Decomposition Methods for Library Binding of Speed-Independent Asynchronous Designs

Polly Siegel Giovanni De Micheli

Center for Integrated Systems
Stanford University, Stanford CA 94305

Abstract

We describe methods for decomposing gates within a speed-independent asynchronous design. The decomposition step is an essential part of the library binding process, and is used both to increase the granularity of the design for higher quality mapping and to ensure that the design can be implemented. We present algorithms for simple hazard-free gate decomposition, and show results which indicate that we can decompose most of the gates in our benchmark set by this simple method. We then extend these algorithms to work for those cases in which no simple decomposition exists.

1. Introduction

Asynchronous design styles have been increasing in popularity as device sizes shrink and concurrency is exploited to increase system performance. However, asynchronous designs are difficult to implement correctly because the presence of *hazards*, which do not affect the correctness of synchronous systems, can cause improper circuit operation. Many asynchronous design styles, together with accompanying automated synthesis systems, address the issues of design complexity and correctness. Typically, these synthesis systems take a high-level description of an asynchronous system and produce logic-level equations that are hazard-free for the given delay assumptions. For burst-mode asynchronous designs ([11], [18], [4]), the designer can then automatically translate this logic-level description into a technology-specific implementation using the asynchronous technology mapper described in [15]. However this technology mapper will not correctly map designs for speed-independent asynchronous design styles, because the different delay assumptions change the types of hazards that are of concern. In particular, the decomposition step of traditional library binding is difficult to perform in a hazard-free manner, requiring development of new theory and algorithms for decomposition of large gates so the general library binding problem can be addressed.

Several researchers have tackled gate-level synthesis of speed-independent designs ([17], [1]), although the gates may not be available in a particular library, due either to their complexity or their size. Varshavsky [17] showed that an implementation using n-input AND-OR-NOT gates or two-input NAND and NOR gates (with limited fanout) can be derived from any speed-independent signal transition graph (STG) without choice. These circuits are larger and more complex than those produced by Beerel's synthesis procedure [1], which uses unlimited-fanout basic gates to produce a hazard-free speed-independent design. No method addresses the general library binding problem, however. Because Beerel's style is more efficient and handles a wider range of specifications, we use it as the starting point for library binding (also called *technology mapping*).

In this paper we tackle the problem of decomposition for library binding of speed-independent designs. The decomposition step is an essential component of the library binding problem which splits each logic function into finer granularity base-functions, both to insure that an implementation composed of elements from the library exists and to improve the quality of the mapped circuit. We first describe the speed-independent design style and the hazards inherent in that design style. Next we analyze the basic library binding procedure presented in [15] to demonstrate its applicability to this design style. After showing that decomposition is the difficult step, we present theory and algorithms for hazard-free decomposition under the speed-independent delay assumptions. We implemented these algorithms in a new tool, and we ran the tool on some benchmark examples. The results demonstrate both the usefulness and limitations of the decomposition techniques, which must be combined with non-standard covering algorithms to yield efficient mapped circuits.

2. Background and definitions

2.1 Design style

There are several popular asynchronous design styles that are the subject of active research. In this paper we are interested in the speed-independent asynchronous design style, in which wires are assumed to have zero delay and gates can have unbounded but finite delay. Each delay element is assumed to exhibit *pure* delay properties [7]. Thus, glitches will not be filtered out by inertial delays and any glitch can be propagated to the output of a gate.

Each gate is assumed to be *atomic*, and can be modeled as an instantaneous Boolean function of its inputs with a single pure delay on the output. The atomic gate assumption implies that each

©1994 ACM 0-89791-690-5/94/0011/0558 $3.50

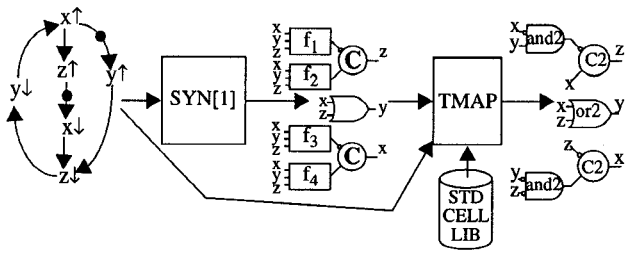


Figure 2.1: Illustration of synthesis flow

gate in the circuit either exists as a single CMOS gate in a library or that a custom gate will be created for it. If a circuit is implemented as an atomic gate, then given any input change the gate will respond with a corresponding output change after some (potentially unbounded) delay. Because of the delay assumptions, the relative ordering of signal transitions on the inputs to a gate will be preserved irrespective of the actual delays. This property will be important later on.

Inherent in the speed-independent model is the *isochronic fork* assumption [7], where an isochronic fork is a multi-fanout point with a single delay. This implies that if the output of the driving gate changes then the inputs to the driven gates will change at the same time. In the physical implementation, the isochronic fork assumption requires help from a block place-and-route tool to balance path delays.

