Automatic Synthesis of Fast Compact Self-Timed Control Circuits

Bill Coates
Al Davis
Kenneth S. Stevens*

Hewlett-Packard Laboratories
Palo Alto, CA USA

*Department of Computer Science
University Of Calgary
Calgary, Alberta T2N 1N4
Canada

ABSTRACT

We present a tool called MEAT which has been designed to automatically synthesize transistor level, CMOS, self-timed control circuits. MEAT has been used to specify and synthesize self-timed circuits for a fully self-timed 300,000 transistor communication coprocessor. The design is specified using finite state machines which permit *burst-mode* inputs. Burst-mode is a limited form of MIC (multiple input change) signalling. The primary goal of MEAT is to produce fast and compact circuits. In order to achieve this goal, MEAT implementations permit timing assumptions which can be verifiably supported at the physical implementation level, and result in significant improvements in speed and area of the design. Since MEAT has been used for large designs, we have also been forced to make the algorithms efficient. The result is a tool which is efficient, easy to use by today's hardware designers since the specification is based on the commonly used finite state machine control model, and synthesizes CMOS transistor implementations that are self-timed, fast and compact. The paper presents a description of the tool, the nature of the algorithms used, and examples of its use.

1 Introduction

Three major constraints – speed of operation, size, and design time must be considered with any large chip design, be it a commercial product or a laboratory prototype. As integrated circuit technology improves, the amount of logic that can be placed on a VLSI circuit increases quadratically, and the speed of the components become faster linearly. Without improvement in design methodology, the design time of circuits will increase *at least* quadratically. In addition, synchronous design techniques are facing critical design and performance difficulties as circuit complexity escalates due to a number of primary factors:

- It is becoming extremely costly to manage quadratically narrowing clock skew requirements.
- An increasingly disproportionate amount of power must be budgeted to the global clock lines. For example, the DEC Alpha CPU [7] uses 17 watts of the chip's massive 30 watt power budget to drive the clock.
- Incremental performance improvements in a synchronous design are extremely costly due to the global nature of the synchronous timing model.

One clear alternative is to adopt an asynchronous design style. Many proponents would also argue that asynchronous circuits are inherently faster since they are controlled by locally adaptive timing rather than the usual global worst-case clock frequency constraints. While we believe that this claim has merit, we feel that in general it is misleading. Asynchronous circuits usually require more components to implement the same function. This may result in longer wires, increased area, and reduced performance. Performance is lost in synchronous systems where there is a significant difference between the average and the worst case operation delay since the clock period has to accommodate the worst case. A simple example of this is seen in ripple carry arithmetic circuits, where operational delay is dominated by the carry propagation times and where the average carry only propagates a short distance. In this simple case, it is relatively inexpensive to adopt more sophisticated carry chain circuits in order to narrow the average to worst case difference. When compared to a well tuned synchronous design where this difference is small, a functionally equivalent asynchronous implementation may actually run slightly slower. Hence the performance advantage of asynchronous circuits, while often valid, must be analyzed carefully. In practice the speed of a design is more dependent on the quality of the design and the fabrication process. Our experience has been that, for large designs, it is easier to achieve the necessary quality using an asynchronous style than it is in the synchronous discipline primarily due to the speed and simplicity obtained from localized communication and control. The speed of asynchronous circuits has been demonstrated to be on par with that of synchronous versions[12,14].

The lack of a global clock in asynchronous designs inherently eliminates the clock skew and disproportionate clock power budget problems. The down-side is that asynchronous circuit implementations must be hazard free which inherently requires additional logic and more careful design. Fortunately these issues need to be considered only at the lowest level and therefore do not become intractable concerns as the design becomes complex.

Perhaps the clearest advantage that asynchronous designs have over the synchronous approach is functional modularity. Asynchronous design modules inherently keep time to themselves, whereas the timing model intrinsic to synchronous methods applies globally. The result is that a change to a synchronous module often requires concomitant changes in the other system modules. Asynchronous system modules connect to other system modules through a functional interface which encodes the temporal constraints. These interfaces impose sequencing constraints on the interconnected modules and frees them from the need to operate consistently in a global timing model [17]. Hence a change in some module, which significantly changes the modules performance but not its interface will not require changes in other modules to maintain the functional consistency of the system. As system complexities escalate, the need to produce designs as composable modules becomes mandatory. Design time and design performance have become equally critical success factors. In addition the ability to reuse modules of previous designs can be an important way to save on the design time of subsequent efforts. Reuse of asynchronous modules is extremely simple while adapting synchronous modules to a new global timing model is potentially very costly.

Still, after a half century of synchronous design momentum there is much to inhibit a change to asynchronous design:

1. For board level designs, there is little in the way of an asynchronous component selection.
2. Complex designs require sophisticated electronic CAD tool kits, and these tools have been constructed to support synchronous design styles.
3. There is a huge pool of experienced synchronous designers who have demonstrated their ability to produce complex working systems. A significant style change will be painful.

The first problem can be bypassed if the design is an IC rather than a board. Recently there have been several attempts which, like MEAT, represent a start on a solution to the second problem. The third problem is significant and will likely take considerable time to solve completely but today's hardware designers are relying on increasingly sophisticated synthesis tools in their CAD suite to produce/synthesize the implementation from a design specification. If an asynchronous synthesis tool permitted a similar specification style then the change would be less traumatic. This has been our approach with MEAT. Hardware designers are used to thinking in terms of finite state machines for control and separate datapath formulation as their design entry specification. It is our view that asynchronous and synchronous datapath design techniques are quite similar, whereas the implementation strategy for the asynchronous control components, i.e. the finite state machines, must satisfy some additional constraints. Most of this additional burden is handled by the MEAT tools. The result is that an experienced synchronous circuit designer will notice a very minor conceptual shift in order to use MEAT in the creation of an asynchronous design.

MEAT is by no means complete or totally original. MEAT is best viewed as an ensemble of methods for asynchronous finite state machine synthesis, many of which were created elsewhere but modified to suit our needs in the implementation of MEAT. Hence the name MEAT for *Modified Ensemble Asynchronous Tool*.

All asynchronous design styles are fundamentally concerned with the synthesis of hazard free circuits. To avoid subsequent confusion, we use the following terms:

- *Self-timed* and *asynchronous* are general terms used to describe any circuit that is not synchronous and therefore exhibits hazard free behavior under some conditions. We use them synonymously.
- *Delay-insensitive* circuits exhibit hazard free behavior with arbitrary delays assigned to both wires and gates.
- *Speed-independent* circuits exhibit hazard free behavior with arbitrary gate delays but assume zero delay wires.

