An Algorithm for the Allocation of Functional Units from Realistic RT Component Libraries

Roger Ang
rang@ics.uci.edu

Nikil Dutt
dutt@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine
Irvine, California USA 92717

Abstract

Existing algorithms in High-Level Synthesis (HLS) typically assume a direct mapping of hardware description language (HDL) operators to RT units. This assumption simplifies synthesis to generic RT components, but prevents effective use of complex databook components, custom designed cells, previously synthesized RT modules, and RT module generators. In this paper, we present an algorithm for allocation in HLS for reuse of existing RT-level components. This approach can be used to customize HLS tools to user-specific RT libraries. Our experiments show improvements of 10–37% in area over conventional approaches.

1 Introduction

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 HDL operator directly maps to a single RT operation which in turn can be performed by a few generic 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. While this assumption simplifies synthesis by the use of simple models of generic RT components, if shifts the task of physical component binding to a later stage of technology mapping that can become very complex for datapath components. Such an approach of delayed technology mapping at the Boolean level can also prevent effective reuse of available, previously designed components such as 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 that are multi-functional and have multiple outputs, with several functions performed simultaneously. Such components produce several outputs in a single state, much like a general functional mapping of inputs to several outputs. Unfortunately, this is a mismatch with HDL semantics which typically have single-function-single-output operators. Consequently, current HLS systems attempting to use such components will typically ignore some of their functionality leading to suboptimal usage of RT components.

Consider the adder shown in Figure 1(a). The behavior of the adder is described abstractly in Figure 1(b). If this description were given to a typical HLS tool, it would be likely to generate the circuit shown in Figure 1(c). This is because current approaches still continue to use a fundamental assumption: each implementation component generates a single data result. Although logic optimization can be applied to the resulting design, this decomposition of the functionality to the logic gate level is counterproductive to design reuse at the RT-level. If synthesis is targeting reuse of RT level components that are automatically generated, previously designed or synthesized, it would not make sense to "reoptimize" these components that already have well-characterized and predictable design characteristics such as timing. In addition, the redundancies that can be eliminated through design reuse are more obvious at the behavioral level than they would be at the logic level.

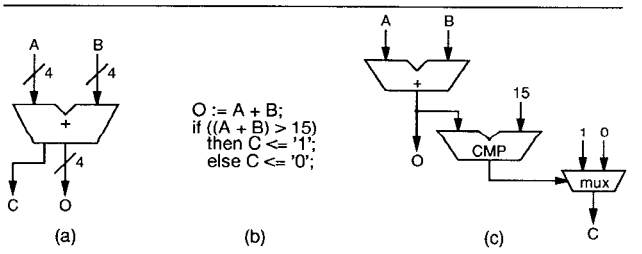

Figure 1: Adder: (a) RT component, (b) behavior in VHDL, (c) circuit synthesized from VHDL.

Our work is a step towards a synthesis approach that attempts to use previously designed or generic components effectively at the behavioral/algorithmic level of synthesis. The underlying models for compo-

0-8186-5785-5/94 $03.00 © 1994 IEEE

nent behavior and how they map to HDLs are geared towards reuse of customized RT components. In this paper, we present an allocation algorithm employing this novel representation to more efficiently map abstract behavior to realistic RT-component behavior. Compared to traditional approaches, this algorithm exploits more of the functionality of previously designed components and modules, making design reuse in high-level synthesis more effective. Consequently, this algorithm produces an improved allocation of functional units from a user-specified library of components.

The rest of this paper is organized as follows: Section 2 describes related work. Section 3 describes our design model. Section 4 outlines the allocation algorithm. Section 5 presents our experiments and results. Section 6 concludes with a summary.

2 Related Work

Traditionally, 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 map to generic logic gates. These generic gates can then be mapped to a technology-specific implementation [8] [9]. For example, "and" maps to an *and* operation implemented as a generic **and** gate, which can map to a 0.5 micron CMOS implementation using **nor** gates. For logic synthesis, the assumptions used 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.

The RT components typically used for implementing designs in HLS 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 and Left/Right Shifters. 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. Several approaches have been used in attempts to solve this mapping problem. In [12], branch-and-bound search was used to cluster generic operations into multifunction nodes which would be performed by generic multifunction units. [4] used graph matching techniques to construct "application specific units" (ASUs) to be used, for example, in the datapath of a DSP circuit. These units were constructed from a predefined library of "abstract building blocks" (ABBs), e.g., an adder/subtractor. A mechanism was defined to map behavior to the ABBs to build the ASUs. A similar idea was used in [10] but for a different goal: generating microcode for a programmable architecture. This system demonstrated a retargetable mapper for matching behavior to a given architecture. However, all these approaches are still based on the assumption that each RT unit 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, as [10] illustrates. Therefore, we are attempting an approach analogous to the approach described in [3]. This approach described a method that used a table to map intermediate code to a specific microcode implementation on a fixed, single processor architecture. However, the problem we are dealing with is larger in scope. For HLS, the allocation/binding problem would be analogous to performing the same tasks for multiple processors, where each processor may have a different instruction set. [7] extends this idea for synthesis. Instead of a table, this work proposes a library of "parts." Each part may be able to perform several functions and each function has a subgraph representation. The proposed algorithm used graph matching to map behavior to generic parts to be generated later. Along this same line of thinking, [11] proposed a "template" library and a regularity extraction algorithm that may be used for synthesis. But in both of these proposed methods, the libraries are not sufficiently defined to deal with components that generate multiple data outputs. In our work, we use the ideas of having a component library with templates for the component, define this library for multi-function, multi-output components, and demonstrate how such a library may be used to allocate instances of components for a given behavioral description.

