Center for Reliable Computing

TECHNICAL REPORT

Reducing Synchronization in Concurrent Behavioral Descriptions

Nur A. Touba

92-3

(CSL TN # 92-389)

August 1992

Center for Reliable Computing
ERL 460
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract:

The necessity to synchronize concurrent modules in a behavioral hardware description adds constraints to behavioral synthesis, thereby limiting the quality of the resulting implementation. This report shows how an improved hardware implementation can be obtained by reducing synchronization requirements in a concurrent behavioral description. Two high-level transformations that reduce synchronization are described along with a method that uses an activation graph to identify when and where the transformations can be applied to a given description. Since not all applications of the transformations result in an improved implementation, the process of selecting which transformations to apply is discussed.

Funding:

This work was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and administered through the Office of Naval Research under Contract No. N00014-92-J-1782, and by the National Science Foundation under Grant No. MIP-9107760.

Imprimatur: LaNae Avra and Francoise Martinolle

Reducing Synchronization in Concurrent Behavioral Descriptions

Nur A. Touba

CRC Technical Report No. 92-3
(CSL TN No. 92-389)
August 1992

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

ABSTRACT

The necessity to synchronize concurrent modules in a behavioral hardware description adds constraints to behavioral synthesis, thereby limiting the quality of the resulting implementation. This report shows how an improved hardware implementation can be obtained by reducing synchronization requirements in a concurrent behavioral description. Two high-level transformations that reduce synchronization are described along with a method that uses an activation graph to identify when and where the transformations can be applied to a given description. Since not all applications of the transformations result in an improved implementation, the process of selecting which transformations to apply is discussed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Behavioral Descriptions use formal procedural languages such as Ada [Girczyc 85], HardwareC [Ku 90], or VHDL [Camposano 91] to model the functionality of hardware. These descriptions are used as input for *behavioral synthesis* in which they are transformed into a hardware implementation. As suggested in [Mostow 84], the process of behavioral synthesis can be thought of as an automated application of a series of transformations which are proven to preserve behavior. Behavior equivalence for synthesis is defined in [Camposano 89] as equal output sequences for equal input sequences and equal initial state (more formally called reduced behavior equivalence in [Manna 91]).

Because of the inherent ability of hardware to do operations in parallel, it is both natural and generally more efficient to model concurrency at the functional level. For this reason, behavioral description languages typically provide some means for specifying blocks of sequential statements that execute concurrently as well as constructs that allow for synchronization between the concurrent modules. A *wait-block*, as defined here, is a block of sequential statements that waits for a particular event (called an *activation event*) to be generated by other wait-blocks or primary inputs before it executes. Wait-blocks can be used as a simple model for the synchronization that exists in a description by abstracting the actual mechanisms and language constructs that specify the synchronization. Examples of how synchronization in concurrent behavioral descriptions can be modeled by wait-blocks will be given in Sec. 2.

An overview of the main steps in behavioral synthesis from a sequential specification is given in [Camposano 90]. When applying these steps to the synthesis of a concurrent behavioral description, synchronization between concurrent modules adds extra constraints. Considering synchronization as modeled by wait-blocks, the quality of the resulting implementation is limited by the following:

1) <u>Any wait-blocks that could potentially execute simultaneously cannot share resources:</u>

   There is no guarantee that more than one wait-block may need a particular resource at the same time.

2) <u>Each wait-block must be scheduled independently:</u>

   The time when one wait-block is executed relative to another wait-block is dynamically determined, thus the relative ordering of operations between wait-blocks is not fixed.

3) <u>Each wait-block requires synchronization logic:</u>

   Logic is needed to determine when the activation event occurs.

Because of these limitations, under some circumstances it is advantageous to perform high-level transformations that reduce the synchronization in a description by combining some of its wait-blocks. The resulting description can then be synthesized such that it results in a better implementation. The operation of combining wait-blocks will be referred to as *fusion* as it is in [Martinolle 91] where it is described for VHDL process statements.

This report describes two high-level transformations that reduce synchronization in concurrent behavioral descriptions (Sec. 3). It then presents a method for analyzing a concurrent description to identify wait-blocks that can be combined using these transformations (Sec. 4). Lastly, the process of selecting transformations to apply is discussed (Sec. 5).

