# Provably Correct High-Level Timing Analysis without Path Sensitization

Subhrajit Bhattacharya [*]
Dept. of Computer Science
Duke University
Durham, NC 27706

Sujit Dey
C&C Research Labs
NEC USA
Princeton, NJ 08540

Franc Brglez [†]
CBL, Dept. of ECE
North Carolina State Univ
Raleigh, NC 27695

**Abstract -** *This paper addresses the problem of true delay estimation during high level design. The existing delay estimation techniques either estimate the topological delay of the circuit which may be pessimistic, or use gate-level timing analysis for calculating the true delay, which may be prohibitively expensive.*

*We show that the paths in the implementation of a behavioral specification can be partitioned into two sets, SP and UP. While the paths in SP can affect the delay of the circuit, the paths in UP cannot. Consequently, the true delay of the resulting circuit can be computed by just measuring the topological delay of the paths in SP, eliminating the need for the computationally intensive process of path sensitization. Experimental results show that high-level true delay estimation can be done very fast, even when gate-level true delay estimation becomes computationally infeasible. The high-level delay estimates are verified by comparing with delay estimates obtained by gate-level timing analysis on the actual implementation.*

## I. Introduction

In the process of designing a circuit from a behavioral specification, parameters like area, delay, or power consumption of the final implementation influence many of the high-level trade-off decisions. Exact values of these parameters are attainable only by way of a time-consuming implementation of the design. Hence the need for fast and accurate estimation techniques. This paper addresses the problem of *true (functional)* delay estimation during high level design.

The topological delay of the longest path in the circuit, while simple to estimate, can be overly pessimistic, since many long paths may not be sensitizable [1, 2]. Several gate-level timing analysis techniques have investigated the exact conditions under which a path can affect the clock period of the circuit [1, 2, 3, 4, 5]. Using static sensitization condition, the delay of the circuit is the delay of the longest statically sensitizable path in the circuit. However, timing analysis using static sensitization may produce inaccurate results. A statically unsensitizable path may be dynamically sensitizable, and determine the delay of the circuit, when delays of circuit elements are considered [2]. Dynamically sensitizable paths are also referred to as simply sensitizable paths, or true paths. The true delay of the circuit is the delay of the longest true path.

Gate-level timing analysis techniques which consider path sensitization for circuit delay estimation have been presented in [1, 2, 3, 4, 5]. However, since they have to check every path for sensitization, the gate-level timing analysis techniques can be computationally very expensive, and are not feasible for circuits having arithmetic functions. Specifically, the gate-level techniques are too slow for the purposes of most high level design tools.

Delay models and estimation techniques for behavioral synthesis have been presented in [6, 7, 8, 9]. These techniques calculate estimates of the topological delay of the circuit at the layout or RT-level. A topological and path sensitization based estimation technique for behavioral synthesis has been proposed in [10]. However, since the method relies on true path delay analysis at the gate-level, it can estimate the true delay of only the control part of the design. In [9], high level functional information has been used for a refined timing analysis at a high-level instead of at the gate-level. Functional false paths, which arise due to unused functionalities of chained multi-function ALU units, are avoided while computing the delay. However, false paths due to resource sharing and the effect of control signals on the datapath delay have been ignored [9].

In this paper, we address the problem of *true* delay estimation during high level synthesis of behavioral specifications. A circuit implementation can have false paths due to presence of false paths in the behavioral specification [11]. However, we show that even when the specification does not have false paths, resource sharing can introduce false paths in the implementation. This has been earlier observed in [12]. We provide a comprehensive analysis of the sources of unsensitizable paths during resource sharing and assignment. We show that the paths in the implementation can be partitioned into two sets, SP and UP. The set SP consists of statically sensitizable paths. While the paths in UP may or may not be sensitizable, we prove that for every dynamically sensitizable path in UP, there is a longer path in SP. Hence, paths in UP can never determine the clock period of the circuit. To compute the true delay of the resulting circuit, we simply measure the topological delay of all paths in SP. The significance of the proposed technique is that it eliminates the need for checking whether long paths in the circuit can be sensitized, a process which makes gate-level timing analysis computationally expensive and impractical for data-path intensive circuits.

In Section II, we review the concept of path sensitization. Section III presents a comprehensive analysis of the sources of false paths due to resource sharing. Section IV discusses how to identify the set SP of paths which alone determine the delay of a circuit. Section V proves that paths in UP can never affect the delay of a circuit. An algorithm for delay estimation is given in Section VI. Experimental results in Section VII demonstrate the validity and efficiency of the proposed high level true delay estimation technique.

[*] S. Bhattacharya was supported by a grant from CCRL, NEC USA.
[†] F. Brglez was supported in part by a grant from SRC.

© 1994 ACM 0-89791-690-5/94/0011/0736 $3.50

## II. Basic Concepts in Timing Analysis

In this section we define certain properties of acyclic combinational circuits. Much of the notation has been taken from [1]. The delay of a gate *G* and lead *f* are denoted by *d(G)* and *d(f)*. Let $P = (f_0, G_1, f_1, \ldots, G_{m-1}, f_{m-1})$ be a path in the combinational circuit, where $f_i$ is a lead and $G_j$ is a gate. The leads $f_0$ and $f_{m-1}$ are a primary input and output respectively. All inputs to $G_j$ other than $f_{j-1}$ are called *side-inputs* of gate $G_j$.

A logic value is the *controlling* value of a gate if the logic value at an input of the gate determines the gate output independently of the other inputs. Otherwise the logic value is called a *non-controlling* value. The controlling and non-controlling values for a gate *G* are called *c(G)* and *n(G)* respectively. For example, if *G* is an AND gate, $c(G) = 0$ and $n(G) = 1$.

