

# Verification of Delayed-Reset Domino Circuits Using ATACS\*

Wendy Belluomini<sup>1</sup>

Chris J. Myers<sup>2</sup>

H. Peter Hofstee<sup>3</sup>

<sup>1</sup>Computer Science Department, University of Utah, Salt Lake City, UT

<sup>2</sup>Electrical Engineering Department, University of Utah, Salt Lake City, UT

<sup>3</sup>IBM Austin Research Laboratory, Austin, TX

## Abstract

*This paper discusses the application of the timing analysis tool ATACS to the high performance, self-resetting and delayed-reset domino circuits being designed at IBM's Austin Research Laboratory. The tool, which was originally developed to deal with asynchronous circuits, is well suited to the self-resetting style since internally, a block of self-resetting or delayed-reset domino logic is asynchronous. The circuits are represented using timed event/level structures. These structures correspond very directly to gate level circuits, making the translation from a transistor schematic to a TEL structure straightforward. The state-space explosion problem is mitigated using an algorithm based on partially ordered sets (POSETs). Results on a number of circuits from the recently published guTS (giga-hertz unit Test Site) processor from IBM indicate that modules of significant size can be verified with ATACS using a level of abstraction that preserves the interesting timing properties of the circuit. Accurate circuit level verification allows the designer to include less margin in the design, which can lead to increased performance.*

## 1 Introduction

In order to be successful and widely used, any circuit design style must have extensive CAD support. Lack of CAD support is one reason that asynchronous design has not been widely accepted, despite its advantages in some areas. However, in order to get high performance, synchronous designs are always pushing the boundaries of existing CAD tools as well. The circuit style being developed and used at IBM's Austin Research Laboratory is a good example of this. Although the circuits are synchronous, their local behavior is asynchronous, and the timing constraints that must be met for them to work correctly are quite com-

plex. Existing synchronous static timing analysis methods can be adapted to analyze this type of circuit [1, 2, 3], but they have some limitations. The approach presented in [1] extends the static timing methodology by changing the standard two event per signal model to a four event per signal model. This allows all of the relevant timing constraints on a domino gate to be verified. However, since the method considers only topological delay and not boolean behavior, it can be overly pessimistic. The method presented in [2] is successful in verifying a large, high performance chip. It adds some boolean behavior to the topological delay calculations in order to improve accuracy, but may still be conservative. The technique presented in [3] is designed to verify self-resetting or delayed-reset circuits at the macro level. A designer must determine an interface specification for each macro through simulation. The timing analysis tool then determines if the combination of all the macros correctly implements all of the interfaces. This works well for chip level timing verification, but the interfaces specified by the designer are never formally verified. A tool that verifies that the gates inside the macro work correctly within the specified interface is required to complete the verification.

Since the behavior inside the self-resetting and delayed-reset macros is very similar to that of an asynchronous circuit, a tool designed for asynchronous circuits is useful in their timing verification. A timing analysis tool for asynchronous circuits must be more general than the static timing tools used for synchronous circuits. No assumptions about the time behavior of the system are made, which means that an asynchronous timing verification tool can handle any timing optimizations made by synchronous designers. This flexibility allows designers to create new circuit styles without modifying the timing analysis tool.

There are two basic approaches to timing analysis for asynchronous systems. One approach, presented in [4], is based on *time separation of events*. This algorithm is efficient for determining an individual time separation. However, verification of a large circuit model requires checking thousands of time separations. Since these algorithms view

---

\*This research is supported by NSF CAREER award MIP-9625014, SRC contract 97-DJ-487, and a DARPA ASSERT Fellowship. The work in this paper was performed while the first author was an intern at the IBM Austin Research Laboratory.

each new time separation as a different problem, there is a point at which exploring the timed state space of the specification becomes faster than computing all of the necessary time separations. The other approach, which is taken in this paper, is based on timed state space exploration. Timed state space exploration algorithms determine the set of all states that are reachable in the specification given the timing information provided. The state space exploration algorithm used for verification in this paper is based on the POSET technique [5], which significantly reduces state space size.

However, any tool designed for asynchronous circuits is limited in the size of specifications it can analyze. Since asynchronous tools explore all possible sequential behaviors of the circuit, their time and space complexity are exponential, and they are best used for gate level analysis of small circuits or abstracted versions of larger macros. This paper describes the application of the asynchronous timing analysis tool ATACS, to the verification of several designs from the guTS (gigahertz unit Test Site) processor designed at IBM's Austin Research Laboratory. It first describes the model that ATACS uses to represent circuits. It then discusses the delayed-reset domino circuit style and its performance advantages. Finally, it presents several case studies from the guTS microprocessor. These case studies show that ATACS is capable of analyzing delayed-reset and self-resetting circuits accurately and that the abstractions necessary to make the problems tractable do not cause the tool to underestimate performance. If it is possible to accurately verify the macros, less margin is necessary to ensure correctness, and higher performance circuits can be achieved.

## 2 Specifying delayed-reset domino circuits

The specifications that are analyzed by ATACS, Timed Event/Level (TEL) structures [6], described formally below, are very well suited to the delayed-reset domino style. TEL structures extend timed ER structures [7], by allowing boolean expressions to be added to the specification. TEL structures represent a set of specifications equivalent to those represented by both time and timed Petri nets, and they have two main advantages over purely event based specifications. The first is that they are easy to generate from a higher level language such as VHDL [8]. Purely event based specification formats do not correspond well to the signal level based semantics of higher level languages. The second advantage is in performance. Specifications expressed as TEL structures are more compact and have fewer markings than those expressed in a purely event based formalism. This results in smaller state spaces and decreased run time and memory usage.