2. WAIT-BLOCKS

A behavioral description can be partitioned into a set of wait-blocks. If the description contains no concurrency or no synchronization, then the set contains only one element, namely the whole description itself. Identifying wait-blocks in a concurrent behavioral description depends on how synchronization is specified. The two common paradigms for synchronization are shared variables and message passing. Synchronization with shared variables is typically specified by a control construct that suspends execution of a concurrent module until a variable is assigned a particular value by another concurrent module. An example of this is the *wait* statement in VHDL which suspends execution of a process until an event on a shared signal is generated by another process. Another example is the *while* statement in HardwareC which can be used to halt execution in a process until a particular value appears on one of its ports. Synchronization with message passing is typically specified by a control construct that suspends execution of a concurrent module until a particular message is received. An example of this is the *send* and *receive* statements in HardwareC which can be used to suspend execution of a process until it receives a message sent by another process.

An example of a concurrent VHDL description and its corresponding set of wait-blocks is shown in Table 1. Each *wait* statement delimits a wait-block. Notice that the first process contains two wait-blocks, so in order to maintain the sequential ordering between the two wait-blocks, their activation events must include a fictional state variable S (used only for modeling, not needed in implementation) that keeps track of which wait-block is currently active. The other processes contain only one wait-block, so the activation event is just the wait condition. Less straightforward is the example of a concurrent HardwareC description and its corresponding set of wait-blocks which is shown in Table 2. The first process contains two *while* statements which provide synchronization, so it is divided into two wait-blocks with appropriate activation events. The second process contains a *receive* statement which provides synchronization, so it is modeled as a wait-block that waits for a message event.

As seen in the examples, a single concurrent module may contain multiple wait-blocks. Wait-blocks in the same module have the property that they never execute simultaneously, thus they can share resources, but they still must be scheduled independently because the operations in one wait-block cannot be done until all the operations in the previous wait-block are completed.

Table 1. Example: Wait-Block Set for a Concurrent VHDL Description

Concurrent VHDL Description:	*Wait-block Set:*
process begin B <= ‘0’; **wait until** A = ‘1’; [statements 1] C <= ‘1’; D <= ‘1’; **wait until** B = ‘1’; [statements 2] **end process**; **process begin** C <= ‘0’; **wait until** C = ‘1’; [statements 3] **end process**; **process begin** D <= ‘0’; **wait until** D = ‘1’; [statements 4] B <= ‘1’; [statements 5] **end process**;	**wait-block 1** activation event (S=0 and A=1) [statements 1] C <= ‘1’; D <= ‘1’; S := ‘1’; **wait-block 2** activation event (S=1 and B=1) [statements 2] B <= ‘0’; S := ‘0’; **wait-block 3** activation event (C=1) [statements 3] C <= ‘0’; **wait-block 4** activation event (D=1) [statements 4] B <= ‘1’; [statements 5] D <= ‘0’;

Table 2. Example: Wait-Block Set for a HardwareC Description

HardwareC Description:	*Wait-block Set:*

process P1(a,b,c)
in port a,b;
out channel c;
[
while (!a);
[statements 1]
send (c,1);
while (!b);
[statements 2]
]
 	
process P2(c,b)
in channel c;
out port b;
[
b = 0;
receive (c)
[statements 3]
b = 1;
[statements 4]
]
	wait-block 1
	activation event (s=0 and a=0)
	[statements 1]
	send (c,1);
	s = 1;
	
	wait-block 2
	activation event (s=1 and b=0)
	[statements 2]
	s = 0;
	
	wait-block 3
	activation event (msg on c)
	[statements 3]
	b = 1;
	[statements 4]
	b = 0;

3. HIGH-LEVEL TRANSFORMATIONS THAT COMBINE WAIT-BLOCKS

As stated in [Camposano 90], the first step of behavioral synthesis is to perform high-level transformations which optimize circuit behavior by transforming one behavioral description into another behavioral description that is closer to optimal. For the reasons discussed in Section 1, given the set of wait-blocks for a concurrent behavioral description, it may be advantageous to combine some of the wait-blocks. Two high-level transformations that do this are sequential fusion and parallel fusion. They will be described in general terms here. More details on how they can be implemented using data and control flow graphs can be found in [Martinolle 91].