**Definition 1** *A path P is called* statically sensitizable*, if for some input vector* v*, none of the side inputs to a gate* G *on* P *has a controlling value for* G.

On applying a primary input vector $v$ at time $t = 0$, eventually the logic value at every node in the circuit will stabilize. The stable logic values under $v$ at any lead $f$ and output of gate $G$ is denoted as $sv(v, f)$ and $sv(v, G)$ respectively. The times when these values become stable are denoted as $st(v, f)$ and $st(v, G)$ respectively.

We say that $f_i$ *dominates* $G_{i+1}$ if any one of the following conditions are true. (1) The only controlling input to $G_{i+1}$ is $f_i$. (2) There are more than one controlling inputs to $G_{i+1}$, but $f_i$ arrives before the other controlling inputs. In other words, $st(v, f_i)$ has the smallest value of all the controlling inputs. (3) Every input to $G_{i+1}$ is non-controlling. However, $f_i$ is the last input to stabilize and $st(v, f_i)$ is the maximum for all the inputs to $G_{i+1}$. Given the above definition of dominating inputs, the stable time of the output of a gate $G_i$ with dominating input $f_{i-1}$ is given by $st(v, G_i) = st(v, f_{i-1}) + d(G_i)$. If the output of $G_i$ is connected to $f_{i+1}$, then $st(v, f_{i+1}) = st(v, G_i) + d(f_{i+1})$.

**Definition 2** *A path* P *is defined to be* dynamically sensitizable *or true if there is at least one input vector* v *under which every lead $f_i$ on P dominates $G_{i+1}$, $0 \leq i \leq (m-2)$. A dynamically unsensitizable path is a false path.*

**Lemma 1** *Every statically sensitizable path is dynamically sensitizable. A statically unsensitizable path may be dynamically sensitizable.*

Only dynamically sensitizable paths determine the output of a circuit. Since the longest path in a circuit may not be dynamically sensitizable, the topological delay is a pessimistic estimate of the delay of a circuit. The delay of the longest statically sensitizable path is optimistic, since there may be a longer statically unsensitizable path which is dynamically sensitizable.

**Definition 3** *The correct clock period, CP, of a circuit is greater than or equal to the delay of the longest* dynamically sensitizable *path.*

We illustrate the above ideas for the circuit in Figure 1. Let us assume that each lead has zero delay and each gate has unit delay. Let input vector $v = (I_1, I_2) = (1, 0)$. It can be easily shown that $(a, b, c, d, e, f, g, o)$ has stable logic values $(0, 1, 1, 1, 1, 1, 1, 1)$ under vector $v$. Since every lead on $P = (b, G_2, e, G_3, f, G_4, o)$ dominates the successor gate in P, P is dynamically sensitizable. However

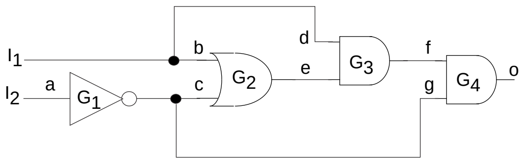

Figure 1: Circuit for illustrating path sensitization.

it can be shown that P is *statically unsensitizable*. The longest statically sensitizable path in the circuit is $(d, G_3, f, G_4, o)$ and has delay 2. The longest path $(a, G_1, c, G_2, e, G_3, f, G_4, o)$ has delay 4 but can not be dynamically sensitized. Hence, even though P is statically unsensitizable, since it is the longest dynamically sensitizable path, *CP* for the circuit is greater than or equal to the delay of P which is 3.

## III. Sources of Statically Unsensitizable Paths

In [13] we identify the sources of long paths due to resource sharing in circuits implementing behavioral descriptions. In this section, we provide a comprehensive analysis of sources of statically unsensitizable paths due to resource sharing. A pair of operations can share the same resource if, (a) they are never executed in the same clock cycle and, (b) it is possible to assign them to the same resource. Two operations are never executed in the same clock cycle if (a) they are scheduled in separate states, or (b) they are in the same state but on mutually exclusive paths.

**Sharing Mutually Exclusive Operations.** Consider the CFG shown in Figure 2(a) which has been extracted from the behavioral description of the *dealer* process of the Blackjack benchmark. The schedule and assignment is given in 2(b). An abstraction of the circuit implementation which is used for delay estimation, the DelayG, is given in Figure 2(e). The actual circuit implementation can be found in [14]. There is a path in the circuit from the (+) unit to which operation 2 has been assigned to the (<) unit to which operation 3 has been assigned, since operation 3 uses the output of operation 2. Operations 6 and 7 are assigned to the$(+/-)$ unit. The (<) unit to which operation 5 has been assigned decides whether operation 6 or 7 should be executed by the ALU. Hence there is a path from the (<) unit to the ALU in the circuit. Since operations 3 and 5 in the CFG are both in state $s_0$ and are assigned to the (<) unit we get a path $((+), (>), (+))$ in Figure 2(e). Assuming the variable *Incr* of operation 2 is stored in the register $Incr$ and the variable Card of operation 6 in $Card$, there is a register to register path $(Incr, (+), (>), (+/-), Card)$ in the circuit implementation of Figure 2(e). However, since operation 2 assigned to the $(+)$ unit is never executed in the same clock cycle as either operation 6 or 7 assigned to the $(+/-)$ unit, the above path from $Incr$ to $Card$ is never completely executed in the same clock cycle and is statically unsensitizable.

