Title
Scheduling for reuse of datapath components in the interactive synthesis environment

Authors
Ang, Roger
Dutt, Nikil

Publication Date
1995-08-25

Peer reviewed

Scheduling for Reuse of Datapath Components in the Interactive Synthesis Environment*

Roger Ang and Nikil Dutt

Technical Report 95-36
August 25, 1995

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 824-8059

rang@ics.uci.edu

Abstract

This report describes an implementation of a scheduling algorithm constructed to use the functionality of existing RT datapath components more effectively. Scheduling algorithms for High-Level Synthesis (HLS) have been based on assumptions that failed to recognize that RT datapath components may have complex, concurrent functionality. The algorithm described in this report takes advantage of a representation that was designed to capture the complex, concurrent functionality of combinatorial RT functional units. This enables the algorithm to utilize RT functions that have been available in existing RT components but had been ignored by other scheduling techniques. The algorithm has been implemented to work with the Interactive Synthesis Environment (ISE).

*This work was supported in part by SRC contract 95-DJ-146

Contents

1 Introduction and Problem Definition

Early work in High-Level Synthesis (HLS) assumed a simple model for the functionality of RT components: 1 component can perform a single operation in a single time step (clock period). This was reflected in the simple way HLS algorithms mapped operations, such as additions or multiplications, directly to RT components, such as adders and multipliers. This was an intuitive assumption and even seemed to work well when using multi-function RT components. However, as we move towards more complex designs, the role of design reuse, i.e., the use of existing components, becomes critical. Hence, a concern that helps speed up the design process is whether pre-existing RT components, such as those found in design databooks, can be *reused* in a new design. Examining the components in a databook reveals that the original HLS assumptions about RT components are insufficient. Some components may have complex functionality. For example, a multiply-accumulator is capable of performing a multiplication and an addition in combination. Also, some commonly used components will have secondary outputs. For example, an ALU may be able to perform an addition of two numbers as well as generate overflow and carry status bits. These separate but simultaneous outputs are equivalent to functions performed concurrently by the ALU. Consequently, most scheduling algorithms for High-Level Synthesis (HLS) have been based on assumptions that failed to recognize that RT datapath components may have complex, concurrent functionality.

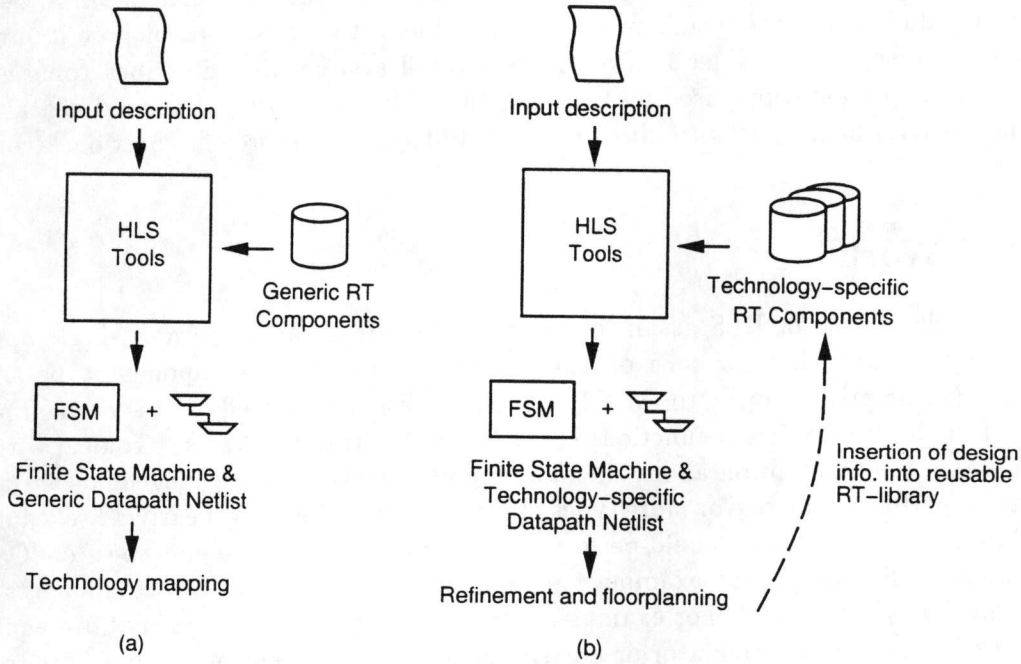

Figure 1: (a) Traditional HLS with generic components, (b) Synthesis for design reuse.

To address the problem of reusing existing RT components efficiently in HLS, our research is investigating these two aspects of the problem:

1. How to represent the full functionality of pre-existing combinatorial RT components.

2. How algorithms for HLS can effectively use the full functionality of pre-existing combinatorial RT components.

[AnDu93] describes our work in representing component functionality. This report describes a scheduling algorithm that is part of the second aspect of our overall research. How our overall approach to HLS differs from traditional HLS is illustrated in Figure 1. Traditional HLS worked with generic models of RT components. These models are based on the simplified assumptions discussed above. The same models are used by the system regardless of the actual technology or library of components used for the final implementation. In contrast, the system we envision will work with different model sets of RT components. These models will represent the library of available components for the specific technology used to implement the design. These models differ fundamentally from traditional, generic RT component models because they capture complex, concurrent functionality. Therefore, new algorithms need to be constructed that are aware of the features of the models and can effectively use the information captured in the models.

The algorithm described in this report is a conversion and enhancement of the algorithm described in [AnDu94]. The algorithm now uses Assignment Decision Diagrams (ADDs) [ChGa92] in the Interactive Synthesis Environment (ISE) [Had95]. As compared to the algorithm in [AnDu94], the algorithm described here is enhanced to deal with control constructs such as conditional execution of operations and loops in the design description. In many situations, our algorithm will find schedules that are an improvement over other techniques, or the algorithm will find a feasible schedule where other techniques will fail.

This report is organized as follows: Section 2 describes previous and related work. Section 3 describes our scheduling algorithm and presents some illustrative examples of how it functions. Section 4 presents how the scheduler is used in the overall system, and describes some external design models and representations used by the algorithm. Section 5 presents experimental results on some standard HLS benchmarks to illustrate the utility of our approach. Section 6 concludes with a summary.

2 Related Work

Early scheduling algorithms in HLS assumed a single component can perform a single operation in a single time step, and that for each operation there was a single component type that the operation could be directly mapped to [McCP88]. But it was recognized that some components are able to perform a set of different functions (e.g., Adder/Subtractor, ALUs). To deal with these multifunction units, the assumption about the mapping of operations was changed so that a single operation may be performed by one or more types of components. However, nearly all scheduling algorithms still assumed a component could perform a single operation in a single time step [Camp91] [PLNG90] [PaKn89] [PaGa87]. But examination of some commonly used RT components shows that this assumption will not hold. For example, there are no single operators that are equivalent to the function of a multiply-accumulator or a parity generator. Consequently, most HLS tools are unable to make effective use of such components. In considering the ongoing use of HLS tools, this can be very detrimental. Designers today can construct modules from the basic RT components types available. But by assumption, it is likely such modules can not be used effectively by the HLS tools if they perform combinations of operations. Obviously, the assumptions about how operators should map to RT components needs to be changed.

Both [LMD94] and [GCDM95] describe formulations that are a step towards improved component reuse. [LMD94] uses implication rules as the mechanism to map possibly complex expressions to RT units using integer programming. Currently, this formulation has not been extended to

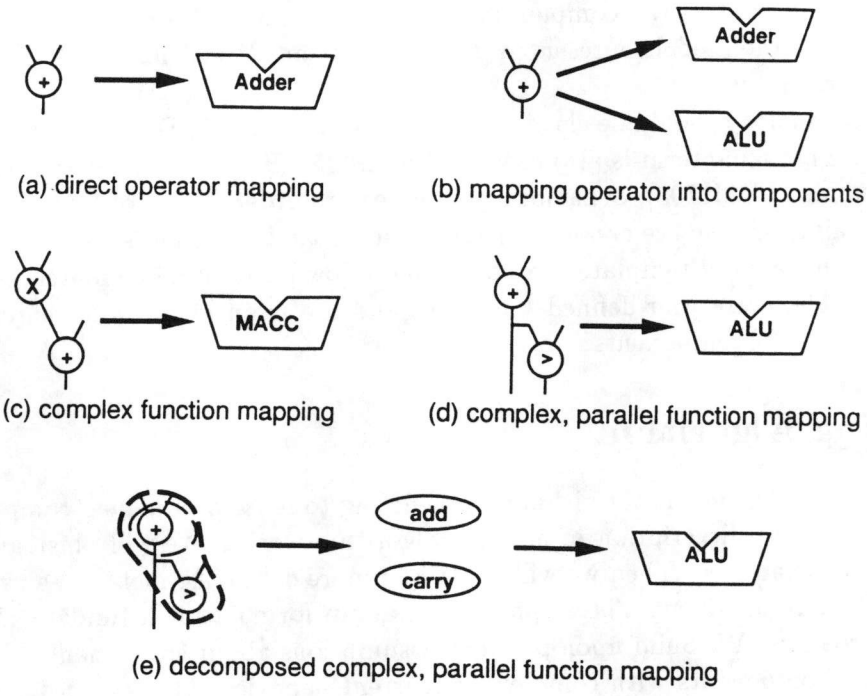

Figure 2: Synthesis RT models.

recognize concurrent functions but this is a viable alternative to our approach. However, one of the goals of our work is integration with an interactive environment. Because of concerns related to how data should be presented interactively and the quick response required for an interactive tool, we opted for an approach based on graphs and algorithms driven by heuristics. Consequently, our work has more in common with [GCDM95] which uses a graph-based representation and graph topology matching to map behavior to RT structures. However, the algorithm in [GCDM95] uses a representation that allows a combination of operators, which produce a single output, to be mapped to a single function of a RT function unit. While this model proves useful for Digital Signal Processing applications, where the functional units fit well into this model, we want to consider a wider range of components. For example, the representation we desire should be able to describe the addition of two numbers and a carry bit as a function producing a sum and carry output. That is, RT function units may have multiple inputs and may produce multiple outputs. Both [KnWi92] and [WPAV92] describe representations that recognize the multi-I/O nature of RT components. With these representations, it is possible to map a set of interconnected operations that produce multiple outputs to an RT component. But when we examined how the behavior of components needs to be used in HLS, we realized that partial matching of functionality was important. For example, it should be possible for an adder to add two numbers and a carry bit and produce a sum and carry output. But that same adder should also be able to just add two numbers to produce a sum. Consequently, we derived a representation that decomposes RT unit behavior into *RT operations* each of which may perform a set of *RT functions* [AnDu93]. In our representation, a component may have various modes of operation, where each output of the components performs a distinct function. Figure 2 illustrates the conceptual differences between the various assumptions and models. Figure 2a is the most basic model early algorithms dealt with: an operator mapped to a

single component type [McCP88]. Figure 2b is the model many well-known scheduling algorithms assumed: a single operator can map to various component types [Camp91] [PLNG90] [PaKn89] [PaGa87]. Figure 2c illustrates the complex function model [LMD94] and [GCDM95] are able to deal with. Figure 2d is the model represented in [KnWi92] and [WPAV92]. Figure 2e is the model our scheduling algorithm uses.

