Assignment Decision Diagram for High-Level Synthesis

Viraphol Chaiyakul
Daniel D. Gajski

Technical Report #92-103
December 12, 1992

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

Abstract

In the past, the research on representation for synthesis systems had been focusing on two main issues, the completeness and the efficiency. There is, however, another important issue that is not addressed by most of traditional representations, the uniqueness. This report proposes a representation for synthesis called the Assignment Decision Diagram (ADD) that is complete, efficient and partially unique. In addition, the ADD also furnishes many synthesis tasks with information that can simplify the tasks, and can enrich the results of the synthesis. Discussion of ADD's properties and its uses in synthesis is provided in this report.

Contents

1 Introduction **5**

2 Assignment Decision Diagrams (ADD) **6**

3 Representing behavior and structure information in the Assignment Decision Diagrams **10**

3.1 Representing behavior information in the ADD 10

3.1.1 Variable type 10

3.1.2 Assignment construct 11

3.1.3 The conditional signal assignment, WHEN, construct 11

3.1.4 IF-THEN-ELSE construct 12

3.1.5 CASE construct 13

3.1.6 FOR construct 15

3.1.7 WHILE construct 15

3.1.8 WAIT construct 15

3.2 Representing structure information in the ADD 18

3.2.1 Functional Unit 18

3.2.2 Storage Unit 21

3.2.3 Interconnect Unit 23

3.2.4 Control Unit 23

4 High-level synthesis using the ADD **25**

5 Minimizing syntactic variance using the ADD **26**

6 Scheduling of the ADD **30**

7 Iterative, interleave and interactive scheduling and binding using the ADD **32**

7.1 Transformations of the ADD for scheduling 34

7.2 Transformation of the ADD for operator binding 41

7.3 Transformation of the ADD for interconnect binding 44

8 Accessing layout quality measures from the ADD **45**

9 "Partial" uniqueness of the ADD **48**

10 conclusion **49**

11 References **50**

List of Figures

1 The Assignment Decision Diagram: a) FSMD model, b) the ADD. 7

2 Representing assignment construct in ADD: (a) output port, registers and scalar variable assignment, (b) local signal, local bus, and temporary variable assignment (b) array variable assignment 12

3 The ADD representation of a WHEN assignment statement. 13

4 An example of the ADD representation for the IF-THE-ELSE construct. . . . 14

5 An example of ADD representation for the CASE construct. 14

6 An example of the ADD representation of a WHILE loop: (a) states table for a sequential representation, (b) ADD representation. 16

7 The ADD representations of different types of functional unit: (a) uni-cycle functional unit, (b) multi-functional unit, (c) multi-cycle functional unit, and (d) pipelined functional unit. 19

8 Representation of register file and memory in ADD: (a) reading from a multiple output port register file, (b) representation of a multiple output port register file in the ADD, (c) writing to a multiple input port register file, and (d) representation of a multiple input port register file in the ADD. 22

9 Examples of the ADD representation for different implementation style of a bus and a multiplexer: (a) representation of an one level bus or multiplexer in ADD, (b) single level bus, (c) single level multiplexer, (d) representation of a segmented bus or a multiple level multiplexer in ADD, (e) segmented bus, (f) multiple level multiplexer. 24

10 An example of a state table. 25

11 A general high-level synthesis approach. 27

12 Overview of the proposed approach. 28

13 An example of transforming a VHDL input description into ADD. 29

14 An example of scheduling the branch conditions and operations in the branch in the same control step: (a) Using the ADD, (b) the scheduling effect. 31

15 An example of scheduling operations across basic blocks: (a) Using the ADD, (b) the scheduling effect. 32

16 An example of merging mutually exclusive operations before scheduling: (a) before merging, (b) after merging. 33

17 An example of merging mutually exclusive operations during scheduling. . . . 33

18 Interactive scheduling and binding using ADD 34

19 Two types of transformations for scheduling in Assignment Decision Diagram: (a) linear state insertion, and (b) state branches insertion. 35

20 Linear state insertion 36

21 An example of transformation for linear state insertion: (a) ADD before the transformation and (b) ADD after the transformation. 38

22 An example of transformation for state branches insertion: (a) ADD before the transformation and (b) ADD after the transformation. 40

23 An example of transformation for operator merging: (a) ADD before the transformation and (b) ADD after the transformation. 43

24 An example of transformation for interconnect merging. 46

25 Estimating layout quality from the ADD. 47

26 Summary of features provided by the ADD that are complicated to obtain from CDFT, VT, and CF-DFG. 49

1 Introduction

High-level synthesis is a process of synthesizing designs from given abstract behavioral descriptions. In general, the process can be divided into several tasks that include: compiling descriptions into an internal representation, transforming the internal representation into a more suitable form for synthesis, partitioning operations into control steps, and binding operations and interconnects to appropriate resources. Most of these tasks require to make design-tradeoff decisions based on information extracted from the internal representation. Hence, generally speaking, the main backbone of a synthesis system is its internal representation.

In the past, the research on representation for synthesis systems had been focusing on two main issues, the completeness and the efficiency. The completeness is defined as the ability to encode every piece of information that is given in the input description and is required in constructing a design. The encoding should be accomplished such that information can be easily and efficiently extracted during synthesis. The simplicity of extracting information from internal representation defines the efficiency of the representation. There is, however, another important issue that is not addressed by most of traditional representations, the uniqueness of the representation. The uniqueness of a representation is defined as the ability to represent a given input description in a unique form. In other words, for every given input description, there must exist a unique way of representing the description that is independent the from writing styles or the usages of language constructs in that description. A unique internal representation can tremendously simplify the use of the synthesis system because users can describe the functionality of the intended design with any language constructs or writting styles and still obtaining the same synthesized hardware.

Traditional representations, such as CDFG [10, 12], VT [11], and CF-DFG [3], satisfy the completeness property by encoding the input description into the representation in a one-to-one mapping manner. In other words, each language construct in the description is realized with a particular topology of nodes in the representation. For example, a VHDL[18] description can be compiled into a CDFG by mapping computations in a basic block to nodes in a data-flow graph, and a conditional constructs to a control nodes [10, 12]. Similar mappings from VHDL to CFG and from ISPS [1] to VT can be found in [3, 4, 11, 16]. Although these representations shown to be efficient, they lack the uniqueness. Since there exists a one-to-one correspondence between the constructs of input descriptions and the schema for the internal representation, different descriptions would result in different representation. The internal representations of two given descriptions could be far different even if the descriptions are semantically equivalent.

This report proposes a representation, the Assignment Decision Diagram (ADD), that is complete, efficient and "partially" unique. The constituents of the Assignment Decision Diagram is discussed in section 2. To demonstrate the completeness of the

ADD, representation of frequently used language constructs and structure are presented in section 3. The efficiency of the ADD is discussed in section 4 and detailed discussion of useful features of the ADD in minimizing syntactic variance, scheduling, interactive synthesis, and quality measures are provided in section 5, 6, 7 and 8 respectively. The ADD is "partially" unique because it can uniquely represent a set of descriptions that are written with different grouping and ordering of conditional and assignment statements. However, descriptions that are differed in the ordering or grouping of unbounded loop will not be represented uniquely in the ADD. Discussion of the ADD's "partial" uniqueness is provided in section 9. Finally, section 10 shows the summary of features furnished by the ADD for synthesis tasks and its efficiency as compare to traditional representations.

2 Assignment Decision Diagrams (ADD)

The primary objectives in deriving ADD is to define a representation that is capable of encapsulating functionality of the described hardware in a unique, precise, and simple manner. We regard these three objectives with utmost importance because of the following reasons.

- The uniqueness of the representation will allow synthesis tools to be independent of syntactic variances that are present in the input description. In [5] we have proposed the most parallel representation to be the unique representation. Hence, the ADD has to be able to depict the most parallel representation of a given description in order to satisfy the uniqueness property.
- In addition to being unique, the representation we are seeking should consists of parts that reflect semantics of the description instead of syntactic constructs. Each constituent of the presentation should have no direct relationship with language constructs. We refer to this property as the preciseness of the representation.
- A simple representation is one that consists of a few number of different object types and relationship between each object type. Such representation can simplify synthesis algorithms because the algorithms have to manage small number of objects. Since most of the synthesis algorithms are topology-graph based, the representation for a synthesis system has to be a form of topology graph. Thus, a simple representation is, ideally, a graph that consists of small number of different types of nodes and edges.

To derive the ADD, let us first define a hardware model that could be used to describe any digital systems.

Digital systems can be classified into two basic groups, namely, the combinational system and the sequential system. Functionality of both combinational and sequential

systems can be described by the Finite-State-Machine and the Datapath model (FSMD) [7] as shown in Figure 1(a). Note that in the case of a combinational system, storage units will be absent from the model. The FSMD model can be viewed as assignments of values to storage units (either the state variable or storage units in the datapath) and output ports based on certain conditions and the state of the system. These conditional assignments are represented in the Assignment Decision Diagram shown in Figure 1(b). It should be noted that the state variable is represented in the ADD the same way as

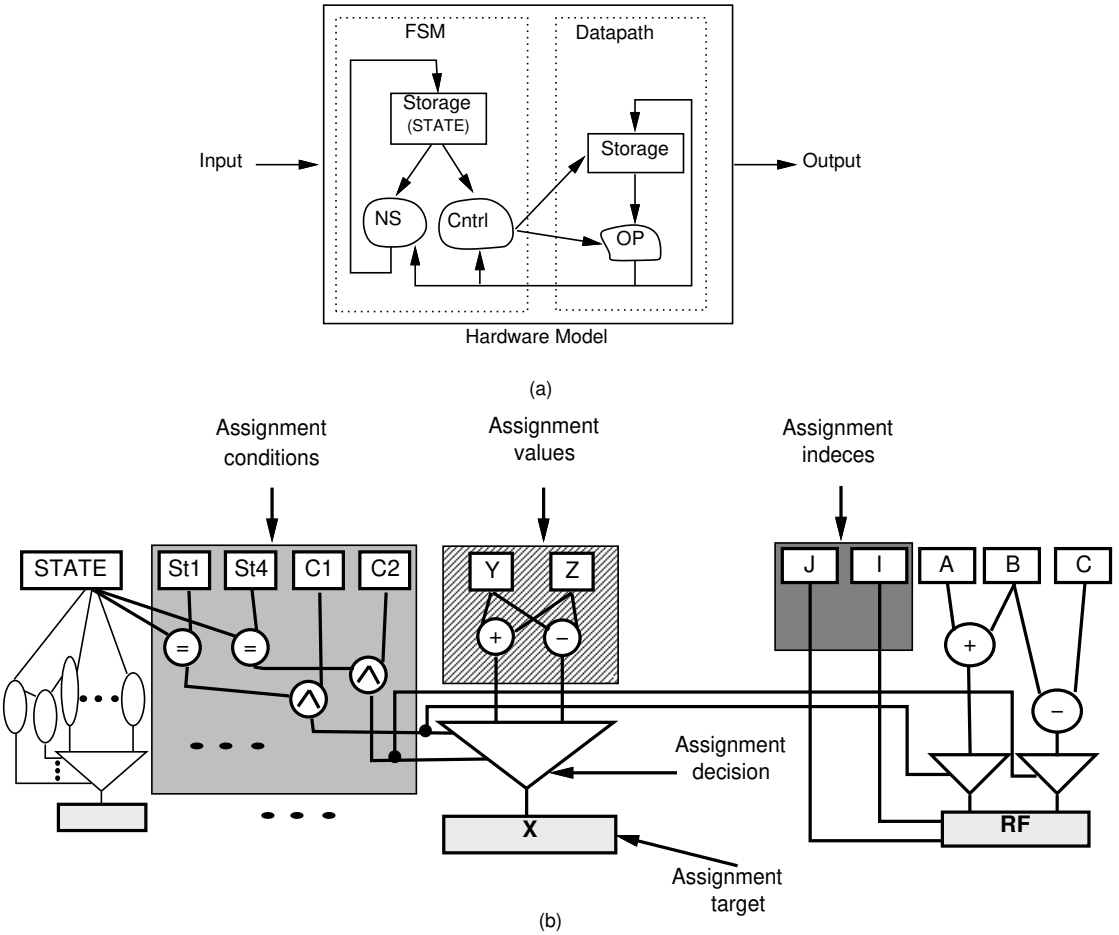


Figure 1: The Assignment Decision Diagram: a) FSMD model, b) the ADD.

any other variables in the design. The ADD representation consists of five parts: (1) the assignment value, (2) the assignment condition, (3) the assignment decision, (4) the assignment target, and (5) the assignment index. These parts are implemented with four types of nodes: operation nodes, read nodes, write nodes and assignment decision nodes (ADN).

The *assignment value* part consists of read nodes and operation nodes. This part represents the computation of values that are to be assigned to a storage unit or an output port. The value is computed from current contents of storage units, input ports, or constants. These are represented by read nodes. The actual computation is represented as a data-flow graph that contains operator nodes, which correspond to the type of operations that are performed.

The *assignment condition* part consists of read nodes and operation nodes that are connected as a data-flow graph to represent computation of a condition. The end product of the condition-computation is a binary value that evaluates to *true* or *false*. This *true/false* value is used as a guarding condition for the assignment value.

The *assignment-decision* part consists of an Assignment Decision Node (ADN). The ADN selects a value from a set of values that are provided to it. These input values are computed by the *assignment-value* part of the ADD. The selection is based on the conditions computed by the *assignment condition* part of the ADD. If one of the conditions to the ADN evaluates to *true* then the corresponding input value is selected. It is also possible that none of the conditions of a ADN evaluate to *true* at a given time. In this case none of the input values are selected.