### 2.1 Timed event/level structures

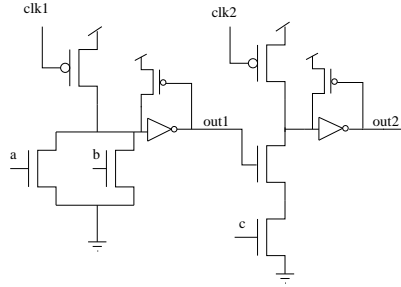
A TEL structure is a 6-tuple  $T = \langle N, s_0, A, E, R, \# \rangle$  where:

1.  $N$  is the set of signals;
2.  $s_0 = \{0, 1\}^N$  is the initial state;
3.  $A \subseteq N \times \{+, -\} \cup \$$  is the set of actions;
4.  $E \subseteq A \times (\mathcal{N} = \{0, 1, 2, \dots\})$  is the set of events;
5.  $R \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\}) \times (b : \{0, 1\}^N \rightarrow \{0, 1\})$  is the set of rules;
6.  $\# \subseteq E \times E$  is the conflict relation.

The signal set,  $N$ , contains the wires in the circuit specification. The state  $s_0$  contains the initial value of each signal in  $N$ . The action set,  $A$ , contains for each signal  $x$  in  $N$ , a rising transition,  $x+$ , and a falling transition,  $x-$ , along with the sequencing event  $\$$ , which is used to indicate an action that does not cause a signal transition. The event set,  $E$ , contains actions paired with occurrence indices (i.e.,  $(a, i)$ ). Pairing actions with occurrence indices allows an arbitrary number of events to be created from each action, including the sequencing action,  $\$$ . Sequencing events are often used to express nondeterminism where a signal may or may not transition. Although, formally the definition requires that all sequencing events be of the form  $\$/i$  where  $i$  is an integer, sequencing events of the form  $\$/s$  where  $s$  is a string are used in this paper in order to make the purpose of the sequencing event more clear. Rules represent causality between events. Each rule,  $r$ , is of the form  $\langle e, f, l, u, b \rangle$  where:

1.  $e$  = enabling event,
2.  $f$  = enabled event,
3.  $\langle l, u \rangle$  = bounded timing constraint, and
4.  $b$  = a boolean function over the signals in  $N$ .

A rule is *enabled* if its enabling event has occurred and its boolean function is true in the current state. There are two possible semantics concerning the enabling of a rule. In one semantics, referred to as *non-disabling semantics*, once a rule becomes enabled, it cannot lose its enabling due to a change in the state. In the other semantics, referred to as *disabling semantics*, a rule can become enabled and then lose its enabling. This can occur when another event fires, resulting in a state where the boolean function is no longer true. A single specification can include rules with both types of semantics. Non-disabling semantics are typically used to specify environment behavior and disabling semantics are typically used to specify logic gates. For the purposes of verification, the disabling of a boolean expression on a disabling rule is assumed to correspond to a failure. Disabling of boolean guards are considered failures since they correspond to a glitch on the input to a gate. A rule is *satisfied* if



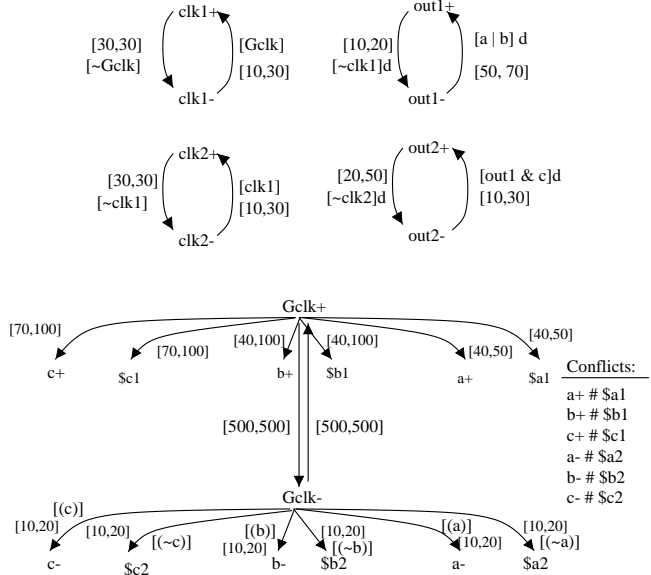
**Figure 1. A delayed-reset domino gate.**

it has been at least  $l$  time units since it was enabled and *expired* if it has been at least  $u$  time units since it was enabled. Excluding conflicts, an event cannot occur until every rule enabling it is satisfied, and it must occur before every rule enabling it has expired.

The conflict relation,  $\#$ , is used to model disjunctive behavior and choice. When two events  $e$  and  $e'$  are in conflict (denoted  $e \# e'$ ), this specifies that either  $e$  can occur or  $e'$  can occur, but not both. Taking the conflict relation into account, if two rules have the same enabled event and conflicting enabling events, then only one of the two mutually exclusive enabling events needs to occur to cause the enabled event. This models a form of disjunctive causality. Choice is modeled when two rules have the same enabling event and conflicting enabled events. In this case, only one of the enabled events can occur. Every pairwise conflict in the TEL structure must be specified, but this does not cause a problem for the user since TEL structures are typically generated from a higher level input language, such as VHDL [8]. If a specification is cyclic, then the TEL structure representing it is infinite. However, due to its repetitive nature, this infinite behavior can be described with a finite model by adding an additional set of rules and conflicts which recursively define the infinite structure [7].

Figure 1 shows an example of a delayed-reset domino gate. The gate computes the function  $(a \vee b) \wedge c$  in two stages. The first stage computes  $a \vee b$  while  $clk1$  is high, and the next stage computes  $out1 \wedge c$  while  $clk2$  is high. Both gates precharge while their respective clocks are low. Since neither n-stack has a “foot” transistor to ensure that the path to ground is turned off during the precharge phase, the timing of the circuit must guarantee that all the inputs to the gate are low by the time the local clock for each stage falls.