3 Model

In [1], we presented a representation capable of describing the mapping of behavioral constructs to multi-function, multi-output RT components. This representation encompassed the mappings between three sets of entities:

1. Intermediate behavioral representation — the internal representation used by a synthesis tool for circuit specifications, e.g., a Control/Data Flow Graph (CDFG). This is typically derived from parsing of an HDL.

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 *RT functions*, which are functional abstractions for each data output of the components.

3. RT structure — instances of components. Each RT component is characterized by sets of RT functions that describe modes of operation for the RT component.

The mapping between these structures, illustrated in Figure 2, is needed for the binding of abstract behavioral operations to RT unit functionality. The RT Data Flow Graph (RTDFG) provides an additional level of abstraction between the abstract HDL behavior and the RT structure, in order to manipulate this

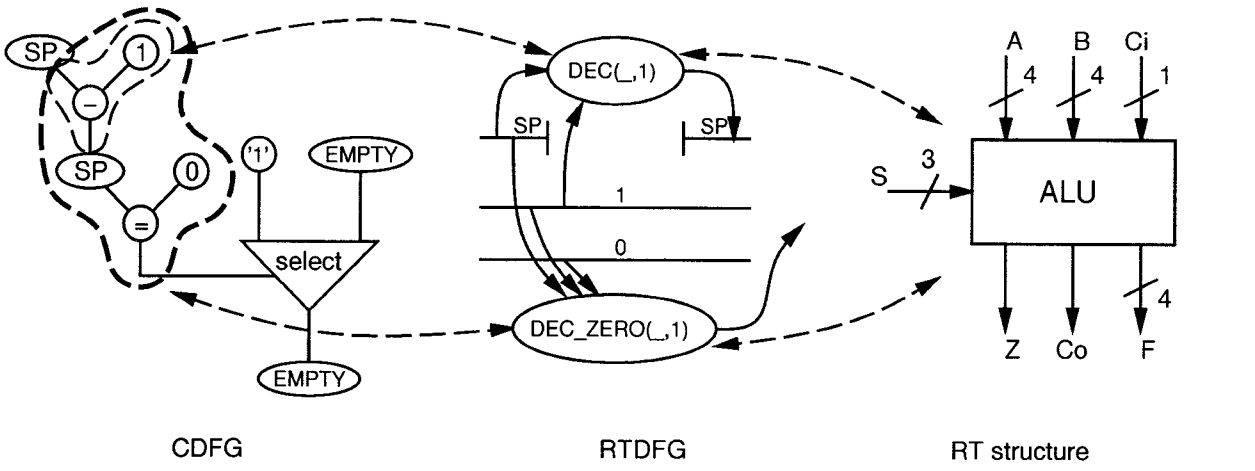

Figure 2: Links between behavior represented as a CDFG, RTDFG transformation nodes, and RT structure.

complex mapping. In the RTDFG, each node of the graph has an associated RT Function. Each node can be matched to a cluster of HDL behavior, as well as linked to an instance of an RT component to associate a component operation to that unit and data values to its pins. Figure 2 illustrates part of a single state in a design. The RT Functions DEC and DEC_ZERO have been bound to an ALU indicating both functions can be performed simultaneously on that unit.

4 Algorithm

Using the representation described in the previous section, the problem of binding behavior to 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. This set is the input for the second phase of binding. In this second phase, these RT Functions are assigned to RT units. The algorithm we present here is for the second phase of this binding process. The algorithm does a branch-and-bound search for possible solutions, recursively transforming the input graph to find possible bindings of the RT Functions to RT Units. The algorithm will try each node in the graph as a starting point. For each node in the graph, the algorithm will then 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 worse than or redundant to an already found solution, further consideration of that partial solution is abandoned.

Algorithm A describes the procedure for the search of possible allocations/bindings of RT units. During the first phase of binding, a set describing which RT Functions can be performed simultaneously is produced. We define this set as a graph **G** where, initially,

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

The edge between the nodes is labeled with the name of the type of RT unit that can perform the RT Functions. Figure 3(a) shows a sample graph.

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.

FORM_START_LIST(G) orders the nodes in the graph G 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:

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

This cost function is used to provide a measure of the potential difficulty to implement a given function using a given component library.

Each function associated with a node in the graph can be performed by one or more units from a component library. Each component must be in a particular mode of operation to perform that function, e.g., shift right, increment, count up. The costs of these *unit operations* 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 performed by different *unit 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.