There are a large number of rather different design styles in today's asynchronous design community. One partition of design styles can be based on the type of asynchronous circuit target: locally clocked [23,11,6], delay-insensitive [15,2,33,20], or various forms of *single-* and *multiple-* input change circuits [31]. Yet another distinction could be made on the nature of the control specification: graph based [24,18,34,4], programming language based [15,2,33,1], or finite state machine based [23,11]. For the finite state machine based styles, there is a further distinction that can be made based on the method by which state variables are assigned [13,29]. The design style space is large and each design style has its own set of merits and demerits. It is worthwhile to note that virtually all of the design styles focus on the design of the control path of the circuit since there is little to distinguish the asynchronous and synchronous datapath design styles.

The methods which produce delay-insensitive circuits, while not perfect [16], are the most tolerant of variations in device and wire delays. This tolerance improves the probability that a properly designed circuit will continue to function under variations in supply voltage, temperature, and process parameters. We chose to slightly expand the domain of timing assumptions which must remain valid to retain hazard free implementation since this permits higher performance implementations at the expense of reduced operational tolerance. Our view is motivated by the reality that our designs have to meet certain performance requirements. For any given layout and fabrication process, we have models which predict the speeds of the wires and transistors for the desired operational window. We also know the percentage of error that can be tolerated in those predictions. We could not live with arbitrary delays for performance reasons and therefore it seems impractical to assume arbitrary delays in order to ensure hazard free operation of the circuits. The approach taken in MEAT has therefore been to insure hazard free operation under sets of timing assumptions that can be verified as being within acceptable windows of fabrication and operational tolerance.

Compiled implementations based on programming language like specifications [15,2,33,1], while elegant and robust, suffer in performance because they are presently compiled into intermediate library modules rather than into optimized transistor networks. The module of greatest concern is the C-element. C-elements are common circuit modules in asynchronous circuits and eliminating them completely is unlikely. However it has been our experience over the past decade that C-elements are similar to the proverbial GOTO statements in programming languages, i.e. too many of them are indications of serious trouble. C-elements are stylized latches and as such are synchronization points. Too much synchronization reduces parallelism and performance. Our design style does not use C-elements for finite state machine implementations, although our designs do use C-elements sparingly in interface circuits such as arbiters.

In order to achieve the necessary hazard free asynchronous finite state machine (**AFSM**) implementation, it is necessary to place constraints on how their inputs are allowed to change. The most common is the *single input change* or SIC constraint [31]. SIC circuits inherently require state transitions after each input variable transition. In cases where the next interesting behavior is in response to multiple input changes, the circuit response will be artificially slow, either due to too many state transitions or due to the external arbiters required to sequence the multiple inputs. *Multiple input change* or MIC circuit design methods have been developed [31,5] but either required input restrictions or implementation techniques that were unsuitable for our purposes. As a result we developed a design style that we call **burst-mode** which permits a certain style of multiple input change. Our burst-mode implementation method does not require performance inhibiting local clock generation or flip-flops.

During the development of MEAT, we were fortunate to have Steve Nowick, a member of David Dill's Stanford University research group, spend two summers with us. He incorporated David Dill's verifier [8] into the tool kit, and modified the verifier to accommodate our burst-mode timing model and our timing assumption based, performance oriented design style. Subsequently the HP and Stanford efforts have had a substantial coupling. In particular, the burst-mode influence can be seen in the work of Ken Yun and Steve Nowick [23,22]. The more theoretically oriented Stanford work has pointed out some serious oversights in our early MEAT algorithms and has influenced our approach to hazard removal.

This paper presents the MEAT synthesis tool, which has proven its ability to greatly reduce design time while also generating compact, high-performance, self-timed circuits. MEAT allows the designer to specify the logical operation of asynchronous control components as a finite state machine.

MEAT synthesizes a verifiably correct, hazard free implementation of the design to produce a *complex gate* CMOS transistor level schematic. A complex gate is a fully complementary CMOS function which implements the sum of products equations that describe the implementation. MEAT has been used to develop a control intensive multicomputer communication chip called the *Post Office* [27]. The Post Office contains 300,000 transistors and has an area of 11 $\times$ 8.3 mm in the 1.2 micron MOSIS CMOS process.

The remainder of the paper describes the nature of the design specification, the MEAT algorithms, and presents several design issues that are exemplar of our design style.

2 MEAT - a Tool for Control Circuit Synthesis

The MEAT tools are fast enough that alternative design options can be explored. The designer is freed from the task of understanding the underlying transformations required to produce hazard-free asynchronous circuits. Asynchronous circuits are specified for MEAT as a burst-mode Mealy state machine. This style of specification provides a powerful way to encapsulate concurrency, communication, and synchronization in an accurate and easily understood form. The input specification is compiled into a set of CMOS complex gate. The result is an implementation which is efficient both in terms of speed and area.

State flow diagrams are used to model the behavior of state machines implemented using MEAT. They provide an intuitive method for defining control functionality, and are similar to the flow charts and state diagrams that are commonly taught in multiple disciplines today[9]. A finite state machine is modeled as a directed graph, where the nodes represent states and arcs represent transitions between states. Each arc is labelled with the set of input firings which trigger the transition and an associated set of output firings. These state diagrams can easily represent parallelism and synchronization, and are reasonably compact when compared to other graphical specification methods.

MEAT state diagrams allow a constrained form of MIC operation, which we refer to as **burst-mode**. When a state change is triggered by a conjunction of input signal transitions (an input burst), these signals are allowed to change in any order and at any time. Allowing MIC operation simplifies the definition of synchronization operations and tends to more closely match the designer's mental model of the hardware. Presently MEAT does not contain a state graph editor so a textual specification format is used. The more natural graphical state machine description may be trivially mapped to the textual version: each arc in the state diagram is mapped to a single statement in the text file, which indicates the source and destination states along with the associated input and output bursts.

Burst-mode state diagrams are reasonably compact when compared to petri-nets, m-nets, STG's, and other graphical representations. These diagrams work well for transition (2 cycle) or level-mode (4 cycle) signalling protocols. Figure 1 shows an example of an STG (a), enhanced STG (b), and burst-mode state diagram (c) for an asynchronous D flip-flop. In this paper we assume positive logic, hence $a\uparrow$ corresponds to a high transition on signal a. In the textual version $a\uparrow$ is represented simply as a and $a\downarrow$ as $a\sim$. The corresponding textual entry version for MEAT is:

```
:fsm Asynch-Flip-Flop ;name of FSM for documentation.

:in  (D Clk) ;list of input variables.

:out (Q) ;list of output variables.

:init-in  ()   ;initial value of inputs, default zero.

:init-out ()  ;initial value of outputs, default zero.
```