The TEL structure representation for the domino gate is shown in Figure 2. It includes one rising and one falling event for each signal. The specification indicates that there is a global clock  $Gclk$  which rises 500 time units after it falls and falls 500 time units after it rises. The inputs to the gate,  $a, b$  and  $c$ , nondeterministically rise some time after the clock rises. The nondeterminism is modeled us-



**Figure 2. TEL structure for the gate in Fig. 1**

ing the conflict relation and sequencing events. Each rising event on an input conflicts with a corresponding sequencing event. Since the rising event and the sequencing event conflict, only one of them can occur. If the rising event for a signal fires, the signal rises in that clock cycle, if the sequencing event fires, it does not. A falling transition on the global clock is followed by falling transitions on all of the inputs, if they have risen. Again sequencing events and conflicts are used to deal with the nondeterminism. If an input signal rises on the rising edge of  $Gclk$  then a falling event for that signal must occur when  $Gclk$  falls. Otherwise, a conflicting sequencing event fires, preventing the falling event on the input signal from becoming enabled as soon as that signal rises again. The  $Gclk$  signal also controls the firing time of the two local clocks,  $clk1$  and  $clk2$ . The local clock  $clk1$  rises between 10 and 30 time units after  $Gclk$  rises and falls 30 time units after  $Gclk$  falls. The other local clock,  $clk2$  and the two gate outputs,  $out1$  and  $out2$  are specified in a similar fashion.

Although the TEL structure is readable for a small circuit, it would be difficult to specify a large macro at this level. ATACS provides support for two higher level input languages, VHDL and CSP. Designers can specify circuits in these languages, and they are compiled into TEL structures using techniques described in [7, 8].

Once a TEL structure for the circuit is created, it can be used by ATACS to find all of the circuit’s possible timing behaviors and check for timing failures. In the case of these domino gates, ATACS is used to check for two types of failures. The first is that once a pulldown stack turns on, it remains on until the dynamic node has switched. The other is that all paths to ground in the n-stack are turned off be-

fore the dynamic node begins to precharge. Both of these conditions must be met in order for the circuit to be electrically sound. If the first condition is not met, an incorrect value for the dynamic node may be latched, and if the second condition is not met the gate will draw a large short circuit current. Either of these conditions would be quite simple to check for statically, since the first can always be met by increasing the time that the inputs are on, and the second can always be met by decreasing the time that the inputs are on. However the combination of the constraints is more difficult to check. There is a range of pulse lengths for each signal that allow the circuit to function correctly, and the range for each signal is correlated to the pulse lengths of many other signals.

Analysis with ATACS allows a designer to check if a set of pulse length ranges works without considering the implications of each range by hand or running a SPICE level simulation. The first constraint, that inputs remain on long enough for a gate to switch, is always checked automatically as ATACS explores the state space of the system. ATACS generates an error if the boolean expression associated with any disabling rule can become true and then false again before the signal transitions. This error is generated regardless of the timing bound on the rule involved. For example, consider the TEL structure for the signal *out1* in Figure 2. When *out1* is low, if the expression  $a \vee b$  becomes true, ATACS generates an error if it becomes false at any time before *out1* rises. The upper bound on the pulse lengths can be checked by adding a simple constraint to the specification which states that for each gate, the inputs must be low at the time the clock falls. This constraint is the same for each gate, so it can be generated automatically. Additionally, any other timing constraint that a designer may wish to check, such as setup and hold times, can also be specified and verified.

After the transistor level design is completed, the best and worst case propagation time through each gate is derived from SPICE. These delays are computed as the time difference between the midpoints of the rising edge of the input change and the rising edge of the transitioning output. After each gate has been characterized, ATACS can be used to check that a circuit is electrically sound without running multiple SPICE runs on the entire circuit to check all possible combinations of gate delay behavior. Since ATACS does not do electrical analysis, it is necessary to add margin to the SPICE derived delays to ensure that electrically marginal circuits do not verify. SPICE simulations show that a 5% margin on the SPICE derived best and worst case gate propagation times is enough to ensure that circuits that verify in ATACS do not have electrical failures due to timing. When circuits are too large to be verified at the gate level, conservative abstractions which preserve the property that a circuit that is verified is electrically sound are used.

### 3 Timed state space exploration

Circuits specified as TEL structures are verified using a depth-first search to find all of the states allowed by the specification. The algorithm uses a number of sets to keep track of the status of all of the rules in the specification during the search. The  $R_m$  set or, *marked set*, contains all events whose enabling event has fired. The state vector,  $s_c$ , contains the state of all the signals. From this information, the algorithm can compute the set of enabled rules,  $R_{en}$ , which includes only those members of  $R_m$  whose boolean expressions are satisfied by the state vector. In order to determine which rules in  $R_{en}$  can fire, timing information,  $TI$  is needed. A *timed state* is defined to be  $R_m \times s_c \times TI$ . A timed state contains all the information necessary to compute the set of rules that can fire,  $R_s$ .

#### 3.1 Geometric regions

Timing information is represented with *geometric regions*, which were first introduced in [9, 10]. In the geometric region approach, part of  $TI$  is defined to be a constraint matrix  $M$  that specifies the maximum difference in time between the enabling times of all the rules in  $R_{en}$ . The  $i$ th row and column of the matrix contain the separations between the enabling times of each rule in  $R_{en}$  and a dummy rule  $r_0$ . The enabling time of  $r_0$  is defined to be uniquely 0. Each entry  $m_{ij}$  in the matrix  $M$  contains the maximum time difference between the enabling time of rule  $j$  and the enabling time of rule  $i$ . Since the enabling time of  $r_0$  is always zero, the maximum time difference between the enabling time of rule  $i$  and the enabling time of rule  $r_0$  ( $m_{0i}$ ) is just the maximum time since  $i$  was enabled. The maximum time difference between the enabling time of  $r_0$  and the enabling time of rule  $i$  ( $m_{i0}$ ) is the negation of the minimum time since  $i$  was enabled. Note that  $M$  only needs to contain information on the timing of the rules that are currently in  $R_{en}$ , not on the whole set of rules. This constraint matrix represents a convex  $|R_{en}|$  dimensional region. Each dimension corresponds to a rule and the firing times of the enabled events for the rules can be anywhere within the space. The result of timed state space exploration using geometric regions is a *region graph*, where the nodes in the graph are geometric regions and the edges are rules. Although the TEL structure is infinite, its repetitive nature allows its behavior to be expressed as a cyclic region graph where each geometric region represents an equivalence class defined by the set of possible time relationships between enabled rules.