A speed-independent implementation composed of combinational logic and C-elements[1] is used as input to the technology mapper. This implementation may have been synthesized from a semi-modular speed-independent signal transition graph or state graph ([3], [10], [1]), or may have been derived by some other means. Although the specification style is not important, knowledge of the environment is necessary to perform library binding for this design style. Figure 2.1 illustrates the synthesis flow starting from an STG and ending in a gate-level implementation using gates from a standard-cell or gate-array library.

There are several aspects of this specification and implementation style that impact library binding. First, since the gates in the initial implementation can have arbitrarily large fan-in, some of the gates produced by the synthesis method may not exist in the given library. Therefore, the library binding algorithm must be able to decompose these large gates into an interconnection of smaller gates while preserving the speed-independence of the design. Second, because of the speed-independent delay assumptions, relative signal ordering through each gate in the combinational logic must be preserved in the final implementation or hazards will result. This means that library binding must be done in the context of the circuit's environment, which has an impact on how gates are decomposed.

1. A two-input Muller C-element is a single-output sequential element with next-state equation C = AB + (A+B)C, where A and B are its inputs, and C is its output. In examples presented later in this paper, each C-element has one inverted input, implying that its output will change when the two inputs change to be of opposite value.

2.2 Hazards and gate decomposition

A *hazard* is an unwanted output glitch in response to a change in some input(s). Hazards may cause a design to operate incorrectly. The initial design (the input to the library binding step) is assumed to be hazard-free within the context of the environment in which it operates. For the library binding problem, focusing on the hazard behavior of the combinational logic is sufficient to insure that the final implementation is still hazard-free.

There are two basic classes of combinational hazards: *function* and *logic* hazards. Function hazards are a property of the logic function, whereas logic hazards are purely a property of the implementation. Although a function hazard cannot be eliminated, it can be avoided by controlling the sequence of input changes to the function. If a network has a function hazard for a given transition, then it cannot also have a logic hazard for that same transition. Each class of hazard includes both *static* and *dynamic* hazards. Given that a transition is being made between two points α and β in the input space, static hazards apply to transitions where $f(\alpha) = f(\beta)$, and dynamic hazards apply to cases where $f(\alpha) \neq f(\beta)$. (More thorough treatment of hazards can be found in [16] and [9].)

Because library binding must not introduce hazards, combinational logic hazards cannot be tolerated in the design, as is the case with burst-mode designs [15]. Logic transformations that alter the order of signal propagation through the network can cause additional hazards to be exercised in speed-independent designs.

In addition to combinational hazards, *sequential hazards* can occur as a result of the delay assumptions. The notion of *acknowledgment* is key to the definition:

Definition 2.1 *An output transition on a gate is* acknowledged *if one of the gates connected to its fanout cannot change until that output transition reaches its input.* [1]

Proposition 2.1 *An unacknowledged transition on a gate output can result in a sequential hazard in the circuit.*

With the burst-mode design style we were able to take advantage of delay assumptions in the feedback path to ignore these types of hazards. However, speed-independent circuits have no such simplifying assumptions. Improper decomposition can yield unacknowledged transitions resulting in hazards.

3. Library binding

As in [15], we started with a synchronous mapping procedure and examined each step to see its effect on the hazard behavior of the design. We found the decomposition step to be the most troublesome and, after briefly describing the overall library binding procedure, we focus on the decomposition step for the remainder of the paper. After characterizing the initial design, we show how improper decomposition can lead to hazards, motivating the need to characterize the circuit environment (a step that was not necessary for library binding of other design styles). We then present theorems supporting a simple hazard-free decomposition algo-

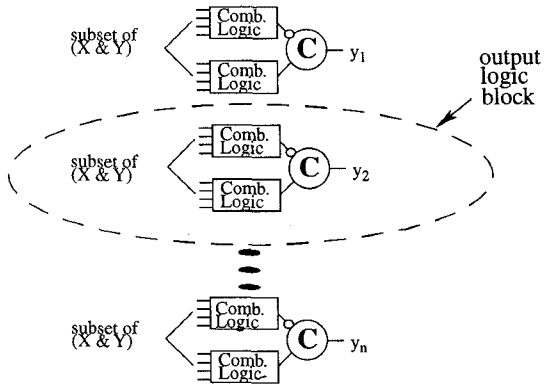


Figure 3.1: Initial Implementation

rithm which works for many gates. Finally, we extend the simple procedure to handle cases where the simple procedure failed.