3.1 Sequential Fusion

Sequential fusion is a transformation that combines a wait-block with all other wait-blocks that can generate an event that activates it. It is similar to in-line expansion of procedures in software languages. The code in a wait-block is placed after all statements that can activate it along with any necessary conditional control statements.

An example of the sequential fusion is shown in Table 3. The transformation expands wait-block 3 after each of the statements that activate it. There is only one such statement which is located in wait-block 1. So the code in wait-block 3 is placed after the statement in wait-block 1 that activates it, thereby eliminating a wait-block while preserving the behavior of the system.

The limitation on sequential fusion transformation is that it is not possible to apply it to a wait-block whose activation event is externally generated. This occurs when a wait-block is waiting for an event that depends on a primary input signal. Since there is no way to determine the behavior of a primary input from the description given, no assumptions can be made about when such a wait-block will be activated.

Table 3. Example: Sequential Fusion Transformation on Wait-Block 3

Original Wait-Block Set:	*Transformed Wait-block Set:*
wait-block 1 activation event (S=0 and A=1) [statements 1] D <= ‘1’; C <= ‘1’; S := '1'; **wait-block 2** activation event (S=1 and B=1) [statements 2] B <= ‘0’; S := ‘0’; **wait-block 3** activation event (C=1) [statements 3] C <= ‘0’; **wait-block 4** activation event (D=1) [statements 4] B <= ‘1’; [statements 5] D <= ‘0’;	**wait-block 1** activation event (S=0 and A=1) [statements 1] D <= ‘1’; C <= ‘1’; [statements 3] C <= ‘0’; S := '1'; **wait-block 2** activation event (D=1 and B=1) [statements 2] B <= ‘0’; S := '0'; **wait-block 4** activation event (D=1) [statements 4] B <= ‘1’; [statements 5] D <= ‘0’;

3.2 Parallel Fusion

Parallel fusion combines a pair of wait-blocks that are activated by events that always occur together. The simplest case of this is if the activation events for two wait-blocks are identical. In general, parallel fusion is possible as long as the activation events always occur a bounded number of clock cycles apart (activation events that meet this criteria will be referred to as *compatible)*. Examples of activation event compatibility within blocks of sequential statements are shown in Table 4. Notice that in order for two activation events to be compatible, there cannot exist a case where one event occurs, but not the other. In each of the examples of non-compatibility, if (Z=0), then wait-block W_1 is activated, but wait-block W_2 is not activated, thus the two wait-blocks could not be combined.

Table 4. Examples of Activation Event Compatibility

X=1 event activates wait-block W_1
Y=1 event activates wait-block W_2

Compatible	**Not Compatible**
[sequential statements]	[sequential statements]
X <= 1	X <= 1
[non-control statements]	if (Z = 1) then Y <= 1
Y <= 1	[sequential statements]
[sequential statements]	
(X=1 $\Leftrightarrow$ Y=1)	(X=1 doesn't imply Y=1)
Compatible	**Not Compatible**
[sequential statements]	[sequential statements]
X <= Z	X <= 1
Y <= Z	Y <= Z
[sequential statements]	[sequential statements]
(X=1 $\Leftrightarrow$ Y=1)	(X=1 doesn't imply Y=1)
Compatible	**Not Compatible**
[sequential statements]	[sequential statements]
X <= 1	X <= 1
Z <= X	Z <= W xor X
Y <= Z	Y <= Z
[sequential statements]	[sequential statements]
(X=1 $\Leftrightarrow$ Y=1)	(X=1 doesn't imply Y=1)

An example of parallel fusion is shown in Table 5. Wait-block 3 is activated when C=1, and wait-block 4 is activated when D=1. Since the statements that activate each of these two wait-blocks are always executed together, the wait-blocks always execute together, and therefore can be combined into a single wait-block without changing the behavior of the system.

The limitation on the parallel fusion transformation is that it is not possible to apply it to a particular pair of wait-blocks w_1 and w_2, if some case exists in which one is activated without the other being activated. So, if there exists some wait-block which contains a statement that can activate w_1, but no compatible statement that activates w_2 under the same conditions, then parallel fusion of (w_1,w_2) is not possible.