When an event fires and causes new rules to be added to  $R_{en}$ , the matrix needs to be updated to reflect the new timing information. Information about the newly enabled rules must be added to the constraint matrix and information about rules that are no longer in  $R_{en}$  must be removed. The

main operation used to do this is *recanonicalization*. Recanonicalization takes a matrix  $M$  where some of the  $m_{ij}$ 's are greater than  $\max(t(\text{enabling}(j)) - t(\text{enabling}(i)))$  and produces a matrix where all the  $m_{ij}$ 's have their maximum allowed value. The assignment of the  $m_{ij}$ 's so that they all have their maximum value is always unique, so the algorithm can determine when a given region is equivalent to or contained in a region that has been seen before. Recanonicalization is essentially the all pairs shortest path problem and can be done in  $O(n^3)$  time with Floyd's algorithm. When the algorithm is used for maintaining a region matrix, it can be done incrementally in  $O(n^2)$  time, since most entries in the matrix already have their canonical value [11].

In our version of the geometric region algorithm [12], timing information is updated whenever a rule fires, and rules are allowed to fire independently of events. This approach is a generalization of the geometric regions technique presented in [11], where timing information only changes when an event fires. In our algorithm, a rule can always fire when it is satisfied. The firing of a rule, however, does not always correspond to the firing of an actual event. An event only fires when all of the rules enabling it have fired. As rules fire, they are projected out of the constraint matrix, and are removed from  $R_m$ ,  $R_{en}$ , and  $R_s$ . They are added to a new set of "fired" rules,  $R_f$ , which is part of the timing information. Since they have fired, timing information about them is no longer needed, but the fact that they have fired must be recorded. When a set of rules sufficient to enable an event  $e$  are in  $R_f$ ,  $e$  can fire.

### 3.2 POSET timing: updating the state

A depth first search is used to find the timed state space of a TEL structure. From a timed state,  $R_m \times s_c \times M \times R_f$ , the search algorithm calculates the  $R_s$  set. It then chooses a rule from  $R_s$  to fire, places the rest of the rules in  $R_s$  on the stack, and calls a function that returns the timed state that results from firing the rule. If the new timed state has been seen before, the algorithm pops an unexplored timed state off the stack and continues the search. If there are no more unexplored states on the stack, the algorithm has completed.

In order to reduce the state explosion problem, an algorithm based on *partially ordered sets* (POSETs) [13, 5] is used. The POSET algorithm uses partially ordered sets of events to create geometric regions rather than linear sequences. This prevents additional regions from being added for different sequences of event firings that lead to the same untimed state. POSET timing results in a compression of the state space into fewer, larger geometric regions that, taken together, contain the same region in space as the set of regions generated by the standard geometric technique. Regions that are generated by the standard geometric technique are only combined by the POSET technique if their

combination is convex. Therefore, all properties of the system that can be verified with the standard geometric technique can be verified with the POSET algorithm.

The core of the POSET algorithm involves updating the timed state when a new rule fires. The function to do this must update all of the sets which indicate the status of rules as well as the constraint matrix and a POSET matrix, which contains time relationships between event firing times. The function first adds the firing rule to the fired set,  $R_f$  and removes it from the marked set  $R_m$ . It then checks if firing this rule causes any event to fire. An event is fired if all of the rules that enable it are either in  $R_f$  or conflict with another rule that is in  $R_f$ . If an event can fire, the algorithm updates the untimed state. If the change in the state vector causes any rule to become disabled, either an error or warning is generated to indicate that this may be a verification failure. Next, the algorithm removes any rules that enable an event in conflict with the firing event from the constraint matrix  $M$ . It also removes any conflicting rules from  $R_m$  and  $R_f$ , and it removes from  $R_f$  any rules that enable the firing event. Next, the algorithm adds any rules whose enabling event is the firing event to the marked set  $R_m$ , and adds any newly enabled rules to the constraint matrix. The age differences between these rules and the previously existing rules are determined by the contents of the POSET matrix, which needs to be updated to include the new event firing. The procedure for updating the POSET matrix and using the new POSET matrix to compute a new constraint matrix is described in the next subsection. After the constraint matrix is updated, the fired rule is eliminated from the matrix, and time is advanced. The recanonicalization step then restricts the maximum ages to those that are allowed given the age differences allowed by the POSET matrix. Finally the algorithm returns a new timed state and an updated POSET matrix.

### 3.3 POSET timing: updating the POSET

The method for updating the POSET matrix is based on the concept of causality. An event that is enabled by multiple rules does not fire until all of these rules have fired. The last rule to fire actually causes the event to fire, and is referred to as the *causal rule*. More formally, a rule  $r_m = \langle e_c, e, l, u \rangle$  is causal to event  $e$  given a rule firing sequence  $r_0 \dots r_n$ , if the firing sequence  $r_0 \dots r_{m-1}$  does not enable  $e$  and the firing sequence  $r_0 \dots r_m$  does enable  $e$ . When doing analysis with ER structures, if  $r = \langle e_c, e, l, u \rangle$  is the causal rule to  $e$ , then the firing time of the event  $e_c$  controls the firing time of event  $e$ .

However, with TEL structures this is not always the case. The event that controls the enabling time of a rule may be its enabling event, or it may be some other event firing which causes its boolean expression to become satisfied. To ana-

lyze TEL structures, the concept of a *causal event* to a rule is also necessary. The causal event of a rule,  $r$ , is the event whose firing caused  $r$  to become enabled. More formally, an event  $e_m$  is causal to a rule  $r$  given an event firing sequence  $e_0 \dots e_n$  if the rule  $r$  is not enabled after the sequence  $e_0 \dots e_{m-1}$  has fired and is enabled after  $e_0 \dots e_m$  has fired. For TEL structures, the causal event,  $e_c$ , to an event  $e$  is the causal event of the causal rule of  $e$ ,  $r_c = \langle e_i, e, l, u, b \rangle$ . Since  $e_c$  controls the time that  $r_c$  becomes enabled, and the firing time of  $r_c$  controls the firing time of  $e$ , it is the firing time of  $e_c$  which determines when  $e$  fires. Once  $e_c$  fires, between  $l$  and  $u$  time units pass before  $e$  fires. Since  $e_c$  is the causal event, no other events are necessary in order for  $e$  to fire, and the upper bound on  $r$ ,  $u$ , cannot be exceeded. For example, consider the TEL structure for the signal *out2+* in Figure 2 when *out2-* has just fired. Suppose that *out1* rises and then  $c$  rises. In this case  $c+$  is causal to the event *out2+*. Assuming that the rule does not become disabled, *out2+* must rise between 10 and 30 time units after  $c$  rises.