The approach to library binding for burst-mode asynchronous designs taken in [15] was based on synchronous mapping procedures ([6], [13], [8]). Since the general library binding problem is known to be intractable, heuristic algorithms are used. First, the initial network, which is represented as a directed acyclic graph (DAG), is decomposed into a multi-level network composed of simple gates (e.g., two-input AND/OR gates). The decomposition step is used both to ensure that the circuit is implementable (the base functions are assumed to be in the technology library), and to increase the size of the solution space for subsequent steps. Next, the circuit is partitioned into sets of single-output *cones* of logic, where a cone of logic represents a subnetwork obtained by cutting the network at points of multi-fanout. All possible matches to library elements are then found for subnetworks within each logic cone. Finally, an optimal set of matching library elements is selected from the set of matches to realize the network without introducing hazards. The procedure is outlined below:

```
procedure bm-tmap(network, library) {
  decomposed_network = tech_decomp(network);
  cones = partition(decomposed_network);
  foreach output in cones {
    find_best_async_cover(output, library);
  }
}
```

For speed-independent designs, the decomposition step requires knowledge of the circuit environment as well as the initial technology-independent circuit description. The partitioning and matching/covering algorithms described in [15] can be used without modification to yield a correct implementation in the given technology. Modifications to the covering routines to give better quality results can be found in [14].

3.1 Initial design description

Although there are several synthesis procedures that produce speed-independent logic, we chose to start with an initial unoptimized speed-independent sequential implementation produced by SYN [1]. This initial design consists of C-elements and logic blocks, where each logic block is modeled as a disjoint sum-of-products (SOP) expression and can be equivalently represented as a set of AND and OR gates. We discuss the SOP form for clarity; the synthesis method can produce multilevel logic. The algorithms can easily be extended to multi-level logic as well.

The circuit is first partitioned by breaking the circuit at the outputs of the C-elements. This induces a partition into blocks of circuitry which implement each primary output. From that, "cones" of combinational logic that feed into the C-elements are extracted and passed along to be further decomposed. We assume that the two-input C-element exists in the library.

Definition 3.1 (Initial implementation) *An initial implementation of a speed-independent circuit consists of two-input C-elements implementing the outputs of the design, and blocks of combinational logic connected to each C-element input.*

Definition 3.2 (Output logic block) *An output logic block consists of a two-input C-element along with the combinational logic connected to each of its inputs. Each output logic block has a single output, and its inputs are a subset of the design's primary inputs and outputs.*

Figure 3.1 shows an example of the initial implementation of a design, composed of n output logic blocks. We can treat each output logic block independently during decomposition.

3.2 How hazards can occur during decomposition

Before characterizing the behavior of the circuit's environment, we will show through an example how knowledge of the environment is necessary to do proper decomposition.This example will illustrate how a static-0 hazard can occur in a speed-independent design due to improper decomposition of a gate.

Figure 3.2a shows a gate implementing the function $f = wxy$ and two different decompositions composed of cascaded connections of two-input AND gates. The input timing as derived from the STG is $y\downarrow$ followed by $x\uparrow$, as shown, with w remaining constant at 1. Because of the input signal ordering, the gate will take the trajectory shown in the Karnaugh map, ensuring that the function hazard in the AND gate is avoided. Thus, the gate's output will remain at 0 during the transitions on x and y.

Now, suppose the gate is decomposed so w and y are placed in a separate AND gate, as shown in Figure 3.2b. When y changes, the intermediate AND gate will go low some time later. Before that change has been seen by the final AND gate, x may have gone high, resulting in a glitch on the output of the gate.

Figure 3.2c shows a decomposition where w and x are connected to the same gate, and y is connected to the AND gate closest to the output. In this case y changes first, presenting a 0 on the output of the final gate. The intermediate AND gate changes after x changes, but the output of the network will not change, since y is guaranteed to be low before the change in the intermediate AND gate reaches q, keeping the final AND gate low.

In this example, there is a multi-input change on the inputs to the AND gate, and proper signal ordering is required to avoid the static-0 hazard. For the decomposition in Figure 3.2b, the relative order between the two signals is not maintained, resulting in the hazard. In Figure 3.2c, the critical signal ordering is maintained, and the decomposed circuit is hazard-free.

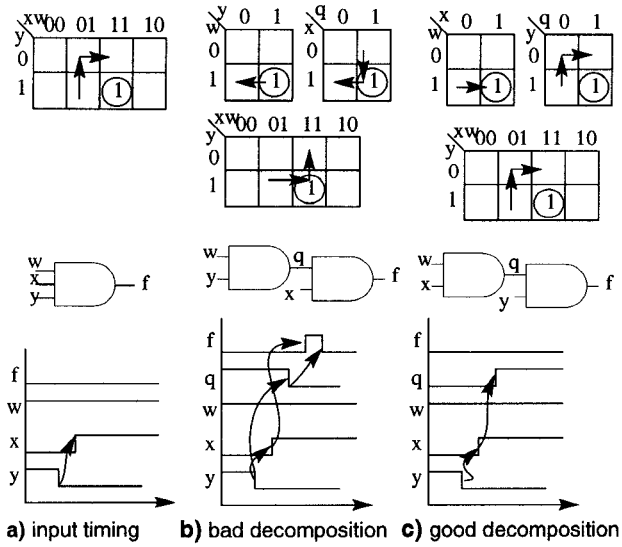


Figure 3.2: Example of AND decompositions

This example suggests that circuit transformations must preserve the ordering of certain signals through internal combinational logic of any decomposed gates. These transformations are dependent upon the specification, which defines the signal ordering.