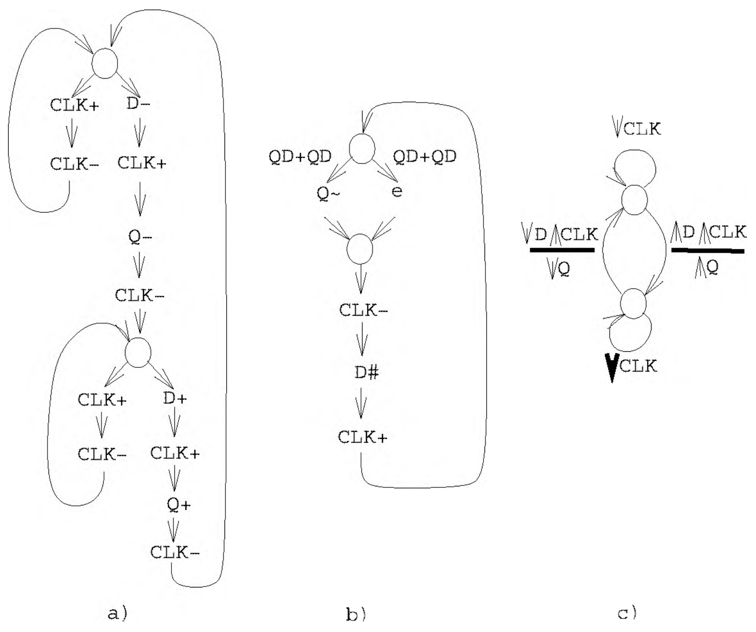

Figure 1: Sample Flip-Flop Specifications

```
:init-state 0 ;initial state, default is State 0.

:state 0 (Clk~) 0 () ;specification of state transitions:

:state 0 (D * Clk) 1 (Q) ;  format is <current state> <input burst>

:state 1 (Clk~) 1 () ;            <next state> <output burst>

:state 1 (D~ * Clk) 0 (Q~)
```

The first automated task performed by MEAT is to generate a primitive flow table [31] from the textual FSM specification. This is a two-dimensional array structure which captures, in a more detailed form, the behavior represented by the state diagram. Each row of this table represents a node in the state diagram; each column represents a unique combination of input signals. Each entry in the table thus represents a position in the possible state space of the FSM.

For each entry, the value of the output signals and the desired next state may be specified. If a next-state value is the same as that of the current row, the state machine is said to be in a *stable state*. If the next-state value specifies a different row, the table entry represents an *unstable state*. A simple way of understanding the flow table is to note that horizontal movement within a row represents changes in the values of input signals, while vertical movement within a column represents a *state transition*. All of our specifications are given in *normal form*, that is, each unstable entry in the table must lead directly to a stable state.

Each allowed input burst will result in a particular path through the FSM state-space, starting at the stable entry where the burst begins. Other entries in the same row may be visited during the course of the input burst. In order for MIC behavior to be correctly represented, it must be guaranteed that the circuit will remain stable in the initial row until the input burst is complete. This is an important point and is a cornerstone of the burst-mode methodology. In essence, any minterm formed from input variables which can be reached during the course of an input burst must be *covered* by a stable entry in the flow table. The minterm defined by the completion of the burst will correspond to an unstable state which will cause a transition to the target row and fire the output burst.

The output burst, if any, may occur concurrently with the state change, or can be constrained to happen *after* the state change has occurred. To allow the flexibility for the later synthesis stages to choose either option, signals in the output burst are labeled as don't cares in the unstable exit state of the flow table. Since all state transitions are STT *Single Transition Time*, the monotonicity of output voltage changes is guaranteed, regardless of whether the value of a given transitioning output in an unstable entry is mapped to logic level zero or one.

Any entry in the flow table not reachable by any allowed sequence of input bursts is labeled as a *don't care* and can take on any value for the outputs or next-state values. As in the case of output bursts discussed above, it is not immediately evident which values will lead to the simplest circuit. Therefore, the assignment of specific values to the don't care entries is deferred for as long as possible. The inclusion of these don't cares can significantly simplify state reduction and boolean minimization, and also lead to more compact circuits.

The next step in the design process is to attempt to reduce the number of rows in the flow table by merging selected sets of two or more rows into one while retaining the specified behavior. This involves first calculating the set of *maximal compatible* states. The set of maximal compatibles consists of the largest sets of state rows which can be merged, which are not subsets of any other such set. There may be various valid combinations of the maximal compatibles that can be chosen to produce a reduced table with the same behavior.

This is essentially the well-known state-reduction problem; unfortunately complications are introduced due to the MIC nature of the input bursts. "Traditional" methods normally apply only to SIC circuits, and when used for our burst-mode specifications may produce hazards in the final implementation.

Nowick et. al.[21] have developed the modifications necessary to the state-reduction and subsequent synthesis steps to guarantee that the resulting implementation will be hazard-free under burst-mode conditions. These modifications are not presently incorporated into MEAT. Currently we use a verifier [8] on the synthesized implementation. The verifier has been modified to operate with explicit timing assumptions. Hazards detected in the implementation are then reviewed to see if the circuit would exhibit correct behavior under reasonable delay assumptions. If these assumptions fall within acceptable bounds of fabrication and operational constraints then the timing assumption is entered into the verifier. If an unacceptable assumption is required then the circuit is fixed either by manual repair or by modifying the state-machine specification. The manual repair usually involves the addition of appropriate inverter chain to delay the race critical path.

The final choice of minimized states is an example of the *binate covering* problem. There are three constraints on this choice. First, and obviously, only compatible states may combined (*compatibility* constraint). Second, each state in the original design must be contained in at least one of the reduced states (*completeness* constraint). Third, selecting certain sets of states to be merged may imply that other states must also be merged (*closure* constraint). Grasselli and Luccio [10] have developed a tabular method for determining a closed cover of states, which is also in the process of being incorporated into MEAT. At present, MEAT requires the user to manually determine and enter a state covering. If any of the necessary constraints are not satisfied, MEAT will inform the user that the covering is invalid.

A new flow table representing the behavior of the minimized FSM is then generated by merging the specified rows of the original flow table. It should be noted that it is not always true that minimizing the number of states will simplify the hardware or increase performance. However, a reduced state machine can result in fewer state variables which in most cases does indeed result in a smaller and faster implementation.

A set of state variables must then be assigned to uniquely identify each row of the reduced flow table. These state variables are used as feedback signals in the final circuit. In contrast to synchronous control logic design, state codes may not be randomly assigned, but must be carefully chosen to prevent races. The MEAT state assignment algorithm is based on a method developed by Tracey[29]. The Tracey algorithm has the advantage that it produces STT state assignments which minimizes delay in the implementation. In cases where two or more state variables must change value when transitioning to a new state, all variables involved are allowed to change concurrently, or *race*. It must be guaranteed that the outcome of the race is independent of the order in which the state variables actually transition in order to produce a *non-critical* race which exhibits correct asynchronous operation. Several valid assignments may be produced, and each will be passed to the next stage for evaluation. Each state assignment will result in a unique implementation.

After state codes are assigned, the next synthesis stage computes a canonical sum of products boolean expression for each output and state variable. A modified Quine-McCluskey minimization algorithm is used. The resulting expression includes all essential prime implicants, and possibly other prime implicants and additional terms necessary to produce a covering free of logic hazards. It may be possible for each output or state variable to be specified using several alternate minimal equations. The large number of don't care entries typically present in the flow table causes the standard algorithm to be rather inefficient and increases the likelihood that more than one minimal expression will be found. The MEAT implementation contains optimizations for don't care dominant functions. Each possible solution is given a heuristic "weight" that indicates the expected speed and area cost of implementation using complex CMOS gates. When multiple state assignments have been produced in the previous step, the total weight of each unique SOP (sum of products) equation is then used to choose between the various instantiations.