The *assignment-target* is represented by a write node. The write node is provided with the selected value from the corresponding ADN. A value will be assigned to the write node, only if one of the conditions to the corresponding ADN is *true*. And since only one value can be assigned to a target at any given time, all assignment conditions for each target are mutually exclusive. For example, Figure 1(b) shows $(Y + Z)$ will be assigned to X if and only if $C1$ is true and the hardware is in state $ST1$. Likewise, $(Y - Z)$ will be assigned to X in state $ST4$ and the condition $C2$ is true. If none of the conditions is true then X will retain its previous value.

The *assignment index* is used in an ADD expression when the corresponding read or write node represents an array (i.e., two dimensional storage units) variable, The assignment index is used to indicate the location where the assignment values are written. Figure 1(b) shows an example of an ADD that represents reading and writing to the array variable RF. The value $(A + B)$ will be assigned to $RF[I]$ if and only if $C1$ is true and the hardware is in state $ST1$. Likewise, $(B - C)$ will be assigned to $RF[J]$ if and only if the hardware is in state $ST4$ and $C2$ is true.

One of the unique features of ADD is its capability to represent conditions and computations in a consistent data-flow fashion. Thus, operations in ADD are ordered by their data dependency only. In other words, ADD is free of control dependency that are introduced in the description. With this capability, ADD can represent the most parallel implementation for a given description.

In addition to representing the most parallel representation, ADD can be used to represent multi-state designs. Such multi-state designs become necessary if the descrip-

tion contains a loop construct with variable bounds. In this case, the corresponding ADD would contain a special storage unit called $State_reg$ that represents the control step-sequencer. This $State_reg$ has the same representation as any other storage units. Assignments to $State_reg$ represent the sequencing of control steps, where each assignment value is a constant that represents a control step, and each assignment condition represents the sequencing between the steps.

ADD can be implemented as an undirected acyclic graph, where each read, write, operation or assignment decision node is implemented as a node with different attributes, and connections are implemented as undirected edges. Representing a description would require, in the worst case, a graph whose size is proportional to the number of conditional assignments to all the ports and storage units.

In addition to the topological graph representation of the ADD, we have derived an equation or tex-based description of the ADD. The ADD equation will be used in many parts of this report especially in describing algorithms. A general equation for an assignment in ADD is as follows:

$$\alpha = \langle \beta_0\{\gamma_0\} \ \oplus \ \beta_1\{\gamma_1\} \oplus \ \ldots \ \oplus \ \beta_n\{\gamma_n\}\rangle[\theta_0] \ , \ \langle \ldots\rangle[\theta_1] \ , \ \ldots \ , \langle \ldots\rangle[\theta_n].$$

where α represents an assignment target or a temporary signal. Each assignment statement consists n number assignment actions ($\langle \ldots\rangle[\theta_i]$). Each assignment actions represents the assignment to an index of the corresponding assignment target. Thus, n is the maximum number of assignment index for the corresponding assignment target (For example, assignment to a scalar variable will contain only one assignment action, i.e., $n = 1$). The expression for each assignment action is separated by "," and the last assignment action ends with a ".". Each assignment action comprises of three components: the assignment value, γ_i, the assignment condition, β_i, and the assignment index, θ_i. Only one assignment index is given for each assignment action and it is enclosed in a pair of "[]". For clarity purposes, assignment values are enclosed in a "{}", and "$\langle\rangle$" are used as delimiters for the assignment index and the rest of components in an assignment action. For example, the assignments to X and RF in Figure 1(b) can be written in the ADD equation as follows:

$$\begin{aligned}
temp1_{(type:signal)} &= \{STATE = St1 \wedge C1\}. \\
temp2_{(type:signal)} &= \{STATE = St4 \wedge C2\}. \\
X_{(type:storage)} &= temp1\{Y + Z\} \ \oplus \ temp2\{Y - Z\}. \\
RF_{(type:storage)} &= \langle temp1\{A + B\}\rangle \ [I] \ , \ \langle temp2\{B - C\}\rangle \ [J].
\end{aligned}$$

The ADD is to be used in representing both behavior and structure information. This is because we conceive the synthesis process as tasks that are directly or indirectly

"massaging" the ADD. Basically, the input behavior description is transformed into an ADD that will be manipulated by synthesis algorithms. The manipulation may involve addition of information or transformation of the ADD topology. For example, scheduling would transform assignments to the $State_reg$ and adding temporary registers to the ADD, whereas, functional unit binding would transform the ADD by merging operator nodes. As the final result, we would obtain an ADD that can be mapped directly to a structure netlist. In the next section we illustrate the completeness of the ADD by discussing the representation of behavior and structure information in the ADD.

3 Representing behavior and structure information in the Assignment Decision Diagrams

In order to be complete, ADD should be able to represent any behavior and structure information. This section is divided into two parts that discuss the representation of behavior and structure information in ADD, respectively.

3.1 Representing behavior information in the ADD

The input behavior description to the synthesis system is usually written in a high-level language such as VHDL [18], ISPS [1], or C. In our synthesis-framework, we have chosen VHDL as the input language due to its popularity and recognition as the standard high-level descriptive language. We demonstrate the ability of representing behavioral level information in the ADD by illustrating the representation schema of a set of commonly used VHDL constructs, namely, the variable type, the assignment construct, the WHEN construct, the IF-THEN-ELSE construct, the CASE construct, the WHILE construct, the FOR construct and the WAIT construct. Nevertheless, the same representation scheme can also be derived for other high level descriptive languages.

3.1.1 Variable type

Each variable in the description carries a set of properties that will be represented in the ADD as attributes. Three basics attributes are considered in the ADD, type attribute, dimension attribute and size attribute. The type attribute can be of three kinds, namely a signal, a port, or a storage unit. The dimension attribute can have value of one, which represent scalar variables, or two, which represent array variables. The size attribute indicates bitwidth of the variable. Any variables in the description that are declared as complex data-type will have to be converted a simple type that can be described by these three basic attributes.

For example, a two dimensional variable A of the size 16x8 is represented in ADD as follows:

$$A_{(type:storage,dim:2,sz:(16,8))}$$

3.1.2 Assignment construct

The assignment in VHDL can be classified into five classes based on the type of the assignment target: namely, assignment to output ports, assignment to registers, assignment to variables, assignment to signals and buses, and assignment to array variables. In ADD, the type of a unit (an assignment target or an operand in the assignment statement) is represented as a type attribute. Input/Output ports are represented as the *port* type, local (w.r.t. the VHDL process) signals, local buses and temporary scalar-variables are represented as the *signal* type, while registers, scalar and array variables are represented as the *storage* type. Global signals and buses have to be resolved using the approach in [15] prior to this representation scheme. A unit of the *port* type or of the *storage* type are represented in the graph as a read or a write node, depending on the usage nature of the unit. On the other hand, a unit of a *signal* type is represented as an edge in the graph. The size and dimension of each unit is realized with the size and the dimension attributes, as described in the previous section.

The following table contains examples that illustrate representation of each assignment type in the ADD.

VHDL assignment construct		
Type of the assignment target	*Assignment statement*	*Representation in the ADD*
16-bit output port A	$A <= B + C;$	$A_{(type:port,dim:1,sz:(16))} = \{B + C\}$.*Figure 2(a)*
16-bit registers A	$A := B + C;$	$A_{(type:storage,dim:1,sz:(16))} = \{B + C\}$.*Figure 2(a)*
16-bit variable A	$A := B + C;$	$A_{(type:storage,dim:1,sz:(16))} = \{B + C\}$.*Figure 2(a)*
16-bit local signals A	$A <= B + C;$	$A_{(type:signal,dim:1,sz:(16))} = \{B + C\}$.*Figure 2(b)*
16-bit local buses A	$A <= B + C;$	$A_{(type:signal,dim:1,sz:(16))} = \{B + C\}$.*Figure 2(b)*
$8 \times$ 16-bit array A	$A[i] := B[j] + C;$	$A_{(type:storage,dim:2,sz:(16,8))} = \langle\{B[j] + C\}\rangle[i]$. *Figure 2(b)*

3.1.3 The conditional signal assignment, WHEN, construct

Conditional signal assignment in VHDL is in the form of a GUARDED construct, a WHEN construct, or a combination of both. VHDL allows the use of GUARDED assignment only if the type of the assignment target is a bus or a register. Whereas, WHEN construct can be used with a signal, a bus or a register. The syntax of conditional assignment is shown as follows:

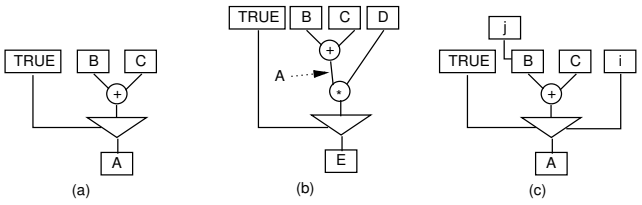


Figure 2: Representing assignment construct in ADD: (a) output port, registers and scalar variable assignment, (b) local signal, local bus, and temporary variable assignment (b) array variable assignment

```
signal <= [guarded]
  value1 when condition1 else
  value2 when condition2 else
  ...
  valueM when conditionM else
  valueN;
```

Values are assigned to the target only if the guard and/or its correspondent condition evaluates to true.

The conditional signal assignment is represented in ADD as a parallel task of conditions evaluation and values assignment. Each value is directly mapped to the assignment value, while its correspondent condition is evaluated and stored in a signal that is used as the assignment condition. For example, the ADD representation for the conditional assignment statements:

```
A := B + C when (E < F) else
     B - C;
```

where A, B, C, E and F are 16-bit signals, is as follows:

$$\begin{aligned} temp_{(type:signal,dim:1,sz:1)} &= \{E < F\}. \\ A_{(type:signal,dim:1,size:(16))} &= temp1\{B + C\} \oplus \overline{temp}\{B - C\}. \end{aligned}$$

The graphical representation of this ADD is shown in Figure 3.

3.1.4 IF-THEN-ELSE construct

IF-THEN-ELSE construct is used in the behavioral description to produce sequential condition-branching effect. The IF part is accompanied by a condition that if evaluates

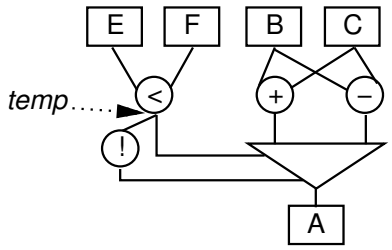


Figure 3: The ADD representation of a WHEN assignment statement.

to true will cause the execution of actions in the THEN part, otherwise actions in the ELSE part is executed (the ELSE part is optional).

A parallel representation of the IF-THEN-ELSE construct requires simultaneous evaluation of conditions and execution of statements in the THEN and ELSE part. For example, consider the following block of VHDL code:

```
if (C == "0010") then
   A := B + D;
else
   A := B - D;
endif;
```

where, A, B, C and C are 4-bit registers. The ADD representation of this example is as follows:

$$\begin{aligned} temp_{(type:signal,dim:1,size:(1))} &= \{C < \text{“0010”}\}. \\ A_{(type:storage,dim:1,size(4))} &= temp\{B+D\} \oplus \overline{temp}\{B-D\}. \end{aligned}$$

The correspondent graphical representation of the above ADD is shown in Figure 4.

3.1.5 CASE construct

Similar to the IF-THEN-ELSE construct, the CASE construct provides multi-way branching capability to the sequential description. Actions in each branch is executed if its companion condition evaluates to true.

The parallel representation of the CASE construct requires simultaneous evaluation of conditions and execution of operations in each branches of the CASE. For example, consider a VHDL CASE statement below:

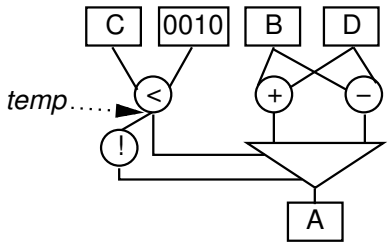


Figure 4: An example of the ADD representation for the IF-THE-ELSE construct.

```
case (C + E) is
   when "0001" => A = "0100";
   when "0101" => A = "1111";
   when "1011" => A = "0110";
end case;
```

where A, C, and E are 4-bit registers. The ADD representation of this example is as follows:

$$
\begin{aligned}
temp0_{(type:signal,dim:1,size:(4))} &= \{C + E\} \\
temp1_{(type:signal,dim:1,size:(1))} &= \{temp0 == \text{“0001”}\} \\
temp2_{(type:signal,dim:1,size:(1))} &= \{temp0 == \text{“0101”}\} \\
temp3_{(type:signal,dim:1,size:(1))} &= \{temp0 == \text{“1011”}\} \\
A_{(type:storage,dim:1,size(4))} &= temp1\{\text{“0100”}\} \oplus temp2\{\text{“1111”}\} \oplus temp3\{\text{“0110”}\}
\end{aligned}
$$

The correspondent graphical representation of this ADD is shown in Figure 5.

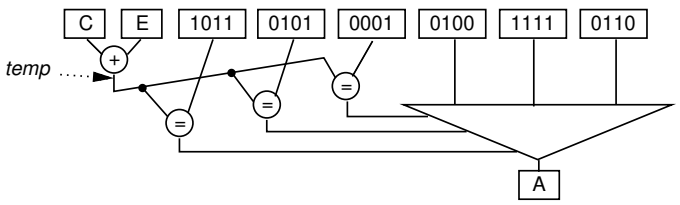


Figure 5: An example of ADD representation for the CASE construct.

