALLY. THE ALU'S EXTENDED OUTPUT ( 5 BITS LONG) IS "result".

--   IN THE ADD/SUBTRACT OPERATIONS, THE CARRY-INPUT "C0" (1 BIT) IS EXTENDED
--   BY 4 BITS ( ALL '0') IN THE MORE SIGNIFICANT BITS TO MATCH ITS LENGTH TO
--   THAT OF "R_ext" AND "S_ext". THEN, THESE THREE OPERANDS ARE ADDED.

--   ADD/SUBTRACT OPERATIONS ARE DONE ON 2'S COMPLEMENT OPERANDS.

        R_ext := '0' & RE;
        S_ext := '0' & S;

        case I(5 downto 3) is
           when "000" =>
                        result := R_ext + S_ext + ("0000" & C0);
           when "001" =>
                        result := S_ext - S_ext - ("0000" & not(C0));
           when "010" =>
                        result := R_ext - S_ext - ("0000" & not(C0));

                 .
                 .
                 .
```

B Extended Information

B.1 Lists of Library Components

Components described with GENUS:

1-bit Adder (ripple carry)
1-bit Adder (carry look-ahead)
1-bit Adder/Subtractor (ripple carry)
1-bit 2-input And gate
1-bit inverter
1-bit LU
1-bit 2-input Multiplexor
1-bit 2-input Nand gate
1-bit 2-input Nor gate
1-bit 2-input Or gate
1-bit D register
1-bit Left/Right Shifter
1-bit 2-input Exclusive Nor gate
1-bit 2-input Exclusive Or gate
4-bit Adder (ripple carry)
4-bit Adder (carry look-ahead)
4-bit Adder/Subtractor (ripple carry)
4-bit ALU
4-bit 2-input And gate
4-bit Barrel Shifter
4-bit 6-function, 2-input Comparator
4-bit Counter
2-to-4 Decoder
4-bit Divider (Non-Restoring)
4-to-2 Encoder
4-bit in/ 1-bit out And gate
4-bit in/ 1-bit out Nand gate
4-bit in/ 1-bit out Nor gate
4-bit in/ 1-bit out Or gate
4-bit in/ 1-bit out Exclusive Nor gate
4-bit in/ 1-bit out Exclusive Or gate
4-bit LU
4-bit Multiplier (Carry-Save Array)
4-bit 2-input Multiplexor
4-bit 2-input Nand gate
4-bit 2-input Nor gate
4-bit 2-input Or gate
4-bit D register
4-bit Left/Right Shifter
4-bit 2-input Exclusive Nor gate
4-bit 2-input Exclusive Or gate
5-bit Adder (ripple carry)
5-bit Adder (carry look-ahead)
5-bit Adder/Subtractor (ripple carry)
5-bit Adder/Subtractor (carry look-ahead)
5-bit Subtractor (ripple carry)
5-bit Subtractor (carry look-ahead)
5-bit 6-function, 2-input Comparator
8-bit Adder (ripple carry)
8-bit Adder (carry look-ahead)
8-bit Adder/Subtractor (ripple carry)
8-bit ALU
8-bit 2-input And gate
8-bit Barrel Shifter
8-bit 6-function, 2-input Comparator
8-bit Counter
3-to-8 Decoder
8-bit Divider (Non-Restoring)
8-to-3 Encoder
8-bit in/ 1-bit out And gate
8-bit in/ 1-bit out Nand gate
8-bit in/ 1-bit out Nor gate
8-bit in/ 1-bit out Or gate
8-bit in/ 1-bit out Exclusive Nor gate
8-bit in/ 1-bit out Exclusive Or gate
8-bit LU
8-bit Multiplier (Carry-Save Array)
8-bit 2-input Multiplexor
8-bit 2-input Nand gate
8-bit 2-input Nor gate
8-bit 2-input Or gate
8-bit D register
8-bit Left/Right Shifter
8-bit 2-input Exclusive Nor gate
8-bit 2-input Exclusive Or gate
24-bit 6-function, 2-input Comparator
28-bit Barrel Shifter
28-bit ALU
28-bit Adder/Subtractor (ripple carry)
28-bit 6-function, 2-input Comparator

Components developed with Synopsys tools:

1-bit Adder (ripple carry)
1-bit Adder (carry look-ahead)
1-bit Adder/Subtractor (ripple carry)
1-bit Adder/Subtractor (carry look-ahead)
1-bit Subtractor (ripple carry)
1-bit Subtractor (carry look-ahead)
1-bit 2-input And gate
1-bit inverter
1-bit 2-input Multiplexor
1-bit 2-input Nand gate
1-bit 2-input Nor gate
