An Introduction to Asynchronous Circuit Design

Al Davis[1]
Steven M. Nowick[2]

UUCS-97-013

[1]Computer Science
University of Utah
Salt Lake City, UT 84112

[2]Computer Science
Columbia University
New York, NY 10027

September 19, 1997

Abstract

The purpose of this monograph is to provide both an introduction to field of asynchronous digital circuit design and an overview of the practical state of the art in 1997. In the early days of digital circuit design, little distinction was made between synchronous and asynchronous circuits. However, since the 1960's, the mainstream of the digital circuit design enterprise has been primarily concerned with synchronous circuits. Synchronous circuits may be simply defined as circuits which are sequenced by one or more globally distributed periodic timing signals called clocks. Asynchronous circuits are an inherently larger class of circuits, since there are may sequencing options other than global periodic clock signals. Asynchronous circuits have been studied in one form or another since the early 1950's [92] when the focus was primarily on mechanical relay circuits. A number of theoretical issues were studied in detail by Muller and Bartky as early as 1956 [138]. Since then, the field of asynchronous circuits has gone through a number of high-interest cycles. In recent years there has been an unprecedented level of interest in both academic and industrial settings [81]. Much of this recent research effort has focused more on theory than practice. Nonetheless, the advance of practical asynchronous circuit design techniques also has an unusual level of interest. The focus of this document is on the aspects of the asynchronous circuit design discipline which are either likely, or have been an influence on the practical design of asynchronous circuits. The attempt is to provide an introduction to the basic concepts which provide the foundation for today's design techniques and to summarize the current practice. The text contains an extensive set of bibliographical pointers to guide the more serious student of the field.

1 Introduction

The intent of this report is to provide an introductory yet comprehensive overview of the field of asynchronous circuit design. The focus on design implies that a number of theoretical aspects of the discipline which do not directly affect the practical design process will be ignored. Given the size of the field and the number of design methods, it is impossible to cover all of the various design methods in depth. On the other hand, little could be learned if all of the methods were just mentioned superficially. The result is that there will be enough depth in this report to introduce the basic concepts and to highlight a few of the design styles. Other design methodologies will be covered more cursorily, at the conceptual level. Differences and similarities between methods will be discussed. The many citations and extensive bibliography provide ample direction for an in-depth study of any particular method.

2 Motivation and Basic Concepts

Circuit design styles can be classified into two major categories, namely synchronous and asynchronous. It is worthwhile to note that neither is independent of the other and that there are many designs that have been produced using a hybrid design style which mixes aspects of both categories. Synchronous circuits may be simply defined as circuits which are sequenced by one or more globally distributed periodic timing signals called clocks. Asynchronous circuits are an inherently larger class of circuits, since there are may sequencing options other than global periodic clock signals. It may be difficult to understand the motivation for asynchronous circuit techniques when the bulk of commercial practice and considerable experience, artifact, and momentum exists for the synchronous circuit design style. For some, the motivation to pursue the study of asynchronous circuits is based on the simple fact that they are different. Others find that asynchronous circuits have a particular modular elegance that is amenable to theoretical analysis. However, for those interested in the practical aspects of asynchronous circuit design, the motivation often comes from some concern with the basic nature of synchronous circuits.

Of common concern are the cost issues associated with the global, periodic, and common clock that is the temporal basis for synchronous circuits. The *fixed* clock period of synchronous circuits is chosen as a result of worst-case timing analysis. It is not adaptive and therefore does not take advantage of average- or even best-case computational situations. Asynchronous circuit proponents view this as an opportunity to achieve increased performance since asynchronous methods are inherently adaptive. Arithmetic circuits provide a good example. Arithmetic circuit performance is typically dominated by the propagation delay of carry or borrow signals. The worst-case propagation situation rarely occurs, yet synchronous arithmetic circuits must be clocked in a manner that accommodates this rare worst-case condition. Some asynchronous circuit designers have made the mistake of generalizing this observation into a view that the inherent adaptivity of asynchronous circuits implies that they are capable of achieving higher performance in general. However, this is not necessarily the case.