3.1.6 FOR construct

FOR loop is used to describe a sequence of statements that is to be executed repeatedly in a deterministic number of time. The FOR construct uses an index variable whose value steps through a specified range for each iteration of the loop. The most parallel representation for a FOR loop is to unroll the loop and substituting values of the index variable for each unrolled instant.

3.1.7 WHILE construct

The WHILE construct is used to describe a non-deterministic loop where loop exit condition is not known before the execution time. Hence, the loop has to be iterated in a sequential manner. The most parallel representation of a WHILE loop requires 3 control steps: namely, the loop-entry step, the loop-body step, and the loop-exit step. In the loop-entry step, the loop condition is evaluated and if the result is true then the loop-body step is assigned as the next control step, otherwise, the next control step will be the loop-exit step. The same branching of control step is performed in the loop-body step. For example, consider a WHILE loop described in VHDL below:

```
D := 0;
while (D < E) loop
   A := D + A;
   D := D + 1;
end loop;
```

where A, D, and E are 4-bit registers. The ADD representation of this description is as follows:

$$
\begin{aligned}
temp_{(type:signal,dim:1,size:(1))} &= \{D < E\}. \\
State_register &= st_2\{st_0\} \oplus ((st_0 \wedge temp) \vee (st_1 \wedge temp))\{st_1\} \\
&\quad \oplus ((st_0 \wedge \overline{temp}) \vee (st_1 \wedge \overline{temp}))\{st_2\}. \\
A_{(type:storage,dim:1,size:(4))} &= st_1\{D + A\}. \\
D_{(type:storage,dim:1,size:(4))} &= st_0\{\text{"0000"}\} \oplus st_1\{D + \text{"0001"}\}.
\end{aligned}
$$

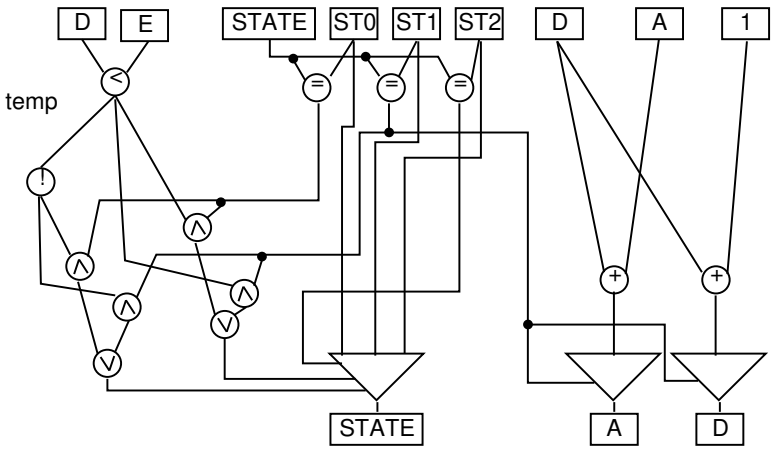
The state transition table and the graphical representation of the ADD is shown in Figure 6.

3.1.8 WAIT construct

Syntax for WAIT statement in VHDL is shown below:

State	Cond	Next St.	Actions
ST0	temp	ST1	temp := (D < E)
	$\overline{\text{temp}}$	ST2	
ST1	temp	ST1	A := D + A; D := D + 1; temp := (D < E)
	$\overline{\text{temp}}$	ST2	
ST2		ST0	

(a)



(b)

Figure 6: An example of the ADD representation of a WHILE loop: (a) states table for a sequential representation, (b) ADD representation.

```
wait [until <condition>] [for <time>];
```

A WAIT statement is used as the synchronization mechanism by suspending the execution of a process until a specified condition is true, or a specified time period elapses. WAIT statement is interpreted as a synchronous event in the ADD representation. In other words, the evaluation of the WAIT condition is performed at the edge of the clock. Hence, WAIT for a condition, C, is represented in a similar manner as an empty WHILE loop (a WHILE loop with empty loop body) that exits on the condition, C. For example, consider the following WAIT statement:

```
wait until (A < B);
```

where A is a 4-bit register and B is a 4-bit input port. This WAIT is represented in ADD the same way as the following WHILE loop:

```
while (A < B) loop
        end loop;
```

that is,

$$\begin{aligned} temp_{(type:signal,dim:1,size:(1))} &= \{A < B\} \\ State_register &= st_2\{st_0\} \oplus ((st_0 \wedge temp) \vee (st_1 \wedge temp))\{st_1\} \oplus \\ &\quad ((st_0 \wedge \overline{temp}) \vee (st_1 \wedge \overline{temp}))\{st_2\} \end{aligned}$$

On the other hand, a WAIT for a specified time period, t, is interpreted as a wait for a constant number of clock cycle, $Count$, where $Count = t/cp$ and cp is an estimated clock period. A WAIT for $Count$ number of clock cycle is represented in ADD as a WHILE loop with a counter that is initialized to 1 and increments by 0 at every loop iteration. The loop is exited when the counter reaches $Count$. For example, consider the following WAIT statement:

```
wait for 200 ns;
```

where the estimated clock period is 20 ns. This WAIT is represented the same way as the following WHILE loop:

```
count = 0;
while (count != 10) loop
  count = count + 1;
end loop;
```

3.2 Representing structure information in the ADD

Structural information consists of description of structure components in the design (e.g., ALU, adder, AND gate, OR gate) and their interconnectivities. In this section, we illustrate methods of representing a set of commonly used structure components and interconnectivities in the ADD.

Structure information can be classified into four categories; namely, functional unit, storage unit, interconnect unit, and control unit.

3.2.1 Functional Unit

Functional unit consists of logic that are used to manipulate data. Components in the functional units can be further classified into five types depending on the number of execution steps they required, or the number of operations they can perform; these are uni-cycle, multi-function, multi-cycle, pipelined, and macro functional unit.

- Uni-cycle functional unit

  A uni-cycle functional unit is represented in the ADD as an operator (arithmetic or logic) in the computation of the assignment value. For example, a 16-bit adder is represented as:

  $$A_{(type:signal,dim:1,size:(16))} = \{B + C\}.$$

  where B and C are inputs and A is the output of the adder. The correspondent graphical ADD is shown in Figure 7(a).

- Multi-functional unit

  A muti-functional unit is a component that has the ability to perform more than one operations. The unit requires control signals to indicate the type of function to be performed at any given instant of time. A multi-functional unit is represented in the ADD similar to the representation of a function in a programing language. The name of the unit is represented as the function name while inputs and control signals are represented as input parameters to the function. The set of functions that the unit can perform are listed as attributes of the output signal. For example, a 16-bit ALU that can perform four functions, namely add, subtract, left shift, and right shift, can be represented in the ADD as follows:

  $$A_{(type:signalsize:(16),func:(+,-,LSH,RSH))} = \{ALU(B, C, cntl)\}.$$

  where B and C are inputs of the ALU, $cntl$ is the control signal that indicates the type of function to be performed, and A is the output of the ALU.

  A multi-functional unit can have more than one type of output. For example, an ALU that performs add ($+$), subtract ($-$), less than comparison ($<$) and greater

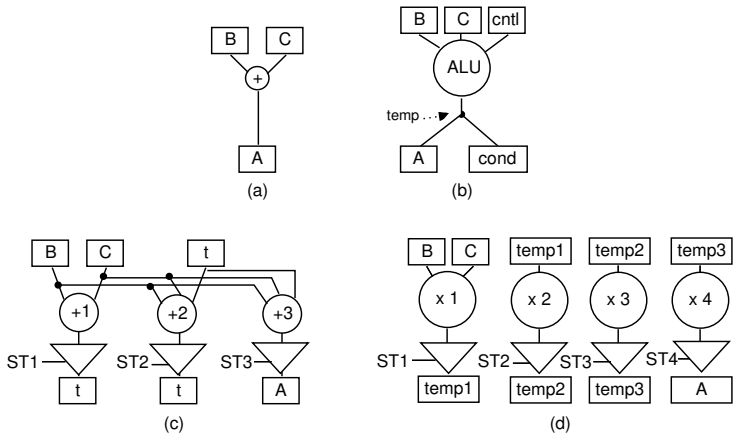


Figure 7: The ADD representations of different types of functional unit: (a) uni-cycle functional unit, (b) multi-functional unit, (c) multi-cycle functional unit, and (d) pipelined functional unit.

than comparison (>) can provide two types of output; data from the two arithmetic operations (+,−), or status from comparison operations (<,>). When representing such unit in the ADD, all outputs are bundled and assigned to a temporary signal. Different output types are then extracted from this temporary signal. For example, an ALU, which has functionalities as described earlier, with output data connects to register A and status data connects to register *cond* is represented in the ADD as follows:

$$\begin{aligned} temp_{(type:signal,size:(17),func:(+,-,<,>))} &= \{ALU(B,C,cntl)\}. \\ A_{(type:register,size:(16))} &= \{tem(0..15)\}. \\ cond_{(type:register,size:(1))} &= \{tem(16)\}. \end{aligned}$$

where, *temp* is a temporary signal. Bit 0 to 15 of *temp* are data output, while the bit 16th is the output of the comparison. The correspondent graphical ADD of the above ALU is shown in Figure 7(b)

- Multi-cycle functional unit

  Each multi-cycle functional unit is represented by n number of nodes, where n is the number of clock cycle required by the unit. Each of the n nodes represents a part of the operation performed in one clock cycle. For example, Figure 7(c) shows the representation of a 3 clock-cycle adder. Intermediate results obtained from each clock cycle are represented by a variable, t, that does not have any structural or

functional meaning. It is used to signify the dependency between operation at each clock cycle. It should be noted that operands of the function are required for all n number of cycle. This is very crucial because it reflects the required life time of the operands. The corresponding ADD equation for Figure 7(c) is as follows:

$$\begin{aligned} t_{(type:dependency,..)} &= ST1\{B +_1 C\} \oplus ST2\{+_2(B,C,t)\}. \\ A &= ST3\{+_3(B,C,t)\}. \end{aligned}$$

- Pipelined functional unit

  In a pipelined functional unit, the computation task is subdivide into a sequence of subtasks, each of which can be executed by a hardware stage that operates concurrently with other stages in the pipeline. Intermediate results from each stage is kept in a storage unit that is used as inputs to the next stage. A pipelined functional unit is represented in the ADD as multiple assignment statements to storage units. Each assignment represents the storing of the intermediate result for a pipelined stage. To indicate the stage in which the assignment is performed, the operator in the assignment computation is subscripted with a stage number. Inputs to the pipelined unit are kept in some storage units, which are embeded in the pipelined unit. Only the computation in the first stage requires content of the operands. Computations in successive stages operate only on output from the previous stages. This is the main difference between a multi-cycle unit and a pipelined unit. It should be noted that the size of storage units that store results of each intermediate pipe stage may not reflect the real size as required in the hardware. Example of the representation of a 16 bits multiplier with 4-stages pipelined is shown as follows:

$$\begin{aligned} temp1_{(type:storage,dim:1,size:(16),func(\times))} &= ST1\{B \times_1 C\}. \\ temp2_{(type:storage,dim:1,size:(16),func(\times))} &= ST2\{\times_2 temp1\}. \\ temp3_{(type:storage,dim:1,size:(16),func(\times))} &= ST3\{\times_3 temp2\}. \\ A_{(type:signal,dim:1,size:(16))} &= ST4\{temp3\}. \end{aligned}$$

  The correspondent graphical ADD representation of the above pipelined multiplier is shown in Figure 7(d).

- Macro functional unit

  A macro unit is a component that is designed to carry out a predefined set of operations; for an example, an incrementer is a macro unit that computes the value of 1 plus its input value. The macro functional unit is represented in the ADD the same way as any other functional unit depending on its execution property, i.e., pipelined, multi-state etc.

3.2.2 Storage Unit

Storage units can be classified into two classes based on the dimension of the storage cells: namely, register (one-dimension), and register file and memory (two-dimensions). Representations of storage unit in ADD are described below:

- Register

  A register consists of a single dimensional array of storage cells with an input port and an output port. Dimension and size of a register is represented in the ADD as attributes of the assignment target. For example, a 16-bit register is represented as

  $$A_{(type:storage,dim:1,size(16))} =$$

  Reading from a register is represented as a variable in the computation of the assignment value. Where as writing to a register in the ADD is represented as a simple assignment to an assignment target of the storage type. For example, consider the assignment $A = B + C$ where A, B and C are 16-bit registers. This is represented in the ADD as follows:

  $$A_{(type:storage,dim:1,size(16))} = \{B + C\}.$$

- Register file and memory

  Register file and memory are storage units that consist of two-dimensional array of storage cells. They can be viewed as an one-dimensional array of registers with direct access mechanism to any register of the array. In the ADD, dimension and size of each unit is represented as attributes of the assignment target. For example, an 8×16-bit register file is represented as

  $$A_{(type:storage,dim:2,size(16,8))} =$$

  where the size in the first dimension (i.e., 16) represents number of registers in the register file or memory, and the second dimension (i.e., 8) represents number of bits for each register.

  A register file or a memory can have more than one input or one output port. Each port contains an index whose bit-width is greater than or equal to $\lceil \log n \rceil$, where n is the number of register in the array. For example, an 16×8 register file requires at least $\lceil \log 16 \rceil = 4$ bits for the index. The index is used to indicate the register that is being accessed. Representations of the reading and write to the register file or the memory are described below.