The minimized equations produced in the previous step are then used to automatically generate transistor netlists, suitable for simulation, representing complex CMOS gates. An interface to the Electric[25] design system is used to automatically produce a schematic diagram to help guide the layout process, which unfortunately has not yet been automated. The complementary nature of CMOS n-type and p-type devices is exploited to generate a single, complex, static gate through simple function preserving transformations. These transformations can increase performance while reducing the area and device count. As a SOP equation is *folded* into a complex gate, the number of logic levels required to generate the output can be reduced. If the function is too large to be implemented as a single module, it can easily be broken up into a tree of complex gates with 2 or more logic levels, but better overall performance[28]. Typical state machine implementations have response times between 3 and 5 2-input NAND gate delays.

Our complex gate design generates negative logic outputs (low voltage levels for asserted signals). A convention of positive logic levels is assumed for all signals external to the state machine, requiring that the outputs be inverted. This is a feature for performance reasons as the gain of the inverter can be used as a driver to increase signal strength and reduce rise and fall times. When outputs need to drive a large load, a buffer tree can be used.

All state machines also require a reset signal to place the storage logic into the correct initial state. Storage in these state machines is implemented via the state variables. If a single complex gate is used to generate the output, the state storage is reset by NOR-ing the output with the reset line. For complex gate trees, a resetable NAND gate is used. Although the performance of the NOR gate is not optimal, the load on the feedback lines is local to the state machine and typically small so a large gain is not required.

3 Design Issues and Examples

Figure 1 essentially shows how AFSM designs are specified using MEAT. Rather than presenting a series of more complex designs which will show roughly the same thing, we will present a number of design vignettes which illustrate interesting points in the design space, and an example of MEAT usage.

3.1 A Story about C-element Design

During a 1986 course on asynchronous circuits taught by Ivan Sutherland and Bob Sproull, the discussion turned to the design of the common C-element. At that time, the standard C-element consisted of a 2-high stack, followed by an inverter. This element had the problem that it was a *dynamic* gate. If the two inputs remained at different voltage levels for long enough the C-element's state would be lost and cause an invalid output transition. This was clearly unacceptable for general asynchronous applications. Static versions of the circuit were created by including a weak "trickle charge" inverter to maintain correct voltage on the internal node in the absence of it being directly driven by the 2-high stack.

The trickle charge inverter was a problem for several reasons. First, it reduced the performance of the circuit. When the internal node $\overline{c}$ (in Figure 2a) needed to be flipped to a different voltage, the trickle inverter would be actively driving the circuit one way, while the 2-high stack was actively driving it another way. The 2-high stack needed to charge the node, as well as dissipate the current supplied from the trickle inverter. This caused increased power consumption due to the existence of a DC path between the power rails during a state change. Secondly, the inherent gain of an inverter is greater than the gain of a 2-high stack. This design requires the 2-high stack to overpower the inverter to flip the state of the device. Unless the drive of the 2-high stack is significantly greater than the inverter, the node becomes susceptible to noise problems which could result in hazards. This gain difference can only be overcome by reducing the size of the inverter and increasing the size of the 2-high stack. Hence the sizing of the components becomes critical. Increasing the size of the 2-high stack slows the circuit by requiring additional input drive. Decreasing the width and increasing the length of the inverter reduces the reliability of the inverter and the portability to other processes.

After the day's discussion, we spent several hours attempting to come up with a better C-element design which eliminated the trickle inverter, yet did not add significant complexity to the component. Ultimately a design was found which was compact and efficient. This design has been widely used in a number of sites. This design required 4 more transistors than the trickle charge design. However, the 2-high stack could be of optimally sized transistors and there was no fight to drive the internal node $\overline{c}$. Although this circuit was larger, and the inputs drive twice the number of devices, it was significantly faster than the original design and avoided the power consumption, noise, portability, and function problems of the old design.

Several years later, curiosity lead us to see what MEAT would produce for a C-element. The exact same circuit was produced from MEAT in an instant. MEAT generated equations for the circuit shown in Figure 2b and the back-end schematic generated the equivalent but optimized version shown in Figure 2c.

3.2 Using Burst-Mode to Increase Performance

Burst-mode assumes that inputs and outputs are generated as discreet sets, or bursts. In general, this violates delay-insensitive and speed-independent assumptions. For example, assume that an input burst has completed, and the resulting output burst causes several outputs to be generated. One of the outputs

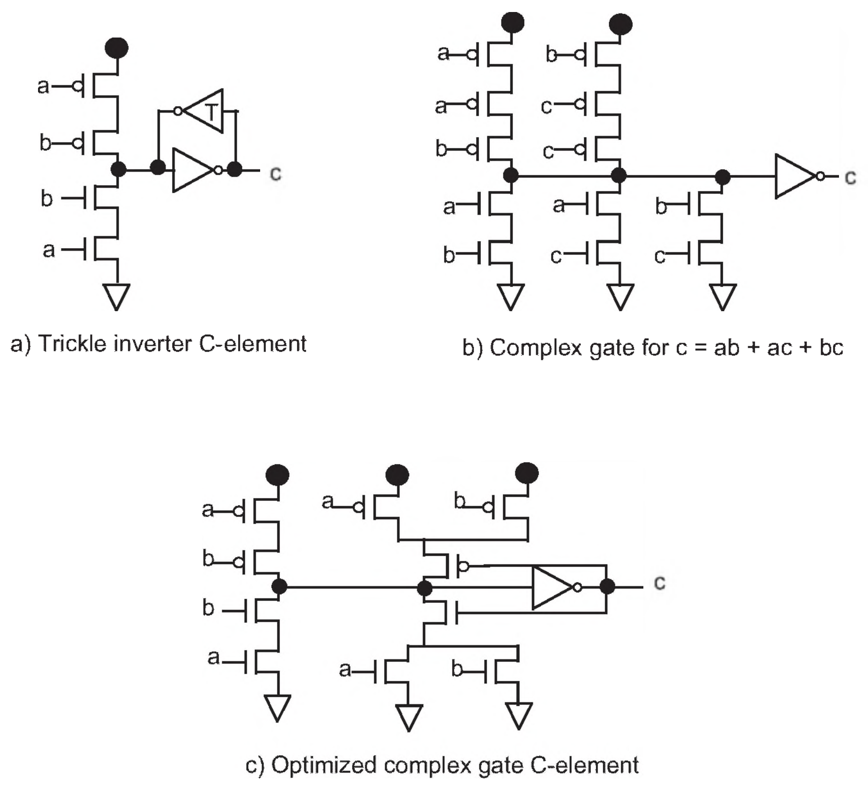
a) Trickle inverter C-element
b) Complex gate for c = ab + ac + bc
c) Optimized complex gate C-element
a
b
c
T

Figure 2: C-Elements, Hand Optimized Matched by MEAT

could be generated before the others. This output can be received by a destination module which could in turn generate an output which is fed back as an input to the original module even *before* the rest of the outputs have been generated. This violates burst-mode operation as the next input burst has occurred before the previous output burst has completed. Burst-mode assumes that all outputs in the burst must be generated before the environment can respond to the output burst or computation interference may occur. The cases where computation interference can occur can be flagged and checked by circuit timing analysis.