3.3 Characterization of the circuit's environment

As we have shown, knowledge of signal transition ordering is essential to be able to decompose the circuit in a hazard-free way. Therefore, we cannot do the decomposition without knowledge of the signal ordering—we must use information from the specification (i.e., the STG or state graph) to drive the decomposition.

Definition 3.3 *An* input burst *is an unordered set of input transitions that can appear at the input to an output logic block.*

Input bursts are denoted as signal transitions separated by vertical bars enclosed within curly braces, for example $\{x_i\downarrow \| x_j\uparrow \| x_k\uparrow\}$. They represent a concurrent portion of a state graph, where the transitions may occur in any order. A degenerate burst consists of a single signal transition.

Definition 3.4 *An* input sequence *is an ordered set of input bursts.*

Input sequences are denoted as input bursts separated by commas enclosed within parentheses. An example of an input sequence is $(x_i\uparrow, x_j\downarrow, x_k\uparrow)$, where the signals are all inputs to the output logic block.

Definition 3.5 *A* context signal *is a signal that remains constant during a given input sequence and is one of the inputs to an output logic block. Furthermore, the* context *for a given sequence is the set of context signals for that input sequence.*

Proposition 3.1 *The input-output behavior of an output logic block for output x_i is completely characterized by the set of possible (input sequence; context) tuples extracted from the STG for each instance of $x_i\uparrow[x_i\downarrow]$, where the starting point of each sequence is the state immediately following the preceding downward [upward] transition on x_i, and the context for that sequence are the variables that don't change during that sequence.*

3.4 Decomposition

Having characterized the initial implementation and circuit's environment, we can now tackle the decomposition step. We first show that the sequential portions of the circuit can be separated from the combinational portions of the circuit. Next, we show that each cone of logic can be treated independently, and furthermore, that each AND gate within a cone of logic also operates independently—i.e., that the inputs to the OR gate in the implementation of an SOP expression operate disjointly. We then formulate legal decompositions of each gate for each type of input-output behavior that the gate may experience (inputs change⇒output rises; inputs change⇒output falls; inputs change⇒output does not change). We propose a simple algorithm for doing a hazard-free decomposition, and present results showing where it succeeds and fails. Finally we extend this algorithm to insert wire forks to eliminate sequential hazards, and show results of applying this algorithm to the cases that failed on the simple algorithm. We present theorems without proof; proofs can be found in [14].

Theorem 3.1 *Given the implementation of an output logic block as in Definition 3.2, each input to the C-element makes a monotonic transition before the C-element's output changes. Furthermore, the inputs to the C-element can change in any order, and we can therefore treat each combinational logic block feeding into the C-element independently.*

Corollary 3.1.1 *During an input sequence, each input to the C-element in an output logic block will change exactly once. Additionally, the inputs will change in opposite directions.*

Now, let us take a look at the hazard-behavior of the individual output logic blocks.

Theorem 3.2 *Given a block of logic that consists of an AND-OR implementation of a sum-of-products expression, the OR gate can be decomposed according to the associative law, independent of signal ordering, and the resulting decomposed network will still be hazard-free.*

Corollary 3.2.1 *Given a block of logic that consists of an AND-OR implementation of a sum-of-products expression, and a set of (input sequence; context) tuples that completely describe the behavior of the block of logic, the OR gate will only see single-input changes.*

Corollary 3.2.1, coupled with Theorem 3.2, allows us to decompose each AND gate independently. We do so by partitioning the input sequences for each AND gate into three sets: one resulting in a falling transition of the AND gate, another resulting in a rising transition of the AND gate, and another resulting in no transition of the AND gate. Any decompositions that are valid for

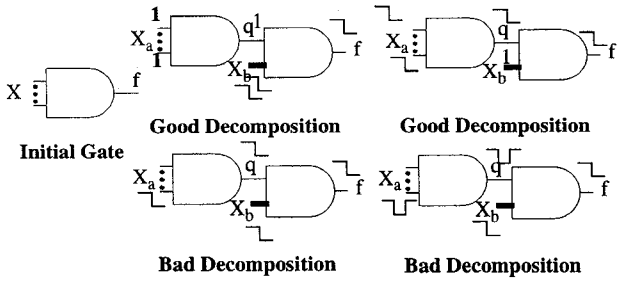


Figure 3.3: Illustration of Theorem 3.3

all three sequence types are valid decompositions. Note that for all cases we only consider disjoint decompositions (i.e., where none of the input variables are replicated).