  Reading from a register file or a memory is represented as a group of assignment to temporary signals each of which represents an output port of the register file or memory. An index signal is assigned to each output port to indicate the location

of the register where the content is read. For example, the ADD representation of read accesses from the register file shown in Figure 8(a) is as follows:

$$
\begin{aligned}
O1_{(type:signal,dim:1,size(16))} &= \{A[\text{“}0100'']\}. \\
O2_{(type:signal,dim:1,size(16))} &= \{A[Oaddr2]\}. \\
O3_{(type:signal,dim:1,size(16))} &= \{A[Oaddr3]\}. \\
B_{(type:signal,dim:1,size(16)))} &= \{O1 + \text{“}0x0001''\}.
\end{aligned}
$$

where A represents the register file or memory (e.g., $A_{(type:storage,dim:2,size(16,8))}$). The correspondent graphical ADD representation is shown in Figure 8(b).

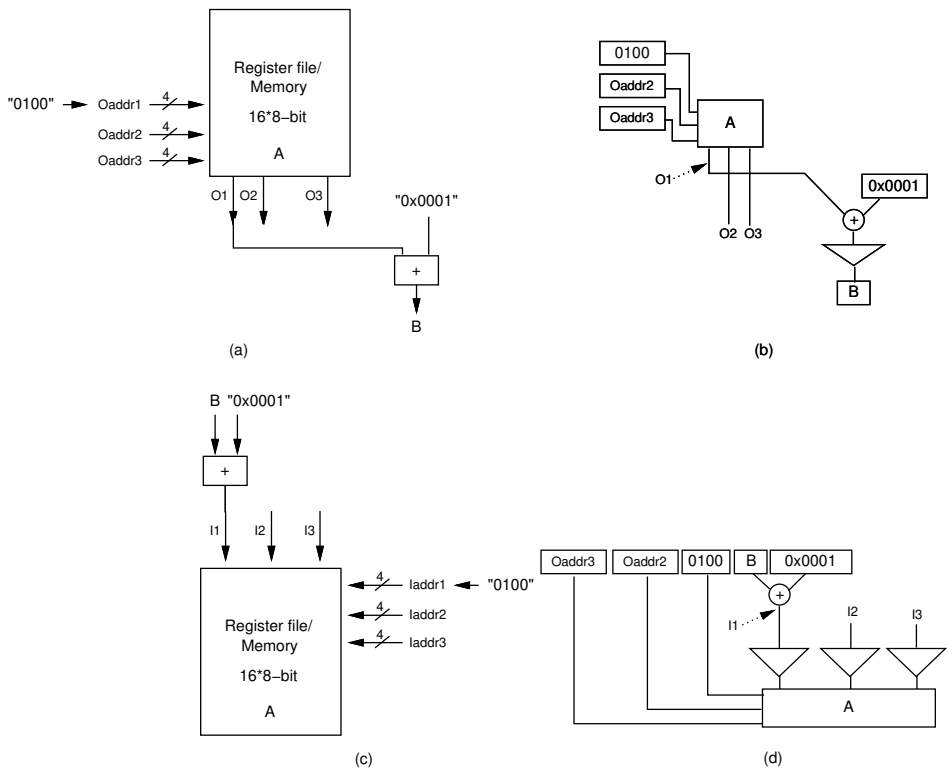


Figure 8: Representation of register file and memory in ADD: (a) reading from a multiple output port register file, (b) representation of a multiple output port register file in the ADD, (c) writing to a multiple input port register file, and (d) representation of a multiple input port register file in the ADD.

Writing to a register file or a memory is represented as a group of assignments to the storage unit. Each assignment represents all write accesses to an input port of the register file or the memory. An index signal is assigned to each input port

to indicate the location where the value is to be assigned. For example, the ADD representation of the register file shown in Figure 8(c) is as follows:

$$A_{(type:storage,dim:2,size(16,8))} = \langle\{B+1\}\rangle[\text{“0010}''] \, , \langle\{I2\}\rangle[Iaddr2] \, \langle\{I3\}\rangle[Iaddr3].$$

The correspondent graphical ADD representation is shown in Figure 8(d).

3.2.3 Interconnect Unit

Interconnect unit can be classified into two classes: namely, signal and bus/multiplexer.

- Signal is the simpliest kind of interconnect unit that allows only one physical driver. A signal has one dimension but can have arbitrary sizes. The size and dimension of a signal is represented as attributes in the ADD. For example, a 16-bit signal A is represented as

  $$A_{(type:signal,dim:1,size(16))} =$$

  A signal is represented in the ADD graph as an edge in the graph.

- Bus/multiplexer is an interconnect type that supports more than one physical drivers. Though the bus/multiplexer can carry values from multiple sources, only one source will be activated at any given instant of time. Thus, a bus or a multiplexer requires control signals to select the active driver.

  Bus and multiplexer share the same representation in the ADD since both of them have the same functionalities. Each bus/multiplexer is represented as an ADN. An attribute is assigned to each ADN to indicate its implementation as a bus or a multiplexer. The ADD representation of a bus or a multipleser is discussed as follows:

  - When representing a *bus*, each assignment condition of the ADN depicts a control to the tri-state driver of the assignment value (e.g. Figure 9(b)). In the case where the assignment value is an ADN (e.g. Figure 9(d)) the assignment condition represent conditions to the switch between the two bus segments (e.g. Figure 9(e)).
  - On the other hand, when representing a *multiplexer*, each assignment condition of the ADN depicts a control signal to the multiplexer (e.g. Figure 9(c)). If the assignment value is an ADN then the corresponding ADD represent a multiple level multiplexers implementation (e.g. Figure 9(f)).

3.2.4 Control Unit

The Control unit is a collection of combinational logic that are responsible for managing the sequence of data computation (next-state logic) and controlling proper execution of

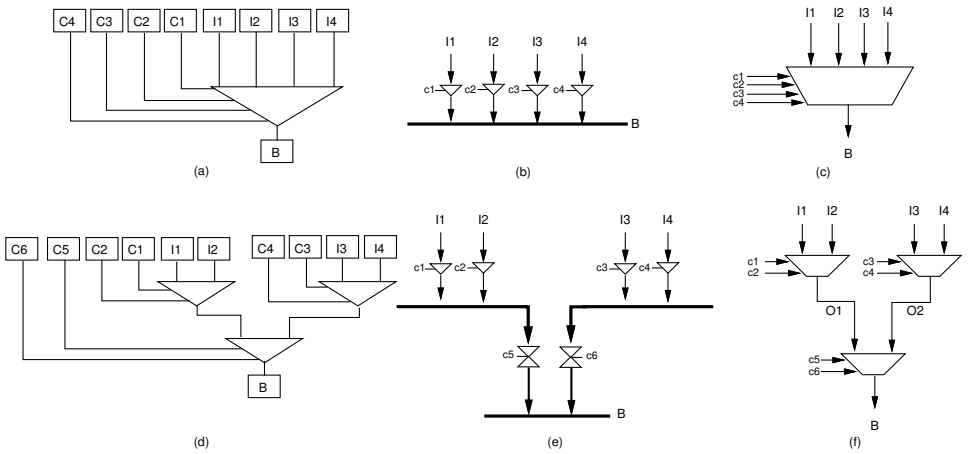


Figure 9: Examples of the ADD representation for different implementation style of a bus and a multiplexer: (a) representation of an one level bus or multiplexer in ADD, (b) single level bus, (c) single level multiplexer, (d) representation of a segmented bus or a multiple level multiplexer in ADD, (e) segmented bus, (f) multiple level multiplexer.

data computation (control logic). Both types of logic are represented in the ADD as the computation in the assignment conditions.

The next-state logic is represented as assignment conditions for the special assignment target called $State_register$. For example, the next-state logic for the state transition shown in Figure 10 is represented in the ADD as

$$State_register = (\mathbf{st_0} \wedge \mathbf{cond})\{st_1\} \ \oplus \ (\mathbf{st_0} \wedge \overline{\mathbf{cond}})\{st_2\} \ \oplus \ \mathbf{st_1}\{st_3\} \ \oplus \ \mathbf{st_2}\{st_3\}.$$

On the other hand, **the control logic** is represented in the ADD as assignment conditions for assignment targets that are not the $State_register$. For example, the control logic for the sequential system shown in Figure 10 is represented in the ADD as

$$\begin{aligned} n1 &= (\mathbf{c1} \wedge \mathbf{st_0})\{A\} \ \oplus \ (\mathbf{c1} \wedge \mathbf{st_2})\{B\} \ \oplus \\ &\quad (\mathbf{c3} \wedge \mathbf{st_2})\{E\}. \\ n2 &= (\mathbf{c4} \wedge \mathbf{st_0})\{\text{“}0x0000\text{”}\} \ \oplus \\ &\quad (\mathbf{c5} \wedge \mathbf{st_1})\{C\} \ \oplus \ (\mathbf{c6} \wedge \mathbf{st_2})\{E\}. \\ n3 &= (\mathbf{c7} \wedge \mathbf{st_0})\{\text{“}10\text{”}\} \ \oplus \ (\mathbf{c5} \wedge \mathbf{st_0})\{\text{“}00\text{”}\}. \end{aligned}$$

Present state	Condition *cond*	Next state	Condition	Assignment
St0	1	St1	c1	n1 = A
			c4	n2 = "0000"
	0	St2	c7	n3 = "10"
			c5	n3 = "00"
St1	–	St3	c5	n2 = C
St2	–	ST3	c2	n1 = B
			c3	n1 = E
			c6	n2 = E
St3				

Figure 10: An example of a state table.

4 High-level synthesis using the ADD

The Assignment Decision Diagram provides high-level synthesis with two major capabilities that are not offered by traditional representations, which are, the minimization of syntactic variances and the models for estimating layout quality metrics during synthesis. In addition, the proposed diagram also simplifies many synthesis tasks such as allocation and scheduling. Overview of advantages using the ADD is given in this section. Detailed discussion is provided in subsequent sections.

- **Minimize syntactic variances**

  Due to the "partially" uniqueness property of the ADD, the ADD can be used to minimize syntactic variances in the input description. To be specific, the ADD can be used to minimize syntactic variances that are caused by ordering and grouping or conditional and assignment statements. A synthesis system that uses the ADD can produce consistent results for descriptions that are differed in such ordering or grouping of statements but are functionally equivalent. Discussion of minimizing syntactic variance scheme is provided in section 5 and [5].

- **Allocation**

  Since control dependencies are represented as data dependencies, the notion of "basic block" is absent in the ADD. This means determining allocation for the whole design would require only a simple data-flow based allocation algorithm.

- **Scheduling**

  Similar to the simplification of the allocation task, a simple data flow scheduling can be used to schedule any design that is represented in the ADD. Operations can be scheduled with global dependencies and criticality consideration. In addition,

since the initial ADD which is obtained from the compilation is in its most parallel form (see section 5), results of the scheduling is free from implicit state boundaries that might have been introduced in the description. Furthermore, each operator in the ADD is represented with a correspondent condition in which the result of operation is to be used. Thus, during scheduling, operators that are mutually exclusive due to their usage conditions can be easily identified. Detailed discussion of scheduling using the ADD is provided in section 6.

- **Iterative, interleave and interactive scheduling and binding tasks**

  Results from both scheduling and binding can be represented in the ADD. In other words, scheduling and binding can be conceived as transformations on the ADD representation. After a scheduling or a binding iteration a new ADD that represents the scheduled or bounded result is created. This new ADD can then be used for further scheduling and binding. Thus, the scheduling and the binding can be applied iteratively and in interleaving manner. This is a very crucial requirement for a representation to be used in the interactive synthesis paradigm.

- **Improve cost function**

  In conjunction with the ADD, we have developed fast estimation techniques that can access layout area and timing information with high fidelity [21, 6, 14]. Thus, synthesis tasks that use the ADD can perform realistic design tradeoff with these layout quality metrics.

- **Binding**

  Although the ADD does not increase the efficiency of the binding task, the ADD neither decrease the efficiency. This is because majority of binding algorithms operate on graphs that show usage exclusivity of operators, storage units, and interconnect (e.g., compatibility graph), instead of the internal representation. These usage-exclusivity graphs are, generally, constructed from results of the scheduling. The same information can be extracted from the ADD with the same complexity as traditional representations.

5 Minimizing syntactic variance using the ADD

The first task in high-level synthesis is to compile the input description into an internal representation that is usually in a form of a topological graph. The compilation is usually accomplished by a one-to-one mapping of the input description into the internal representation. In other words, each language construct in the description is realized with a particular topology of nodes in the representation.

Due to the one-to-one correspondence that exist between the constructs of the input descriptions and the schema for the internal representation, these compilers produce

different representations for different descriptions. The internal representations of two given descriptions could be far different even if the descriptions are semantically equivalent.

Graphs obtained from the compiler are used by high-level synthesis tasks. Hence, majority of synthesis algorithms are topological-graph based. These algorithms produce results that are generally depended on topology of the graph. Meaning, the algorithms would produce different results for graphs with different topology, even though those graphs have the same semantics. And since the compiler produces different graph topologies for different descriptions, as the result, synthesis tasks would produce different hardware for each of the topology, as illustrated in Figure 11.

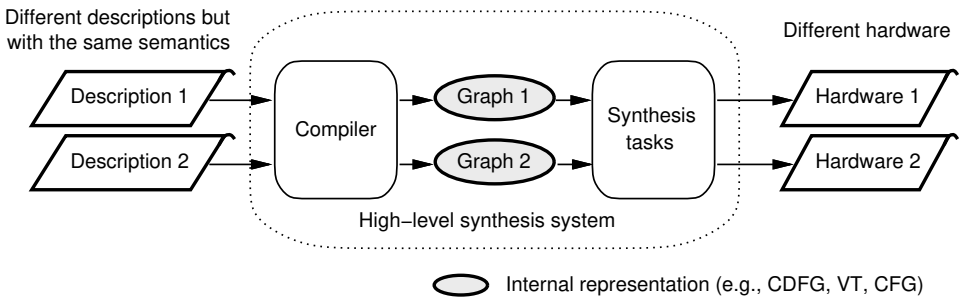


Figure 11: A general high-level synthesis approach.