**Explicit Sharing across States.** Consider the CFG shown in Figure 2(a). Operations 2 and 8 are in different states. Since they will never be executed in the same clock cycle, they are assigned to the (+) unit in Figure 2(e). Since variable *Incr* is an input to operation 2 and variable *Avalue* is an output of operation 8, a path $(Incr, (+), Avalue)$ is created in the circuit of Figure 2(e). However, since *Incr* is an input to the $(+)$ unit only in state $s_0$ and the $(+)$ unit output goes to *Avalue* only in state $s_1$, the above path is false. It arises due to operations 2 and 8 which are in different states sharing the same resource.

**Implicit Sharing across States.** Figure 3 shows a portion of the CFG, schedule, assignment and a circuit implementation for the Unmanned Aerial Vehicle (UAV) controller [15]. Note that operation 6 is sched-

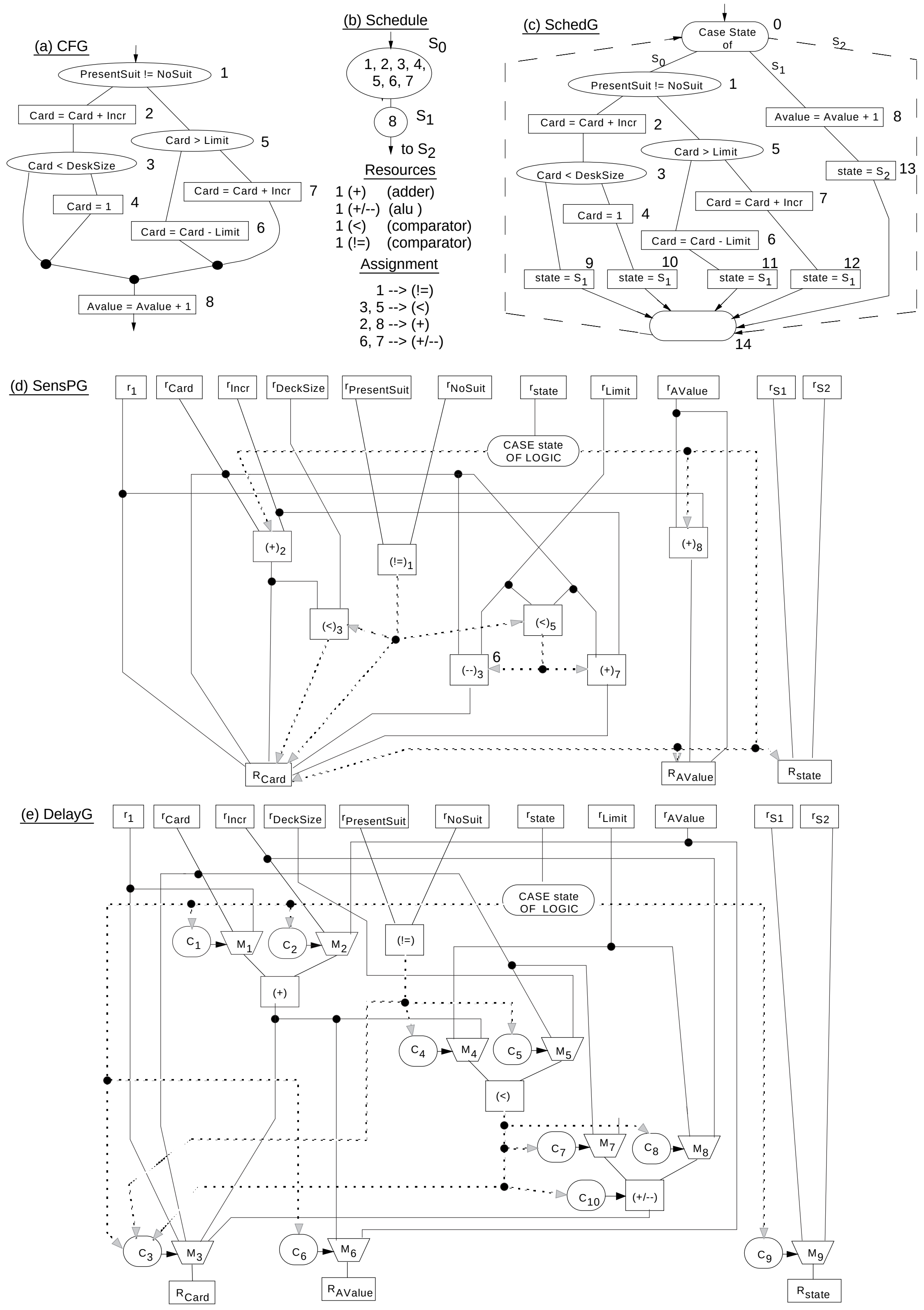

Figure 2: The dealer example. Its (a) control flow graph *CFG*, (b) resource allocation, schedule and assignment (c) schedule graph *SchedG*, (d) sensitizable path graph *SensPG*, and (e) delay graph *DelayG*.

uled in both state $s_0$ and $s_2$. Assigning both occurences of operation 6 to the same resource unit, ALU2, is termed *implicit sharing across states* [13]. In the corresponding implementation shown in Figure 3(c), a path (*MaxVal*, $ALU1$, $ALU2$, $cmp2$, *RTI*) is created. The part (*MaxVal*, $ALU1$, $ALU2$) is exercised in state $s_0$ and corresponds to execution of operations 2 and 6 in $s_0$. In state $s_2$ the part ($ALU2$, $cmp2$, *RTI*) is exercised and corresponds to execution of operations 6 and 7 in state $s_2$. Since these two parts are never exercised simultaneously, the complete path is false. This false path was a result of the implicit sharing of operation 6 across states.