Theorem 3.3 (Falling Transition of an AND gate) *Given an atomic AND gate with inputs X, a set of (input sequence; context) tuples resulting in a falling transition on its output, and a decomposition of the gate into two cascaded AND gates, where the inputs to the intermediate AND gate are $X_A \subset X$, the output of the intermediate AND gate is a new signal q, and the inputs to the final AND gate are $X_B \cup q$, where $X_B = X - X_A$, then this decomposition is hazard-free for this set of input sequence; context) tuples if and only if a transition on q for a given input sequence implies that no transition can occur on the inputs in X_B, and a transition on one or more inputs in X_B for a given input sequence implies that no transition can occur on q.*

Let us now apply Theorem 3.3 to one of our previous examples. In Figure 3.4 we have one (input sequence; context) tuple for *f↓: (w↓, y↓; x=1)*. We partition the gate such that X_A={*w, x*} and X_B = {*y*}, where *q* is the output of the intermediate AND gate. We observe that there is a transition on *w*, leading to a transition on *q*, along with a transition on *y*. Therefore the decomposition is invalid. In Figure 3.4b, X_A={*w, y*}, and X_B = {*x*}. Since *x* is a context signal, it does not change during the input sequence and this is a valid decomposition for that sequence.

Corollary 3.3.1 *Given a hazard-free decomposition of AND gates as defined by Theorem 3.3, and a set of (input sequence; context) tuples that result in falling transitions of the final AND gate, then the only valid decompositions for a given (input sequence; context) tuple in that set are those in which the signals that change in the input sequence are connected to a single gate (i.e., either X– {context signals} ⊆ X_B or X – {context signals} ⊆ X_A).*

Again, referring to Figure 3.4b we can see that *X - {context signals} = {w, x, y} - {x} = {w,y}* = X_A, satisfying the corollary.

For the rising transition of an AND gate, there are several important points to consider. First, the AND gate will change when the last arriving signal goes to 1. This means that all intermediate nodes will need to be 1 before the AND gate can change, and therefore there can be no unacknowledged signals. Second, it is possible for a subset of signals to change during a given input sequence so they would cause an intermediate AND gate to expe-

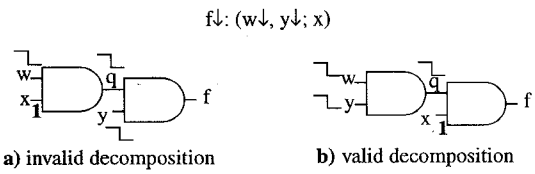


Figure 3.4: Illustration of Theorem 3.3: decompositions for a falling transition.

rience a dynamic function hazard. This second point is the critical point for the decomposition, since without that we could decompose the AND gate in any logic-hazard-free way. In other words, we need to break up the gate so intermediate gates change monotonically during an input sequence.

Theorem 3.4 (Rising Transition of AND) *A decomposition of an atomic AND gate into two cascaded AND gates is hazard-free with respect to its set of rising (input sequence; context) tuples if and only if for each (input sequence; context) tuple in the set, the output of the intermediate AND gate of the decomposed circuit makes at most a single monotonic transition.*

Corollary 3.4.1 *Given the above assumptions on the AND gate, if the transitions on the inputs of all intermediate AND gates are monotonic then the decomposition is hazard-free for rising transitions of the AND gate.*

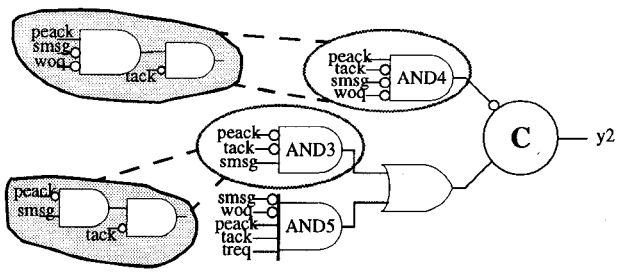
Given a proposed decomposition, for each input sequence where the output of the gate rises we must look at the signals within the input sequence to tell whether the decomposition is valid. If each signal that is connected to the intermediate AND gate makes a single monotonic transition, then the intermediate AND gate will in turn make a single monotonic transition and the theorem will be satisfied. However, if a signal feeding an intermediate AND gate changes non-monotonically, the gate can transition high and then low before settling at its final value, leading to a hazard.