One point of terminology should be clarified. In this work and in [AnDu93], the term "behavioral template" is used. This same term is also used in [LKMM95]. However, these are different objects with different uses. In [LKMM95], behavioral templates are ordered sets of nodes used to specify local timing constraints. These are created, merged, and expanded during scheduling. In this work and in [AnDu93], a behavioral template is a partial data flow graph used to match behavior to RT unit functionality. These are user-defined templates and are maintained in an external database for each specific library of components.

3 Scheduling Algorithm

In this section we will describe our algorithm for scheduling to reuse pre-defined components. First, we will give a very general description of how the algorithm works. We will illustrate the general workings with simple examples. Then we will explain in more detail the metrics and method of our scheduling algorithm. Again, it must be emphasized that our formulation is fundamentally different from previous approaches. We build upon previous assumptions about component behavior in HLS in order to consider components with complex, concurrent behavior. Consequently, our algorithm considers a greater variety of ways that operators can be mapped to component types as illustrated in Figure 3. This means that determining how a given operator can possibly map to a set of allocated components becomes a more complicated search than previous scheduling algorithms had formulated.

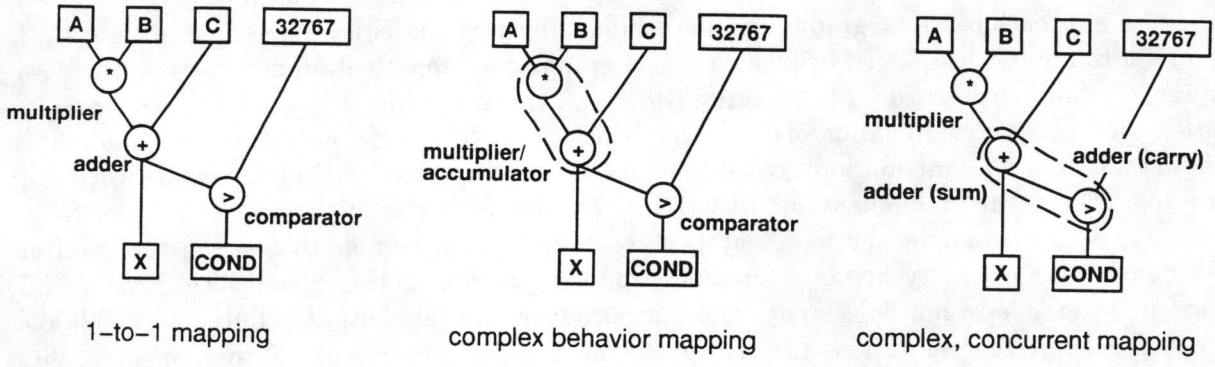

Figure 3: Example of alternative ways to map operators to RT units that scheduler considers.

3.1 Algorithm Overview and Examples

Obviously, there are many approaches that can be used to solve this formulation of the scheduling problem. Our approach is to divide the design into a set of scheduling subtasks, and then use a branch-and-bound search with various search reduction heuristics to solve each subtask. We treat each ADD basic block as a separate scheduling subtask. So for each block, the algorithm:

1. estimates the critical path through the block. This involves a "quick-and-dirty" mapping of operators to components so that component pin-to-pin delays are used for the operators.

2. schedules all operators related to the critical path. A more thorough search of mapping of operators to components is done in this phase. Our algorithm will be given a constraint on the allowed propagation delay through function units in a single clock cycle. This constraint will be used to determine chaining and multicycling of RT unit operations.

3. schedules the off-critical path operators. The algorithm determines whether remaining unscheduled operators can be assigned to an already scheduled RT unit (i.e., searches for concurrent operations).

At this point, some control flow characteristics of the ADD representation should be noted. In ADDs, conditional assignments do not imply state transitions. In a Control/Data Flow Graph (CDFG) representation, a construct like an "if-then-else" statement is usually translated to a control flow branch node which, for scheduling, is usually interpreted into branching state transitions (see Figure 4). However, this is a scheduling assumption based on language syntax. "if-then-else" statements only attach conditions to assignments. The typical interpretation that all assignments and operations that appear before an "if-then-else" should be executed before the evaluation of the condition for the "if", is an assumption used to simplify scheduling algorithms based on CDFGs. In contrast, the ADDs used in ISE are not built with such an assumption. The effect is that descriptions that consist of only unconditional or conditional assignments are represented by a single ADD basic block.

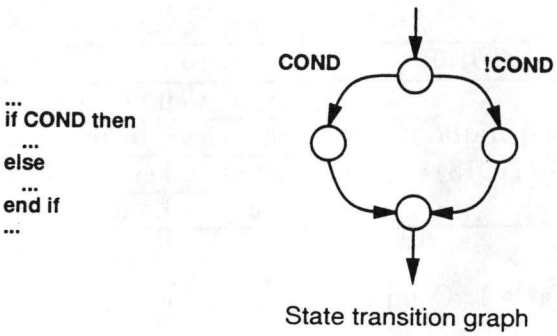

Figure 4: Conditional branch translated to state transitions.

For conditional loops in ADDs, states must be introduced into the description. For loop constructs like "while" loops, we concluded that three separate blocks of assignments were needed: before the loop, for the body of the loop, and after exiting the loop. For scheduling, this would require state transitions like those shown in Figure 5. Consequently, ADD does not use any special loop construct. Instead, it uses an attached state transition graph. In the finite-state machine represented in this manner, each *state* represents an ADD basic block, while the state transitions represent the loop.

We illustrate this representation and the operation of the algorithm on three simple examples that respectively deal with unconditional assignments, conditional assignments, and loops. For these examples, three RTL components were captured through ISE: a 16-bit Adder/Subtractor,

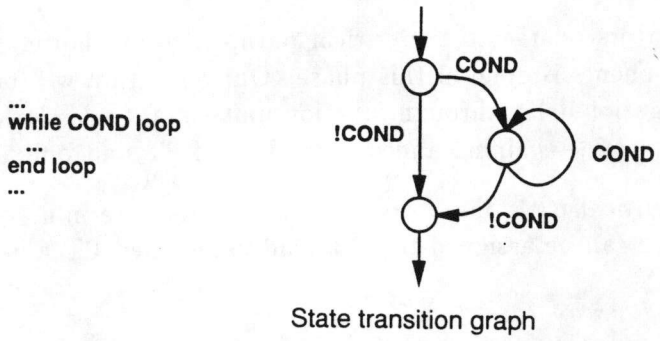

Figure 5: Conditional loop translated to state transitions.

a 16-bit Multiplier, and a 16-bit Subtractor/6-function Comparator (see Table 1). The area and delay characteristics of these components were based on the 0.8 micron CMOS VTI library [VTI92]. For a design description, we assume that an allocation of components is given. Also, we assume a delay value is specified for the maximum allowed data propagation delay through the datapath components. Note, that this is not a clock period length since it does not account for delays related to registers, interconnect and wiring, or memory accesses.

Component	delays	
16-bit Adder/Subtractor (DPASB001S)	input to sum	11 ns
	input to carry/overflow	7 ns
16-bit Multiplier (DPMLT010H)	input to product	25 ns
16-bit Subtractor/Comparator (DPSUB001S + DPZDT001S)	input to difference	11 ns
	input to Greater-or-Equal/Less	12 ns
	input to Less-or-Equal/Greater	14 ns
	input to Equal/Not-Equal	14 ns

Table 1: Components used for examples.

3.1.1 Example 1: Unconditional Assignments

This example will illustrate how the algorithm estimates the critical path and schedules operations. The description for this example is shown in Figure 6. In this example, COND describes the overflow output for the addition A + B. A 16-bit Adder/Subtractor and a 16-bit Multiplier were allocated for this example.

The first step in scheduling a set of assignments is to estimate the critical path. To do this we calculate the *delay to output* (DTO) for data flowing through this basic block. We will need to find the operator delays for this. Operator delays are found by mapping the operator to an allocated component. The matching for this example is shown in Figure 7. Note how the match for the **overflow** overlaps with the match for the **add** and how this single delay match encompasses more than one operator. Once the delays for the operators are found, the DTOs are calculated by the formula:

```
entity example1 is
  port (A: in BIT_VECTOR(15 downto 0);
        B: in BIT_VECTOR(15 downto 0);
        C: in BIT_VECTOR(15 downto 0);
        COND: out BIT;
        X: out BIT_VECTOR(15 downto 0));
end example1;

architecture BEHAVIOR of example1 is

begin

process(A,B,C)
  variable TEM1: BIT_VECTOR(15 downto 0);
begin
  TEM1 := A + B;
  COND := TEM1 > B"0111111111111111";
  X := TEM1 * C;
end process;
end BEHAVIOR;
```

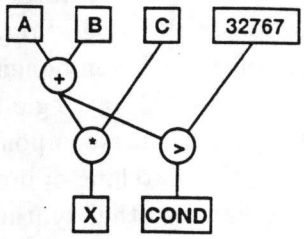

Figure 6: Example 1 - unconditional assignments: VHDL and equivalent ADD.