A simple solution to avoid the inefficiences caused by syntactic differences is to force the designer to write descriptions that fit the algorithm used inside the synthesis system. This solution is impractical because the designers would need to acquire detailed knowledge of the synthesis algorithms that are used.

We believe that the effect of syntactic variances can be minimized without unnecessarily increasing the complexity of synthesis tasks by (1) improving the internal representation and (2) modifying the compilation scheme, as discussed as follows:

1. We need a representation that is capable of representing different descriptions that have the same semantics in one "unique" topology. There are many ways of representing a given description starting from the most sequential, which is inherent from the description, to the most parallel representation. We choose the most parallel representation to be the "unique" representation because it does not contain implicit sequentiality that are found in the description. Thus, the representation we are seeking should be able to depict the most parallel representation of any given descriptions. Traditional representations can not provide such capability because their constructs inherit the sequentiality from the description. For example,

the control dependency in CFG implies the sequentiality in the execution of each computation as specified in the description.

In addition, the representation, which we are seeking, should consist of parts that reflect semantics of the description and not syntactic constructs. This rules out traditional representations because their constructs are closely related to language constructs. For example, IF-branch and IF-join nodes in CDFG are used to represent the begining and the ending of an if branch, respectively.

Hence, to resolve this issue, we developed the ADD that is capable of representing the description in its the most parallel form using parts that have no direct relationship with language constructs.

2. Having defined the ADD, we need to develop a compilation scheme from the input description into the new representation. The compilation transforms/converts a given description into its most parallel representation and, at the same time, resolve the discrepancies that are caused by the ordering and grouping of conditional branches and/or computations. As a result, different descriptions that contain such discrepancies can be transformed into a "unique" graph so that result obtained from synthesis tasks is consistent. The proposed approach is illustrated in Figure 12.

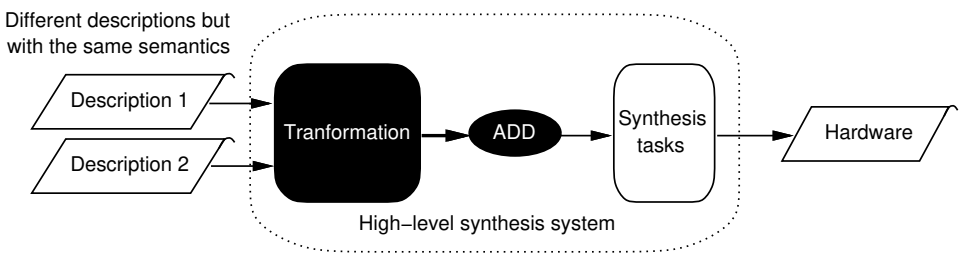


Figure 12: Overview of the proposed approach.

The transformation process converts a given input description into its most parallel representation in ADD. The process assumes that a state boundary will be introduced for every loop, with variable bound, and synchronization (e.g., WAIT statement of VHDL) constructs. On the other hand, each statement block of assignment statements and conditional branches that are defined before, within, and after the assumed state boundaries are transformed into an ADD that shows the execution of the statement block in one state (i.e., the most parallel representation). Figure 13(a) shows the assumed state boundaries for a VHDL example that contains a WHILE loop.

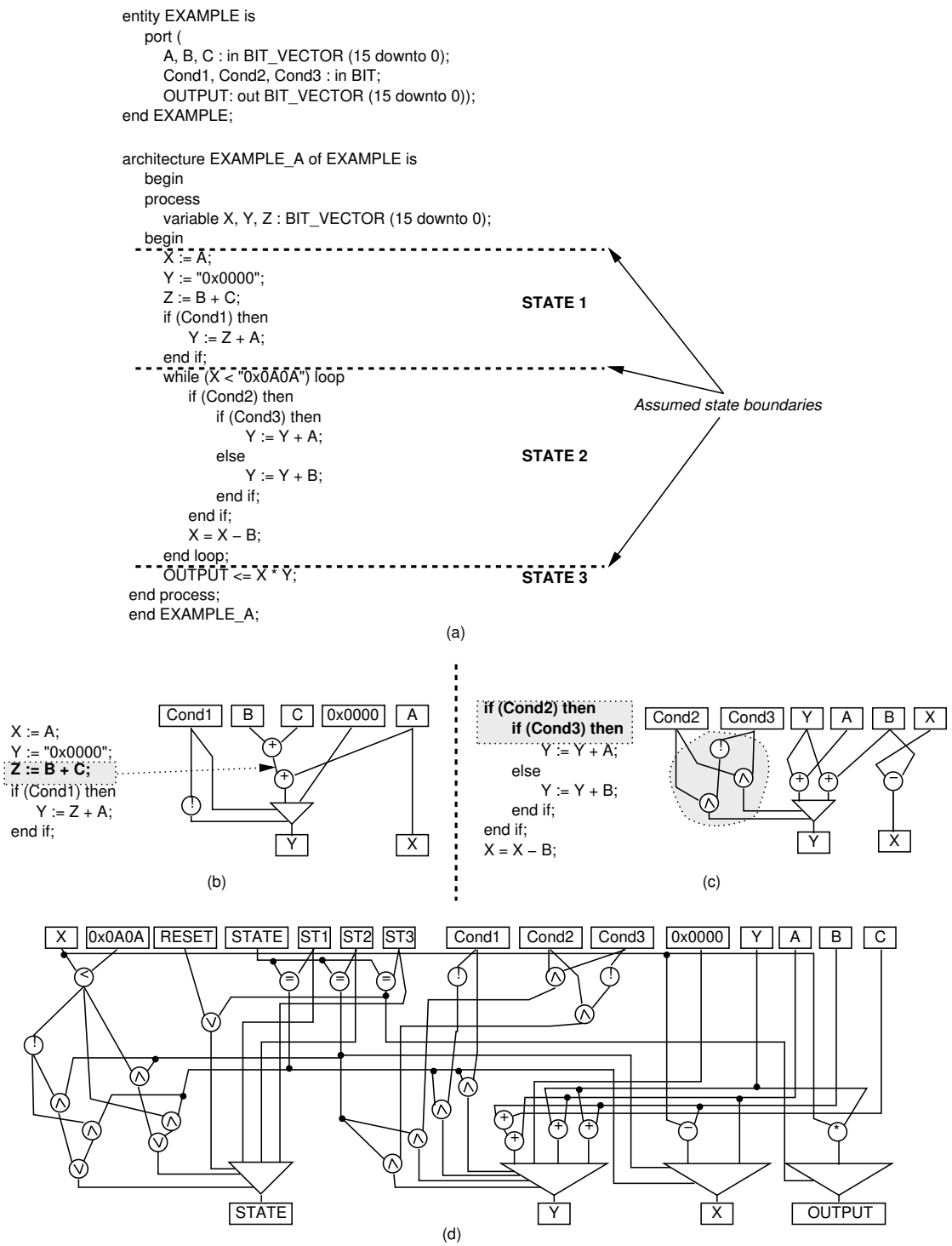


Figure 13: An example of transforming a VHDL input description into ADD.

For each variable in a statement block, the transformation process determines all possible assignment values and their correspondent conditions for that variable. Both assignment values and assignment conditions are expressed in their most flattened form. In other words, every use of temporary variables are replaced with their actual values. As the result, computations within the assumed state boundaries are represented as a dataflow graph that required inputs only from two sources; namely, input ports and content of variables from previous state. By flattening the computation, the resultant representation is free of implicit state boundaries that might be introduced due to writing and reading of temporary variables. Figure 13(b) shows an example of the computation flattening process.

In addition, to flattening of computation, the transformation also flattens nested conditional branches. Basically, assignment conditions for each computation in a nested conditional branch is transformed into boolean functions of conditions that determine the path to that computation. By doing so, the resultant representation is free of any grouping and ordering of conditional branches that are specified in the description. Figure 13(c) shows an example of the flattening of a two level nesting conditional branches.

Detailed transformation algorithms for converting input description into ADD can be found in [5]. The conversion process includes algorithms for resolving data dependencies (i.e., read after read, write after write, read after write, and write after read), flattening of computation and flattening of conditional branches. Figure 13(d) shows an example of ADD obtained from applying transformation algorithms on the VHDL description shown in Figure 13(a).

6 Scheduling of the ADD

With the ADD representation, any description that contains straight-line code, conditional branches, loops, or synchronization constructs (WAIT statement) can be scheduled using a simple dataflow scheduling technique (e.g., the FDS [13]). In addition to simplifying scheduling algorithm, ADD provides crucial information for scheduling that could be complicated to obtain from traditional representations, such as CDFG. Scheduling features that can be obtained from scheduling of the ADD are discussed as follows:

- **Scheduling of conditions of the branches and operations of the branches in the same control step**

  With the ADD, operations in conditional branches can be scheduled before, after, or in the same control step as the condition of the branch. Scheduling of operations in conditional branches to available operators, even before the branch is decided, can increase the component utilization factor and reduce total number of control

steps in the design [17]. In addition, number of control steps on the critical path can be reduced if the scheduled operations are on the critical path [3, 8].

For example, Figure 14(a) shows the scheduling of the branch condition $(A + B)$, which is to be used in $((A + B) < C)$, and a computation in a branch $(C * J)$ in the same control step. $(C * J)$ is selected because it is on the critical path. Results of $(C * J)$ are stored in a temporary variable that will be used in later control steps. The effect of such scheduling can be illustrated with the CDFG shown in Figure 14(b).

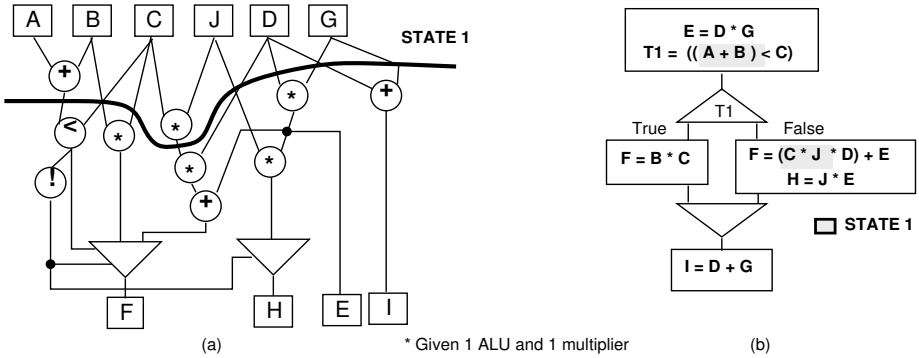


Figure 14: An example of scheduling the branch conditions and operations in the branch in the same control step: (a) Using the ADD, (b) the scheduling effect.

- **Scheduling across basic blocks**

  In the ADD, operations that are described in different basic blocks can be scheduled in the same control step. This is because the ADD does not contain the notion of a basic block. Selection of operations to be scheduled can be based on the data dependencies, and resource constraints. Thus, allowing the scheduler to consider the criticality of all operations globally before making any decision.

  For example, Figure 15(a) shows the scheduling of operations $(A + B)$ and $(D * G)$ in the same control step. The effect of such scheduling can be illustrated with the CDFG shown in Figure 15(b).

- **Merging of mutually exclusive operation before scheduling**

  Each operator in the ADD is accompanied with a condition under which the result of the operation is to be used. The mutual exclusivity of operators can be determined by deciding the mutual exclusiveness of these conditions. Since the result of merging can be represented in the ADD, exclusive operators can be merged before applying the scheduling algorithm.

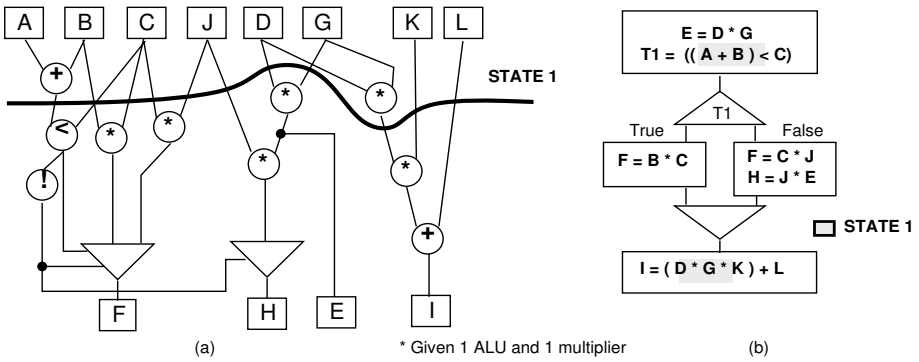


Figure 15: An example of scheduling operations across basic blocks: (a) Using the ADD, (b) the scheduling effect.

For example, Figure 16(a) shows two multiplication that are used in mutually exclusive condition, namely, when $(A + B) < C)$ and when $(NOT((A + B) < C))$. The resultant ADD after merging of these two multiplications is shown in Figure 16(b). Further scheduling can then be applied to resultant of the merge.

- **Merging of mutually exclusive operations during scheduling**

  In the ADD, operators that are mutually exclusive can be merged during scheduling. The merging is possible provided that conditions for the merged operators are scheduled in a control step prior to the operator or in the same control step as the merged operators. This form of merging can be achieved in the ADD using a simple dataflow scheduling technique as illustrated in an example shown in Figure 17. The example shows assignment of two multiplication operators to the same multiplier in the same control step $STATE3$. This is possible because the two multiplications are used in mutually exclusive conditions, namely when $((A + B) < C)$ and when $(NOT((A + B) < C))$.