All asynchronous circuits have additional operational constraints when compared to their synchronous counterparts. Ideally, digital signals represent binary values and therefore model 2 distinct voltage levels. For convenience let them be called 0 and 1. These signals then have the possibility of either remaining constant or changing as a result of a circuit action or *event*. When signals change, the change may not be a monotonic transition between one voltage level and the other. Such a non-monotonic change is often called a *glitch.* A circuit producing an output which may glitch is said to contain a *hazard.* There are many types of hazards which are discussed in Section 4.3. A glitch on a clock signal of a synchronous circuit will typically cause the circuit to malfunction. Glitches on non-clock signals do not cause a malfunction as long as the signal is stable at its new value for a certain time before and after a clock signal transition. This glitch issue is both the advantage and the disadvantage of synchronous circuits. It implies that non-clock signals need not be designed to be hazard free which often times results in smaller circuits. However the clock must be carefully controlled and since it must be globally distributed, this often proves to be difficult. All forms of asynchronous circuits are concerned with providing hazard- or *glitch-free* outputs under some timing model.

In order to achieve hazard-free behavior, an asynchronous circuit will often contain more gates than a functionally equivalent synchronous circuit. Therefore in terms of the number of basic components, asynchronous circuits are often somewhat larger than synchronous circuits. More gates implies more wires, and this may result in slower rather than faster circuit latencies. Furthermore in order to achieve their inherently adaptive nature, asynchronous circuits must *explicitly* generate sequence control signals such as a request and an acknowledge signal. The request signal can be used to signal initiation of some action and the corresponding acknowledge signal indicates completion of that action. In synchronous circuits much of this type of control signaling is *implicit* in the common clock signal. The generation of these explicit control signals further exacerbates the complexity of asynchronous circuits, and may lead to a further performance degradation.

The adaptive potential remains where the worst-case situation is rare and when the difference between the worst-case and average-case latencies is significant. However, synchronous circuit designers are also well aware of this situation and take considerable care to create a clock model and circuit structure that can take advantage of these differences. The most notable example of this tactic is in the finely-grained pipeline structures of modern floating point units. Yet, for very large circuits, such as microprocessors, balancing all the timing constraints of a large computational space to minimize the difference between the worst and average case timing models is a difficult task. The work by Mark Dean on the STRiP processor [54] provides an interesting example. Dean showed that even a well-balanced and well-designed processor such as the MIPS-X CPU could be sped up if the instruction set were split into three classes, and the clock period adjusted appropriately to match the temporal needs of each class.

Dean also demonstrated that an even greater performance enhancement could be achieved due to the tighter margins which are possible with adaptive clocking. Synchronous systems usually rely on an externally-generated clock signal which is distributed as the common timing reference to all of the system components. The speed at which integrated circuits operate varies with the circuit fabrication process, and fluctuations in operating temperature and supply voltage. In order to achieve a reasonable shield against these variables, the clock period is extended by a certain *margin.* In current practice, these margins are often 100% or more in high-speed systems. Adaptive clocking cannot be generated externally, and therefore must be provided internally to each device. The fact that the clock generator is affected by the same process, temperature, and supply variations as the rest of the chip permits the safety margin to be reduced significantly.

Clock distribution is becoming an increasingly costly component of large modern designs. Today's microprocessors contain over ten million transistors and their clock rates are around 200 to 400 MHz. The clock period is determined by adding the worst case propagation delay, the margin and the maximum clock skew. Clock skew is simply the maximum difference in the clock arrival as seen by all clocked points in the circuit. The latency of the clock pulse to the reception points is not a concern. With today's large VLSI circuits exceeding 20 mm per side, several nanoseconds of skew is easily possible. However with a 5 nanosecond clock period, several nanoseconds of skew is a disaster. Clock distribution and de-skewing methods are abundant but they share the common characteristic of being expensive in either power or area and they become more so as clock speeds increase. A common method is to distribute the clock via a balanced H-tree configuration [9] with amplifying buffers placed at the fanout points. The problem with this approach is that as more buffers are added to a clock path, larger skew results. The designers of the DEC Alpha CPU [193] took the opposite approach. The Alpha contains 1.68 million transistors and is fabricated in a .75 micron, 3.3 volt CMOS process. Even with three layers of metal, the chip is 16.8 mm by 13.9 mm. In order to keep clock skew to within 300 picoseconds, the Alpha's designers localized the clock buffering to minimize process induced variations and therefore the skew induced by the buffers. Details of the method can be found in [60] but the result is a clock driver circuit that occupies about 10% of the chip area, and consumes over 40% of the 30 watts of power dissipated by the chip. 19 mm^2 of area and over 12 watts of power is a very high price to pay for keeping the skew under control. Power concerns in particular will limit the use of this technique as circuit speeds and transistor counts increase.