## IV. Identification of Sensitizable Paths

In this section we develop an efficient technique to identify the subset of paths in a circuit which are sensitizable without explicitly sensitizing the paths as done at the gate level. The paths we consider include both the control and data path of the circuit. After identifying the set of sensitizable paths, a simple topological delay analysis on the paths in the set gives the true delay of the circuit. We define various graph structures that will be required for defining the set of statically sensitizable paths, SP, in a circuit implementation of a behavioral description. The schedule graph (*SchedG*) is a representation of the schedule suitable for defining the sensitizable paths graph (*SensPG*). Every path in *SensPG* belongs to the set SP. We define a third graph structure, the delay graph or *DelayG*, which is an abstraction of a RT-level implementation of the circuit and allows us to do fast delay estimation without actually deriving the RT-level implementation. Instead of specifying the rules for the derivation of each of these graphs, we illustrate how to construct them by examples.

### A. SchedG - The Schedule Graph

Scheduling a CFG consists of clustering the operations of the CFG into states. A schedule can be represented in more than one way. A traditional representation of a schedule is given in Figure 2(b). We define an alternative representation shown in Figure 2(c) called the schedule graph (*SchedG*), which is suitable for our purposes. Executing the schedule is equivalent to executing the CFG. We assume that each state of a schedule requires one clock cycle for execution.

Consider execution of the path $(1, 5, 7, 8)$ in the CFG of Figure 2(a). The schedule executes this path in two clock cycles. Assuming that the initial state of the schedule is $s_0$, the schedule executes the path $(0,1,5,7,12, 14)$ of the *SchedG* shown in Figure 2(c) in the first clock cycle . Since operation 12 assigns $s_1$ to the state variable *state*, the operations $(0, 8, 13, 14)$ are executed in the next clock cycle. This is equivalent to executing operations $(1, 5, 7)$ in the first clock cycle and operation $(8)$ in the second. It should be noted from Figure 2(c) that although there are paths in state $s_0$ other than $(0,1,5,7,12, 14)$, only one path is executed in any clock cycle depending upon the value of the conditional operations in nodes $1, 3$ and $5$.

**Lemma 2** *In any clock cycle, only one path in the* SchedG *is executed. In a circuit implementation of the scheduled CFG, only the operations on the path which is executed determine the new value of the* SchedG *variables.*

It should be noted that if there is no dependency between two operations on the same path in a *SchedG*, then they may be executed in parallel in the same clock cycle. For example, in state $s_0$ of Figure 3(b), operation 5 does not depend on operation 3, and are executed in parallel whenever state $s_0$ is executed.

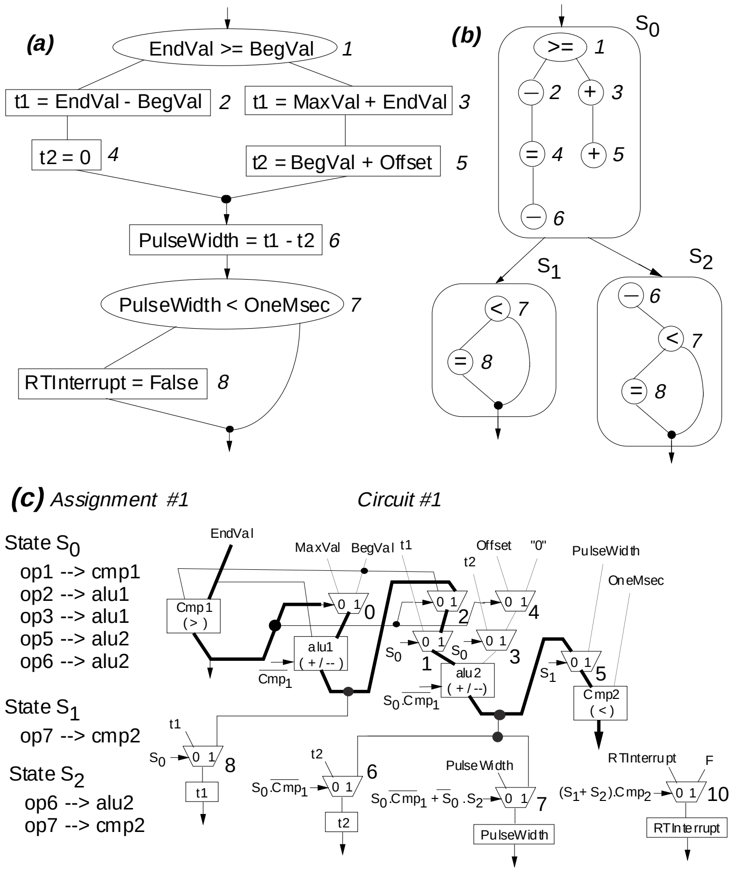

Figure 3: The UAV example for illustrating statically unsensitizable paths due to implicit sharing across states.

### B. SensPG - The Sensitizable Paths Graph.

The sensitizable paths graph (*SensPG*) represents the dependencies between the variables and the operations in the *SchedG*. The dependencies give rise to paths in the *SensPG*. Corresponding to each path in the *SensPG*, there exists one or more paths in the circuit implementation. As we show later, these are the only paths in the implementation whose delay has to be determined for clock period calculation. The *SensPG* can be constructed from the *SchedG* and the assignment. We explain how to construct the *SensPG* in Figure 2(d) from the *SchedG* and assignment of Figure 2.