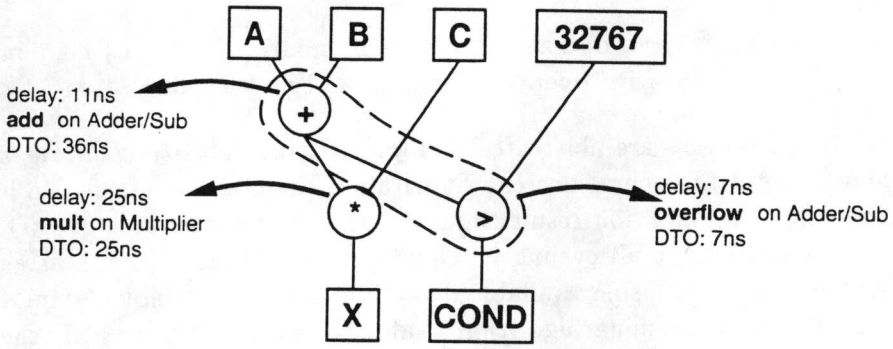

Figure 7: Operator mapping for calculating delay.

DTO = delay of operator + max. DTO of dependent operators

where a dependent operator is an operator that uses the data output of this operator. Once the DTOs have been calculated, we find the critical path by starting at the operator with the highest DTO. We then proceed to the dependent operator with the highest DTO. Once we reach an operator that outputs to a write node, we have found the bottom of the critical path.

Now that the critical path has been found, the algorithm will try to schedule the critical path operators. At this point, it is necessary to describe the scheduling options available with our algorithm. Since delay information about RT functions is available, our scheduling algorithm makes no assumptions about operator delays. Consequently, the algorithm can be configured to do various combinations of RT operation chaining and multicycling. We define operation chaining as linking the input of one RT component to the output of another so that data is propagated from one unit to the other within a single state. We define operation multicycling as allowing an RT operation to be executed on a component for more than one state. For operation multicycling, we assume that appropriate latching of inputs will be implemented, either built into the RT component or added to the design by other synthesis tools.

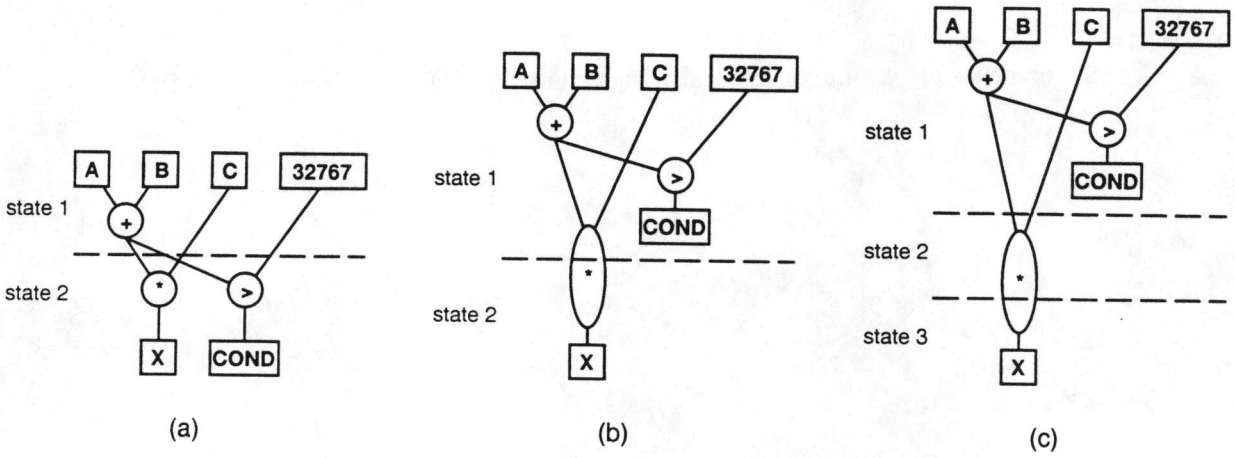

Figure 8: Scheduling options: (a) no operator chaining or multicycling. (b) combined operator chaining and multicycling. (c) exclusive operator chaining and multicycling.

Some of the scheduling options are illustrated in Figure 8. In *exclusive chaining/multicycling*, operations are chained only if the operations can fit within a single state. As shown in Figure 8b, chaining the multiplication and addition results in a propagation delay that is greater the allowed maximum. For exclusive chaining/multicycling in Figure 8c, the multiplication starts execution in state 2. As Figure 8 illustrates, by using available delay information and not making assumptions about component delays, our scheduling algorithm is able to explore a variety of schedules based on the configured options and the allowed propagation delay. Of course, this can produce infeasible combinations. For this example, if the allowed datapath propagation delay is only 20 nanoseconds and no multicycling is allowed, there is no way to execute the multiplication that takes 25 nanoseconds. During scheduling, when a template match is scheduled to a component during a given state, the appropriate operator inputs and outputs are bound to the component pins in that state. This information is needed when the algorithm tries to schedule other operations to the same component in the same state.

Once the critical path has been scheduled, the algorithm will try to schedule the rest of the description in an As-Soon-As-Possible (ASAP) manner. For this example, we will now illustrate some of the key advantages of our models. For the Adder/Subtractor, a template for the overflow output of an addition matches the portion of the graph shown in Figure 7 for the **overflow** function. Our scheduling algorithm is able to find this template match and query the component database to find that the allocated Adder/Subtractor can perform this template concurrently with the already scheduled addition. To confirm this concurrency, the algorithm checks that the pin bindings for both templates are compatible, i.e., the pins of the Adder/Subtractor will be bound to the same inputs and outputs. Once the compatibility of the pin bindings has been confirmed, the "overflow" is scheduled to the Adder/Subtractor in the same state as the "addition." This is why the comparison in Figure 8b and Figure 8c is scheduled to the same state as the addition.

3.1.2 Example 2: Conditional Assignments

```
entity example2 is
  port (A: in BIT_VECTOR(15 downto 0);
        B: in BIT_VECTOR(15 downto 0);
        C: in BIT_VECTOR(15 downto 0);
        D: out BIT_VECTOR(15 downto 0));
end example1;

architecture BEHAVIOR of example2 is

begin

process(A,B,C)
  variable TEM1, TEM2: BIT_VECTOR(15 downto 0);
  variable COND: BIT;
begin
  COND := C > 16;
  TEM1:= A * B;
  if COND then
    TEM2 := TEM1 – C;
  else
    TEM2 := A – B;
  end if;
  D := TEM2;
end process;
end BEHAVIOR;
```

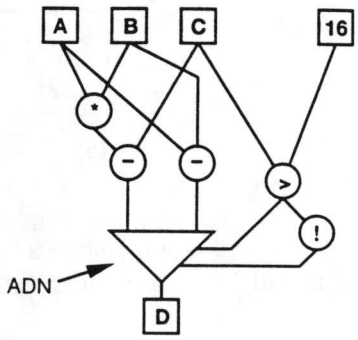

Figure 9: Example 2 - conditional assignments: VHDL and equivalent ADD.

The second example will illustrate how the algorithm handles conditional assignments. The description of this example is shown in Figure 9. For this example, a multiplier and a subtractor/comparator were allocated. For conditional assignments, the ADD has an Assignment Decision Node (ADN) which serves a function similar to the *select* in the Value-Trace representation [McFar78]. An ADN has data inputs, each of which has an associated condition input. Each input can be an ADD graph for an expression. Each data input of the ADN must be the same bit-width as the ADN's output. Each condition input must evaluate to a Boolean value (i.e. 1 bit wide) and must be mutually exclusive to the other condition inputs. That is, only one condition input should **true** at any time. The value of a data input is transferred to the output of an ADN when the appropriate condition input becomes true.

Special consideration must be given to ADNs to correctly schedule a design. First, consider that a condition input applies to all the operators in the expression "rooted" at a data input to the ADN. For scheduling, this is important when we are determining under what condition a given operator needs to be executed. This becomes the problem of determining the *condition vectors*

for each operator. For ADDs, the generation of condition vectors is described in [JCG94]. These condition vectors are used to determine whether operators have mutually exclusive conditions, and thus, can possibly share the same component. Another point about ADNs is that they do not impose an order of evaluation between the data and condition inputs. There is no requirement stating that the data input must be ready before the condition input is evaluated or vice versa. For scheduling, it is necessary to have both the data and condition values ready when a data transfer through an ADN node is to be assigned to a state. These ideas will be further illustrated in the example.

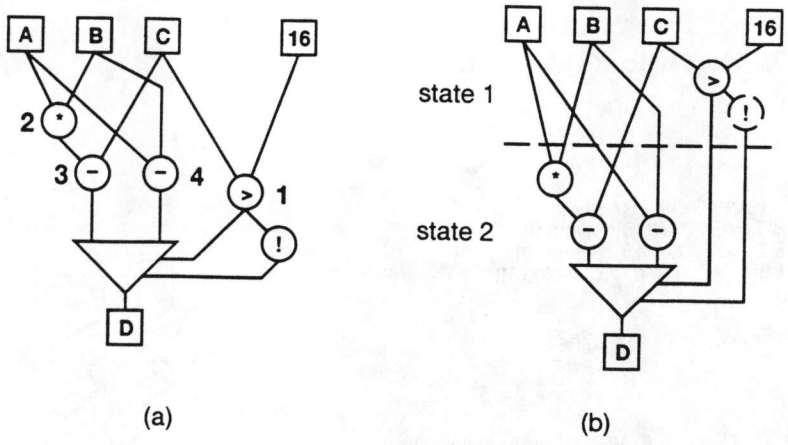

Figure 10: Scheduling conditions: (a) order operators are scheduled. (b) schedule for example with allocation of a multiplier and subtractor/comparator.

Our scheduling algorithm makes certain assumptions to deal with conditional assignments. Conditional assignments do not affect the way delays or DTOs are calculated. But for determining the critical path, only one data or condition input to an ADN will be considered critical. For scheduling the critical path, we need to consider all operators related to the critical path, i.e., operators that have data or *condition* dependencies with operators on the critical path. For this example, scheduling the critical path means not only scheduling the multiplication and a subtraction but also the "greater than" comparison, since it determines the assignment condition of the critical path. To ensure that the corresponding condition input is ready when a data input arrives at a ADN, we schedule the condition vector operators first, then we schedule the data input operator to a state after the condition vector operators are executed. For this example, that means we first try to schedule the "greater than," then the multiplication, then the subtraction, and the subtraction must be scheduled to a state after the "greater than" is performed. Figure 10a shows the order in which the operators are scheduled. Likewise for the other operators in the description, the condition vector operators are scheduled first. So to complete the schedule for this example, the "not" operator should be scheduled before the subtraction. However, we assume that some basic logic functions, such as 1-bit AND, OR, NOT, are better implemented as random logic or can be incorporated into the control logic. Consequently, for our experiments and examples, we do not allocate gates for these logic operations and we do not schedule them because we assume they will be implemented as random logic or will be part of the control unit.