Table 5. Example: Parallel Fusion Transformation on Wait-Blocks 3 and 4

Original Wait-Block Set:	*Transformed Wait-block Set:*
<u>wait-block 1</u> activation event (A=1 and B=0) A <= '0'; [statements 1] C <= '1'; D <= '1';	**<u>wait-block 1</u>** activation event (A=1 and B=0) A <= '0'; [statements 1] C <= '1'; D <= '1';
<u>wait-block 2</u> activation event (D=1 and B=1) [statements 2] B <= '0';	**<u>wait-block 2</u>** activation event (D=1 and B=1) [statements 2] B <= '0';
<u>wait-block 3</u> activation event (C=1) [statements 3] C <= '0';	**<u>wait-block 3</u>** activation event (D=1) [statements 3] C <= '0'; [statements 4] B <= '1'; [statements 5] D <= '0';
<u>wait-block 4</u> activation event (D=1) [statements 4] B <= '1'; [statements 5] D <= '0';	

4. ANALYZING CONCURRENT DESCRIPTIONS

Because there are restrictions on which wait-blocks can be used for sequential and parallel fusion, a concurrent behavioral description must be analyzed in order to determine the possible transformations that may exist. In this section, a method using an activation graph is presented.

4.1 Activation Graph

An activation graph is a directed graph which shows the possible interactions among the wait-blocks. It is formally defined as activation graph $G = (V,E)$ where V is the set of vertices and E is the set of directed edges where each edge $e \in E$ has a condition $c(e)$ associated with it. Let W be a set of vertices containing one vertex for each wait-block in the description, and let P be a set of vertices containing one vertex for each primary input signal in the description, then $V = W \cup P$. Let $S(w)$ be the set of statements and primary inputs that activate a wait-block $w \in W$, let $l(s) \in V$ be the vertex corresponding to the location (either wait-block where statement is found, or primary input) of $s \in S(w)$, and let $c(s)$ be the condition under which s activates w, then $E = \{ (l(s),w) \mid w \in W, s \in S(w) \text{ and } \forall[(l(s),w) \in E]\ c((l(s),w)) = c(s) \}$.

The condition associated with each edge (v_1,v_2) is determined by considering the vertex from which the edge emanates, v_1. Note that v_1 can represent either a primary input or a wait-block, but v_2 must represent a wait-block. If v_1 represents a primary input, then the condition is simply the value that the primary input must change to in order to generate an activation event for wait-block v_2. If v_1 represents a wait-block, then the condition is an expression in terms of variables in v_1 that indicate whether or not an activation event for v_2 will be generated when v_1 is executed. Therefore, if $p \in P$ changes value such that $c((p,w))$ is satisfied, then w is activated, and if $w_1 \in W$ is executed such that $c((w1,w2))$ is satisfied, then w_2 is activated.

An example of an activation graph is shown in Figure 1. There is one vertex corresponding to the primary input RESET and four vertices corresponding to the four wait-blocks. Wait-block 1 is activated when RESET = 0, so there is a directed edge from the RESET vertex to the W_1 vertex with the condition RESET = 0 associated with it. When C = 1, wait-block 1 activates both wait-block 3 and wait-block 4, so there are directed edges from the W_1 vertex to the W_3 and W_4 vertices with the condition C = 1 associated with each edge. Similarly, there are edges for rest of the statements in the description that can activate a wait-block. Thus, the graph shows all synchronization in the description.

Figure 1. Example of Activation Graph

Wait-Block Set:

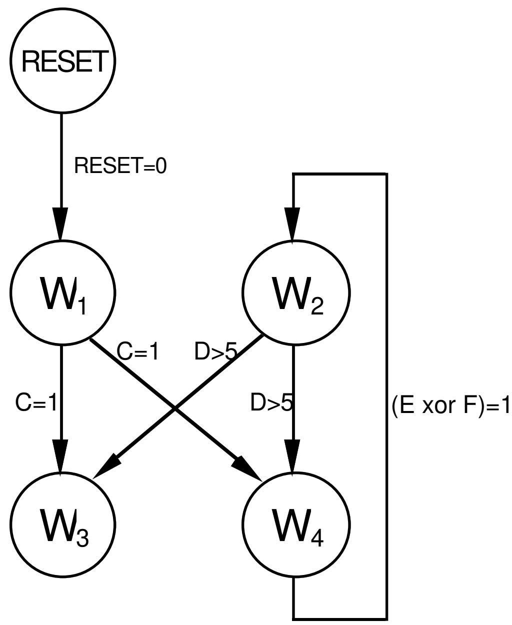
```
wait-block 1
activation event (RESET=0)
        A <= C;
        [statements 1]
        B <= C;

wait-block 2
activation event (G=1)
        G <= 0;
        [statements 2]
        if ( D > 5 ) then
     A <= 1;
     B <= 1;
        endif;

wait-block 3
activation event (A=1)
        A <= 0;
        [statements 3]

wait-block 4
activation event (B=1)
        B <= 0;
        [statements 2]
        G <= E xor F;
```