We explain creation of the nodes of the *SensPG* first. Consider the nodes on any path *P* in the *SchedG*.
(1) If the variable *y* is being used in the node $n_1$ on *P*, create a node $r_y$ in the *SensPG*. Consider path $P = (0, 1, 2, 3, 9, 14)$ in the *SchedG* and let $n_1$ be operation 1. Nodes $r_{PresentSuit}$ and $r_{NoSuit}$ are created in the *SensPG* for the variables *PresentSuit* and *NoSuit*.
(2) For every variable $v$ that is assigned to in a node on P, create a node $R_v$ in the *SensPG*. For example, variable *Card* is assigned to in node 2 in *P* and hence there is a node $R_{Card}$ in the *SensPG*.
(3) For every operation $op$ in node $n_1$ on *P*, create an operation node $(op)_{n_1}$ in the *SensPG*. Node 3 in *P* has the operation $(<)$ for which there is a corresponding node $(<)_3$ in the *SensPG*.

There are two principal types of arcs in the *SensPG*. The data dependency arcs are shown as solid lines, the control dependency arcs as dashed lines. We first explain creation of the data dependency arcs. Data dependency arcs arise to data dependencies between operations

in the *SchedG*.
(4) Consider the $(+)$ operation in node 2 of the *SchedG*. It has inputs *Card* and *Incr*. Hence, in the *SensPG*, there are data dependency arcs from $r_{Card}$ to $(+)_2$ and from $r_{Incr}$ to $(+)_2$. Since *Card* is being assigned to in operation 2 and on $P$ there is no further assignment to *Card*, there is a data dependency arc from $(+)_2$ to $R_{Card}$.
(5) Consider the $(<)$ operation in node 3 of the *SchedG*. It has operands *Card* and *DeckSize*. Since *Card* was assigned to in operation 2 which precedes operation 3 in *P*, there is a data dependency arc from $(+)_2$ to $(<)_3$. There is another arc from $r_{DeckSize}$ to $(<)_3$.

Control dependency arcs in the *SensPG* can arise due to assigning multiple operations to the same resource unit. They can also arise due to assigning a variable from multiple sources. In an actual implementation, control dependency arcs create a path from comparators or control logic to muxes at the inputs of functional units or registers. To create these arcs, we need to know the assignment.
(6) Operation 3 and 5 have been assigned to the same comparator, a $(<)$ unit. Since the two operations are on mutually exclusive branches of operation 1, operation 1 decides which operation, 3 or 5, should be executed. In the *SensPG*, the nodes corresponding to operations 1,3 and 5 are $(!=)_1$, $(<)_3$ and $(>)_5$. Hence in the *SensPG* there is a control dependency arc from $(!=)_1$ to both $(<)_3$ and $(>)_5$.
(7) The variable *Card* is being assigned from various sources, as can be seen from nodes 2, 4, 6 and 7 in the *SchedG*. Along the path (0, 8, 13, 14) in the *SchedG*, *Card* is not assigned and remains unchanged. Given the assignment, it can be shown that the conditionals in nodes 0, 1 and 2 of the *SchedG* decide the source from which *Card* is assigned. Hence in the *SensPG*, there are arcs from (CASE_state_of) node, $(!=)_1$ node and $(<)_3$ node to $R_{Card}$. Note that there is no control dependency arc from $(>)_5$ to $R_{Card}$ even though *Card* is assigned to on the mutually exclusive branches of operation 5. The reason is, both operation 6 and 7 are assigned to the same ALU. Hence the source of the input is same for *Card* irrespective of the result of the conditional operation 5.
**Paths in the SensPG corresponding to a path in the SchedG.** Given a path *P* in the *SchedG*, there exists one or more corresponding paths in the *SensPG*. Consider the path $P = (0, 1, 2, 3, 9)$ in the *SchedG*. One of the paths in the *SensPG* corresponding to P is, ($r_{State}$,$CASE\_state\_of$,$(+)_2$,$(<)_3$,$R_{Card}$). Since the CASE operation in node 0 of the *SchedG* decides whether operation $(+)_2$ should be executed, this creates the subpath ($r_{State}$,$CASE\_state\_of$,$(+)_2$). Since the operation in node 3 uses the result of the operation in node 2 of the *SchedG*, there is an arc from $(+)_2$ to $(<)_3$ in the *SensPG*. Also, if the result of the '$<$' comparison in node 3 is true, only then $Card$ is assigned the output of the '$+$' operation in node 2. Hence there is a control arc from $(<)_3$ to $R_{Card}$ in the *SensPG*.

Let $P$ be any path in the *SchedG*. The set of paths corresponding to $P$ in the *SensPG* are denoted by:
SensPGmap(P) $= \{p \mid$ every node on $p$ corresponds to a node on P $\}$.

**Lemma 3** *In a circuit implementation of the scheduled CFG, in any clock cycle, let P be the path in the* SchedG *which is executed. Then the operations on the paths SensPGmap(P) in the* SensPG *decide the new value of the* SchedG *variables.*

## *C. DelayG - The Delay Graph.*

The delay graph *DelayG* is derived from the *SensPG* and the assignment. It closely resembles a circuit implementation of the behavioral description while hiding the actual implementation details. For example, the *DelayG* shown in Figure 2(e) corresponds to the assignment in Figure 2(b) and the *SensPG* in Figure 2(d). We prove some key results based on our *DelayG*, thus making the *results independent of the underlying implementation details.* Also, since the *DelayG* is used for the timing estimation, depending upon requirements, we can make the estimation as accurate as we like by controlling the details of the implementation. In the following discussion, we refer to the *DelayG* shown in Figure 2(e).