When conditional assignments are used, an additional consideration for sharing of components is

mutual exclusiveness of operators. The condition vectors from [JCG94] are used to determine this. Consequently, when our algorithm attempts to schedule operators to the allocated components, it not only checks for data and condition dependencies but also for mutual exclusiveness of condition vectors. In this example, the algorithm schedules the two subtractions to the same state on the same component. Figure 10b shows a complete schedule for this example. Note, that regardless of the allowed propagation delay, at least two states are required for this example because of the assumptions we make for scheduling conditions.

3.1.3 Example 3: Conditional Loops

```
entity example3 is
  port (A: in BIT_VECTOR(15 downto 0);
        B: in BIT_VECTOR(15 downto 0);
        X: out BIT_VECTOR(15 downto 0));
end example1;

architecture BEHAVIOR of example3 is

begin

process(A,B)
  variable TEM1, TEM2: BIT_VECTOR(15 downto 0);
begin
  TEM1:= A–B;
  TEM2 := 0;
  while (TEM1 > 0) loop
    TEM1 := A–B;
    TEM2 := TEM2 + 1;
  end loop;
  end if;
  X := TEM2;
end process;
end BEHAVIOR;
```

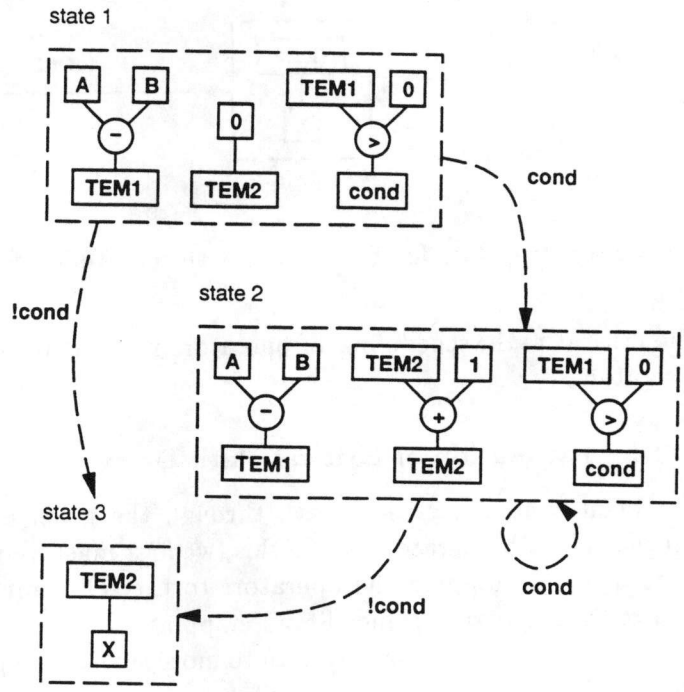

Figure 11: Example 3 - conditional loop: VHDL and equivalent ADD.

This example will illustrate the scheduling of a loop. The description of this example is shown in Figure 11. As previously mentioned, to introduce looping into an ADD description requires introducing states. For scheduling, each of these states can be treated as a basic block, So, scheduling a description with loops can be done by scheduling the individual blocks. Thus, a description with loops simply becomes a set of scheduling tasks similar to the two previous examples, and the scheduler will "split" each state into a set of states. Figure 12 illustrates the scheduling of this example for an allocation of one Adder/Subtractor and one Subtractor/Comparator.

3.2 Basic Block Scheduling In-Depth

Now that we have explained how control-flow constructs are dealt with by our scheduling algorithm, we will go into a more detailed discussion of how the algorithm schedules individual basic blocks. We will describe in detail the three phases of basic block scheduling for ADDs: estimation of

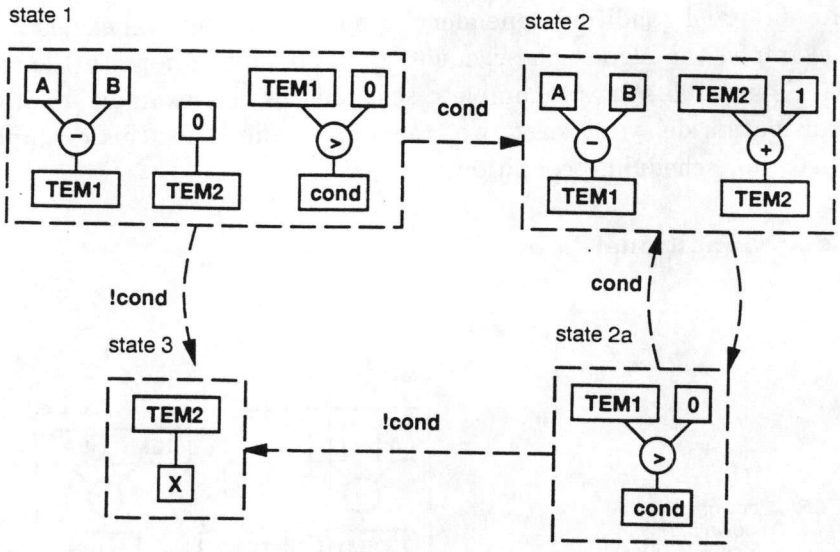

Figure 12: Schedule for Example 3 with one Adder/Subtractor and one Subtractor/Comparator.

the critical path, scheduling of operators related to the critical path, and scheduling of remaining operators.

3.2.1 Estimation of Critical Path Delay

For identifying the critical path through the given description we calculate the *delay to output* (DTO) for each operator. To do this, we first must find the input-to-output delay of each operator. This involves mapping the operators to the RT components allocated from the library. For this phase, we do a "quick and dirty" mapping, trying to map each operator directly to an allocated component. An operator may map to more than one type of component, so we select the fastest, i.e., the component with the minimum input-to-output delay. But there is a side-effect of this approach that could lead to a false critical path: since only individual operators are considered, the best mapping of operators to components may not be used. For example, if a multiply-accumulator has been allocated and the description has multiplication outputting to an addition (e.g., **A*B+C**), both operators can be mapped to a single RT operation (a *multiply-accumulate*), but for estimating delay, we will try to map each to separate RT operations (*multiply* and *addition*). The delays for a *multiply* plus an *addition* could be greater than a *multiply-accumulate*. However, if the mapping of **A*B+C** to a *multiply-accumulate* is the only one possible, the delay of a *multiply-accumulate* is used for **A*B+C**. For this first pass, we are simply checking if there exists at least one allocated component that can execute each part of the description. For each operator we find the delay by locating which allocated components can perform this operator and checking the pin-to-pin delays.

Once the delays for the operators are found, the DTOs are calculated by propagating the delays to their inputs. For each operator match, this formula is used:

$$\text{DTO} = \text{delay of operator} + \text{max. DTO of dependent match}$$

A dependent match is one that uses the data output of this operator match. For example, in **A*B+C**, the addition is dependent on the multiplication. Once the DTOs have been calculated,

we find the critical path by starting at the operator with the highest DTO. We then proceed to the dependent operator with the highest DTO. Once we reach an operator that outputs to a write node, we have found the bottom of the critical path. During the entire scheduling algorithm, the set of operators that we have calculated delays for is maintained as the set of all operators that need to be scheduled. When operators are scheduled, they are marked but remain in the set.

Note, as previously mentioned, we assume that some basic logic functions, such as 1-bit AND, OR, NOT, are better implemented as random logic or can be incorporated into the control logic. During calculation of delays, if we encounter a 1-bit AND, OR, NOT that cannot be mapped to an allocated component, we ignore it assuming that subsequent synthesis will implement it as random logic or it will be part of the control logic.

3.2.2 Scheduling the Critical Path

The second phase of basic block scheduling will schedule the operations related to the critical path. These include the operators that have condition or data dependencies with the critical path operators. This portion of the algorithm will assign the operations to a sequence of states, starting with the operators at the top of the critical path (the ones receiving data that is coming into this basic block) and working down to the output operators. In the process, the algorithm determines if multicycling or chaining of the operations is possible or needed.

The scheduling of the critical path is described in procedure Sched_CP() in Figure 13. To handle condition and data dependencies between operators, we build a "critical path stack" prior to calling this procedure. For each critical path operator, operators with data or condition dependencies are above it on the stack. During scheduling of the critical path, we pop operators off the stack to ensure that the dependencies for an operator have already been scheduled. The internal looping and recursive procedure calls of Sched_CP() implement a "branch and bound" search, trying various schedules and stopping on alternatives that become longer than the shortest schedule found so far. During scheduling of the critical path, the algorithm does not try to evaluate all possible combinations of template matches for the entire critical path. Instead, it evaluates only a portion of the critical path at any time. In effect, the algorithm schedules a small "window" of the critical path, sliding the window down the critical path at each step of the scheduling process. The size of this window is the variable *depth*.

The procedure begins by finding an unscheduled critical path operator on the stack (line 1). If the stack has been exhausted, all operators related to the critical path have been scheduled. Then we first try to add this operator to the existing schedule at line 3, i.e., schedule it to a state before the current end of the schedule. If that is possible we go on to the next operator at line 4. Regardless of whether we can schedule the operator to a previous state, we will still explore the possibility of scheduling the operator to this state, i.e. lines 5 and on will still be executed whether line 3 fails or not. At line 6, the procedure tries different size "windows" of the critical path. The current critical path operator to be scheduled is always at the top of this "window" and each iteration of line 6 shrinks the window. All possible template matches in this window that "cover" the critical path operator are examined at lines 7 and 8. For each template match we try to schedule it to all possible RT functions and RT units (lines 9 and 10). If the selected function cannot be performed within the current time step, it is scheduled to multiple time steps at lines 11 and 12, i.e., a multi-cycle operation. If the function can be performed within the time left for the current step, it will be scheduled to this step, i.e., it will be chained to other operators at line 13. Chaining is possible because we maintain the amount of delay the needed for the operations already scheduled to the current step. We also assume an additive delay model (i.e., the delay to perform two operations is equal to the sum of each operation's delay). Of course, multicycling and

```
Sched_CP()
1  Pop an unscheduled node from the stack
2  if there is an unscheduled operator on the critical path
3    if can ASAP schedule node to existing schedule.
4      Sched_CP() next node on stack
5    /* Try other schedules of node */
6    for depth = max. downto 0 do
7      find covers of node of size depth
8      for each cover of node do
9        for each RT function cover can bind to do
10         for each RT unit RT Function can bind to do
11           if RT function needs to be multi-cycled,
12             add states and schedule
13           if RT function can be chained, schedule.
14           if schedule not shorter than best schedule found,
15             cancel this search.
16           if could not schedule RT function,
17             if could not schedule RT function to an empty state,
18               cancel this search.
19             else maybe no RT units were available,
20               add a state to the schedule and retry
21           else could schedule RT function
22             Sched_CP() next node on stack
```

Figure 13: Scheduling of critical path related operators

chaining is only done when these options are allowed. If the function was scheduled, we go on to the next operator at line 22. If the function was not scheduled, the algorithm will add a state (line 20) and attempt to schedule the function again. If the algorithm fails to schedule an operator when given an empty state, then the algorithm returns a failure to find a feasible schedule for the critical path (lines 16-18). This can be due to a combination of no multicycling and the allowed maximum propagation delay being too short for an operation.