1-bit 2-input Or gate
1-bit D register
1-bit Left/Right Shifter
1-bit 2-input Exclusive Nor gate
1-bit 2-input Exclusive Or gate
4-bit Adder (ripple carry)
4-bit Adder (carry look-ahead)
4-bit Adder/Subtractor (ripple carry)
4-bit Adder/Subtractor (carry look-ahead)
4-bit Subtractor (ripple carry)
4-bit Subtractor (carry look-ahead)
4-bit ALU
4-bit 2-input And gate
4-bit Barrel Shifter
4-bit 6-function, 2-input Comparator
4-bit Counter
2-to-4 Decoder
4-bit Divider (Non-Restoring)
4-to-2 Encoder
4-bit in/ 1-bit out And gate
4-bit in/ 1-bit out Nand gate
4-bit in/ 1-bit out Nor gate
4-bit in/ 1-bit out Or gate
4-bit in/ 1-bit out Exclusive Nor gate
4-bit in/ 1-bit out Exclusive Or gate
4-bit LU
4-bit Multiplier (Wallace Dadda)
4-bit Multiplier (Carry-Save Array)
4-bit 2-input Multiplexor
4-bit 2-input Nand gate
4-bit 2-input Nor gate
4-bit 2-input Or gate
4-bit D register
4-bit Left/Right Shifter
4-bit 2-input Exclusive Nor gate
4-bit 2-input Exclusive Or gate
5-bit Adder (ripple carry)
5-bit Adder (carry look-ahead)
5-bit Adder/Subtractor (ripple carry)
5-bit Adder/Subtractor (carry look-ahead)
5-bit Subtractor (ripple carry)
5-bit Subtractor (carry look-ahead)
5-bit 6-function, 2-input Comparator
8-bit Adder (ripple carry)
8-bit Adder (carry look-ahead)
8-bit Adder/Subtractor (ripple carry)
8-bit Adder/Subtractor (carry look-ahead)
8-bit Subtractor (ripple carry)
8-bit Subtractor (carry look-ahead)
8-bit ALU
8-bit 2-input And gate
8-bit Barrel Shifter
8-bit 6-function, 2-input Comparator
8-bit Counter
3-to-8 Decoder
8-bit Divider (Non-Restoring)
8-to-3 Encoder
8-bit in/ 1-bit out And gate
8-bit in/ 1-bit out Nand gate
8-bit in/ 1-bit out Nor gate
8-bit in/ 1-bit out Or gate
8-bit in/ 1-bit out Exclusive Nor gate
8-bit in/ 1-bit out Exclusive Or gate
8-bit LU
8-bit Multiplier (Wallace Dadda)
8-bit Multiplier (Carry-Save Array)
8-bit 2-input Multiplexor
8-bit 2-input Nand gate
8-bit 2-input Nor gate
8-bit 2-input Or gate
8-bit D register
8-bit Left/Right Shifter
8-bit 2-input Exclusive Nor gate
8-bit 2-input Exclusive Or gate
24-bit 6-function, 2-input Comparator
28-bit Barrel Shifter
28-bit ALU
28-bit Adder/Subtractor (ripple carry)
28-bit 6-function, 2-input Comparator

B.2 Allocation Results

The following are the component allocations for the listed examples. Also, numbers are given for the estimated gate counts, the number of calls of BRANCH_ON_BIND(), the number of graph nodes stored (these were the complete and partial graphs stored for redundancy checking), and the number of complete solutions found.

B.2.1 Using GENUS library

Facet+:

	Multi-I/O allocation	Single output allocation
Allocation	4-bit Multiplier 4-bit Divider 4-bit Adder 4-bit Adder/Subtractor 4-bit 2-input Or gate 4-bit 2-input And gate	4-bit Multiplier 4-bit Divider 3 4-bit Adder 4-bit Adder/Subtractor 4-bit 2-input Or gate 4-bit 2-input And gate
Gate count	745	831
BRANCH_ON_BIND() calls	42	180
Nodes stored	201	1203
Solutions found	2	3

Floating Point Adder:

	Multi-I/O allocation	Single output allocation
Allocation	28-bit ALU 28-bit Barrel Shifter 8-bit Adder/Subtractor 8-bit 6-function Comparator 24-bit 6-function Comparator 3 1-bit 2-input And gate 1-bit 2-input Or gate 1-bit Inverter 1-bit 2-input Exclusive Or gate 1-bit 2-input Exclusive Nor gate	28-bit Adder/Subtractor 28-bit Barrel Shifter 8-bit Adder/Subtractor 2 8-bit 6-function Comparator 24-bit 6-function Comparator 28-bit 6-function Comparator 6 1-bit 2-input And gate 1-bit 2-input Or gate 1-bit Inverter 2 1-bit 2-input Nor gate 1-bit 2-input Exclusive Or gate 1-bit 2-input Exclusive Nor gate 1-bit 2-input Nand gate