Activation Graph:



$SEQFUSION = \{w_2, w_3, w_4\}$

$PARFUSION = \{ (w_3,w_4), (w_4,w_3) \}$

$PARCLASSES = \{ (w_3,w_4) \}$

4.2 Analyzing Activation Graph for Sequential Fusions

The condition for which the sequential fusion transformation can be applied to a wait-block is that its activation event cannot be externally generated (i.e. depend on primary input). An activation graph can be used to check for this condition by seeing that the vertex for a wait-block doesn't have any incident edges emanating from a primary input vertex. The set of wait-blocks for which this condition is met is formally defined as:

$$SEQFUSION = \{\ w \mid w \in W \text{ such that } \forall[(v,w) \in E]\ v \notin P\ \}$$

Each element in the *SEQFUSION* set represents a possible sequential fusion transformation. The *SEQFUSION* set for the example activation graph is shown in Figure 1.

4.3 Analyzing Activation Graph for Parallel Fusions

The condition for which the parallel fusion transformation can be applied to a pair of wait-blocks is that their activation events must be generated by the same set of wait-blocks and primary inputs under the same conditions. An activation graph can be used to check for this condition by seeing that the incident edges for each vertex in the pair emanate from the same set of vertices A and the condition on each edge directed from each $a \in A$ to each vertex in the pair is equivalent. The set of all pairs of wait-blocks for which this condition is met is formally defined as:

$PARFUSION = \{\ (w_1,w_2) \mid A_1 = A_2 \text{ and } \forall[a \in A_1 \text{ and } (a,w_1), (a,w_2) \in E]\ c((a,w_1)) = c((a,w_2))\ \}$

where $A_i = \{\ a \mid (a,w_i) \in E\ \}$. Each pair in the *PARFUSION* set represents a possible parallel fusion transformation. Note that the *PARFUSION* set is a relation on the set of wait-blocks W and has the following three properties:

1) Trivially reflexive
2) Symmetric since w_1 and w_2 are interchangeable in the definition, so
   $(w_1,w_2) \in PARFUSION \Leftrightarrow (w_2,w_1) \in PARFUSION$
3) Transitive since $(w_1,w2) \in PARFUSION$ and $(w_2,w_3) \in PARFUSION$
   $\Rightarrow A_1 = A_2$, $A_2 = A_3$, $c((a,w_1)) = c((a,w_2))$, and $c((a,w_2)) = c((a,w_3))$
   thus, $A_1 = A_3$ and $c((a,w_1)) = c((a,w_3))$ so $(w_1,w_3) \in PARFUSION$

These three properties imply that *PARFUSION* is an equivalence relation. Therefore, it partitions W into disjoint equivalence classes. Let *PARCLASSES* be defined as the set of equivalence classes that have more than one member. The wait-blocks in each equivalence class contained in *PARCLASSES* can be transformed into a singe wait-block using the parallel fusion transformation. Both the *PARFUSION* and *PARCLASSES* sets for the example activation graph are shown in Figure 1.

5. SELECTING TRANSFORMATIONS TO APPLY

Once the sets of possible transformations have been found, the next problem is to determine which of these transformations should be applied. In general, reducing synchronization in a concurrent description will increase its size, thus offsetting the advantages of having fewer wait-blocks. Typically, only a small subset of the possible transformations for reducing synchronization will produce a more optimal implementation. If the number of wait-blocks is small, it may be possible to exhaustively try each of the transformations and keep only those which improve the resulting implementation. When this is not practical, a smaller subset of the possible transformations for which the probability of improvement is highest may be tried.