MEAT's burst-mode MIC model is similar to the fundamental mode assumptions for traditional SIC AFSM designs. Namely we assumes that once an input burst has arrived the AFSM will settle in a stable state before the next burst can arrive. If this assumption cannot be met then external arbitration will be required to enforce the assumption.

If an input burst changes an internal state variable, speed-independent operation will generally require the state variable to stabilize before the output can be changed. Performance can be improved if outputs can change concurrently with state changes. MEAT accomplishes this by making the transitioning output a don't care in the unstable exit point of a row in the flow table. This places a priority on logic minimization, but usually will produce a circuit which can generate an output concurrent with state changes. The fundamental mode assumption guarantees that the AFSM is ready to accept the next input burst when it arrives, as the state variable transition has completed and the logic has stabilized. Unger has shown that it is possible to weaken this fundamental mode assumption [30], although his method is not presently incorporated into MEAT.

3.3 When Speed-Independent Circuits Fail: The Isochronous Fork

Ideally all asynchronous circuits should be designed as delay-insensitive modules. However, performance requirements may force one to make weakening assumptions about circuit behavior. Many of these assumptions are realistic, as physical devices and wires do not require unbounded delays to generate and propagate signals. However, care must be used to assure that the circuit complies to these assumptions under all operating conditions or the design will be unsafe and costly failures may occur.

Simplifying assumptions are best exploited when they are constrained to a fixed extent physical domain as is the case with AFSM modules. Hierarchical composition of these modules can then proceed conforming to delay-insensitive rules since all of the external interfaces should be designed avoid timing assumptions. Inside an AFSM, the relative delay of wires and gates can be more easily controlled, analyzed, and modified as the constraints are all local. When these timing assumptions apply outside an individual module then the entire system must be analyzed to assure compliance with the timing assumption set. At this point there is little to distinguish the circuit from a synchronous one.

A common performance and synthesis assumption made by many asynchronous circuit designers is that of speed-independence. The assumption that wire delay is zero leads to the **isochronous fork assumption**. This implies that multiple devices driven by a single component react to the signal change at approximately the same time. This model works well for situations where the transistors are slow and the paths are fast. Unfortunately this model becomes less valid as IC technology progresses and is certainly suspect even today.

Furthermore, whenever the rise or fall time of an isochronous fork is greater than the switching delay of any physical device, failure may occur due to variances in switching thresholds. Noise, long wires, and high-capacitance paths exacerbate the problem. Within a particular AFSM module, this problem can be managed successfully but between modules it is difficult. Martin [14] and Van Berkel [32] have both described circuit failures due to paths which did not behave in an isochronous fashion. Both failures were the result of using C-elements in module interfaces. C-elements inherently contain an isochronous

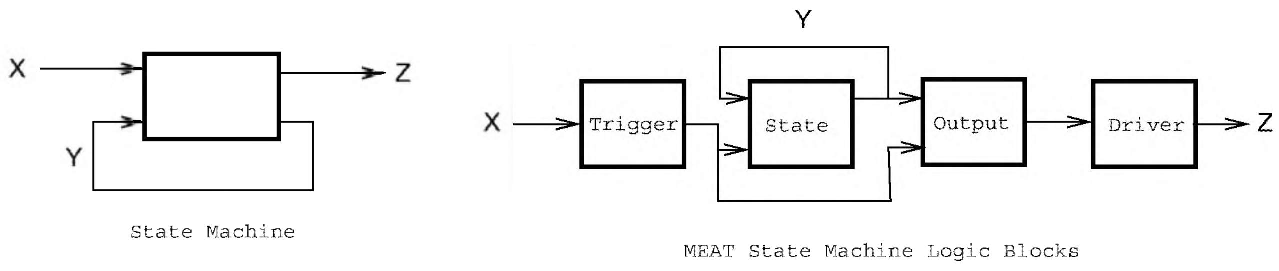

Figure 3: State Machine Generation

fork. Namely the output of the C-element will be an output of the module as well as being fed back locally to maintain the C-element's state.

The philosophy we have used in the MEAT tool and in the design of our circuits is to remove isochronous forks from external interfaces. MEAT state machines are broken into the partitions shown in Figure 3. Our philosophy is that we would rather increase the cost and difficulty of designing modules if it can simplify the composition of systems. Timing assumptions are always easier to analyze and fix in a small, local cell rather than across a series of modules. Systems are hard to design and low-level modules are relatively easy. If by making the module design harder, it becomes easier to do the inherently complex task then the overall difficulty is reduced.

The trigger box has two functions. First, high capacitance inputs (inputs with a slow rise time) will be passed through an inverter or Schmitt trigger. This will reduce the load on the input line, which can increase circuit performance. It also results in crisp rise and fall times of signals internal to the AFSM. Secondly when an unasserted input signal is required by the state or output boxes, the trigger box will invert that signal. Each input will have its inverted and uninverted signal shared among all function blocks in the state machine to eliminate hazards and create a smaller implementation. The isochronous forks created by sharing the inverters are easily controlled within the AFSM domain. Components within a particular AFSM are physically close. Hence wire delays of the internal signals and the trigger box delay are normally insignificant.

The driver block is used to generate positive output voltage levels and to increase the signal strength when the output is heavily loaded. Circuit performance is enhanced since it is sized to drive its output load appropriately. Isochronous forks in MEAT will only exist when a state variable is used directly as an output. In such cases, the output can be buffered by one or two inverters to assure the fork is isolated within the AFSM. While this decreases the performance of the circuit, the module can function in a delay-insensitive manner and can be safely used without analyzing it's load in a broader context.

This design style has been tested continuously over the last five years. We have designed several large asynchronous circuits which have generally worked the first time, merely using simulators to verify correct composition of the modules. The result of this experience has led to a high confidence factor in the method.

3.4 An AFSM example

In order to illustrate exactly what MEAT does, we will transcribe an actual synthesis run using MEAT to create a Post Office state machine called the SBUF-SEND-CTL. The behavior is initially specified as a burst-mode AFSM as shown in Figure 4. This example is taken from the suite of Post Office state machines publicly available for use by other researchers [26,23].