In the decomposition of AND4 shown in the inset of Figure 3.5, for sequence 1 the output of the intermediate AND gate is high at the beginning of the sequence, and then transitions to low when *peack* goes low. Finally, it goes high again when *smsg* goes low. Therefore, this decomposition is invalid since a hazard on the intermediate AND gate is exercised. In this situation both *peack* and *smsg* changed non-monotonically, leading to the hazard. This suggests the following corollary:

Corollary 3.4.2 *Given the assumptions in Theorem 3.4, any decomposition which has all non-context signals connected to a single gate is hazard-free for the rising transition of the AND.*

The final situation we must examine is when there are a set of transitions on the inputs to the AND gate, but the AND gate remains low. In this situation, splitting up the gate incorrectly may lead to a static 0-hazard on the output of an intermediate gate.

Theorem 3.5 (AND gate remains stable) *A decomposition of an atomic AND gate into two cascaded AND gates is hazard-free with respect to the set of (input sequence; context) tuples where its output remains stationary if for each (input sequence; context)*



(Input sequence; context signal) tuples:

y2↓:

1. (peack↓, smsg↑, {treq↓ || peack↑}, {tack↓ || smsg↓}; woq')
2. (peack↓, woq↑, treq↓, tack↓, peack↑, woq↓; smsg')
3. (peack↑, smsg↓; woq', treq', tack')

y2↑:

4. (peack↓, woq↑, treq↑, tack↑, peack↑, woq↓; smsg')
5. (peack↓, smsg↑; woq', tack', treq')
6. ({peack↓ || treq↑}, tack↑, treq↓, tack↓, woq↑, treq↑, tack↑, peack↑, woq↓; smsg')

Figure 3.5: Decomposition for rising transition

tuple where the AND gate does not change, the intermediate AND gate does not change.

We can now combine these three conditions together to come up with conditions for valid decompositions for all input sequences.

Theorem 3.6 *Decomposition of an AND gate in accordance with the constraints imposed by Corollary 3.3.1 and Theorems 3.4 and 3.5 is hazard-free.*

Having established the key theorems for decomposition of AND gates, we can now analyze the set of (input sequence; context) tuples for a given output to determine whether the AND gate can be decomposed. Any conflict in the constraints imposed by the theorems implies that the gate cannot be decomposed in a simple manner.

We will now illustrate the use of these theorems in the example we used previously, shown in Figure 3.5. This example is taken from the *pe-rcv-ifc* circuit, a circuit that was part of a real design done at Hewlett-Packard [5]. Let us try to decompose the instance AND5, since it is the most difficult. For y2↓, AND5 will stay low during sequence 3, and will transition low for sequences 1 and 2. By Corollary 3.3.1, we know that {*peack*, *woq*, *treq*, *tack*} must be placed in the same gate (from sequence 2), and {*peack*, *smsg*, *treq*, *tack*} must be placed in the same gate (from sequence 1). These two partitions overlap, and therefore we must include the signals {*peack*, *woq*, *treq*, *tack*, *smsg*} in the same gate, which means we cannot split up that gate. We do not need to look at any other input sequences at this point.

Instance AND3 may be easier to decompose. Instance AND3 falls during sequence 3. From Corollary 3.3.1, only {*peack*, *smsg*} must be placed in the same gate. This leads to the only possible decomposition (modulo permutations) shown in the inset in Figure 3.5. Instance AND3 rises during input sequence 5. Since the changes on the inputs are monotonic during that sequence, any decomposition will work (by Theorem 3.4), so the decomposition shown also works for the rising transition. AND3 stays low during sequences 1, 2, 4 and 6. Unfortunately, if we look at sequence 1, we see that the intermediate AND gate will undergo a non-monotonic transition, so we cannot decompose AND3 either.

Finally, let us take a look at instance AND4. For this gate, when y2↑, AND4 falls. By Theorem 3.3, we see that, from sequence 4, {*peack*, *woq*, *tack*} must be in the same gate. From sequence 5, we must include {*peack*, *smsg*} in the same gate. So we know that the gate cannot be decomposed further. From this example we see that we cannot decompose any of the gates in a simple manner. This motivates us to look for ways in which we can eliminate some of the hazards introduced by the decompositions, which will be the topic of subsection 3.6.

3.5 Basic decomposition algorithm

Our basic decomposition algorithm is as follows:

```
decompose(sequences, logic):

foreach output {
  foreach block of combinational logic {
    foreach gate in block {
      if (gate == AND) {                          /* AND gate */
        (fallingSeqs, risingSeqs, stationarySeqs) =
                      partition(sequences,inputs(gate));
        falling = decompFalling(gate, fallingSeqs);
        rising = decompRising(gate, risingSeqs);
        stationary = decompNo(gate,stationarySeqs);

        decompositions = falling ∩ rising ∩ stationary;
      }
      else {                                      /* OR gate */
        decompositions = fundmodeDecomp (gate);
      }
    }
  }
}
```

The input sequences which are one of the inputs to the decomposition procedure are derived from the STG via a state graph. Although the number of distinct input sequences for any STG is finite, it can be quite large for an entire design. However, since the number of distinct input sequences for any given output logic block and gate is small, by extracting the sequences for the individual gates the procedure becomes quite efficient.