7 Iterative, interleave and interactive scheduling and binding using the ADD

Scheduling and binding can be conceived as tasks that "massage" the ADD. In other words, during each step of scheduling or binding, the ADD representation is transformed or annotated until the final design is obtained. The process can be illustrated by the Figure 18

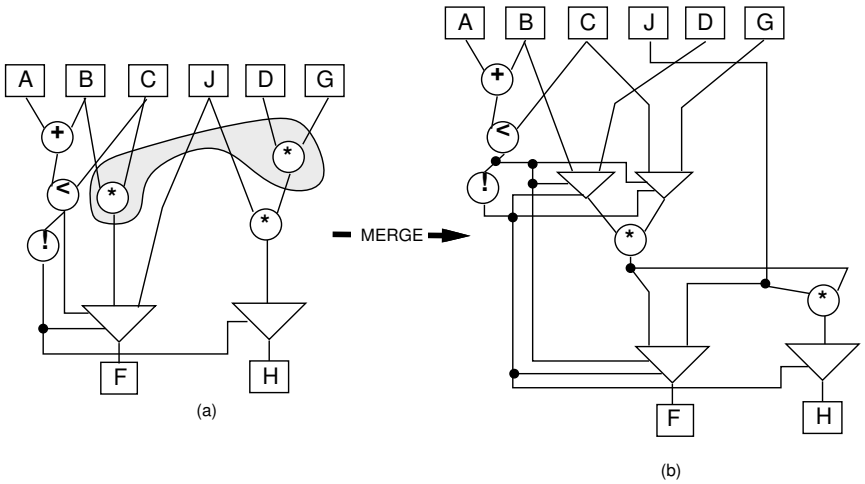


Figure 16: An example of merging mutually exclusive operations before scheduling: (a) before merging, (b) after merging.

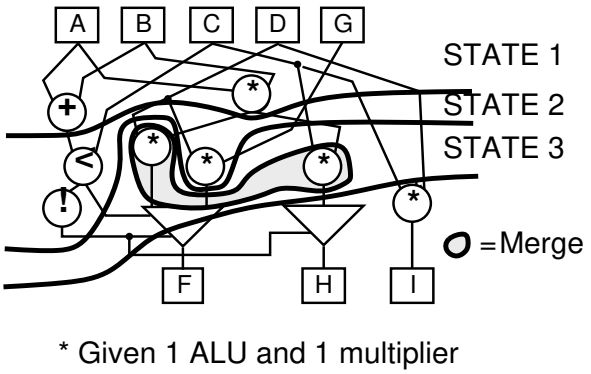


Figure 17: An example of merging mutually exclusive operations during scheduling.

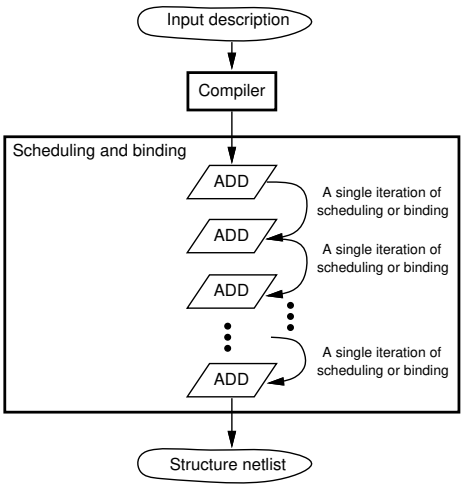


Figure 18: Interactive scheduling and binding using ADD

The transformation process consists of rules that, when invoked, alters the topology of the ADD to reflects the intended scheduling or binding decisions. These rules are described as follows:

7.1 Transformations of the ADD for scheduling

Transformations that are used in scheduling can be categorized into two major classes: namely, linear state insertion and state branches insertion. Generally, the linear state insertion is used for partitioning of a state to reduce the clock period or to reduce the required number of resources. On the other hand, the state branches insertion is used for partitioning actions in a state into substates based on the conditions under which the actions are performed. The state branches insertion can lead to minimization of number of states on the execution path.

The main difference between these two transformation rules is the construction of the next-state logic. In linear state insertion, a state is partitioned (scheduled) in such a way that transition into the new state (i.e., the state formed by partitioning an existing state) is based on the same condition as the partitioned state. The effect of the linear state insertion can be thought as the insertion of a new state right before the partitioned state. For example, Figure 19(a) shows a state transition diagram before and after the linear state insertion. In this example $ST1$ and $ST2$ are being partitioned and the states $STnew_1$ and $STnew_2$ are linearly inserted, respectively. On the other hand, state branches insertion creates new conditions for the transition to the inserted states

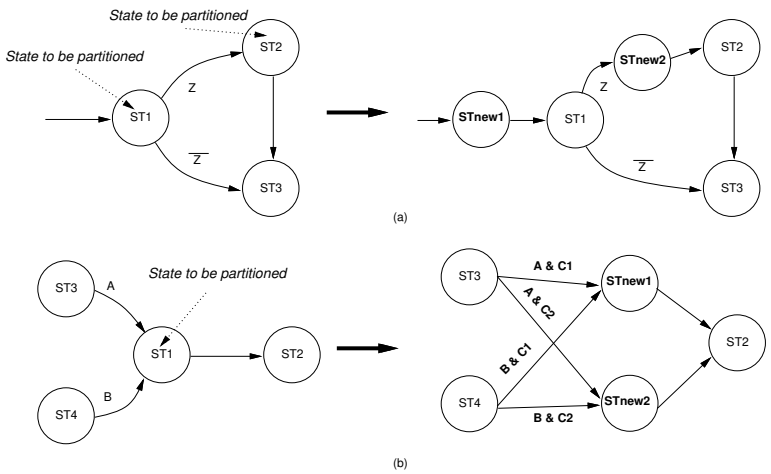


Figure 19: Two types of transformations for scheduling in Assignment Decision Diagram: (a) linear state insertion, and (b) state branches insertion.

based on the branching conditions. For example, Figure 19(b) shows an example of a state branches insertion. In this example, $ST1$ is partitioned into two states, $STnew_1$ and $STnew_2$ based on conditions $C1$ and $C2$ respectively. Both transformations required update of assignments to the $State_register$ such that the state transition is preserved.

Transformation procedures for linear state insertion and state branches insertion are discused as follows:

- **Linear state insertion**

  Given an ADD,

$$\begin{aligned} X &= \ldots \oplus AC\{\gamma_1 OP \gamma_2\} \oplus \ldots . \\ &\vdots \\ State_register &= \ldots . \end{aligned}$$

  where AC is an assignment condition of the form

$$AC = \bigvee (ST_i \wedge cond_i),$$

  ST_i is a state i, $cond_i$ is a boolean expression, and γ_1 and γ_2 are arithmetic expressions. Let Φ be a set of states that are used in the assignment condition AC. That is

$$\Phi = \{\phi_1, \ldots, \phi_n\} = \{ST_i, \ldots, ST_j\}.$$

where n is the total number of states which are used in AC, and $ST_i \ldots ST_j$ are the actual states that are used in AC.

We want to insert a state boundary right after the computation of γ_1 such that the result of γ_1 will be stored in a register, $temp$, and the result of $temp$ will be used by OP in the successive clock cycle, as illustrated in Figure 20. A new ADD that reflects the insertion of this state boundary can be created using the following algorithm.

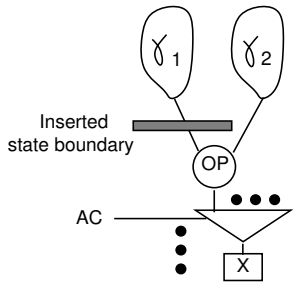


Figure 20: Linear state insertion

Algorithm: Linear state insertion

1. Create a set of new states, i.e., $\{STnew_1, STnew_2, \ldots, STnew_n\}$
2. Create a temporary register to hold the result of γ_1. Let the name of this new register be $temp$. Assign γ_1 to $temp$ with assignment condition $ACnew$, where
$$ACnew = \bigvee_{i=1}^{n} (STnew_i).$$
That is
$$temp = ACnew\{\gamma_1\}.$$
3. Replace γ_1 with the register $temp$, that is
$$X = \ldots \oplus AC\{temp\ OP\ \gamma_2\} \oplus \ldots.$$
4. Update the state transition (i.e., update assignment to the $State_register$ assignment target) as follows:
   (a) For all $i = 1$ to n
      i. Replace the assignment condition, β_i, for ϕ_i (i.e.,
$$state_register = \ldots \oplus \beta_i\{\phi_i\} \oplus \ldots.$$

with $STnew_i$. That is

$$State_register = \ldots \oplus STnew_i\{\phi_i\} \oplus \ldots .$$

ii. Add the assignment value $STnew_i$ to $state_register$ and guard the assignment with the condition, β_i, that is

$$state_register = \ldots \oplus \beta_i\{STnew_i\} \oplus \ldots .$$

To illustrate the Linear state insertion, let us consider the following ADD example,

$$\begin{aligned} A &= ST_0\{B + C\} \\ F &= ((ST_1 \wedge X) \vee (ST_2 \wedge Y))\{(A + D) + E\} \\ State_register &= (init)\{ST_0\} \oplus ST_0\{ST_1\} \oplus (ST_1 \wedge Z)\{ST_2\} \oplus \\ &\quad (ST_1 \wedge \overline{Z})\{ST_2\} \oplus ST_2\{ST_3\}. \end{aligned}$$

whose graphical representation is shown in Figure 21(a). If there is a resource constraint of one adder then the chaining of additions for F, $(A + D) + E$, has to be partitioned. And let's assume that the computation is partitioned such that the value $(A + D)$ is stored in a temporary register $(TEMP)$ and to be used in the following clock cycle as $F = TEMP + E$, which is shown in Figure 21(a). In addition, the new state is assumed to be inserted linearly. Following the above transformation algorithm we would obtain the following results:

- Step 1
  new states $= \{STnew_1, STnew_2\}$
- Step 2

$$TEMP = (STnew_1 \vee STnew_2)\{A + D\}$$

- Step 3

$$F = ((ST_1 \wedge X) \vee (ST_2 \wedge Y))\{TEMP + E\}$$

- Step 4

$$\begin{aligned} State_register &= (init)\{ST_0\} \oplus ST_0\{STnew_1\} \oplus STnew_1\{ST_1\} \oplus \\ &\quad (ST_1 \wedge Z)\{STnew_2\} \oplus (ST_1 \wedge \overline{Z})\{STnew_2\} \oplus \\ &\quad STnew_2\{ST_2\} \oplus ST_2\{ST_3\}. \end{aligned}$$

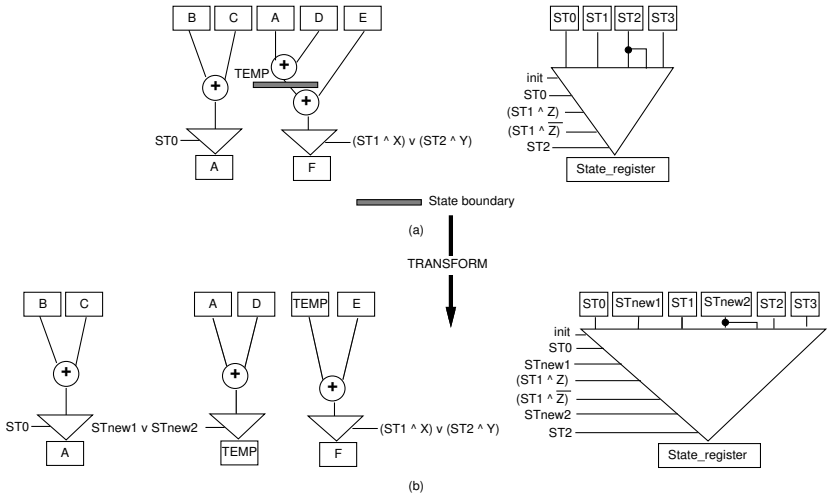


Figure 21: An example of transformation for linear state insertion: (a) ADD before the transformation and (b) ADD after the transformation.

The final resultant ADD after the transformation is:

$$\begin{aligned}
A &= ST_0\{B+C\} \\
TEMP &= (STnew_1 \vee STnew_2)\{A+D\} \\
F &= ((ST_1 \wedge X) \vee (ST_2 \wedge Y))\{TEMP+E\} \\
State_register &= (init)\{ST_0\} \oplus ST_0\{STnew_1\} \oplus STnew_1\{ST_1\} \oplus \\
&\quad (ST_1 \wedge Z)\{STnew_2\} \oplus (ST_1 \wedge \overline{Z})\{STnew_2\} \oplus \\
&\quad STnew_2\{ST_2\} \oplus \oplus ST_2\{ST_3\}.
\end{aligned}$$

The corresponding graphical representation is shown in Figure 21(b).