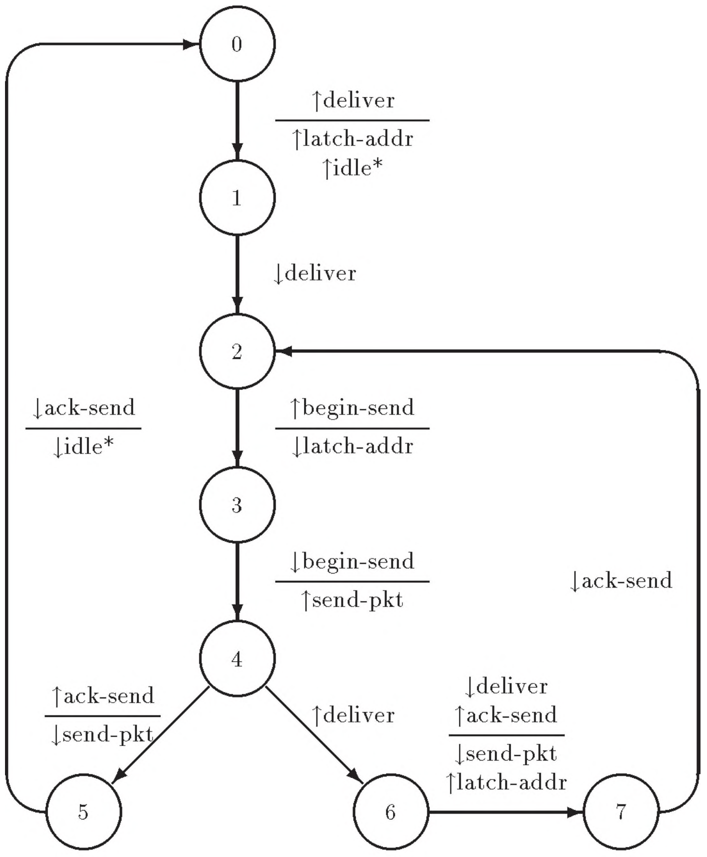

Figure 4: Sbuf-send-ctl State Machine

The specification of sbuf-send-ctl from Figure 4 is textually entered for MEAT as follows:

```
:fsm sbuf-send-ctl
:in  (Deliver Begin-Send Ack-Send)              ;list of input variables
:out (Latch-Addr IdleBAR Send-Pkt)              ;list of output variables
:state  0 (Deliver)
        1 (IdleBAR * Latch-Addr)
:state  1 (Deliver~)
        2 ()
:state  2 (Begin-Send)
        3 (Latch-Addr~)
:state  3 (Begin-Send~)
        4 (Send-Pkt)
:state  4 (Ack-Send)
        5 (Send-Pkt~)
:state  5 (Ack-Send~)
        0 (IdleBAR~)
:state  4 (Deliver)
        6 ()
:state  6 (Deliver~ * Ack-Send)
        7 (Send-Pkt~ * Latch-Addr)
:state  7 (Ack-Send~)
        2 ()
```

The following is a transcript from a MEAT session. The specification resulted in a single implementation with two state variables.

```
> (meat "sbuf-send-ctl.data")

Max Compatibles: ((0 5) (1 2 7) (3 4) (6))
Enter State set: '((0 5) (1 2 7) (3 4) (6))

SOP for "Y1":
  18: DELIVER + Y1*BEGIN-SEND~
SOP for "Y0":
  28: BEGIN-SEND + Y0*ACK-SEND~ + Y0*DELIVER
SOP for LATCH-ADDR:
  12: Y1*Y0~
SOP for IDLEBAR:
  30: ACK-SEND + BEGIN-SEND + Y0 + Y1
SOP for SEND-PKT:
  12: Y0*BEGIN-SEND~
 HEURISTIC TOTAL FOR THIS ASSIGNMENT: 100
```

The implementation can then be verified for hazard-free operation by the verifier. The verifier reads the specification and implementation. For this example, the state variables and outputs generated by MEAT are implemented as two-level AND/OR logic. Each signal is generated independently of the others. Only direct inputs are shared, so the same inverted signal in different output logic blocks will use separate inverters. Separate inverters will result in verification errors in the burst-mode speed-independent analysis. In this example, the $\overline{\textit{begin-send}}$ signal is shared by *Y1* and *send-pkt*. The two inverters are merged and the output is forked to both logic blocks. This implementation is then verified.

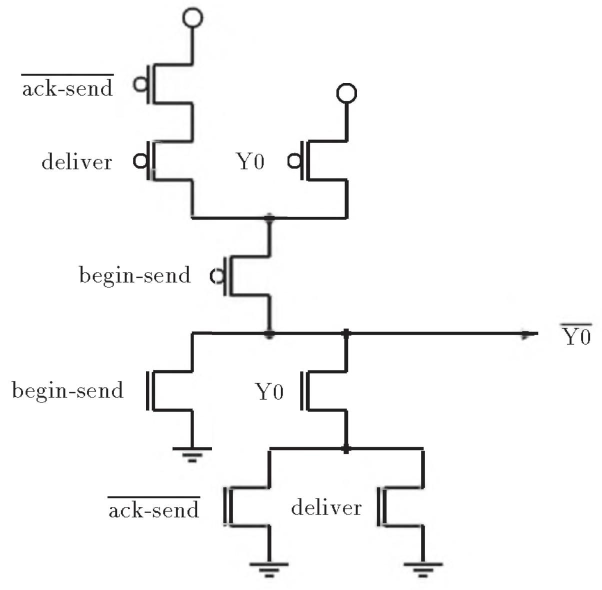

Figure 5: Complex CMOS Gate for sbuf-send-ctl Y0

The verifier points out a d-trio hazard [31] which is removed by adding an inverter to change the sequencing of begin-send into the *Y0* logic. The implementation is then verified as hazard free as follows:

```
> (verifier-read-fsm "sbuf-send-ctl.data")

Max Compatibles: ((0 5) (1 2 7) (3 4) (6))
Enter State set: '((0 5) (1 2 7) (3 4) (6))

> (setq *impl* (merge-gates '(1 11) *impl*))
> (verify-module *impl* *spec*)
10 20 30 40 50
Error:  Implementation produces illegal output.

> (setq *impl* (connect-inverter 10 6 *impl*))
> (verify-module *impl* *spec*)
10 20 30 40 50 60 70 79 states.
T
```

The canonical SOP equations generated by MEAT are then transformed into complex gates for implementation. The CMOS circuit for *Y0* is shown in Figure 5. The complex gates are then manually implemented using the Electric [25] layout editor. The physical layout is then simulated with COSMOS [3] to check for layout errors. Cooperating sets of state machine cells are interconnected to form larger modules, integrating clocked datapath logic when necessary.