3.5.1 Results from simple decomposition procedure

We ran the above procedure on a number of examples from the asynchronous community, as shown in Table 1. Over half the AND3 gates and all but one of the AND4 gates in the designs were decomposable. Thus, using library of AND gates with three or fewer inputs, we were able to decompose over 95% of the designs. For about half the designs we were also able to decompose all of the AND3 gates. Although this doesn't affect the implementability of the designs, this has a positive impact on the quality of the final mapped circuit. By decomposing the gates into gates of finer granularity, we can find a better quality cover of the design during the matching/covering step.

The only design that we could not decompose for gates of size

four or more was *pe-rcv-ifc*. The next section examines the source of the failures, and presents a method for successful decomposition of these gates.

Design	# of gates	# of AND gates (≥ 3 inputs) can decompose	# of AND gates (≥ 3 inputs) cannot decompose
a_to_d_controller	6	2 AND3	0
atod	6	2 AND3	0
chu133	6	2 AND3	0
chu150	7	0	2 AND3
converta	13	0	2 AND3
dff	6	2 AND3	0
ebergen	9	0	2 AND3
half	6	0	1 AND3
hazard	6	2 AND3	0
mp-forward-pkt	10	2 AND4	0
nak-pa	10	0	1 AND3
nowick	11	2 AND3	0
pe-rcv-ifc	36	5 AND3, 2 AND4	4 AND3, 1 AND4, 2 AND5
qr42	9	0	2 AND3
ram-read-sbuf	11	2 AND3	0
rcv-setup	3	1 AND4	0
rpdft	4	1 AND3, 3 AND4	0
sbuf-ram-write	11	3 AND3	1 AND3
sbuf-send-pkt2	13	2 AND3	1 AND3
two_phase_fifo	7	0	2 AND3
vbe5b	6	0	1 AND3
vbe5c	4	0	1 AND3

TABLE 1. Results from simple decomp procedure

3.6 Extensions to enforce sequencing

As we saw from our previous results, there were several large gates we could not decompose. There are two situations we need to address, assuming we have avoided introducing any logic hazards. One is where the decomposition no longer preserves the ordering of the signals in the circuit's internal logic, and thus exercises a function hazard on an intermediate gate within the space of an input sequence. The second situation, which is of more interest, is where there is a hazard due to the sequential nature of the circuit; i.e., an unacknowledged transition. The second situation is the one we address.

Suppose that we have a transition that is not acknowledged by a gate in the output logic block. If we can ensure that it is acknowledged somewhere else in the circuit *before* the output change, then we can enforce the ordering requirement we had previously. Beerel refers to such connections as *acknowledgment wire-forks* [1], and proposed their use during decomposition in [2].

Recall that in our earlier example we were unable to decompose the two larger gates because of the requirements of Theorem 3.3 for the falling transition of an AND gate. In these cases, we encountered a sequential hazard which prevented us from decomposing the gate.

Our general decomposition procedure must change to accommodate addition of wire-forks. Recall that an output logic block has a two-input C-element driven by separate blocks of combinational logic. During any input sequence we have one block of combinational logic that is rising and one that is falling.

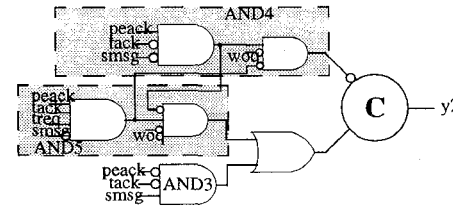


Figure 3.6: Decomposition using AND5 to acknowledge the decomposed AND4

Theorem 3.7 *Given a decomposed AND gate within the combinational portion of an output logic block that satisfies the conditions of Theorems 3.4 and 3.5, the falling transitions of the intermediate AND gate can be acknowledged by adding an inverted input to one or more AND gates in the complementary output logic block and the resulting circuit will be hazard-free.*

Now that we have shown the cases in which we can add a connection, we want to modify our decomposition procedure to take advantage of these situations in case we cannot decompose it by the simpler methods presented in the previous section.

3.6.1 Extended decomposition procedure

```
if (decompositions == NULL) {
  gates2do = undecomposable gates;
  foreach gate in gates2do
    if (gate ∉ library)
      extendedDecomp(gate,decomps,seqs);
  }
}
procedure extendedDecomp(gate, decomps, seqs) {
  gateDecomps = findAllFHFDecomps(gate);
  foreach intermedGate in gateDecomps {
    hazardousSeqs = findHazards (intermedGate,
                                          fallingSeqs);
    ackingGates = findRisingGates(hazardousSeqs);
    connect (intermedGate, ackingGates);
    /* Must check to see if we have exceeded fanin
            of rising gates by adding connection */
    foreach ackingGate in ackingGates {
      if (ackingGate ∉ library)
        gates2do += ackingGate;
    }
  }
}
```

With this extended decomposition procedure we can then decompose the AND5 gates in the design at the penalty of adding a few wire interconnections to the intermediate gate. However, since the fanout of the intermediate gates we added during the decomposition are small to begin with, this does not pose much of a problem.