5.1 Probability of Improvement using Sequential Fusion

First consider the impact of sequential fusion of a wait-block on the area of an implementation. The change in area of an implementation is strongly correlated to the number of places that a wait-block has to be expanded because the code in a wait-block has to be replicated after each statement that can activate it, thus increasing the size of the description. This increase in size is offset by the fact that more efficient scheduling and resource allocation is possible. Formal bounds on the change in area can be determined. Assuming that the changes in the area required by control logic is negligible, when wait-block w is sequentially fused with one or more other wait-blocks:

$$[-\,\mathit{resource\_area}(w) - \mathit{sync\_area}(w)] \leq \text{change in area} \leq [(n-1)\mathit{resource\_area}(w) - \mathit{sync\_area}(w)]$$

where w is the wait-block that sequential fusion is applied to, *resource_area(w)* is the area required by resources used in w, *sync_area(w)* is the area needed for synchronization logic for w, and n is the number of statements that can activate w. The lowerbound is determined by considering the best case, where it is possible to schedule all additional operations (formerly done in w) such that they are performed using existing resources. In this case, the area of the implementation is reduced by the area that was formerly used for w. The upperbound is determined by considering the worst case, where it is not possible to use any existing resources for the additional operations (formerly done in w). In this case, extra resources are needed for each expansion of w, so the area of the implementation is this new resource area minus the area that was formerly used for w. Note that if a wait-block is activated by only one statement ($n = 1$), the change in area will always be negative.

Now consider the impact of sequential fusion of a wait-block on the performance of an implementation. By expanding wait-block *w* within another wait-block *u*, the number of clock cycles required to complete *u* may be increased. Formal bounds on the change in the timing of a wait-block are:

$$0 \leq \text{change in } clock\_cycles(u) \leq (\text{n})[clock\_cycles(w)]$$

where *w* is the wait-block that sequential fusion is applied to, *u* is a wait-blocks that can activate *w*, *clock_cycles(w)* is the number of clock cycles needed to complete *w*, *clock_cycles(u)* is the number of clock cycle needed to complete *u*, and n is the number of statements in *u* that can activate *w*. The lowerbound occurs in the best case, where it is possible to schedule all additional operations (formerly done in *w*) within the number of clock cycles required for *u*. The upperbound occurs in the worst case, where the additional operations (formerly done in *w*) require as many additional clock cycles in *u* as they formerly required in *w*.

Whether a sequential fusion transformation improves the implementation depends on what factors are important for a particular system. Usually the goal is to minimize area while maintaining certain timing constraints. So the probability of improvement will generally be highest for sequential fusion of wait-blocks that are activated by the fewest number of statements. In general, it will only be worthwhile to spend computation time trying sequential fusion transformations that require few expansions.

5.2 Probability of Improvement using Parallel Fusion

First consider the impact of parallel fusion of wait-blocks on the area of an implementation. Formal bounds on the change in area that results from applying parallel fusion to all the members of an equivalence class contained in PARCLASSES can be determined. Assuming changes in the area required by control logic is negligible:

$$-\sum[area(w_i)] + \text{MAX}[area(w_i)] \leq \text{change in area} \leq -\sum[sync\_area(w_i)] + \text{MAX}[sync\_area(w_i)]$$

where w_i is the i-th element in the equivalence class, $area(w_i)$ is the total area required by w_i, and $sync\_area(w_i)$ is the area needed for synchronization logic for w_i. The lowerbound occurs in the best case, where all operations can be scheduled using only the resources needed by the largest wait-block in the equivalence class, thus eliminating the area used for each wait-block in the equivalence classes except for the largest. The upperbound occurs in the worst case where no operations can share resources, so the only savings is the elimination of duplicate synchronization logic. Note that the change in area will always be negative when the parallel fusion transformation is applied to all the members of an equivalence class.

Now consider the impact of parallel fusion of wait-blocks on the performance of an implementation. By applying parallel fusion to all members of an equivalence class, formal bounds on the change in timing for each wait-block in the class are as follows:

$$0 \leq \text{change in } clock\_cycles(w) \leq \sum[clock\_cycles(w_i)] - clock\_cycles(w)$$

where w is a wait-block in the equivalence class, w_i is the i-th element in the equivalence class, $clock\_cycles(w)$ is the number of clock cycles needed to complete w, and $clock\_cycles(w_i)$ is the number of clock cycles needed to complete w_i. The lowerbound occurs in the best case where it is possible to schedule all operations such that they are completed in the same number of clock cycles as w. The upperbound occurs in the worst case, where it is not possible to overlap the operations from each wait-block in the equivalence class, thus the number of additional clock cycles is the sum of the clock cycles needed for each wait-block in the equivalence class minus those needed in w. In general, parallel fusion will worsen the local timing of some wait-blocks.