3.5 D-Trio Hazards, Assumptions, and Possible Elimination

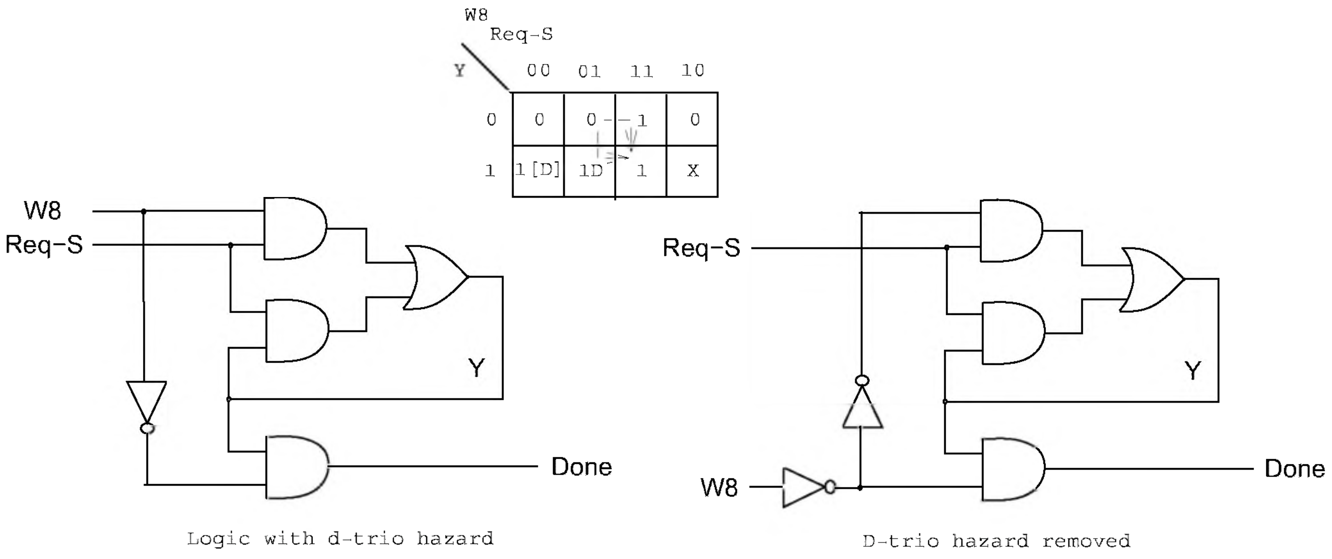

Figure 6: Hazard removal from "Sendr-Done" state machine

Figure 6 shows a static **d-trio** or nonessential function hazard which is found in some of the state machines produced by MEAT. D-trio hazards are fundamental and cannot be removed in every case, but they will be detected by the verifier In this cases the hazard occurs because the input burst resulted in an internal state change while the output burst contained no transition for the *Done* signal. The d-trio hazard in this example can produce a static 1-hazard on the *Done* signal. The input burst is perceived by the *Done* output logic *after* the state change burst thereby creating the hazard.

The $W8$ signal of the logic with the d-trio also contains an isochronous fork. If we ignore the potential threshold deviations then timing analysis shows that the physical behavior will not exhibit the hazard. However, this circuit cannot be included in a system without analyzing the driver, load, and stray capacitance on the $W8$ input or errors will result.

By modifying the trigger logic in the Sendr-Done state machine shown in Figure 6, we can both eliminate the d-trio hazard and the external isochronous fork. This incurs *no* performance penalty. The $\overline{W8}$ signal to the *Done* logic remains delayed by a single inverter, while the $W8$ signal to the state logic becomes double inverted rather than fed directly into the logic from the input.

The double inversion has the effect enforcing correct *sequencing* of the order of arrival of the $\overline{W8}$ signal to the *Done* logic. Transitions on $\overline{W8}$ will always be perceived by the *Done* logic before changes in the state variable, resulting in hazard-free circuit operation. Transitions are ordered such that the assertion of the state variable is not critical to the performance of the circuit, so the double inversion of $W8$ into the state logic has no deleterious effect.

3.6 When MIC Circuits Cannot Be Designed: The NAKing Arbiter

MEAT state graphs must be unambiguous and deterministic. Nondeterministic behavior inside a state graph is not allowed as it can result in metastability. However, the operation of a state machine may be nondeterministic if a mutual exclusion element (ME) is used to order the arrival of two or more concurrent inputs into the state machine. ME's are analog devices, and are the only external device that may be required to implement control functions using the MEAT methodology. They are easily

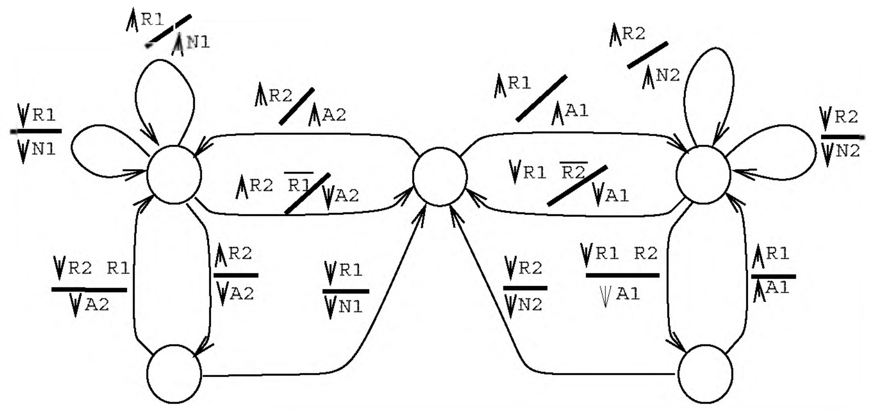

Figure 7: Naking Arbiter SIC State Machine Specification.

fabricated in most VLSI technologies, requiring 12 transistors in CMOS.

When multiple edges exit a single state, there must be at least one pair of mutually exclusive signals for all pairs of edges exiting the state[19]. If there is no pair of mutually exclusive signals for all pairs of edges then the state machine can only operate in *single input change (SIC)* mode for those signals. This has been referred to as the *semi-modularity* property [4].

Arbiters are inherently nondeterministic circuits which cannot be directly implemented as an AFSM. The Naking Arbiter of Figure 7 is an SIC state machine. Since the environment permits the R1 and R2 signals to arrive concurrently these signals pass through a *sequencer* before entering the state machine. A sequencer consists of a set of ME gates, AND gates, and latches, with an input to enable the next transition. Sequencers are nondeterministic and rather expensive to build in terms of size and speed.

4 Summary

The goal in the development of the MEAT tool was to generate fast, compact, efficient circuits. Showing the excellent performance that can be achieved with asynchronous designs is an important part of forwarding this technology to the general circuit design community. While experienced asynchronous designers understand that there are more benefits in the asynchronous approach than speed, it is clear that the dominant metric in evaluating circuit design styles in the commercial arena is performance. Our Post Office design was no exception; as long as the circuit was fast nobody cared how we did it except us. We view this as a sad reality, since it relegates the impact of the conceptual elegance of asynchronous circuits to the academic community.

Building a large, fully self-timed circuit has resulted in many insights. The need for synthesis and analysis tools that compare with those available to the synchronous design community is of primary importance. We hope that MEAT is a step in the direction of attracting more broad based interest. We have publicly offered both the MEAT tool and many of the Post Office state machines to the IC CAD design community in hopes that others will improve on this step. The need for more robust circuit behavior and for higher performance levels is ubiquitous.