For any given rule firing sequence, there is a well defined causal event for each event firing,  $e$ . The timing of this causal event completely determines the firing time of  $e$ . Events that are concurrent with  $e$  do not effect its firing time. The purpose of the POSET matrix is to keep track of the time separations between event firing times that are allowed by the causality in the firing sequence without forcing the timing behaviors represented by the geometric regions to conform to the total order of the firing sequence. This prevents a new region from being generated for every possible firing sequence leading to an untimed state and drastically reduces the size of the state space.

When a new event,  $f$ , is added to the POSET matrix, the time relationships between  $f$  and the events already in the POSET matrix must be calculated. These relationships depend on the causality in the firing sequence being explored. If an event  $e_i$  in the POSET matrix is causal to  $f$ , the possible time relationships between  $e_i$  and  $f$  are completely determined. The maximum time that can pass between  $e_i$  and  $f$  is  $u$ , and the minimum time is  $l$ . If an event  $e_i$  is not causal to  $f$ , but is the enabling event of a rule  $r$  that enables  $f$ , then the minimum separation between  $e_i$  and  $f$  is the lower bound on  $r$ , since the lower bounds on all rules enabling  $f$  must be met. Due to the level expressions, the event  $e_i$  may be necessary in order to fire  $f$  even if it is not the enabling event of a rule enabling  $f$ . If this is the case, then  $e_i$  is *and-context* to  $f$  and must fire some amount of time before  $f$ . For example, if event  $c+$  is enabled by a rule with the boolean expression  $[a \wedge b]$ , then the rising transitions on  $a$  and  $b$  in the POSET matrix must occur a certain amount of time before  $c+$ . A different restriction is necessary for boolean expressions involving **or** expressions. If an event  $a+$  is causal to an event  $c$  through a rule with a boolean expression  $a \vee b$ , then  $a+$  must have happened before

any rising events on  $b$ , otherwise  $b$  would have been causal. Finally, if none of these conditions apply, the minimum and maximum are set to infinity since nothing is known about them. The matrix is then recanonicalized, which constrains all of the separations down to the maximums allowed by the known constraints. Finally, any events that are no longer relevant to future behavior of the system are removed from the matrix and it is recanonicalized. The result is a POSET matrix that constrains the minimum and maximum separations between events that are allowed by the causality in the firing sequence. This new POSET matrix is used to update the constraint matrix, which contains the minimum and maximum differences between the enabling times of rules. A rule becomes enabled when its causal event fires. Therefore, the firing time difference between their causal events is used to determine the age difference between two rules.

For TEL structures with arbitrary boolean expressions, determining whether events are *and-context* or *or-context* could be quite complex. Therefore, if POSET timing is used, the TEL structures are limited to those where each boolean expression is either a single *and* term or a single *or* term. In practice, on the delayed-reset domino circuits this did not prove to be a significant limitation. If a more complex boolean expression is required, the results from the POSET algorithm are conservative. If an exact result with arbitrary expressions is needed, then either the simpler, geometric algorithm for TEL structures [6] can be used, or the specification can be transformed in a straightforward way into one that contains only simple *and* and *or* expressions.

This algorithm extends the benefits of POSET timing to specifications with level expressions. The additions that are necessary to support levels do not add significantly to computation time, since they simply consist of determining causality and context relationships. When TEL structures are limited to simple *and* or *or* terms, these relationships can be determined by checks that occur when a rule becomes enabled, and require very little computation time. The reduction in state space that is generated by the POSET algorithm allows circuits of significant size to be verified.

## 4 Circuit analysis

ATACS was used to analyze several circuits from the guTS (gigahertz unit Test Site) integer microprocessor designed at IBM's Austin Research Laboratory [14]. The purpose of this design is to demonstrate the performance gains that can be achieved using aggressive circuit design. The architecture is a fairly simple, forwarded, four-stage pipeline which implements 96 instructions from the integer part of the PowerPC instruction set. It is implemented in a  $0.25\mu$  CMOS process available in 1997. The high-performance of the circuit is a result of the circuit design, which is nearly 100 dynamic logic, and the microarchitecture, which allows

as much concurrency as possible [15].

#### 4.1 Delayed-reset domino logic

The circuits in the guTS processor are designed using a dynamic circuit style known as delayed-reset domino [16, 17]. A microprocessor designed in this style contains a set of macros which operate synchronously. A delayed-reset domino macro consists of a number of levels of dynamic gates, each of which receives inputs from preceding layers. Standard domino gates use a common clock that acts as a timing reference. In a delayed-reset design, each level of dynamic gates receives its own, precisely timed clock, which is generated by a buffer chain within the macro. The local clocks travel through the logic along with the data, a reset wave preceding each computation wave. This technique allows approximately one-half cycle for each gate to reset and one-half cycle for each gate to evaluate. The cycle time for a delayed-reset domino macro is set by adding the necessary precharge and evaluate times for a single gate. If multiple gates operate on the same precharge signal, cycle time is set by adding the evaluate delay through all the stages to the precharge delay. Due to the overlapping of the precharge and evaluate phases, the delayed-reset domino approach significantly increases the amount of dynamic logic that can be placed in a macro at a given clock frequency.

The delayed-reset domino gates used in the guTS processor lack the “foot” device that is included in a standard domino gate. The purpose of this device is to turn off the gates’ pulldown stack during the precharge phase. Removing this device allows the gate to switch 5% to 15% faster. Alternatively, the gate can compute a more complex logic function using the same transistor stack height if the “foot” transistor is removed [16]. In order to remove this transistor, it is necessary to ensure that the evaluate logic is not on during the precharge phase. This is the case if all inputs to the gate are guaranteed to be low during the precharge phase. To meet this requirement, the inputs to the macro must be pulsed. Combined with the requirement that the inputs to each gate remain stable high long enough to switch the dynamic node, this results in a two sided timing verification problem which is unusual for a synchronous design.