3.2.3 Scheduling Remaining Operators

After the critical path is scheduled, the algorithm will schedule the remaining operators. For these operations, the algorithm will try to share functional units. That is, schedule an operation to a state when a unit is free or when other operations scheduled to that unit have mutually exclusive condition vectors. The algorithm will also try to schedule operations concurrently, i.e., determine if an operator can be scheduled to a function that can execute concurrently with other functions on an RT unit.

The scheduling of remaining operators is described in Figure 14. In procedure ASAP_nCP(), the algorithm will first attempt to schedule the operators ASAP, without additional time steps. If this fails, the algorithm will schedule the operators ASAP with time steps added to the schedule. During the ASAP scheduling of an operator condition and data dependencies are checked as well as exploration of concurrent RT functions.

Lines 1 and 2 in ASAP_nCP() check if the current partial schedule for the critical path is shorter than the best solution found so far. If not, there is no point in finishing this schedule. Line 3 will call ASAP_schedule() to try to schedule the remaining operators into the existing schedule. If that

was not possible at line 4, the loop through lines 6 to 14 will alternately add states to the end and beginning of the schedule to try to make room for the remaining operators. Lines 8 and 13 check that the current partial schedule created in the loop is shorter than the best solution found so far. As line 5 indicates, the procedure will add states and attempt to schedule until a schedule is found or the total number of states added is greater than or equal to 3 times the number of operators we are trying to schedule. This limit on the number states is arbitrary and was derived on the assumption that efficient designs will not have a large number of operators multicycled for 4 states or more.

The procedure ASAP_schedule( ) attempts the actual scheduling of each of the remaining operators. Since this procedure tries to schedule operations as early as possible in the schedule, it makes sense to try to first schedule the operators at the top of long expressions. Consequently, at line 19, ASAP_schedule( ) looks for the unscheduled operator with the highest DTO. But before any operator can be scheduled, the operators that it is dependent on, for its data or condition, must be scheduled first (line 20). If the data and condition dependencies are already schedule, then the procedure will try to insert the operator into the schedule at line 21. It is here that the algorithm will search for concurrent functions and try to map the operator to them. If for any reason the operator cannot be scheduled, line 22 will cause the procedure to quit and return that a feasible schedule could not be found.

```
ASAP_nCP()
1   if not shorter than best schedule found,
2     cancel this search.
3   ASAP_schedule()
4   if failed
5     while not successful or num. added states < 3× num. unscheduled ops. do
6       add state to end of schedule
7       ASAP_schedule()
8       if not shorter than best schedule found,
9         cancel this search.
10      if successful exit while loop.
11      add state to front of schedule
12      ASAP_schedule()
13      if not shorter than best schedule found,
14        cancel this search.
15  if a schedule found
16    if shorter than best schedule found,
17      save as best schedule found

ASAP_schedule():
18  do
19    find unscheduled operator with highest DTO.
20    check if for unscheduled op with data/condition dependency
21    ASAP schedule operator to available units
22    if fails return failed.
23  while there is an unscheduled node.
```

Figure 14: Scheduling of remaining operators

3.2.4 Search Reduction

The search space for this scheduler can be envisioned as a search graph that is a tree. The first branches from the root determine which critical path operators are scheduled to RT units for the first state of the schedule. Subsequent branches add more operators and states to the schedule. Obviously, the size of the search space grows proportional to the number of operators, and the number of different ways each operator can be mapped to an allocated RT unit.

Our scheduling algorithm conducts a depth-first search of this tree. It "prunes" off searches that will obviously not find a better solution than the current solution. That is, it will abandon a schedule that is not shorter than the current schedule already found. However, this still could leave a very large space to search. From experiments we have conducted, we find that the final solution returned is most often found very early in the search. Consequently, we have two additional strategies to limit the search space further: progressive backtracking and a maximum search limit.

Backtracking is an inherent part of depth-first search where the search "backtracks" up one level of the search tree when searching of a branch completes or is pruned. With only one level of backtracking, each schedule the search explores is only slightly different from the previous schedules that have been searched. This is desirable early in the search process when few or no solutions have been found. But as the search progresses, we want to look at schedules that differ more, in order to explore a wider variety of schedules faster. Consequently, in our algorithm, we progressively increase the number of levels of backtracking as the search progresses. This is done by maintaining the number of attempts to schedule operators, i.e., the number of edges the search has traversed. Whenever a search of a branch fails, the amount of backtracking is calculated.

$$\text{num. levels to backtrack} \sim \frac{\text{num. attempts to schedule operators}}{\text{num. operators in design}}$$

Further, since the best solution found is usually discovered early in the search of the entire tree, we allow setting a limit on the number of attempts to schedule operators. Potentially, the limit could be set so that no solution is found but for our experiments the limit was set so that at least several dozen schedules for very complex designs would be searched. This search limit was implemented with the idea that the algorithm would be used in an interactive environment where deriving a "good" solution in a short amount of time was desirable.

4 System Overview

To illustrate how the scheduling algorithm reuses components, it is helpful to describe its use within a system. The scheduling algorithm is constructed to work within the Interactive Synthesis Environment (ISE) [Had95]. Consequently, various types of information used by the scheduling algorithm are entered through the ISE interface or are represented by ISE-related data structures. In order to use an existing component for scheduling, we need to do the following:

- Capture the component's physical characteristics and behavior.
- Incorporate the component's characteristics and behavior into a library database.
- Specify the allocation of RT function units for a design description.

4.1 Component Capture

Figure 15 shows the graphical interface in ISE used to capture component information. The upper left of the display shows a diagram for a behavioral template. The lower part of the display shows a table for the mapping of component pins. These will be described later. The upper right of the display is primarily used to capture physical information about the component.

To describe an RT component in ISE, the designer can specify the height and width of the component in microns. This yields a rectangular box representing the physical shape of the component. A set of ports for the component can be specified. Ports can be classified as input, output, input/output, or control. A port's bitwidth and placement on the edge of the component box can also be specified. Delay values between pairs of ports can be specified. The value can be in picoseconds, nanoseconds, or microseconds. In this way, this portion of the ISE interface captures the area and delay information for a component. The component information for this interface can be saved to and loaded from an external file.

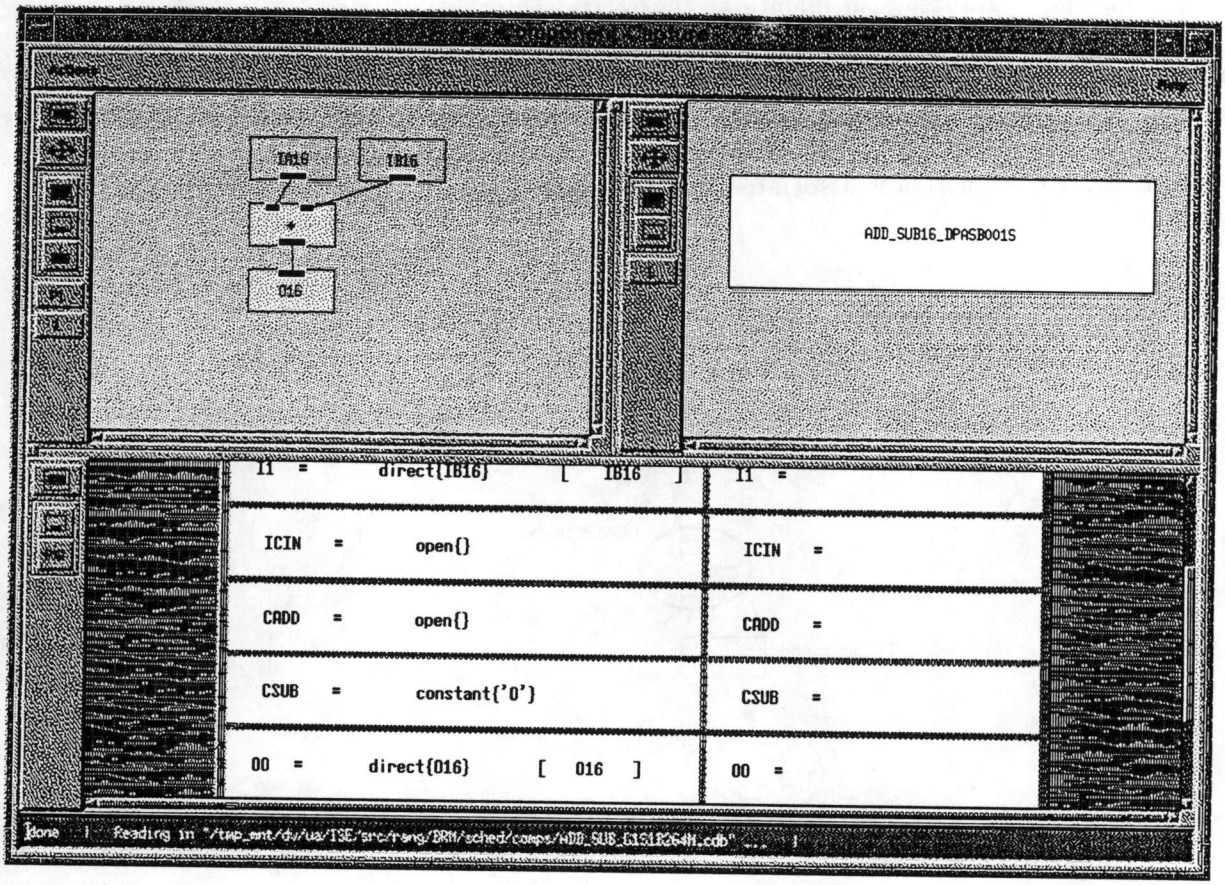

Figure 15: ISE component capture interface.