- **State branches insertion**

  Given an ADD,

$$\begin{aligned}
X &= \ldots \oplus ((ST_i \wedge cond_1 \wedge C_1) \vee D_1)\{\gamma_1\} \\
&\quad \oplus((ST_i \wedge cond_2 \wedge C_2) \vee D_2)\{\gamma_2\} \oplus \ldots. \\
&\vdots \\
Y &= \ldots \oplus ((ST_i \wedge cond_j \wedge C_j) \vee D_j)\{\gamma_3\} \oplus \ldots. \\
&\vdots \\
State_register &= \ldots.
\end{aligned}$$

where ST_i is the state where branches are to be inserted, $cond_j$ is a branching condition, and C_j and D_i are boolean equations. Let Φ be the set of all assignment conditions in the given ADD that contain the state ST_i. That is,

$$\Phi = \{\forall_{j=1}^{n}((ST_i \wedge cond_i \wedge C_i) \vee D_i)\}$$

Let γ_k be the assignment value for the kth assignment condition in Φ. And let ST_f be the next state which transits from ST_i (i.e., ST_f is the assignment value to the $State_register$ that has the assignment condition ST_i). The insertion of state branches can be achieved as follows:

Algorithm: State branches insertion

1. Create a set of new state, i.e., $\{STnew_1, \ldots, STnew_n\}$
2. For $j = 1$ to n
   (a) Replace ϕ_j, $((ST_i \wedge cond_j \wedge C_j) \vee D_j)$, with $((STnew_i \wedge C_j) \vee D_j)$.
3. For $j = 1$ to n
   (a) insert $STnew_j$ as assignment value to the $State_register$ and guard the assignment with condition $AC \wedge cond_i$, where AC is, originally, the assignment condition that guards the assignment of ST_i to the $State_register$. That is,

$$state_register = \ldots \oplus (AC \wedge cond_j)\{STnew_j\} \oplus \ldots.$$

4. Create a new assignment condition, AC_{new}, which will be used in the next step, $AC_{new} = (AC \bigvee_{j=1}^{n}(\overline{cond_j}))$.
5. Replace the assignment condition that is guarding ST_f in the $state_register$ with $(\bigvee_{j=1}^{n}) \vee AC_{new}$. That is,

$$state_register = \ldots \oplus (STnew_1 \vee STnew_2 \vee \ldots \vee AC_{new})\{ST_f\} \oplus \ldots.$$

6. Remove ST_i and any logic that uses ST_i.

To illustrate the state branches transformation algorithm, let us consider the following ADD example,

$$\begin{aligned} A &= ST_0\{B + C\} \\ F &= (ST_1 \wedge X)\{A + D\} \oplus ST_2\{F + G\} \\ State_register &= (init)\{ST_0\} \oplus ST_0\{ST_1\} \oplus ST_1\{ST_2\}. \end{aligned}$$

whose graphical representation is shown in Figure 22(a). The given ADD requires 3 control steps to complete the computation. However, it is possible to reduce one control step if X is false. This is because the computation for F in state ST_1 is required if, and only if the value of X is true. Thus, we can reduce one control step by branching from state ST_0 to ST_2 if X is false. Applying the state branches insertion procedure we can obtain the following:

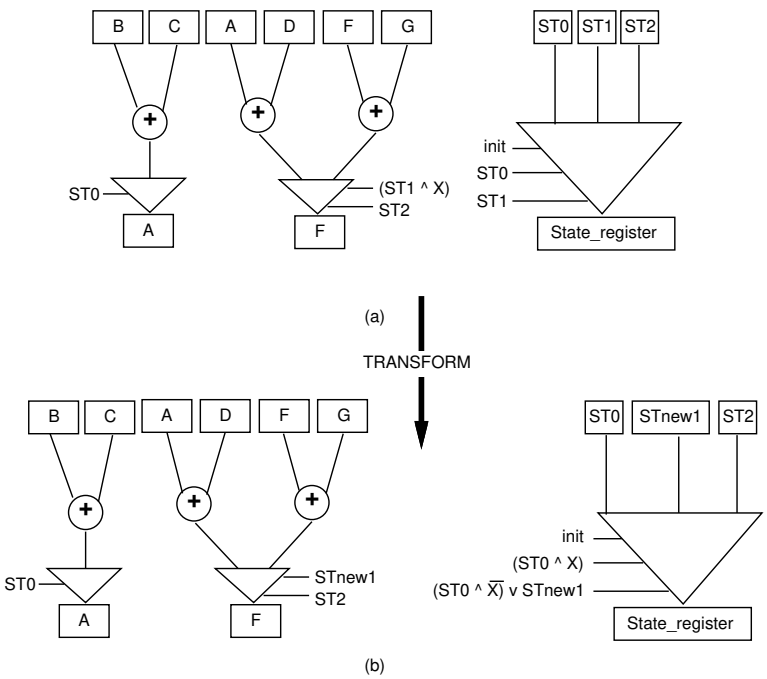


Figure 22: An example of transformation for state branches insertion: (a) ADD before the transformation and (b) ADD after the transformation.

- Step 1
  new states = $\{STnew_1\}$
- Step 2

$$F = STnew_1\{A + D\} \oplus ST_2\{F + G\}$$

- Step 3

$$\begin{aligned} State_register &= (init)\{ST_0\} \oplus (ST_0 \wedge X)\{STnew_1\} \oplus \\ &\quad ST_0\{ST_1\} \oplus ST_1\{ST_2\}. \end{aligned}$$

- Step 4

$$AC_{new} = (ST_0 \wedge \overline{X})$$

- Step 5

$$\begin{aligned} State_register &= (init)\{ST_0\} \oplus (ST_0 \wedge X)\{STnew_1\} \oplus \\ &\quad ST_0\{ST_1\} \oplus (STnew_1 \vee (ST_0 \wedge \overline{X})\{ST_2\}. \end{aligned}$$

- Step 6
  Remove any logic that uses the value of ST_1.

Thus, the resultant ADD after transformation is as follows:

$$\begin{aligned} A &= ST_0\{B + C\} \\ F &= STnew_1\{A + D\} \oplus ST_2\{F + G\} \\ State_register &= (init)\{ST_0\} \oplus (ST_0 \wedge X)\{STnew_1\} \oplus \\ &\quad (STnew_1 \vee (ST_0 \wedge \overline{X}))\{ST_2\}. \end{aligned}$$

The corresponding graphical representation of this ADD is shown in Figure 22(b).

7.2 Transformation of the ADD for operator binding

Operators that are exclusively used in different conditions or states can be bounded to the same component. In ADD, the mutual exclusiveness of operator usage can be identified by testing the exclusivity of their corresponding assignment conditions.

Given a set of operators, $\{OP_1, \ldots, OP_n\}$, that are to be bounded to the same component and the following ADD

$$\begin{aligned} X &= \ldots \oplus \beta_1\{\gamma_{1,1} OP_1 \gamma_{1,2}\} \oplus \ldots. \\ Y &= \ldots \oplus \beta_2\{\gamma_{2,1} OP_2 \gamma_{2,2}\} \oplus \beta_3\{\gamma_{3,1} OP_3 \gamma_{3,2}\} \oplus \ldots. \\ &\vdots \\ Z &= \ldots \oplus \beta_n\{\gamma_{n,1} OP_n \gamma_{n,2}\} \oplus \ldots. \\ &\vdots \\ State_register &= \ldots. \end{aligned}$$

where β_i is the assignment condition for OP_i, and $\gamma_{i,1}$ and $\gamma_{i,2}$ are the first and second operands for the operator OP_i, respectively. Let Φ be the set of assignment conditions for the operators to be bounded, (i.e., $\Phi = \{\beta_1, \ldots, \beta_n\}$). Merging of operators $\{OP_1, \ldots, OP_n\}$ to an OP_{merged} can be achieved as follows:

Algorithm: Operator merging

1. Create two temporary signals, $temp_1$ and $temp_2$, that will be used to hold the operands for OP_{merged}.
2. For $i = 1$ to n
   (a) Assign value $\gamma_{i,1}$ to $temp_1$ using β_i as the guarding assignment condition. That is,
   $$temp_1 = \ldots \oplus \beta_i\{\gamma_{i,1}\} \oplus \ldots.$$
   (b) Assign value $\gamma_{i,2}$ to $temp_2$ using β_i as the guarding assignment condition. That is,
   $$temp_2 = \ldots \oplus \beta_i\{\gamma_{i,2}\} \oplus \ldots.$$
3. Create a temporary signal $temp_{merged}$ that will be used to represent output signal of the OP_{merged}. Then, assign value ($temp_1$ OP_{merged} $temp_2$) to $temp_{merged}$. That is,
   $$temp_{merged} = \{temp_1\ OP_{merged}\ temp_2\}.$$
4. For $i = 1$ to n
   (a) Replace all the computations that use operator OP_i, which is merged to OP_{merged}, with the signal $temp_{merged}$. That is
   $$\begin{aligned} X &= \ldots \oplus \beta_1\{temp_{merged}\} \oplus \ldots. \\ Y &= \ldots \oplus \beta_2\{temp_{merged}\} \oplus \beta_3\{temp_{merged}\} \oplus \ldots. \\ &\vdots \\ Z &= \ldots \oplus \beta_n\{temp_{merged}\} \oplus \ldots. \\ &\vdots \\ State_register &= \ldots. \end{aligned}$$

To illustrate the operator merging algorithm, let us consider the following ADD example

$$\begin{aligned} t1 &= X\{B\} \oplus \overline{X}\{C - D\} \\ E &= Y\{A + t1\} \\ I &= \overline{Y}\{(F * G) + H\} \end{aligned}$$

whose graphical representation is shown in Figure 23(a). Two addition operations are required for the computation in the example ADD; namely $(A+t1)$ and $((\ldots)+H)$. However, these two additions are mutually exclusive since the value of $(A+t1)$ is used when Y is true, while $((\ldots)+H)$ is used when Y is false. Thus, these two addition operations can be merged using the above transformation algorithm, as follows:

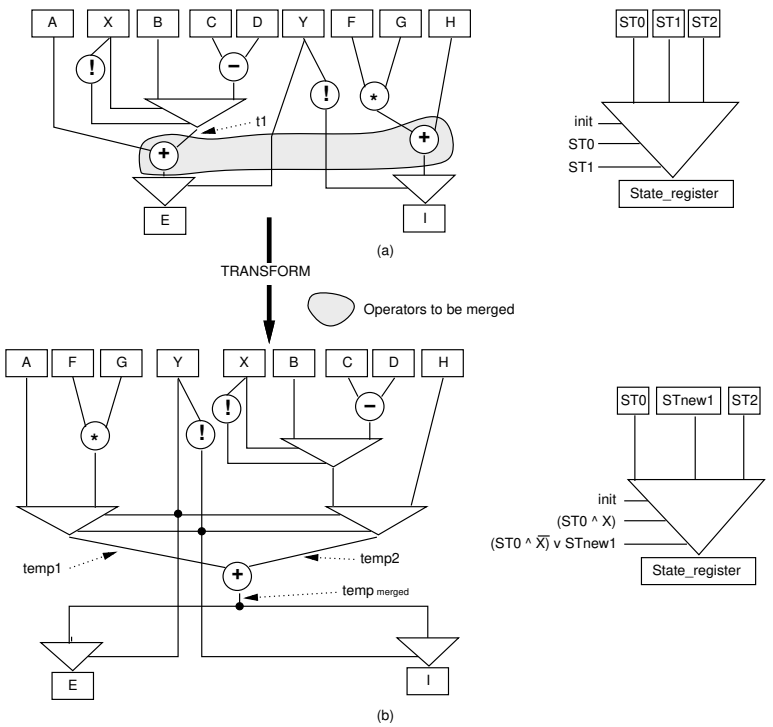


Figure 23: An example of transformation for operator merging: (a) ADD before the transformation and (b) ADD after the transformation.

- Step 1
  Create $temp_1$ and $temp_2$.

– Step 2

$$\begin{aligned} temp_1 &= Y\{A\} \oplus \overline{Y}\{F * G\} \\ temp_2 &= Y\{t1\} \oplus \overline{Y}\{H\} \end{aligned}$$

– Step 3

$$temp_{merged} = \{temp_1 + temp_2\}$$

– Step 4

$$\begin{aligned} E &= Y\{temp_{merged}\} \\ I &= \overline{Y}\{temp_{merged}\} \end{aligned}$$

Thus, the resultant ADD after transformation is as follows:

$$\begin{aligned} t1 &= X\{B\} \oplus \overline{X}\{C - D\} \\ temp_1 &= Y\{A\} \oplus \overline{Y}\{F * G\} \\ temp_2 &= Y\{t1\} \oplus \overline{Y}\{H\} \\ temp_{merged} &= \{temp_1 + temp_2\} \\ E &= Y\{temp_{merged}\} \\ I &= \overline{Y}\{temp_{merged}\} \end{aligned}$$

The corresponding graphical representation of the resultant ADD is shown in Figure 23(b).

7.3 Transformation of the ADD for interconnect binding

Interconnect binding reduces the total number of interconnect units required in the design. The binding can be accomplished by merging interconnect units, which are represented by ADNs, that are exclusively used. The mutual exclusiveness of the usages can be determined from the assignment decision conditions of the ADNs. The interconnect merging algorithm is described below.