Another common modern synchronous technique for controlling clock skew is to use phased locked loops (PLL's). PLL's are essentially an analog circuit that can be used to dynamically adjust the phase of a signal to match it with the phase of another signal. For clock deskewing purposes the local clock is kept in phase with an external reference clock. There are many PLL design variants. We describe a simple voltage controlled version [9] even though in higher speed circuits a current controlled methodology is more common. A phase detection circuit is used to decide whether the internal clock is behind or ahead of the external clock and produces a *ChargeAdd* or *ChargeRemove* signal accordingly. These signals are smoothed by a low pass filter to provide a signal that controls a voltage controlled delay line or a voltage controlled oscillator (VCO). The VCO sits between the clock generator and buffering logic. It is important to note that this PLL based technique is only capable of eliminating the components of clock skew that are the result of the clock generator and clock buffering circuits. PLL's cannot remove the skew components that are caused by the clock distribution tree. Using multiple PLL's at various points in the distribution tree does not help since the area penalty is potentially severe and minor differences in the individual PLL operations will cause an increase in the amount of jitter in the resulting clock system. More importantly this technique exacerbates the basic problem since now the problem shifts to distribution of the reference clock. The result is that PLL's are an important and useful clock distribution technique which can be used to solve part of the deskewing problem. However this approach will be inadequate in the long run since as circuit feature sizes shrink and die sizes grow, a larger fraction of signal speed is due to wire delays and not the logic delays in the clock driver and buffer circuits. New approaches will be necessary or performance will be adversely affected.

A similar skew problem exists for circuit boards as well as chips. The literature contains an abundance of methods for de-skewing clocks [2, 36] on a board but most of them are also costly in either area or complexity, and some will probably not be robust enough for use in commercial circuits. An interesting example is the Monarch [168] processor chip which used active signal selection on each input pad. In this instance, a five slot delay line was used to skew signals to match the clock skew. The appropriate tap in the delay line was selected based on analyzing the clock vs. the incoming signal. While the technique did work, its cost and complexity are probably more instructive in a pathological sense. The bottom line is that clock management is a difficult problem and solving it in today's high-speed complex designs is costly. Asynchronous circuit proponents advocate a simple solution, namely throw away the whole concept of a global clock. This is not a free solution since global absolute timing must be replaced with the relative and sequential mechanisms which lie at the heart of asynchronous circuit signaling protocols. Chuck Seitz wrote an excellent introduction to this general topic in his chapter on *System Timing* in the classic VLSI book by Mead and Conway [129]. The next section of this treatise presents some of the more commonly used protocols and terminology.

Another common motivation for pursuing the asynchronous circuit option is the quest for low-power circuit operation. The consumer market's hunger for powerful yet portable digital systems which run on lightweight battery packs is growing at a rapid rate. Hence there is a strong commercial interest in low-power design methods which extend the operational life of a particular battery technology. CMOS circuits have a particular appeal, since they consume negligible power when they are idle. This would not be true, however, if the clock of a synchronous circuit were to continue running. Therefore, low-power synchronous circuits usually involve some method of shutting down the clock to subsystems which are not needed at a particular time. Clocks must be continuously supplied the subcomponent that must monitor the environment for the next call to action. The result is that power must be consumed even during idle periods. Furthermore, these clock switches exacerbate the clock skew issues which limit performance and also reduce the circuit's ability to provide maximum performance when it is needed. Asynchronous circuits have the advantage that they go into idle mode for free since, by nature, when there is nothing to do there are no transitions on any wire in the circuit. Another advantage is that even for an active system, only the subsystems that are required for the computation at hand will dissipate any power. Researchers such as Kees van Berkel [211] and Steve Furber [70] are pursuing asynchronous circuit designs in an attempt to exploit this feature.