As mentioned above , the upper left portion of the display in Figure 15 displays behavioral information about a component. Each component in ISE can have an associated set of *behavioral templates*. Each template is an Assignment Decision Diagram (ADD) [ChGa92] describing, in a data-flow like manner, a function the component can perform. A template typically is an inter-

connected set of operations, read nodes, and a single write node. The table in the lower part of Figure 15 describes how a template is mapped onto the component. Usually, read nodes are simply mapped to a component's input ports, and a write node is mapped to a component's output port. However, simple logic operations can also be involved in the port mappings (e.g. ANDing two values to obtain the input value to a port) or ports can be set to a binary constant value. The ADD for the behavioral template is input from an external file. The port mapping table is entered manually through the ISE interface.

For our model of component behavior, we are interested in describing concurrent functions of an RT component. For a template, we interpret the constant values assigned to the control ports of the component as the mode of operation for that component. Consequently, for a given component, templates that have the same values on the control ports are interpreted to be concurrent functions. This information is maintained in our component data base.

4.2 Component Database

The component database organizes the physical and behavioral information about all available components. The database also maintains the relationships between the templates and components. The behavior of each component is decomposed into RT operations and RT functions. Each component can have a set of RT operations. Each RT operation represents a distinct mode of operation that the RT component can be configured for. Each RT operation can have a set of RT functions that can be performed concurrently by the component under this particular mode of operation. Each RT function is associated to a template. Figure 16 illustrates this hierarchy.

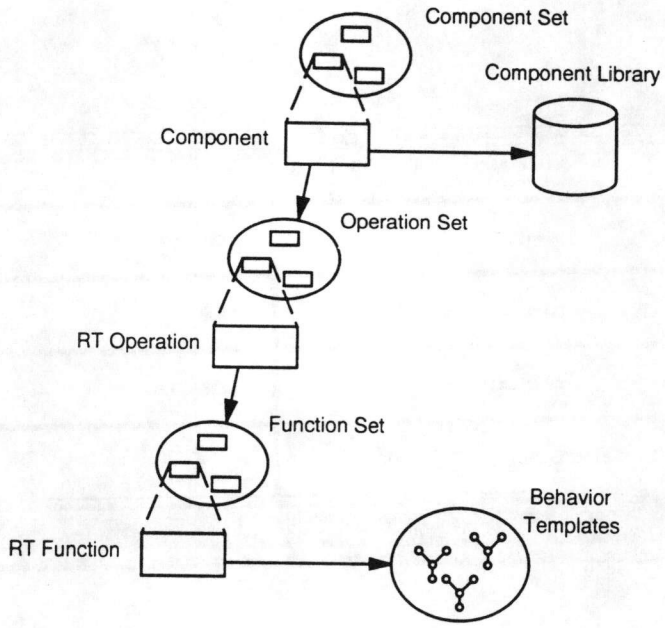

Figure 16: Representation hierarchy for component to behavior mappings.

A basic task in many HLS algorithms is determining how behavior can be mapping to RT components. With our representation, this task is equivalent to trying to find a behavior template that matches to some segment of a given ADD. To do this effectively, we construct a *template parse*

tree structure from all the behavioral templates of the components in a library database. Figure 17 shows some example templates for the addition of two 4-bit numbers, addition of two 4-bit numbers and a carry-in bit, the "carry" function for the addition of two 4-bit numbers, decrementing of a 4-bit number, and a "is-zero" function for the decrementing of a 4-bit number. Figure 18 is the parse tree constructed from the templates in Figure 17. The first level of internal nodes of the tree are write nodes that describe the outputs of the templates, i.e., the output bitwidth. There will be only one write node for all templates with output of a particular bitwidth. The other internal nodes of the tree are operators. The leaf nodes of the trees are read nodes which describe the inputs to the template. Edges (going down) from an internal node are alternative input sets; in this case, sets of input pairs. This arrangement maintains the context of the templates, i.e., the particular combinations of inputs and operations. Input pairs that have leaf nodes are labeled with the RT functions that can perform that template. Figure 19 illustrates how a match is done. First the appropriate write node is found from the root. Then the operator that outputs to the write node is matched. Now for operators, the set of inputs to the operator is matched. In this case, $+_2$ has as inputs $+_1$ and a constant **1**. Matching of input sets is recursively done for each subexpression. In this case, we continue by matching the two 4-bit variable inputs of the $+_1$ subexpression.

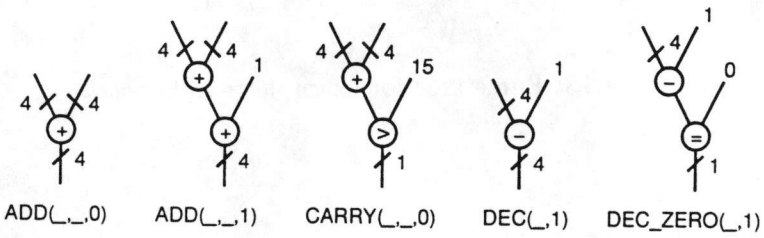

Figure 17: Example behavioral templates of RT functions.

The component database is used in combination with the template parse tree to determine how behavior can map to components. Given some design, the template parse tree is searched to find templates that match segments of the design. Querying the component database determines what components can perform each template, and which templates can be performed concurrently on a single component.

4.3 Input of Design and Allocation of Units

For our scheduling algorithm we assume we are given a design with an allocation of RT functional units. For our examples and experiments, designs were translated from VHDL or entered manually through an ISE interface. The VHDL to ADD translator is still under development and can only be used at the moment for descriptions with no control constructs, i.e., "straight-line code."

Allocation of functional units for the design is done through ISE. A component type is brought into ISE through the component capture interface. "Dragging and dropping" a component from the capture interface into the allocation interface allocates an instance of that component type for the design.

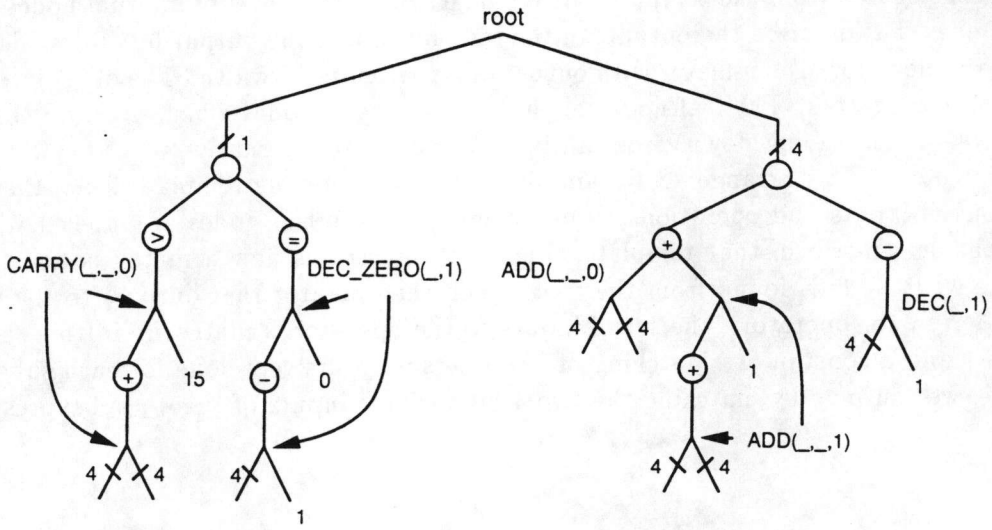

Figure 18: Parse tree for example templates.

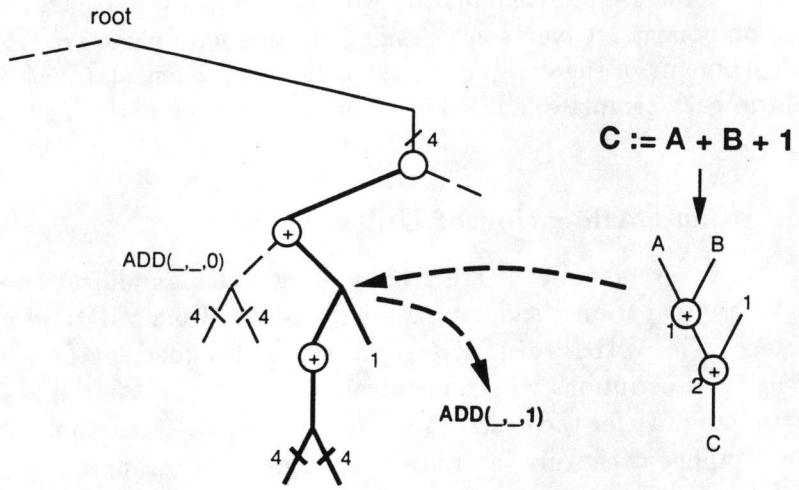

Figure 19: Example parse tree match.

5 Experimental Results

For our experiments, we used descriptions of the Fifth Order Elliptical Wave Filter [PaBu87], a greatest common divisor algorithm [BrBr88], and an 32-bit floating point adder/subtractor [HLSD95]. We ran our experiments to explore the possible schedules for different allocations and delay limits, and to examine the effects of combining operation multicycling and chaining. These experiments also demonstrate how our algorithm is able to use components with different functions and delays.