Given an ADD,

$$\begin{aligned} X &= \beta_{x1}\{\gamma_{x1}\} \oplus \beta_{x2}\{\gamma_{x2}\} \oplus \ldots \oplus \beta_{xn}\{\gamma_{xn}\} \\ Y &= \beta_{y1}\{\gamma_{y1}\} \oplus \ldots \beta_{yp}\{\gamma_{yp}\} \\ &\vdots \\ State_register &= \ldots. \end{aligned}$$

where X and Y are signals, and $\beta_{x1} \ldots \beta_{yp}$ are assignment conditions that are mutually exclusive. Let Φ be the set of all assignment to X and Y (i.e., $\Phi = \{\beta_{x1}\{\gamma_{x1}\} \oplus \ldots \oplus \beta_{xn}\{\gamma_{xn}\}, \ldots, \beta_{y1}\{\gamma_{y1}\} \oplus \ldots \oplus \beta yp\{\gamma_{yp}\}\}$ Merging of interconnect units X and Y can be accomplished as follows:

1. Create a temporary signal, $temp_{merge}$, that will be used to represent the merged interconnect unit.

2. Assign values in Φ to $temp_{merge}$, that is

$$temp_{merge} = \beta_{x1}\{\gamma_{x1}\} \oplus \ldots \oplus \beta_{xn}\{\gamma_{xn}\} \oplus \ldots \oplus \beta_{y1}\{\gamma_{y1}\} \oplus \ldots \oplus \beta yp\{\gamma_{yp}\}.$$

3. Replace all instances of X to Y with $temp_{merge}$.

Merging of interconnect units can be illustrated by the following example. Consider the following ADD:

$$\begin{aligned} t1 &= X\{A+B\} \\ t2 &= \overline{X}\{D*E\} \\ C &= \{t1\} \\ H &= \{(t2+G)+I\}. \end{aligned}$$

where $t1$ and $t2$ represent signals. The corresponding graphical representation for the example ADD is shown in Figure 24(a). Two interconnect units (i.e., one for $t1$ and the other for $t2$) will be required to implement the given ADD. However, $t1$ and $t2$ are exclusively used; $t1$ is used when X while $t2$ is used when $\overline{X}$, respectively. Thus, $t1$ and $t2$ can be merged using the transformation algorithm described above. The result of merging is as follows:

$$\begin{aligned} temp_{merge} &= X\{A+B\} \oplus \overline{X}\{D*E\} \\ C &= \{temp_{merge}\} \\ H &= \{(temp_{merge}+G)+I\}. \end{aligned}$$

The corresponding graphical representation of the merging result is shown in Figure 24(b).

8 Accessing layout quality measures from the ADD

In conjunction with the development of ADD, we have designed estimation algorithms [21, 6, 14] that can provide estimates of layout area and timing for a given ADD. The estimation algorithm is based on models that take into account layout characteristics, layout design process (i.e., placement, routing etc.), and layout technology (such as 3.0 micron CMOS technology). Preliminary experiments have shown that these models are efficient (i.e., linear complexity), accurate, and have high fidelity [21, 6, 14]. Thus, if synthesis tasks are realized as procedures that "message" the ADD then this estimation algorithm can be used to provide realistic layout quality metrics for design tradeoffs in those synthesis tasks.

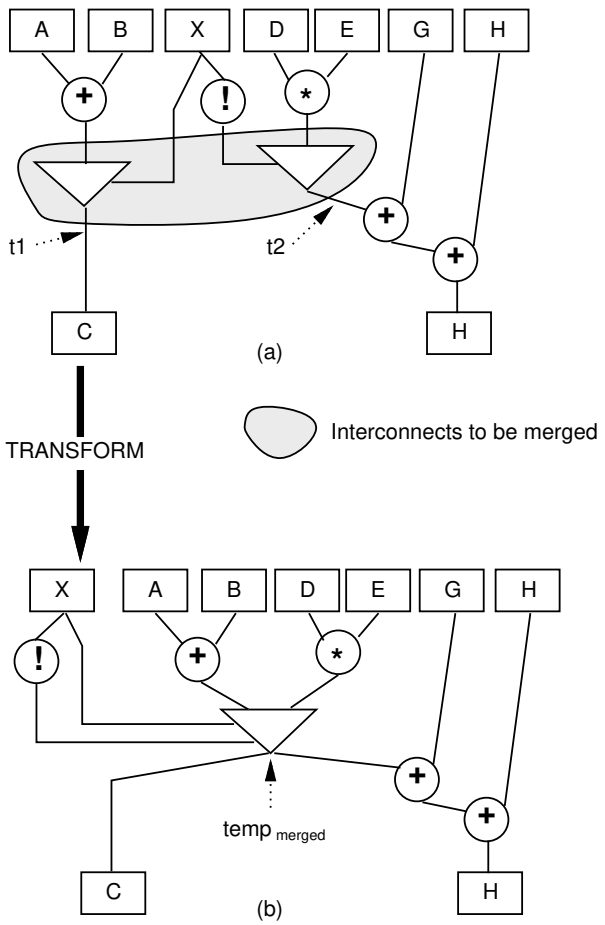


Figure 24: An example of transformation for interconnect merging.

The process of obtaining layout estimates for a given ADD can be divided into 3 major steps: 1) technology mapping, 2) component grouping, and 3) quality estimation, which are illustrated in Figure 25 and described as follows:

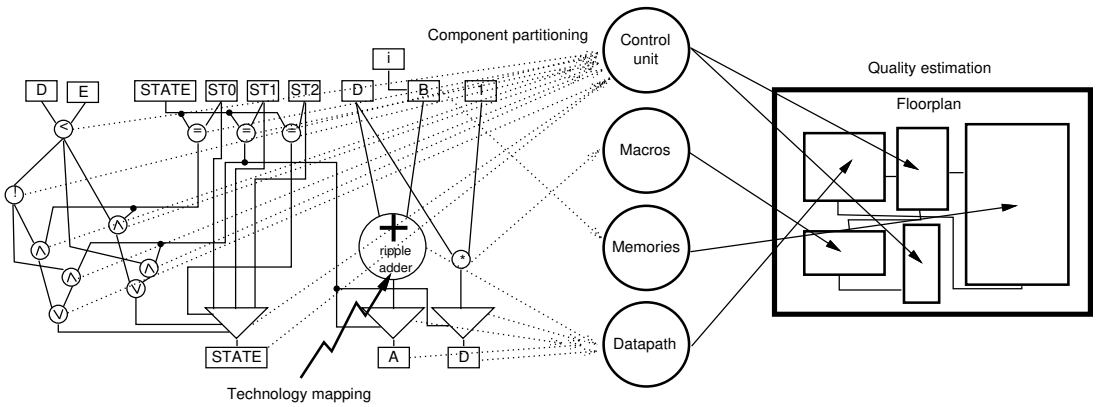


Figure 25: Estimating layout quality from the ADD.

1. Technology mapping

   The goal of the estimation is to provide layout quality measures of an ADD at any instant in the design process. This means, the estimates may have to be derived from an ADD that contains partial or no binding information. To achieve this, the first step is to ensure that all nodes and edges in the given ADD is mapped to a specific structural construct. It is important to realize that this mapping is not a binding process. It is merely a temporary assignment of hardware implementation to ADD constructs so that we can estimate the layout quality. The mapping will not take into account the possibility of component sharing which is done during binding.

   Since the ADD is derived from a hardware model, the mapping can be easily accomplished by a one-to-one mapping of each ADD's construct to a hardware construct; read and write nodes are mapped to storage units, operator nodes are mapped to functional units, ADNs are mapped to buses or multiplexers, and edges are mapped to wires. Although the mapping of the ADD constructs to the hardware constructs can be achieved in a one-to-one manner, there can be multiple choices of implementation style for the hardware constructs. For example, an add operation can be mapped to an adder that can be implemented as a ripple or a carry-look-ahead. To resolve the issue of implementation style we assume that there is a preselected implementation style for each construct that would be used

by the technology mapper. The preselected implementation style can be defined by the user or randomly chosen from the database during synthesis.

2. Component grouping

   After all nodes in the ADD are assigned to a specific hardware implementation, the next step is to assign each node to one of the four component classes as defined in the layout models, namely, datapath, control logic, macros, and memories. Datapath consists of a set of regular components such as adders/subtracters, ALUs, MUXs, or registers that are to be layout in a bit-sliced architecture. Control logic consists of a set of random gates or a PLA associated with the datapath to execute data operations that are to be layout in a standard cells or PLA. Macros include some predefined components such as multipliers. Memories include register files, RAMs, and ROMs. Layouts of macros and memories are predesigned and their layout characteristics are given.

3. Quality estimating

   The estimation algorithm requires components to be identified into different classes because each class has different layout characteristics and design styles and would required different estimation techniques and models. Detailed discussion of the area and the timing estimation algorithm using layout models can be found in [21, 6, 14]. The algorithm provides estimates for chip area in terms of number of square microns. The chip area includes floor plan, routing, and cell areas. In addition, timing estimate is provided by the algorithm in terms of the delay of each register-transfer path and the clock period of the design, (the clock period is defined to be the most critical register transfer path).

9 "Partial" uniqueness of the ADD

One of the ultimate goal in the research on representation for synthesis is the search of a unique representation. A unique representation is one that able to represent any descriptions that have the same functionality in one unique topology. Synthesis system that uses a unique representation would be independent from variances in the descriptions and, thus, would be easy to use. Such unique representation exists in the logic synthesis, for example the Binary Decision Diagram [2, 9]. However, to the best of the authors' knowledge, a unique representation for the high-level synthesis is not known to exist.

Deriving a unique representation is as difficult as proving the equivalence of two descriptions/programs. Hence, deriving a unique representation for the high-level synthesis is a very difficult task, if it is possible. We consider the ADD representation as a one step closer to a unique representation because the ADD can provide the uniqueness for a certain class of descriptions. To be specific, descriptions that are differed in the ordering

and grouping of conditional and assignment statements can be uniquely represented in the ADD [5]. However, the ADD will not be uniqued for descriptions that are differed due to loop constructs. This is because the transformation algorithm from an input description to a ADD assumes that there is a state boundary at each loop boundary.

10 conclusion

This report proposes a representation for high-level synthesis called the Assignment Decision Diagram (ADD) that is complete, partially unique and efficient. The completeness of the ADD is demonstrated by illustrating methods of representing commonly used VHDL constructs and Register-Transfer-Level components. The ADD can uniquely represent functionally equivalent descriptions that are differed in the ordering and grouping of conditional and assignment statements. In addition, the ADD also furnishes many synthesis tasks with information that can simplify the tasks and can enrich the results of the synthesis. Figure 26 shows the summary of features that can be obtained from the ADD but are not available from the three commonly used representation, namely, CDFG, VT, and CF-DFG.

High–level synthesis tasks	Features that are provided by the ADD but are complicated to obtain from CDFG, VT, and CF–DFG
Minimizing of syntactic variances	Reduces syntactic variances that are due to : –ordering and grouping of assignment statements –ordering and grouping of conditional statements
Scheduling	Provides the following features even with a simple data–flow scheduling technique 1) scheduling across basic blocks, 2) scheduling of branch conditions and computations in the branch at the same time, 3) scheduling and merging of mutually exclusive operators at the same time.
Interleave scheduling and binding	–Allows scheduling and binding in any order –Allows scheduling with partial binding information –Allows binding with partial scheduling information
Quality measures	Provides fast estimates of layout area and timing characteristics with high fidelity.

Figure 26: Summary of features provided by the ADD that are complicated to obtain from CDFT, VT, and CF-DFG.

11 References

[1] M.R. Barbacci, G.E. Barnes, R.G. Cattell, and D.P. Siewiorek, "The ISPS Computer Description Language," Technical Report, Department of Computer Science, Carnegie-Mellon University, 1977.

[2] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. CAD*, vol.C-15, no.8, pp.677-689, Aug. 1986.

[3] R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Trans. CAD*, Vol.10, no.1, pp.85-93, Jan. 1991.

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

[5] V. Chaiyakul, D.D. Gajski and L. Ramachandran, "High-level Transformations for Minimizing Syntactic Variances," *Proc. 30th DAC*, pp. 413-418, 1993.

[6] V. Chaiyakul, A. C.-H. Wu and D.D. Gajski, "Timing Models for High-Level Synthesis," *Proc. EURO-DAC*, pp.60-65, 1992.

[7] D.D. Gajski, N. Dutt, A. Wu and S. Lin, *High-level Synthesis, Introduction to Chip and System Design,* Kluwer Academic Publishers, 1992.

[8] T. Kim, J.W.S. Liu, and C.L. Liu, "A Scheduling Algorithm For Conditional Resource Sharing," *Proc. ICCAD 91*, pp.84-87, 1991.

[9] C.Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell. Syst. Tech. J.,* vol.38, pp.985-999, Jul 1959.

[10] J.S. Lis and D.D. Gajski, "Synthesis from VHDL," *Proc. IEEE Int. Conf. on Computer Design'88*, pp.378-381, 1988.

[11] M.C. McFarland, "The Value Trace: A Data Base for Automated Digital Design," PhD. Dissertation, Department of Electrical Engineering, Carnegie-Mellon University, 1978.

[12] A. Orailoglu and D.D. Gajski, "Flow Graph Representation," *Proc. 23rd DAC.*, pp.503-509, 1986.

[13] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Trans. CAD*, vol.8, no.6, pp.661-679, Jun. 1989.

[14] C. Ramachandran, F.J. Kurdahi, D.D. Gajski, A. C.-H. Wu and V. Chaiyakul, "Accurate Layout Area and Delay Modeling for System Level Design," *Proc. ICCAD 92*, pp.355-361, 1992.

[15] L. Ramachandran, F. Vahid, S. Narayan and D.D. Gajski, "Semantics and Synthesis of Signals in Behavioral VHDL," *Proc. EURO-DAC 92*, pp.616-621, 1992.

[16] E.A. Snow, "Automation of Module Set Independent Register-Transfer Level Design," PhD. Dissertation, Department of Electrical Engineering, Carnegie-Mellon University, 1978.

[17] C.-J. Tseng, R.W. Wei, S.G. Rothweiler, M. Tong and A.K. Bose, "Bridge: A Versatile Behavioral Synthesis System," *Proc. 25th DAC.*, pp.415-420, 1988.

[18] *Standard VHDL Language Reference Manual.* New York: The Institute of Electrical and Electronics Engineers, Mar. 1988.

[19] R.A. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, 1991.

[20] R.A. Walker and D.E. Thomas, "Behavioral Transformations for Algorithmic Level IC Design," *IEEE Trans. CAD*, vol.8, no.10, pp.1115-1128, Oct. 1989.

[21] A. C.-H. Wu, V. Chaiyakul and D.D. Gajski, "Layout-area models for high-level synthesis," *Proc. ICCAD 91*, pp. 34-37, 1991.