Every node in the *SensPG* is mapped to a corresponding node in the *DelayG*. Every node of type $r_x$, $R_y$ and $CASE$ in the *SensPG* has an identical node in the *DelayG*. The assignment decides the mapping of the operation nodes in the *SensPG* to the resource unit nodes in the *DelayG*. For example, $(<)_3$ and $(<)_5$ in the *SensPG* are operation nodes that correspond to operations 3 and 5 in the *SchedG*. From the assignment, we see that operations 3 and 5 are both assigned to the $(<)$ resource unit. Hence, both these nodes are mapped to the $(<)$ node in the *DelayG*.

Every resource node and register node of type $R_x$ in the *DelayG* has a *Mux* node at each data input. *Mux* nodes have multiple data inputs, but only a single output. For example, the $(<)$ node has $M_4$ and $M_5$ at its two data inputs. For every *Mux* node, there is a *control* node, whose output goes to the corresponding *Mux* node. Control nodes $C_4$ and $C_5$ correspond to $M_4$ and $M_5$ in our example.

At any time, the *Mux* node allows only one of its data input to be connected to its data output. The decision as to which of its data inputs should be connected to the output is made by the outputs of the control node. Consider nodes $C_3$ and $M_3$ corresponding to $R_{Card}$. Node $M_3$ has a data dependency input from the $(+)$ node. This corresponds to the data dependency arc from $(+)_2$ to $R_{Card}$ in the *SensPG*. It can be seen from the *SchedG* that this assignment was made in state $s_0$ if the conditional $(PresentSuit != NoSuit)$ evaluated to *true*. The node $C_4$ has control dependency arcs from the nodes $(CASE\_state)$ and $(!=)$, and when the former takes on value $s_0$ and the latter takes on the value *true*, the input from $(+)$ to $M_4$ is connected to the output of $M_4$.
**Paths in the DelayG corresponding to a path in the SensPG.** Let $P$ be a path in the *SensPG*. The set of paths corresponding to $P$ in the *DelayG* are denoted by DelayGmap(P). Consider the path $P_1 = (r_{Card}, (+)_2, (<)_3, R_{Card})$ in the *SensPG*. The corresponding path in the *DelayG* is $p_1 = (r_{Card}, M_1, (+), M_4, (<), C_3, M_3, R_{Card})$. Similarly the path $P_2 = (r_{Avalue},\ (+)_8,\ R_{Avalue})$ in the *SensPG* has a corresponding path $p_2 = (r_{Avalue},\ M_2,\ (+),\ M_6,\ R_{Avalue})$ in the *DelayG*. However, there are paths in the *DelayG* which arise due to resource sharing and do not have any corresponding path in the *SensPG*. Operation $(+)_2$ on $P_1$ and operation $(+)_8$ on $P_2$ share an adder for which the corresponding node in the *DelayG* is $(+)$. This creates a path $p_3 = (r_{Avalue}, M_2, (+), M_4, (<), M_3, R_{Card})$ where $(r_{Avalue}, M_2, (+)$ belongs to $p_1$ and $(+), M_4, (<), M_3, R_{Card})$ belongs to $p_2$. No path exists in the *SensPG* which corresponds to $p_3$.

**Definition 4** $SP = \{path\ p\ in\ DG \mid p \in DelayGmap(P)$ *and P is a path in the* SensPG $\}$. $UP = \{$ *path p in* DelayG $\} - SP$.

The set SP consists of statically sensitizable paths in the *DelayG* while the set UP consists of the *statically unsensitizable* paths. The latter set arises due to sharing of the same resource between multiple *SensPG* operations.

**Lemma 4** *In a circuit implementation of the scheduled CFG, in any clock cycle, let P be the path in the* SchedG *which is executed. The* *subset of the paths in SP which correspond to P are the only paths whose operations decide the new value of the* SchedG *variables.*

**Corollary 1** *A path in UP never decides the new value of* SchedG *variables.*

## V. Paths in UP Can Not Affect Circuit Delay

We prove that paths in UP do not affect the delay of the circuit even if they are dynamically sensitizable. Let $p = (f_0, G_1, f_1, \ldots, G_{m-1}, f_{m-1})$ be a path in the combinational circuit, where $f_i$ is a lead and $G_j$ is a gate. The leads $f_0$ and $f_{m-1}$ are a primary input and output respectively. Given an input vector $v$, let $P_v = \{p_j\}$, where path $p_j$ has the property that if an $f_i$ on $p_j$ has a non-controlling value for $G_{i+1}$, then every side input to $G_{i+1}$ is on some path $p_k \in P_v$ and has a non-controlling value. Such a set of paths is said to determine the output completely for the vector $v$.

**Theorem 1** *Let $P_v = \{p_i\}$ completely determine the output of a circuit for the input vector $v$. If there exists a path $p_j \notin P_v$ and $p_j$ gets sensitized for the input $v$, there exists a path $p_i \in P_v$ such that delay($p_i$) $\geq$ delay($p_j$).*

**Proof** There must be a gate G at which $p_j$ meets a path $p_i$ in $P_v$ such that after $G$, all the leads and wires on both $p_i$ and $p_j$ are the same. From the property of paths in $P_v$, the lead $f_i$ on $p_i$ which is an input to $G$ must have a controlling value, since $p_j \notin P_v$. For $p_j$ to be sensitized, the lead on $f_j$ on $p_j$ which is an input to $G$ needs to have a controlling value and also, st(v, $f_j$) $\leq$ st(v, $f_i$). However, this immediately implies that delay($p_j$) $\leq$ delay($p_i$). $\Box$