Allocation (16-bit components)	Scheduling options	Max. delay	# states in schedule
1 Adder/Subtractor 1 Multiplier	no chaining no multicycling	20 ns	not possible
		50 ns	27
	no chaining multicycling	20 ns	28
		50 ns	27
	chaining multicycling	20 ns	29
		50 ns	26
	exclusive chaining & multicycling	20 ns	28
		50 ns	26
1 Adder/Subtractor 1 Multiplier/Accumulator	no chaining no multicycling	20 ns	not possible
		50 ns	16
	no chaining multicycling	20 ns	24
		50 ns	16
	chaining multicycling	20 ns	27
		50 ns	16
	exclusive chaining & multicycling	20 ns	25
		50 ns	16
2 Adder/Subtractor 1 Multiplier	no chaining no multicycling	20 ns	not possible
		50 ns	15
	no chaining multicycling	20 ns	21
		50 ns	15
	chaining multicycling	20 ns	23
		50 ns	14
	exclusive chaining & multicycling	20 ns	21
		50 ns	14
2 Adder/Subtractor 2 Multipliers	no chaining no multicycling	20 ns	not possible
		50 ns	15
	no chaining multicycling	20 ns	17
		50 ns	15
	chaining multicycling	20 ns	18
		50 ns	14
	exclusive chaining & multicycling	20 ns	17
		50 ns	14

Table 2: Results for the Elliptic Filter

Table 2 shows results of experiments done with the Fifth Order Elliptical Wave Filter. As shown in the first column of the table, we conducted experiments with various allocations of adder/subtractors, multipliers, and multiply/accumulators. The second column indicates the type of scheduling attempted. "no chaining" means no operator chaining was allowed. "no multicycling" means no operator multicycling was allowed. "exclusive chaining & multicycling" means operations

System	Allocation	their result	our result
simulated annealing [SaZa90]	2 adders, 2 multicycle mult.	19	17[1]
	2 adders, 1 multicycle mult.	21	21[1]
PBS [PLNG90]	2 adders, 2 multicycle mult.	17	17[1]
	2 adders, 1 multicycle mult.	21	21[1]
FDLS [PaKn89]	2 adders, 2 multicycle mult.	18	17[1]
	2 adders, 1 multicycle mult.	21	21[1]
CATHEDRAL 2nd [LCGM93]	2 adders, 1 single-cycle mult.r	15	15[2]
OSCAR [LMD94]	1 adder, 1 single-cycle mult./accum.	15	16[2]

Table 3: Comparisons to results from previous work. [1] operator multicycling, no operator chaining, 20 ns propagation delay. [2] no operator multicycling, no operator chaining, 50 ns propagation delay.

could be chained only if such chains could fit within a single state. The third column indicates the amount of time allowed for data propagation through the function units. The fourth column indicates the number of states in the solution found by the algorithm.

The Fifth Order Elliptical Wave Filter is a well-known example used in previous works. But because of the assumptions used by previous scheduling algorithms, only a subset of our results are comparable to previous results. Table 3 compares some of our results with results to some previous work. Only specific instances can be compared because only these restricted instances use assumptions and allocations equivalent to these previous works. Table 3 shows that our scheduler is able to produce solutions equivalent to most previous results. Compared to the published results for simulated annealing [SaZa90] and force-directed list scheduling [PaKn89], our algorithm was able to find an improved schedule. One of our results compared with OSCAR [LMD94] reflects the trade-off of guaranteed optimality vs. algorithm speed. While OSCAR was able to find an optimal solution with one less control step, it reportedly took over 10 minutes to find the solution on a SPARC 10. Our scheduler took less than 2 minutes on a SPARC 2. On other experiments with the Fifth Order Elliptical Wave Filter, OSCAR took over an hour to find a solution. In contrast, our scheduler took a maximum of approximately 5 minutes for any experiment.

From our experiments with the Elliptic Wave Filter, we found that allowing a combination of operation chaining and multicycling sometimes produced poorer results. This problem arises because, by allowing combined chaining and multicycling, the critical path can excessively monopolize the available resources. An example of this problem is shown in Figure 20. If two `ADD()` operations are chained and the execution of the second `ADD()` must be carried over to the next state (as in State 1 of Figure 20), the second Adder will be unavailable for two states, even though that Adder could perform an addition in a single state. This leads to the heuristic that operation multicycling and chaining should be done exclusive of each other, i.e., operator chaining should not be done when the chaining forces the operator to be multicycled.

A schedule for the greatest common divisor algorithm is shown in Figure 21. One subtractor/comparator component was allocated for the algorithm. Note, in states 1 and 2, multiple comparisons of **X** and **Y** have been scheduled to the same state. That is because a single comparator can generate each of these tests simultaneously as separate outputs. This demonstrates how parallel functionality in the component library can be exploited.

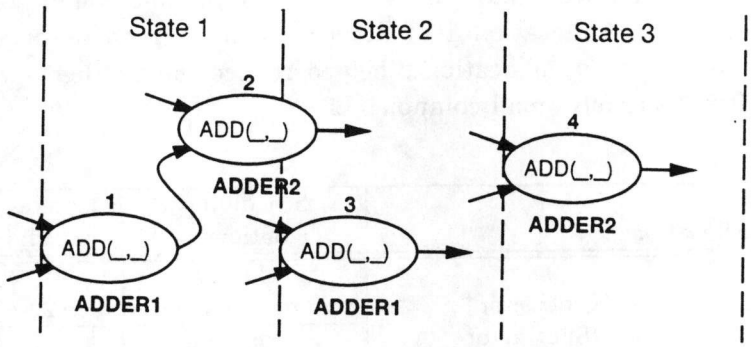

Figure 20: Monopolizing resources by combining chaining and multicycling: the chaining and multicycling of ADD() **1** and ADD() **2** forced ADD() **4** to be scheduled to a third state. But all four additions can be performed in only two states.

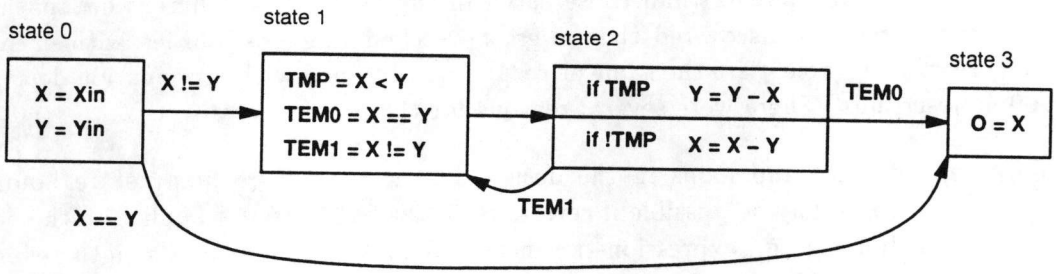

Figure 21: Schedule for greatest common divisor algorithm. 1 Subtractor/Comparator allocated.

Table 4 shows results of experiments done with a description of an IEEE standard 32-bit floating point adder/subtractor. These experiments demonstrate how a wide variety of components can be utilized by our scheduler because it recognize complex behavior. In this example, the library contained specialized components with unusual functionality that other scheduling algorithms are not formulated to deal with (e.g., 0-detection, concurrent subtraction and comparison). These experiments *reused* custom components which were incorporated into the system described in the previous section. We were able to explore different allocations and found that we could replace the 8-bit Adder/Subtractor in allocation 1 of Table 4 with an Incremetor/Decrementor. Also, allocating additional 0-detectors to allocation 2 helped reduce the number of states for the design while adding only a few, relatively small components.

	Allocation	Scheduling options	# states in schedule
1	8-bit Adder/Subtractor 27-bit Adder/Subtractor 27-bit Shifter 8-bit Subtractor/Comparator 23-bit Subtractor/Comparator 27-bit 0 Detector	no chaining no multicycling	22
		no chaining multicycling	22
		exclusive chaining & multicycling	22
		chaining multicycling	22
2	8-bit Incrementor/Decrementor 27-bit Adder/Subtractor 27-bit Shifter 8-bit Subtractor/Comparator 23-bit Subtractor/Comparator 27-bit 0 Detector 8-bit 0 Detector 23-bit 0 Detector	no chaining no multicycling	20
		no chaining multicycling	20
		exclusive chaining & multicycling	19
		chaining multicycling	19

Table 4: Results for 32-bit Floating Point Adder/Subtractor. Permitted datapath propagation delay: 20 ns.

We conducted more experiments than those listed in the table, with different datapath propagation delay limits. But we discovered there were few scheduling possibilities with each given allocation. The results in Table 4 are the same as results we obtained with propagation delay limits of 50 and 100 nanoseconds. There were several reasons for this:

1. The data-dependencies and loops in the design determined where many state boundaries were. More schedules may be possible if certain optimizations are first applied to the description before scheduling (e.g., expression-tree height reduction, operator strength reduction). However, the work described in this report is not concerned with such "code" transformations.

2. Most of the arithmetic components used had approximately the same delay. Lowering the propagation delay limit would force nearly all operations to be multicycled which is very impractical.

3. In the description, there were no long sequences of interdependent arithmetic operations. Consequently, there were no possibilities for chaining of operations.

You will note that the second allocation in Table 4 produced different length schedules of 20 and 19 states. The schedule with 20 states actually has an unneeded state which was introduced because of the heuristics used in our algorithm. However, even in such cases, the search done by our algorithm greatly minimizes such inefficiencies and derives a schedule close to the best possible.

6 Conclusions

In this paper, we presented a scheduling technique that enables reuse of existing datapath components from user-defined libraries. The reusable datapath components can exhibit more parallelism than simple, generic models of RT components used in the past. We believe this is an important issue that will enable the acceptance of HLS techniques for datapath-oriented designs – an area where traditional HLS tools are often discounted as "unrealistic" and "too naive" by real designers.

Our scheduling approach uses a heuristic-guided, branch-and-bound algorithm on our novel design representation for RT-functionality. The scheduling algorithm has been implemented in C on a Sun SPARC workstation. We presented experimental results on some HLS benchmarks and demonstrated the practical effectiveness of our approach. There are several fundamental advantages of our algorithm and formulation:

1. We believe this is the first scheduling algorithm based on a model of RT functional units that recognizes complex and concurrent functionality.
2. The algorithm and formulation are integrated into a system such that the user is able to describe the RT components they wish to use. The information captured and used by the algorithm includes not only the physical features of a component (e.g., pin-to-pin delays) but also how to recognize the abstract behavior of the RT component.
3. The algorithm allows exploration of the design space with different options for operation chaining and multicycling as well as specifying permitted functional unit propagation delays.

In our experiments, we observed that operation chaining and multicycling are most effective when done exclusive of each other.

Our approach is a step towards techniques for design reuse of realistic datapath components in HLS, but currently has some limitations:

- We only consider reuse of non-pipelined, combinatorial datapath components.
- The effects of interconnect, storage, and physical layout have not been incorporated into our approach.
- Timing constraints between operators has also not been explored.