Figure 3.6 shows how we used a decomposed AND4 and AND5 to mutually acknowledge falling transitions through addition of wire forks. (See [14] for more details.) The final decomposed *pe-rcv-ifc* design had 14 AND3s and two AND4s and many AND2s.

The algorithms we described were implemented in approximately 5000 lines of C code, using Tcl and Tk as the user-interface [12]. All examples ran in a few seconds on an HP 9000/730 workstation.

4. Conclusions

We have characterized the problem of decomposing gates in a speed-independent design in the context of library binding for speed-independent designs. Given a technology-independent netlist for a speed-independent design, environmental information, and a logic-hazard-free library, we can decompose the gates into smaller gates. The decomposition is done both for feasibility and to expand the solution space during the matching/covering step of technology mapping. Building on the basic decomposition algorithm, we also showed how to introduce extra circuit connections to remove hazards in a decomposed design.

We implemented the decomposition algorithms as part of a new technology mapper. We ran the routines on asynchronous benchmarks and showed that we could decompose most of the gates with the simple algorithm, but for larger gates we had to rely on the extended algorithm for a successful decomposition.

In this paper we have only addressed decomposition algorithms for library binding. New matching/covering algorithms can be used to take advantage of sharing across logic blocks, given that the granularity of the decomposed design is coarser than with synchronous or burst-mode designs.

5. Acknowledgments

We are indebted to Peter Beerel and Prof. David Dill for insightful discussions and helpful comments on this paper. Prof. Luciano Lavagno also gave us valuable criticisms which we incorporated into our work. This work was supported under SRC contract 93-DJ-205 and under ARPA/NSF contract MIP 9115432.

6. References

[1] P. Beerel and T. H.-Y. Meng, "Automatic gate-level synthesis of speed-independent circuits," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 581–586, Nov. 1992.

[2] P. Beerel and T. H.-Y. Meng, "Logic transformations and observability don't cares in speed-independent circuits," in *TAU*, Aug. 1993.

[3] T.-A. Chu, "Synthesis of self-timed VLSI circuits from graph-theoretic specifications," Technical Report MIT-LCS-TR-393, MIT, 1987.

[4] W. S. Coates, A. L. Davis, and K. S. Stevens, "Automatic synthesis of fast compact self-timed control circuits," in *IFIP Workshop on Asynchronous Circuits*, Manchester, UK, 1993.

[5] W. S. Coates, A. L. Davis, and K. S. Stevens, "The post office experience: Designing a large asynchronous chip," *INTEGRATION, the VLSI journal*, 15(4):341–366, 1993.

[6] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *24th Design Automation Conference*, pages 341–347, IEEE/ACM, 1987.

[7] L. Lavagno and A. Sangiovanni-Vincentelli, *Algorithms for Synthesis and Testing of Asynchronous Circuits*, Kluwer Academic Publishers, 1993.

[8] F. Mailhot and G. De Micheli, "Algorithms for technology mapping based on binary decision diagrams and on boolean operations," *IEEE Transactions on CAD/ICAS*, pages 599–620, May 1993.

[9] E. J. McCluskey, *Logic Design Principles With Emphasis on Testable Semicustom Circuits*, Prentice-Hall, 1986.

[10] T. H. Meng, *Synchronization Design for Digital Systems*, Kluwer Academic, 1990.

[11] S. M. Nowick and D. L. Dill, "Synthesis of asynchronous state machines using a local clock," in *ICCD, Proceedings of the International Conference on Computer Design*, pages 192–197, IEEE Computer Society Press, 1991.

[12] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1993.

[13] R. Rudell, *Logic Synthesis for VLSI Design*, PhD thesis, U. C. Berkeley, Apr. 1989, Memorandum UCB/ERL M89/49.

[14] P. Siegel, *Technology Mapping for Asynchronous Designs*, PhD thesis, Stanford University, 1994.

[15] P. Siegel, G. De Micheli, and D. Dill, "Automatic technology mapping for generalized fundamental-mode asynchronous designs," in *DAC, Proceedings of the Design Automation Conference*, pages 61–67, June 1993.

[16] S. H. Unger, *Asynchronous Sequential Switching Circuits*, New York: Wiley-Interscience, 1969.

[17] V. I. Varshavsky, editor, *Self-Timed Control of Concurrent Processes*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[18] K. Yun and D. Dill, "Automatic synthesis of 3D asynchronous finite-state machines," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 576–580, Nov. 1992.