**Theorem 2** *Given any path $p_1$ in UP that can be sensitized, there exists a path $p_2$ in SP such that delay($p_2$) $\geq$ delay($p_1$).*

**Proof** In any clock period, all the operations on one of the paths in the *SchedG* determine the new value of all the circuit variables (Lemma 2). If the path in the *SchedG* is called $P$, then the paths SensPGmap(P) in the *SensPG* determine the new value of all the circuit variables (Lemma 3). Similarly, the paths in the *DelayG* which correspond to paths in SensPGmap(p) determine the value of all the inputs to the register nodes storing the value of the variables (Lemma 4). The above paths in the *DelayG* by definition are in SP. Hence, in any clock period, a subset of paths in SP completely determine all the circuit outputs. Hence for a given input, if the sensitized path belongs to UP, since $SP \cap UP = \emptyset$, from Theorem 1, there must be a path $p_2 \in SP$, such that delay($p_2$) $\geq$ delay($p_1$). $\Box$

Paths in UP are created due to sharing of resources as explained in Section IV-C. From Theorem 2, paths in UP can never determine the clock period of the circuit. Hence, to compute the true delay of the resulting circuit, we just have to measure the topological delay of the SP paths without considering path sensitization.

## VI. Algorithm for Clock Period Estimation

The clock period estimation algorithm, FEST, finds an estimate of the correct clock period of the circuit implementing a behavioral description. The inputs to the algorithm are the schedule of the behavior in the form of a schedule graph, *SchedG*, an assignment of the operations in the *SchedG* to resource units, and a component library such as shown in Table 1.

An outline of the algorithm FEST is given below. The algorithm first creates the *SensPG* and the *DelayG* as outlined in Section IV. It next creates SP, the set of paths in the *DelayG* which have corresponding paths in the *SensPG*. The algorithm takes every path which is in SP and finds an estimate of the path delay using the function *topo_delay_est*. The maximum of the delay of the paths in the SP is an estimate of the minimum clock period of a circuit implementation of the *DelayG*.

An SP path in a *DelayG* starts at a node of the type $r_x$ and ends at a node $R_y$ and consists of intermediate nodes. The intermediate nodes can be mux nodes, control nodes or resource unit nodes. To estimate the delay of a SP path, the function *topo_delay_est* uses knowledge of the implementation of the nodes in the *DelayG*.

We assume that an n-input *Mux* node is implemented as a balanced tree of (n-1) 2-input *Muxes*. Similarly a block of control logic which controls multiplexor inputs is implemented as a balanced tree of simple gates. For every mux or control node that is on a SP path, given the underlying implementation for the mux and control nodes, the delay of the node is estimated and added to the delay of the path.

The delays of individual resource units as well as the delay of cascades of resource units are precomputed and stored in a table, as in Table 1, for fast lookup. The delay of a cascade of resource units may be less than the sum of the delay of the individual units [7]. For example, from Table 1 we find that the delays of an 8 bit alu unit is 14.33ns. However, the delay of a cascade of two alu units is 19.58ns and not 28.66ns as might be expected. If the path has a cascade of resource units, the delay of the cascade is used rather than the delay of individual components. Details of the delay estimation process can be found in [14].

Table 1: Individual and cascaded library modules in SCMOS 2.0.

| **Unit** | **4 bit Delay [ns]** | **8 bit Delay [ns]** |
|---|---|---|
| Comp(=) | 3.59 | 5.45 |
| Comp(<) | 6.35 | 10.87 |
| adder | 9.62 | 20.69 |
| alu(+, −) | 9.44 | 14.33 |
| mux | 2.95 | 4.19 |
| $(+,-) \rightarrow (+,-)$ | 13.76 | 19.58 |
| $(+,-) \rightarrow (<)$ | 14.96 | 24.48 |
| $(<) \rightarrow (+/-)$ | 14.42 | 24.66 |
| $(=) \rightarrow (<)$ | 8.13 | 13.38 |

```
FEST(SchedG, assignment, library) {
1.  SensPG ← create_senspg(SchedG, assignment);
2.  DelayG ← create_delayg(SensPG, assignment);
3.  SP ← ∅;
4.  for (every path $p_i$ in SensPG) do
5.          SP ← SP ∪ DelayGmap($p_i$);
6.  max_delay ← 0;
7.  for (every path $p_j$ in SP) do
8.          max_delay ← max(max_delay, topo_delay_est($p_j$, DelayG));
}
```

The partitioning of the *DelayG* paths into SP and UP are implementation independent and hence the result in Theorem 2 is implementation independent too. However, the estimated delay value as returned by function *topo_delay_est* will not only depend upon the implementation of the *DelayG* but also on the accuracy of the delay modelling of the implementation.

## VII. Experimental Results

We have implemented the high-level timing analysis tool FEST in C. To evaluate the effectiveness of FEST, we synthesize the following conditional-intensive VHDL descriptions: the dealer process of Blackjack and the controller for the AutoPilot of an Unmanned Aerial Vehicle (UAV) [15]. Each description is scheduled to satisfy the resource constraints specified in Figure 2 for the dealer process and in Figure 3 for the UAV example. The relevant portions of the CFG for the Blackjack process is shown in Figure 2, and the mapping of the CFG operations under the assignment is shown in Figure 2(e). The relevant portions of the CFG and the mapping for the UAV AutoPilot process is shown in Figure 3.