MEAT, like any CAD tool, is incomplete. The back-end only produces schematics. Manual layout is prohibitively time consuming. Some form of automatic layout is necessary unless we abandon the complex gate approach in order take advantage of standard cell and technology mapping approaches.

Automatic layout is a difficult task and should also include automatically sized transistors for the performance needs of the design. Using standard cells will result in some lost performance but the synthesis task is easier. We are investigating both options. There are other performance oriented factors that should be included. As a design is passed down through the different stages of MEAT, some information is lost. The complexity of the algorithms and simplicity of the circuits could be enhanced by preserving some of this information. State graphs lack the formalisms required to analyze compositions of these circuits for safety, liveness, deadlock, and other properties. We are currently investigating a process calculus as a means of specifying and generating MEAT state graphs as well as proving correct operation and construction. MEAT also needs to be connected to existing CAD tools. An example is the connection to a timing analyzer so that the timing assumptions can be automatically analyzed for compliance. Since datapath design is similar to that of synchronous designs, we need to integrate the MEAT capability into an existing CAD framework. Presently, too much designer interaction is required to traverse the seams separating MEAT and other pieces of our tool environment.

Approximately a fifth of the Post Office control path design was done manually, and the rest was done using MEAT. The automated part of the design took one-fourth the amount of design time and was virtually error free. Those errors were corrected when Steve Nowick pointed out a flaw in our minimization algorithms. Our design style has proven to be a very natural transition for existing hardware designers, primarily since it is based on traditional finite state machine control. Our synthesis techniques have generated compact high-performance circuits that work, and the complexity of the synthesis algorithms has proven to be viable for large designs.

References

[1] Erik Brunvand and Robert Sproull. Translating Concurrent Programs into Delay-Insensitive Circuits. In *IEEE International Conference on Computer Aided Design: Digest of Technical Papers*, pages 262–265. IEEE Computer Society Press, 1989.

[2] Steven M. Burns and Alain J. Martin. *The Fusion of Hardware Design and Verification*, chapter Synthesis of Self-Timed Circuits by Program Transformation, pages 99–116. Elsevier Science Publishers, 1988.

[3] Carnegie-Mellon University. *User's Guide to COSMOS*.

[4] Tam-Anh Chu. On the models for designing VLSI asynchronous digital systems. Technical Report MIT-LCS-TR-393, MIT, 1987.

[5] Henry Y. H. Chuang and Santanu Das. Synthesis of multiple-input change asynchronous machines using controlled excitation and flip-flops. *IEEE Transactions on Computers*, C-22(12):1103–1109, December 1973.

[6] A. L. Davis. The Architecture of DDM1: A Recursively Structured Data-Driven Machine. Technical Report UUCS-77-113, University of Utah, Computer Science Dept, 1977.

[7] Digital Equipment Corporation., Maynard, MA. *Alpha Architecture Handbook*, 1992.

[8] David Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. An ACM Distinguished Dissertation.* MIT Press, 1989.

[9] William I. Fletcher. *An Engineering Approach to Digital Design*. Prentice-Hall, 1980.

[10] A Grasselli and F. Luccio. A Method for Minimizing the Number of Internal States of Incompletely Specified Sequential Networks. *IEEE TEC*, June 1965.

[11] A. B. Hayes. Stored State Asynchronous Sequential Circuits. *IEEE Transactions on Computers*, C-30(8), August 1981.

[12] A. B. Hayes. Self-Timed IC Design with PPL's. In R. E. Bryant, editor, *Third Caltech Conference on Very Large Scale Integration*, pages 257–274, Rockville, Maryland, 1983. Computer Science Press, Inc.

[13] Lee A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.

[14] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. "The Design of an Asynchronous Microprocessor". In C.L. Seitz, editor, *Advanced Reserach in VLSI: Proceeedings of the Decennial Caltech Conference on VLSI*, pages 351–373. MIT Press, 1989.

[15] Alain Martin. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1(1):226–234, 1986.

[16] Alain Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[17] C. Mead and L. Conway. *Introduction to VLSI Systems.* McGraw-Hill, 1979. Chapter 7.

[18] Teresa Meng. *Synchronization Design for Digital Systems.* Kluwer Academic, 1990.

[19] R.E. Miller. *Switching Theory, II: Sequential circuits and machines.* Wiley, 1965. Chapter 10.

[20] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of Delay-Insensitive Modules. In Henry Fuchs, editor, *Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[21] S. M. Nowick and D. L. Dill. Synthesis of asynchronous state machines using a local clock. In *1991 IEEE International Conference on Computer Design: VLSI in Computers and Processors.* IEEE Computer Society, 1991.

[22] S. M. Nowick, K. Y. Yun, and D. L. Dill. Practical asynchronous controller design. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors.* IEEE Computer Society, 1992.

[23] Steven M. Nowick and David L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *1991 IEEE International Conference on Computer-Aided Design.* IEEE Computer Society, 1991.

[24] S.S. Patil. Coordination of asynchronous events. Technical Report TR-72, MIT Project MAC, June 1970.

[25] Steven M. Rubin. *Computer Aids for VLSI Design.* VLSI Systems. Addison-Wesley, 1987.

[26] L. Lavagno; K. Keutzer; A. Sangiovanni-Vincentelli. Synthesis of Verifiably Hazard-Free Asynchronous Control Circuits. Technical Report UCB/ERL M90/99, Univ. of California at Berkeley, November 1990.

[27] Kenneth S. Stevens, Shane V Robison, and A.L. Davis. "The Post Office – Communication Support for Distributed Ensemble Architectures". In *Proceedings of 6th International Conference on Distributed Computing Systems*, pages 160 – 166, May 1986.

[28] Ivan E. Sutherland and Robert F. Sproull. Logical effort: Designing for speed on the back of an envelope. In Carlo H. Sequin, editor, *Proceedings of the 13th Conference on Advanced Research in VLSI*, pages 1–16. UC Santa Cruz, March 1991.

[29] J. H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15:551–560, August 1966.

[30] S. H. Unger. A Building Block Approach to Unclocked Systems. In *Proceedings of the 26th HICSS Conference*, January 1993. To appear.

[31] S.H. Unger. *Asynchronous sequential switching circuits*. Wiley-Interscience, 1969.

[32] C. H. van Berkel. Beware the Isochronic Fork. Technical Report Nat. Lab Rep. UR 003/91, Philips Research Laboratories, January 1991.

[33] C. H. (Kees) van Berkel. *Handshake circuits: an intermediary between communicating processes and VLSI*. PhD thesis, Technical University of Eindhoven, May 1992.

[34] Peter Vanbekbergen, Francky Catthoor, Gert Goossens, and Hugo De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *International Conference on Computer-Aided Design*. IEEE Computer Society Press, 1990.