Gate count	3453	4462
BRANCH_ON_BIND() calls	364	336
Nodes stored	832	690
Solutions found	2	1

Greatest Common Divisor:

	Multi-I/O allocation	Single output allocation
Allocation	8-bit Adder/Subtractor 8-bit 6-function Comparator	8-bit Adder/Subtractor 2 8-bit 6-function Comparator
Gate count	265	423
BRANCH_ON_BIND() calls	9	10
Nodes stored	5	6
Solutions found	1	1

AM2901:

	Multi-I/O allocation	Single output allocation
Allocation	4-bit ALU 4-bit 2-input Exclusive Nor gate 4-bit 2-input Or gate 4-bit 2-input And gate 4-bit in/1-bit out Nand gate 5 1-bit 2-input And gate 3 1-bit 2-input Or gate 1-bit inverter 4-bit in/1-bit out Nor gate	2 5-bit Adder/Subtractor 2 4-bit 2-input And gate 2 4-bit 2-input Or gate 4-bit 2-input Exclusive Nor gate 4-bit 2-input Exclusive Or gate 3 1-bit inverter 2 1-bit 2-input Exclusive Or gate 6 1-bit 2-input And gate 4-bit in/1-bit out Nand gate 3 1-bit 2-input Or gate 4-bit in/1-bit out Nor gate
Gate count	130	202
BRANCH_ON_BIND() calls	216	254
Nodes stored	2032	5147
Solutions found	3	4

B.2.2 Using Synopsys library

Facet+:

	Multi-I/O allocation	Single output allocation
Allocation	4-bit Multiplier 4-bit Divider 4-bit Adder 4-bit Adder/Subtractor 4-bit 2-input Or gate 4-bit 2-input And gate	4-bit Multiplier 4-bit Divider 3 4-bit Adder 4-bit Adder/Subtractor 4-bit 2-input Or gate 4-bit 2-input And gate
Gate count	514	598
BRANCH_ON_BIND() calls	34	193
Nodes stored	392	3879
Solutions found	2	2

Floating Point Adder:

	Multi-I/O allocation	Single output allocation
Allocation	28-bit ALU 28-bit Barrel Shifter 8-bit Adder/Subtractor 8-bit 6-function Comparator 24-bit 6-function Comparator 3 1-bit 2-input And gate 1-bit 2-input Or gate 1-bit Inverter 1-bit 2-input Exclusive Or gate 1-bit 2-input Exclusive Nor gate	28-bit Adder/Subtractor 28-bit Barrel Shifter 8-bit Adder/Subtractor 2 8-bit 6-function Comparator 24-bit 6-function Comparator 28-bit 6-function Comparator 6 1-bit 2-input And gate 1-bit 2-input Or gate 1-bit Inverter 2 1-bit 2-input Nor gate 1-bit 2-input Exclusive Or gate 1-bit 2-input Exclusive Nor gate 1-bit 2-input Nand gate
Gate count	2865	3366
BRANCH_ON_BIND() calls	458	535
Nodes stored	634	690
Solutions found	2	1

Greatest Common Divisor:

	Multi-I/O allocation	Single output allocation
Allocation	8-bit Adder/Subtractor 8-bit 6-function Comparator	8-bit Adder/Subtractor 2 8-bit 6-function Comparator
Gate count	247	420
BRANCH_ON_BIND() calls	16	19
Nodes stored	13	15
Solutions found	2	2

AM2901:

	Multi-I/O allocation	Single output allocation
Allocation	4-bit ALU 4-bit 2-input Exclusive Nor gate 4-bit 2-input Or gate 4-bit 2-input And gate 4-bit in/1-bit out Nand gate 5 1-bit 2-input And gate 3 1-bit 2-input Or gate 1-bit inverter 4-bit in/1-bit out Nor gate	2 5-bit Adder/Subtractor 2 4-bit 2-input And gate 2 4-bit 2-input Or gate 4-bit 2-input Exclusive Or gate 4-bit 2-input Exclusive Nor gate 3 1-bit inverter 2 1-bit 2-input Exclusive Or gate 6 1-bit 2-input And gate 4-bit in/1-bit out Nand gate 3 1-bit 2-input Or gate 4-bit in/1-bit out Nor gate
Gate count	142	209
BRANCH_ON_BIND() calls	748	471
Nodes stored	3938	9898
Solutions found	12	4