A netlist is generated for each RT-level circuit using OASIS [16]. The generated netlists are subjected to technology-dependent delay optimization, including fanout optimization, using the SIS technology mapper [17] and the *lib2.genlib* standard cell SCMOS 2.0 library [16]. The gate count for each circuit as reported by SIS is given under column *Gates*.

Table 2 shows the results of topological delay (*Top Delay*) and true delay (*True Delay*) estimation. To estimate the true delay, FEST considers paths only in SP. To estimate the topological delay, FEST considers delay of paths in both SP and UP. To establish the accuracy of FEST, the actual topological delay was computed by SIS on the gate level circuit, reported under *SIS* in the column *Top Delay*. The true delay computed by the gate level tool gate-TA [4] on the technology mapped netlist is reported under *gate-TA* in the column *True Delay*.

For the 4-bit implementation of the UAV example, the high-level topological delay estimate computed by FEST is 37.68 ns while the gate-level value computed by SIS is 31.60 ns. The true delay estimated at the high-level by FEST is 28.88 ns while at the gate-level, gate-TA computes a value of 28.00 ns. The data shows that topological delay can be pessimistic at both the gate and the high-level. It also shows that the delay estimates at the high-level returned by FEST compare well with the results of gate-level tools.

The *cpu* time taken by FEST and gate-TA is reported under the column *cpu*. In all cases, high-level delay estimates by FEST is very fast even when the gate level timing analyzer fails to complete.

Table 2: Results of Topological and True Delay Estimation

| Circuit | # Gates | Top Delay[ns] | |
|---|---|---|---|
| | | FEST | SIS |
| UAV (4 bits) | 315 | 37.7 | 31.6 |
| UAV (8 bits) | 746 | 56.4 | 48.4 |
| dealer (4 bits) | 316 | 38.1 | 34.3 |
| dealer (8 bits) | 578 | 61.3 | 56.4 |

| Circuit | True Delay[ns] | | cpu[s] | |
|---|---|---|---|---|
| | FEST | gate-TA | FEST | gate-TA |
| UAV (4 bits) | 28.9 | 28.0 | 0.46 | 1153 |
| UAV (8 bits) | 40.1 | NA | 0.52 | abrt |
| dealer (4 bits) | 24.1 | 20.0 | 0.42 | 403 |
| dealer (8 bits) | 45.94 | NA | 0.43 | abrt |

## VIII. Conclusions

We prove that paths created in a circuit due to resource sharing need not be considered for calculating the true delay of a circuit. We give a delay model and estimation techniques for a fast and accurate estimation of the true delay of a circuit without doing path sensitization. The accuracy and speed of our delay estimation technique makes it feasible to be invoked repeatedly in a high level design process. Our technique can also be extended for a more accurate delay estimation at the RT-level, making it possible to estimate the delay of large RT-level circuits which could not be handled by traditional gate-level tools.

**Acknowledgments.** We acknowledge Toshio Misawa for the formulation of the Schedule Graph, and Pranav Ashar for his help in using his gate-level timing analysis tool [4].

## References

[1] H-C. Chen and D. H. C. Du. Path Sensitization in Critical Path Problem. In *ICCAD*, 1991.

[2] S. Devadas, K. Keutzer, and S. Malik. Delay Computation in Combinational Logic Circuits: Theory and Algorithms. In *ICCAD*, 1991.

[3] P.C. McGeer, A. Saldanha, P.R. Stephan, and R.K. Brayton. Timing Analysis and Delay-Fault Test Generation using Path-Recursive Functions. In *ICCAD*, Nov 1991.

[4] P. Ashar, S. Malik, and S. Rothweiler. Functional Timing Analysis using ATPG. In *EDAC*, 1993.

[5] S. Perremans, L. Claesen, and H. De Man. Static Timing Analysis of Dynamically Sensitizable Paths. In *26th DAC*, 1989.

[6] 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. In *ICCAD*, August 1992.

[7] A. Kuehlmann and R. A. Bergamaschi. Timing Analysis in High-Level Synthesis. In *ICCAD*, August 1992.

[8] P.K. Jha and N.D. Dutt. Rapid Estimation for Parameterized Components in High-Level Synthesis. *IEEE Transactions on VLSI Systems*, 1(3):296–303, Sept 1993.

[9] C. Safinia, R. Leveugle, and G. Saucier. Taking Advantage of High Level Functional Information to Refine Timing Analysis and Timing Modeling. In *ED&T*, 1994.

[10] C. Ramachandran and F. J. Kurdahi. Combined Topological and Functionality Based Delay Estimation Using a Layout-Driven Approach for High Level Applications. In *EDAC*, 1992.

[11] R. A. Bergamaschi. The Effects of False Paths in High-Level Synthesis. In *ICCAD*, 1991.

[12] B. Gregory, D. MacMillen, and D. Fogg. ISIS: A System for Performance Driven Resource Sharing. In *29th DAC*, 1992.

[13] S. Bhattacharya, S. Dey, and F. Brglez. Clock Period Optimization During Resource Sharing and Assignment. In *31st DAC*, 1994.

[14] S. Bhattacharya, S. Dey, and F. Brglez. Provably Correct High-Level Timing Analysis without Path Sensitization. Technical report, C&C Research Labs, NEC USA, June 1994.

[15] Kenneth Hintz and Daniel Tabak. *Microcontrollers: Architecture, Implementation, and Programming.* McGraw-Hill, New York, NY 10020, 1992.

[16] K. Kozminski (ed.). *OASIS Users Guide.* MCNC, Research Triangle Park, N.C. 27709, 1992.

[17] E.M. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design using Synthesis and Optimization. In *ICCD*, October 1992.