In the guTS processor, the macro level timing verification is done using extensive SPICE level circuit simulation[18]. After the delay behavior of the macros is characterized by designers in SPICE, it is incorporated into a chip level timing model for chip level static timing verification. This was a successful approach for this processor since it worked in first silicon. However, in order to ensure the correctness of the processor over all variations in delay, large amounts of delay margin are included in the design of the macros. If it is possible to formally verify the macros, less margin is necessary to have confidence in

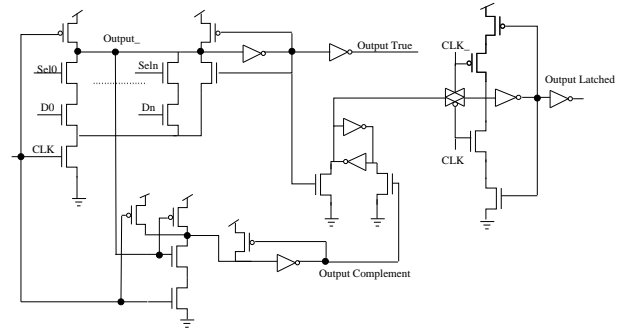


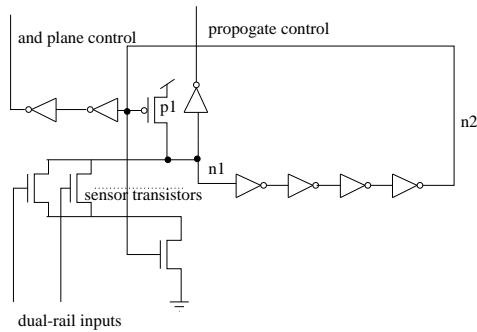
Figure 3. The MLE circuit.

the processor’s correctness, which can result in higher performance. The timing constraints that need to be checked in the delayed reset domino macros are very similar to the correctness constraints necessary for asynchronous circuits, and the delayed reset domino circuits are quite similar to asynchronous circuits. Therefore, an asynchronous timing verification tool is a natural choice to be used for formal verification of the macros.

#### 4.2 Verification of gate level models

The asynchronous timing verification tool ATACS was used to verify several of the macros from the guTS processor. The first circuit is a combined multiplexor and latch (MLE). This circuit is small enough verify at the gate level, and is shown in Figure 3. The goal with this circuit is to verify that the timing specification which is supplied with the circuit indeed guarantees that the circuit works correctly. The timing specification describes the timing requirements that must be met by any circuit communicating with the MLE. It is derived from SPICE level simulation and the circuit designers knowledge of how the circuit works. The timing specifications are also used as the basis for chip level static timing analysis. In order to ensure that the chip-level static timing analysis is modeling all timing behavior, each macro needs to be formally verified in the environment described by the timing specification. ATACS verifies the MLE circuit in a few seconds on a 400MHz Pentium II.

The MLE circuit contains both static and dynamic gates. The inputs to static gates are allowed to be unstable since this does not immediately cause a failure. However, if a glitch on the output of a static gate propagates to the input of a dynamic gate, it can cause a failure. In the MLE circuit, the gate driving the signal “output complement” is static. In every cycle where “output complement” does not fall, there is a glitch on its inputs. At the end of the precharge phase, the signal “Output\_” is always high and it feeds one of the inputs to the static gate. When the clock rises, “output complement” always begins to fall. However, the signal “Output\_” falls later in the clock cycle if the selected data



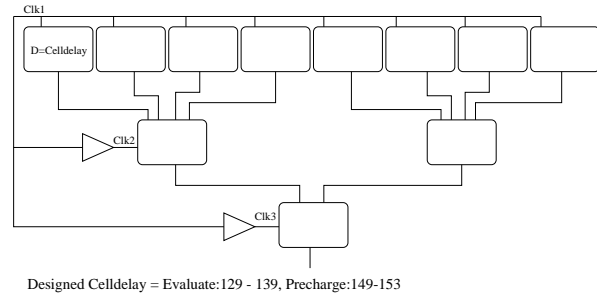
**Figure 4. PLA control.**

value is high. When “Output..” falls, one of the inputs to the static gate is driven low and “output complement” rises again, producing a glitch. ATACS detects this glitch and determines that it cannot propagate to the output of the circuit.

The next circuit is a dynamic PLA that is used in the processor’s control circuitry. Dynamic PLAs are easy to generate automatically and have predictable area and delay. In order to make the PLAs fast, they are controlled using self-resetting circuitry. An example of the control circuitry is shown in Figure 4. The circuit uses a very aggressive technique to determine when its inputs are valid. The inputs are presented to the circuit dual-rail. When the inputs are valid the sensor transistors are turned on. These transistors are all connected to a single node,  $n1$ , which has been precharged high. The sensor transistors are sized so that one of them must be turned on for each input in order for  $n1$  to discharge quickly. However, if one input arrives much earlier than the rest, eventually its single sensor transistor can discharge  $n1$ , erroneously causing the PLA to begin evaluating early. This completion detection circuit is highly timing dependent and only works if the inputs are guaranteed to arrive within a narrow time interval. After the falling edge of  $n1$  propagates through four inverters, the node  $n2$  falls. When this node falls, transistor  $p1$  is turned on which raises node  $n1$ , resetting the completion detection circuit. The line “and plane control” is used to gate transistors which determine if the and-plane of the PLA is in precharge or evaluate mode. The line “propagate control” is used in a similar manner to control whether the output of the and-plane can propagate to the or-plane of the PLA, which is not shown. This control circuitry is essentially asynchronous. Asynchronous, self-resetting circuits are difficult for static tools to handle since they often assume that a transition on an input will cause only a single transition on an output. ATACS is able to verify the circuit using the designed delays in a few seconds.