Future work will address some of these issues with respect to using pre-defined components. Lastly, while the implementation of algorithm works with data structures for ISE, it is currently a separate tool that produces output readable by ISE. A tighter integration with ISE is planned.

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.

[AnDu94] R. Ang and N. Dutt, "Scheduling for Design Reuse of Datapath Components," Tech. Report 94-16 University of California at Irvine, May 1994.

[BrBr88] G. Brassard and P. Bratley, *Algorithmics: Theory and Practice*, Prentice-Hall, New Jersey, p. 15, 1988.

[Camp91] R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Transactions on Computer-Aided Design for Integrated Circuits and Systems*, vol. 10, no. 1, pp. 136–141, January 1991.

[ChGa92] V. Chaiyakul and D.D. Gajksi, "Assignment Decision Diagrams for High-Level Synthesis," Tech. Report 92-103 University of California at Irvine, Dec. 1992.

[GCDM95] W. Guerts, F. Catthoor, and H. De Man, "Quadratic Zero-One Programming-Based Synthesis of Application-Specific Data Paths," *IEEE Trans. on CAD*, vol. 14, no. 1 pp. 1–11, Jan 1995.

[Had95] T. Hadley, "A System for Interactive High-Level Synthesis," Ph. D. dissertation, U. C. Irvine, April, 1995.

[HLSD95] P. Panda and N. Dutt, "1995 High-Level Synthesis Design Repository," Tech. Report 95-04 University of California at Irvine, Feb. 1995. *available for anonymous FTP from ftp.ics.uci.edu.*

[JCG94] H.-P. Juan, V. Chaiyakul and D.D. Gajski, "Condition Graphs for High-Quality Behavioral Synthesis," *Proc. ICCAD 94*, 1994.

[KnWi92] D. Knapp and M. Winslett, "A Prescriptive Formal Model for Data-Path Hardware," *IEEE Transactions on Computer-Aided Design for Integrated Circuits and Systems*, vol. 11, no. 2, pp. 158–184, February 1992.

[LMD94] B. Landwehr, P. Marwedel, R. Domer, "OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming," *Proceedings of EURO-DAC*, 1994.

[LCGM93] D. Lanneer, M. Cornero, G. Goossens, and H. De Man, "An assignment technique for incompletely specified data-paths," *Proceedings of EDAC*, pp. 284-288, 1993.

[LKMM95] T. Ly, D. Knapp, R. Miller, D. MacMillen, "Scheduling using Behavioral Templates," *Proceedings of DAC*, 1995.

[McFar78] M. C. McFarland, "The Value Trace: A Data Base for Automated Digital Design," Ph. D. Dissertation, Dept. of Electrical Enginerring, Canegie-Mellon University, 1978.

[McCP88] M. C. McFarland, A. C. Parker, R. Camposano, "Tutorial on High-Level Synthesis," *Proceedings of DAC*, pp. 330-336, 1988.

[PaGa87] B. Pangrle and D. Gajski, "Slicer: A State Synthesizer for Intelligent Silicon Compilation," *Proceedings of the International Conference on Computer Design*, pp. 42–45, 1987.

[PaBu87] T.W. Parks, C.S. Burrus, *Digital filter design*, Wiley, New York, 1987.

[PaKn89] P. Paulin and J. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design for Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, June 1989.

[PaPM86] A. Parker, J. Pizarro, M. Mlinar, "MAHA: A Program for Datapath Synthesis," *Proceedings of the* 23^{rd} *Design Automation Conference*, pp. 461–466, 1986.

[PLNG90] R. Potasman, J. Lis, A. Nicolau, D. Gajski, "Percolation Based Synthesis," *Proceedings of the* 27^{th} *Design Automation Conference*, pp. 444–449, 1990.

[SaZa90] A. Safir and B. Zavidovique, "Towards a Global Solution to High Level Synthesis Problems," *Proceedings of EDAC*, pp. 283-288, 1990.

[VTI92] *0.8 Micron CMOS VCC4DP3 Datapath Library*, VLSI Technology, Inc., 1992.

[WPAV92] A. van der Werf, M. Peek, E. Aarts, J. Van Meerbergen, P. Lippens, W. Verhaegh, "Area Optimization of Multi-Functional Processing Units," *Proceedings of the International Conference on Computer-Aided Design*, pp. 292–299, 1992.

A Components

8-bit Adder/Subtractor (DPASB001H)
Input Ports: I0 (8-bit), I1 (8-bit), ICIN (1-BIT)
Output Ports: O0 (8-bit), OCOUT (1-BIT)
Delays:
I0 to O0: 9.3 ns
I1 to O0: 8.4 ns
ICIN to O0: 7.5 ns
I0 to OCOUT: 9.5 ns
I1 to OCOUT: 10.5 ns
ICIN to OCOUT: 8.3 ns

16-bit Adder/Subtractor (DPASB001S)
Input Ports: I0 (16-bit), I1 (16-bit), ICIN (1-BIT)
Output Ports: O0 (16-bit), OCOUT (1-BIT)
Delays:
I0 to O0: 11.0 ns
I1 to O0: 11.1 ns
ICIN to O0: 8.7 ns
I0 to OCOUT: 7.1 ns
I1 to OCOUT: 7.2 ns
ICIN to OCOUT: 4.8 ns

27-bit Adder/Subtractor (DPASB001S)
Input Ports: I0 (27-bit), I1 (27-bit), ICIN (1-BIT)
Output Ports: O0 (27-bit), OCOUT (1-BIT)
Delays:
I0 to O0: 11.4 ns
I1 to O0: 11.6 ns

ICIN to O0: 9.0 ns
I0 to OCOUT: 9.5 ns
I1 to OCOUT: 9.7 ns
ICIN to OCOUT: 7.1 ns

8-bit 0 detector (DPZDT001S)
Input Ports: I0 (8-bit)
Output Ports: O0 (1-bit)
Delays:
I0 to O0: 1.91 ns

23-bit 0 detector (DPZDT001S)
Input Ports: I0 (23-bit)
Output Ports: O0 (1-bit)
Delays:
I0 to O0: 3.95 ns

27-bit 0 detector (DPZDT001S)
Input Ports: I0 (27-bit)
Output Ports: O0 (1-bit)
Delays:
I0 to O0: 3.95 ns

8-bit 1s detector (DP1DT001S)
Input Ports: I0 (8-bit)
Output Ports: O0 (1-bit)
Delays:
I0 to O0: 1.52 ns

23-bit 1s detector (DP1DT001S)
Input Ports: I0 (23-bit)
Output Ports: O0 (1-bit)
Delays:
I0 to O0: 3.18 ns

8-bit Incrementor/Decrementor
Input Ports: I0 (8-bit)
Output Ports: O0 (8-bit)
Delays:
I0 to O0: 6.24 ns

16-bit Multiplier (DPMLT010H)
Input Ports: I0 (16-bit), I1 (16-bit)
Output Ports: MSB (16-bit), LSB (16-bit)
Delays:
I0 to LSB: 22.9 ns
I1 to LSB: 25.1 ns
I0 to MSB: 31.2 ns

I1 to MSB: 32.8 ns

16-bit Multiplier/Accumulator (DPMLT011H + DPADD001S)
Input Ports: I0 (16-bit), I1 (16-bit), I2 (16-bit)
Output Ports: O0 (16-bit)
Delays:
I0 to O0: 32.6 ns
I1 to O0: 34.7 ns
I2 to O0: 10.3 ns

27-bit L/R Shifter (using DPMUX2021)
Input Ports: I0 (27-bit), ILIN (1-bit), IRIN (1-bit)
Output Ports: O0 (27-bit)
Delays:
I0 to O0: 2.04 ns
ILIN to O0: 2.04 ns
IRIN to O0: 1.71 ns

8-bit Subtractor/Comparator (DPSUB001H + DPZDT001S)
Input Ports: I0 (8-bit), I1 (8-bit)
Output Ports: O0 (8-bit), OEQ (1-bit), ONEQ (1-bit), OGE (1-bit), OGT (1-bit), OLE (1-bit), OLT (1-bit)
Delays:
I0 to O0: 8.3 ns
I1 to O0: 9.0 ns
I0 to OEQ: 10.2 ns
I1 to OEQ: 10.9 ns
I0 to ONEQ: 10.7 ns
I1 to ONEQ: 11.4 ns
I0 to OGE: 12.8 ns
I1 to OGE: 13.5 ns
I0 to OGT: 14.2 ns
I1 to OGT: 14.9 ns
I0 to OLE: 14.2 ns
I1 to OLE: 14.9 ns
I0 to OLT: 12.8 ns
I1 to OLT: 13.5 ns

16-bit Subtractor/Comparator (DPSUB001S + DPZDT001S)
Input Ports: I0 (16-bit), I1 (16-bit)
Output Ports: O0 (16-bit), OEQ (1-bit), ONEQ (1-bit), OGE (1-bit), OGT (1-bit), OLE (1-bit), OLT (1-bit)
Delays:
I0 to O0: 10.2 ns
I1 to O0: 10.7 ns
I0 to OEQ: 13.4 ns
I1 to OEQ: 13.9 ns
I0 to ONEQ: 13.9 ns

I1 to ONEQ: 14.4 ns
I0 to OGE: 11.9 ns
I1 to OGE: 12.4 ns
I0 to OGT: 13.3 ns
I1 to OGT: 13.8 ns
I0 to OLE: 13.3 ns
I1 to OLE: 13.8 ns
I0 to OLT: 11.9 ns
I1 to OLT: 12.4 ns

23-bit Subtractor/Comparator (DPSUB001S + DPZDT001S)
Input Ports: I0 (23-bit), I1 (23-bit)
Output Ports: O0 (23-bit), OEQ (1-bit), ONEQ (1-bit), OGE (1-bit), OGT (1-bit), OLE (1-bit), OLT (1-bit)
Delays:
I0 to O0: 10.5 ns
I1 to O0: 11.0 ns
I0 to OEQ: 14.5 ns
I1 to OEQ: 15.0 ns
I0 to ONEQ: 15.0 ns
I1 to ONEQ: 15.5 ns
I0 to OGE: 13.4 ns
I1 to OGE: 13.9 ns
I0 to OGT: 14.9 ns
I1 to OGT: 15.4 ns
I0 to OLE: 14.9 ns
I1 to OLE: 15.4 ns
I0 to OLT: 13.4 ns
I1 to OLT: 13.9 ns