In algorithm A, BRANCH_ON_BIND(n) is the main procedure that recursively builds possible solutions. RT_FUNCTION(n) is the RT Function associated with node n. The graph transformation

Algorithm A:

```
START_LIST = FORM_START_LIST(G),
for all n ∈ START_LIST do BRANCH_ON_BIND(n);

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

ALLOCATE_AND_BIND_NEW_UNIT(n) allocates a new RT unit and binds the RT Function of node n to that unit. BIND_RT_FUNCTION_TO_UNIT(n) binds the RT Function of node n to an already allocated unit. AREA(`solution`) is the estimated area cost for the RT units allocated to `solution`. REDUNDANT_SOLUTION(`solution`) determines if `solution`, the current partial solution, is a subset of a possible solution already found. FIND_BINDABLE_RT_FUNCTION() returns a node from the graph whose RT Function can bind to an already allocated unit.

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 Figure 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

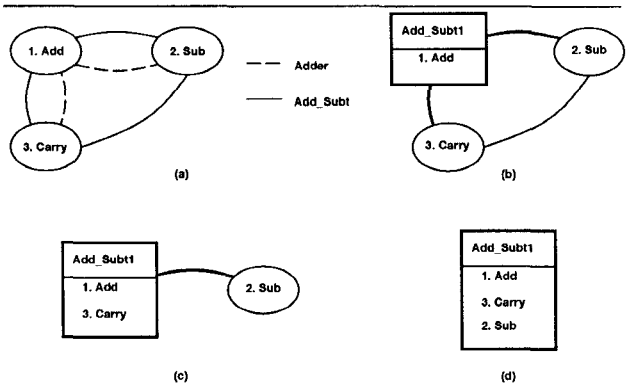

Figure 3: Example bindings. (a) Initial Graph. (b) First possible allocation and binding of node 1. (c) Binding of node 3 to Add_Subt1. (d) Binding of node 2 to Add_Subt1 to form solution.

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 Figure 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 Figure 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 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.

On first examination, the theoretical complexity of algorithm A seems prohibitive. In the worst case, algorithm A has a complexity of $m^n + n$, where n is the number of nodes, or RT functions, in the graph and m is the number of components in the user-specified library. However, such a case can only occur if every RT function in the graph could be performed by any RT component in the library. From a practical standpoint, large instances of such a case will never happen because each RT function typically can only be performed by a few types of components. Consequently, algorithm A ran on the order of n^3 complexity for the experiments we have run. We believe further experiments will show this algorithm runs in the average case in n^c time, where c is a constant.

	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 1: Allocation results (in 2-input nand/nor gates) of two approaches and percentage differences.

5 Experiments

We conducted experiments with this algorithm using two component sets. For the first set, the components were either generated from modules available with the Synopsys tools or were synthesized from VHDL models [13]. Gate counts for the components in the library were derived using Synopsys® 3.0 design tools targeting LSI® 1.0 micron CMOS technology. The figures represent the equivalent number of 2-input nand/nor gates required to implement the allocated units. For the second component set, gate counts for the components in the library were derived either using gate estimators in GENUS, a generic component database [6], or by hand optimized and counted schematics. These numbers represent the number of technology independent, equivalent 2-input logic gates to implement the allocated units.

Table 1 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 [14], 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 [5]. GCD is a description of an 8-bit greatest common divisor circuit [5]. The AM2901 is a model of a 4-bit ALU [5].

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. More detailed descriptions and discussions of the algorithm, component libraries, and examples can be found in [2].

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 fewer 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 needed 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.

Our experiments on these examples show significant savings in area (between 10% to 37%). We believe our approach can be used effectively to allocate realistic, user-defined library components efficiently.

6 Summary

We have presented an algorithm for allocating RT components for the abstract operations within a behavioral description of a digital circuit. The 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 for complex estimation and characterization, and technology mapping in HLS. It also enables effective reuse of previously designed custom components, and customization of synthesis tools to user-specific RT libraries. 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.

Acknowledgements

This work was partially supported by SRC contract 93-DJ-146, a UCI GPOP fellowship, and a UCI Faculty Research grant.

References

[1] 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.

[2] R. Ang and N. Dutt, "Allocation of Functional Units from Realistic Component Libraries," Tech. Report 93-47, University of California at Irvine, October 1993.

[3] 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.

[4] W. Guerts, F. Catthoor, H. De Man, "Heuristic Techniques for the Synthesis of Complex Functional Units," *Proceedings of EDAC*, pp. 552–556, 1993.

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

[6] 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.

[7] M. Kahrs, "Matching a parts library in a silicon compiler," *Proceedings of ICCAD*, pp. 169–172, 1986.

[8] K. Keutzer "DAGON: Technology Binding and Local Optimization by DAG Matching," *Proceedings of DAC*, pp. 617–623, 1987.

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

[10] P. Marwedel, "Tree-Based Mapping of Algorithms to Predefined Structures," *Proceedings of ICCAD*, pp. 586–593, 1993.

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

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

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

[14] C. Tseng and D. Siewiorek "Facet: A Procedure for Automated Synthesis of Digital Systems" *Proceedings of DAC*, pp. 490–496, 1983.