### 4.3 Verification of abstracted models

The next circuit is a compare unit for two 64 bit quantities. It consists of 3 stages of delayed-reset domino logic.



**Figure 5. Model for the compare unit.**

The logic in each stage is exactly the same. A stage consists of a set of blocks that produce an output which indicates whether its two four bit inputs are equal. To do a 64 bit compare, a tree structure is used where the first stage has 16 logic blocks, the second stage has 4 logic blocks, and the final stage has 1 logic block. Unlike the previous two examples, this circuit is too large for ATACS to verify it using a representation derived directly from its transistor level schematic. However, with a small amount of abstraction, it can be verified quickly. It is not necessary to model each of the 64 bits entering the compare unit. Each block in the first level of logic is modeled as a gate that waits for a single input and produces its output some variable amount of time later. Variability in input signal arrival times is accounted for by putting an independent delay range on the arrival time of the abstracted input signal for each of the blocks in the first level of logic. When this signal rises in the abstracted model, it is equivalent to all eight input bits to a block becoming stable in the actual circuit. Additionally, since the timing behavior of each block is the same, the number of input blocks can be reduced from 16 to 8 without effecting the timing behavior of the circuit. Figure 5 shows the structure of the model. Each block is represented as a TEL structure which raises its output signal 129 to 139 time units after the block receives all of its inputs, and lowers its output 149 to 153 time units after its local clock falls. A global clock which controls the transition times of the local clocks is also modeled but not shown. It takes less than five seconds to explore the state space of this model using the POSET state space algorithm on a 400MHz Pentium II. For comparison the model is also analyzed using the standard, geometric region based method [9, 19, 10]. This method requires 196 seconds to analyze the model. The iteration time provided by the POSET algorithm makes it reasonable to iteratively adjust the Celldelay values, global clock speed, and local clock timings to determine the working ranges of the circuit under a variety of assumptions. The circuit verifies for global clock cycles up to 100ps less than the clock cycle necessary for correct operation in the Gigahertz processor.

Since state space exploration is an exponential problem,

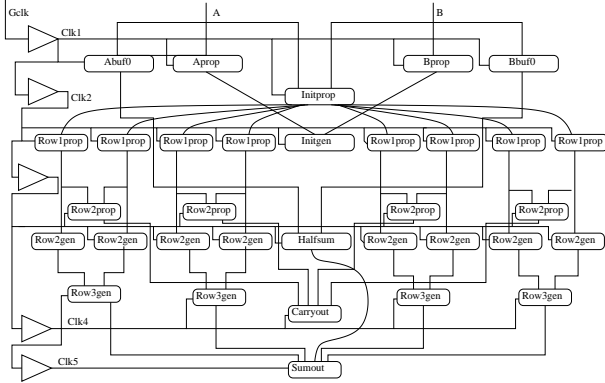


Figure 6. MFXU structure.

large specifications can only be verified at a high level of abstraction. This is illustrated by the verification of the 64-bit adder portion of the Multifunction Fixed Point Unit (MFXU). This unit computes the results of the add, subtract, and compare instructions for the processor. The core of the unit is the 64-bit parallel prefix adder design presented in [15], which is based on the algorithm described in [20]. The MFXU adder contains five stages of delayed reset domino logic. The first stage contains a true/complement mux, stages two through four compute the propagate and generate signals for the adder, and the fifth stage implements a large mux which merges many different signals. Each block contains a few domino gates, which can vary in delay. Attempts to verify this circuit at the gate level quickly use more than half of a gigabyte of memory and do not complete. However, a conservative abstraction of the MFXU verifies in ATACS using the POSET algorithm in about 2 minutes. The verification does not complete using the geometric algorithm.

The structure of the MFXU abstraction is shown in Figure 6. There are two steps involved in creating the conservative abstraction of the MFXU. The first is to reduce the complexity of each block by lumping the delay ranges for all of the different gates into one delay range which represents the minimum and maximum time difference between the block receiving all of its inputs and generating all of its outputs. For example, suppose a block contains two domino gates  $d_1$ , which takes 100ps to evaluate and  $d_2$  which takes 150ps to evaluate. It is conservative to make a model for the block where the minimum evaluate time for the block is 100ps and the maximum evaluate time for the block is 150ps. This abstraction does not capture the gate level behavior that one output of the block is available after 100ps and the other is available after 150ps, but if a circuit verifies using the abstraction, its actual behavior verifies also. An abstraction like this is made for the precharge phase and the evaluate phase of each block. Then the number of blocks is decreased. The goal is to reduce the number of blocks,

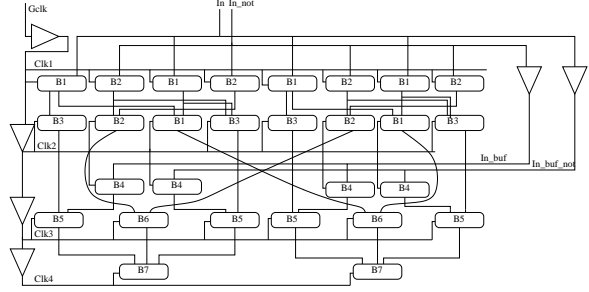


Figure 7. The CLZ circuit.

without hiding any interesting block interactions. This is done by analyzing a 32-bit wide slice of the design. Since each block operates on four bits of input, this corresponds to a model that is eight blocks wide at its input. This model is large enough to include all of the types of interblock relationships of the larger design, and is small enough to verify quickly.

This is done by starting at the last stage and working toward the first. Every block in the last stage is included. Then, for every block in the last stage, at least two instances of each type of block that provides inputs to the last stage are included in the fourth stage. In this case, four instances of the *row3gen* block which feeds *sumout* block in the fifth stage are included. Only one instance of the *halfsum* block is included since there is only one *halfsum* block in the complete circuit. This process is then repeated for the fourth through first stages. The resulting model represents a conservative model of the possible timing relationships in the circuit, and is small enough to verify quickly.