Since the parallel fusion transformation always reduces the area of an implementation, the probability of improvement is high. However, it can worsen the performance of a system, thus it is necessary to check that the resulting implementation meets other design constraints. Because of the high probability of improvement, it is worth the additional computation time to try all possible parallel fusion transformations.

6. SUMMARY AND CONCLUSIONS

This report described how a concurrent behavioral description can be partitioned into a set of interacting wait-blocks that provide a simple model for the synchronization that exists in the description. It was shown that under some circumstances, a better hardware implementation may result from reducing the synchronization by combining some of the wait-blocks. Two high-level transformations that can be used to combine wait-blocks are sequential and parallel fusion. Sequential fusion expands a wait-block after all the statements that can activate it. Parallel fusion merges a pair of wait-blocks that always execute together into a single wait-block. An activation graph can be used to find all possible sequential and parallel fusion transformations. If the number of possible transformations is small, they can be exhaustively tried keeping only those that improve the implementation. When this is not possible, it was shown that it is most worthwhile to try sequential fusion transformations that require few expansions and all parallel fusion transformations.

ACKNOWLEDGMENTS

The author would like to thank LaNae Avra and Francoise Martinolle for their valuable comments and suggestions. This work was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and administered through the Office of Naval Research under Contract No. N00014-92-J-1782, and by the National Science Foundation under Grant No. MIP-9107760.

REFERENCES

[Andrews 83] Andrews, G.R., and F.B. Schneider, "Concepts and Notation for Concurrent Programming," *ACM Computing Surveys*, Mar. 1983, pp. 3-43.

[Avra 90] Avra, L., and E.J. McCluskey, "Behavioral Synthesis of Testable Systems with VHDL," *Dig. COMPCON Spring '90*, San Francisco, Feb. 26 - Mar. 2, 1990.

[Camposano 89] Camposano, R., "Behavior-Preserving Transformations for High-Level Synthesis," *Proc. Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects.* Springer Verlag, Berlin. 1989.

[Camposano 90] Camposano, R., "From Behavior to Structure: High-Level Synthesis," *IEEE Design & Test of Computers*, Oct. 1990, pp. 8-19.

[Camposano 91] Camposano, R., L.F. Saunders, and R.M. Tabet, "VHDL as Input for High-Level Synthesis," *IEEE Design & Test of Computers*, Mar. 1991, pp. 43-49.

[Chartrand 86] Chartrand, G., and L. Lesniak, *Graphs & Digraphs, Second Edition*, Wadsworth & Brooks, 1986.

[Girczyc 85] Girczyc, E.M., R.J.A. Buhr, and J.P. Knight, "Applicability of a Subset of Ada as an Algorithmic Hardware Description Language for Graph-Based Hardware Compilation," *IEEE Transactions on Computer-Aided Design,* Vol. 4, No. 2, Apr. 1985, pp. 134-142.

[Hoare 85] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hopcroft 79] Hopcroft, J.E., and Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[IEEE 88] IEEE Standard 1076-1987, "IEEE Standard VHDL Languages Reference Manual", IEEE Standards Board, 345 East 47th St., New York, NY 10017, 1988.

[Ku 90] Ku, D.C., and G. De Micheli, "HardwareC - A Language for Hardware Design, Version 2.0," *CSL TR 90-419*, Stanford University, Stanford, CA, Apr. 1990.

[Manna 91] Manna, Z., and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer Verlag, New York. 1991.

[Martinolle 91] Martinolle, F., “Fusion of VHDL Processes,” *CRC TR 91*, Stanford University, Stanford, CA, Dec. 1991.

[Mastow 84] Mastow, F., and Baizer, B., “A Program-Transformation Approach to VLSI Design,” *IEEE 1984 Workshop on the Engineering of VLSI and Software*, 1984, pp. 126-133.

[Thomas 87] Thomas, D.E., R.L. Blackburn, J.V. Rajan, “Linking the Behavioral and Structural Domains of Representation for Digital Systems Design,” *IEEE Transactions on Computer-Aided Design*, Vol. 5, No. 1, Jan. 1987, pp. 103-110.