The circuit, abstracted in this way, verifies at its intended clock speed. Therefore, any gate-level timing relationships that are missed by the abstraction are not necessary in order for the circuit to run at the specified speed. If this is not the case, then the blocks on the failure path can be specified in more detail. Although this increases verification time, it should not make the problem intractable since the additional detail is limited to a few blocks. Even the abstracted version of this circuit is quite large, it has complex timing relationships and which provide many possibilities for error. Formal verification gives confidence that all of the timing behaviors have been considered. Currently, ATACS does not have an automated method for generating circuit abstractions, and the abstraction described for this example is done manually. It may be possible to adapt techniques from [21] to develop an automated method for abstracting blocks of domino gates.

The final circuit is an arithmetic circuit used in the integer execution unit. It is of moderate complexity and therefore can be used to test the accuracy of an abstracted model vs. a gate level model. The gate level model is still somewhat abstract in that it does not include the full 64-bit dat-

apath, but each instance of a block is described at the gate level. The results on this macro indicate that the limiting factor in clock speed is the time that the inputs arrive to the macro, not gate to gate interactions inside the macro. Because of this, the maximum clock speeds allowed by the abstracted model and the gate level model are the same. In order for a gate level model to allow a circuit to verify at a higher clock speed than an abstracted model there need to be instances of fast gates in one stage feeding slow gates in another block in the next stage. Such instances do not occur in this example.

## 5 Conclusions and future work

ATACS can be very effective in verifying delayed-reset domino circuits. TEL structures are well-suited to specifying domino gates at both the gate level and at a higher level of abstraction. Since ATACS is designed for asynchronous circuits, it is very flexible and it can be used to verify many different circuit styles by varying the constraints that are checked. Since ATACS can verify circuit level timing specifications, less margin is necessary in each circuit to ensure that the circuit works correctly, which can result in higher performance. ATACS does a complete state space exploration. Therefore, its complexity is exponential and it is not practical to verify large circuits at the gate level. However, for most circuits, a higher level of abstraction is sufficient to verify that that the circuit can run at the desired speed. If this is not the case, it is possible to locally specify more detail on paths that fail without causing a state explosion. Most importantly, this work shows how tools developed for asynchronous circuits can be of great use to synchronous designers when they choose aggressive circuit styles.

Although the results in this paper are somewhat preliminary, resulting from five weeks of work at IBM, they are very promising. In order to make this method practical for circuit designers, more work is needed to develop a more automated method of abstracting circuits, and to develop a method of verifying circuits hierarchically. Additionally, all of the circuits described in this paper are completed and no failures are found by ATACS when designed delays are used. It would be interesting to study how ATACS can help designers determine which delay ranges result in correct circuits closer to the beginning of the design cycle, as well as how it can be used on early versions of circuits to find actual failures. Finally, we would like to explore how the synthesis capabilities of ATACS can be used to help automate the design of delayed-reset domino and self-resetting circuits.

## 6 Acknowledgments

We would like to thank all of the guTS design team at IBM, especially Dr. Kevin Nowka, for explaining the oper-

ation of the circuits used in guTS.

## References

- [1] D. Van Campenhout, T. Mudge, and K. Sakallah. Timing verification of sequential domino circuits. In *International Conference on Computer-Aided Design*, November 1996.
- [2] E. J. Shriver, D. H. Hall, N. Nassif, N.E. Raham, N.L. Rethman, G. Watt, and J.A. Farrell. Timing verification of the 21254: A 600mhz full-custom microprocessor. In *International Conference on Computer Design*, pages 96–103, October 1998.
- [3] V. Narayanan, B. Chappel, and B. Fleischer. Static timing analysis for self-resetting circuits. In *International Conference on Computer-Aided Design*, November 1996.
- [4] Henrik Hulgaard. *Timing Analysis and Verification of Timed Asynchronous Circuits*. PhD thesis, Department of Computer Science, University of Washington, 1995.
- [5] W. Belluomini and C.J. Myers. Verification of timed systems using POSETs. In *International Conference on Computer Aided Verification*. Springer-Verlag, 1998.
- [6] W. Belluomini and C. J. Myers. Timed event/level structures. In collection of papers from TAU'97.
- [7] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Stanford University, 1995.
- [8] Hao Zheng. Specification and compilation of timed systems. Master's thesis, University of Utah, 1998.
- [9] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, 1989.
- [10] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Transactions on Software Engineering*, 17(3), March 1991.
- [11] T. G. Rokicki. *Representing and Modeling Circuits*. PhD thesis, Stanford University, 1993.
- [12] Wendy Belluomini and Chris J. Myers. Efficient timing analysis algorithms for timed state space exploration. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.
- [13] T. G. Rokicki and C. J. Myers. Automatic verification of timed circuits. In *International Conference on Computer-Aided Verification*, pages 468–480. Springer-Verlag, 1994.
- [14] H. P. Hofstee, S. H. Dhong, D. Meltzer, K. J. Nowka, J. A. Silberman, J. L. Burns, S. D. Posluszny, and O. Takahashi. Designing for a Gigahertz. *IEEE MICRO*, May-June 1998.
- [15] J. Silberman et al. A 1.0 GHz single issue 64-bit PowerPC integer processor. *IEEE Journal of Solid-State Circuits*, page accepted for publication, November 1998.
- [16] K. Nowka, T. Galambos, and S. Dhong. Circuit design techniques for a Gigahertz integer microprocessor. In *International Conference on Computer Design*, October 1998.
- [17] Terry I. Chappell, Barbara A. Chappell, Stanley E. Schuster, J.W. Allan, S.P. Klepner, R.V. Joshi, and R.L. Franch. A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture. *IEEE Journal of Solid-State Circuits*, 26(11):1577–1585, November 1991.
- [18] S. Posluszny et al. Design methodology for a 1.0 GHz microprocessor. In *International Conference on Computer Design*, 1998.
- [19] H. R. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical report, Harvard University, July 1989.
- [20] P.M. Kogge and H.S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence relations. *IEEE Transactions on Computers*, pages 786–793, August 1973.
- [21] Y. Kukimoto and R. K. Brayton. Delay characterization of combinational modules. In *International Workshop on Logic Synthesis*, 1998.