The final motivation of asynchronous design is the inherent ease of composing asynchronous subsystems into larger asynchronous systems. While there is still room for doubt about whether asynchronous circuits can generally achieve their potential advantages in terms of higher performance or lower power operation than synchronous circuits, there is little doubt that asynchronous circuits do have a definite advantage with respect to composability. Asynchronous circuits are functional modules in that they contain both their timing and data requirements *explicitly* in their interfaces. In a sense, they "keep time for themselves", hence the term self-timed circuits. Synchronous circuit modules contain only data requirements in their interfaces and share *the clock*. However, important temporal issues, such as when data must be valid to avoid set-up and hold time violations between modules, are *implicit* at best. In contrast, composing asynchronous modules is almost trivial. If the interfaces match and observe the same signaling protocol then they can simply be connected. The same cannot be said for synchronous circuits with their global timing requirements and clock-based sequencing. The result is that a more detailed knowledge of module internals is required before synchronous subsystems can be connected.

The problem of combining synchronous systems is exacerbated when each module has a separate clock, each running at a different frequency. The effects of this problem are numerous and probabilistically involve some variant of metastability failure [34]. It is commonly accepted, although not definitively proven to the authors' knowledge, that it is impossible to build a perfect synchronizer. Many of the subsystems in today's computers run on clocks which are not synchronized with the CPU. A good example is the I/O subsystem. Often these subsystems are confusingly called asynchronous or are considered to have an asynchronous interface. In reality, they are synchronous systems which use some sort of synchronizing scheme in their interface. Synchronizers, while imperfect, effectively trade increased latency for more reliable synchronization. The reliability is adjusted to meet the MTBF (Mean Time Before Failure) requirements of the system, and the resulting decreased performance is simply viewed as the price that must be paid for the required reliability.

The ease of composing asynchronous subsystems is a clear advantage. It allows components from previous designs to be reused, it allows modification of slower components which may result in incremental performance improvements without impacting the overall design, and it facilitates behavioral analysis by formal methods. However, asynchronous circuits are not presently the mainstay of commercial practice. The definite advantage of composability is not a strong enough factor to counter the significant synchronous circuit momentum, and the promises of improved performance and decreased power consumption remain to be generally realized. There is also a clear gap in the quality of the design infrastructure, e.g. CAD tools, libraries, etc. In addition, the level of synchronous design experience dwarfs the small experience base in asynchronous circuit design. The subsequent sections on current research are indications that this gap is narrowing. The asynchronous circuit discipline is becoming more viable, even though much work remains to be done before they will be competitive in the commercial sector.

3 Controlling Asynchronous Circuits

3.1 Signaling Protocols

Most asynchronous circuit signaling schemes are based on some sort of protocol involving *requests*, which are used to initiate an action, and corresponding *acknowledgments*, used to signal completion of that action. These control signals provide all of the necessary sequence controls for computational events in the system. Strictly speaking these *handshake* signals are independent of any global system time and are only concerned with the local relative temporal relationships between two subsystems sharing a common interface. The resulting computational model is very much like the dataflow model [49, 1], where the arrival of the necessary operand data triggers an operation. Similarly there is a concept of a sender of information and a corresponding receiver. From the circuit perspective, and ignoring data transmission issues for now, these request and acknowledge control signals typically pass between two modules of an asynchronous system. For example let there be two modules, a sender **A** and a receiver **B**. A request is sent from A to B to indicate that A is requesting some action by B. When B is either done with the action or has stored the request, it acknowledges the request by asserting the acknowledge signal, which is sent from B to A. Most asynchronous signaling protocols require a strict alternation of request and acknowledge events. These ideas can be extended to interfaces shared by more than 2 subsystems, although this is not the common case due to performance and circuit complexity issues.

There are several choices of how these alternating events are encoded onto specific control wires.

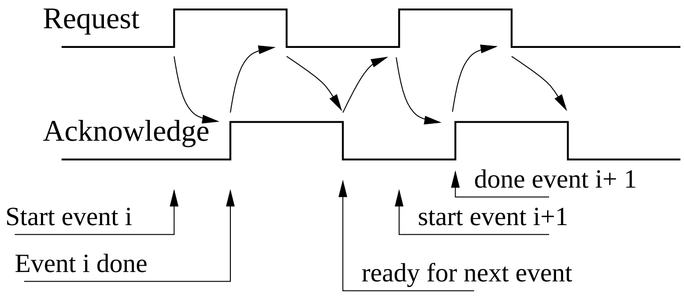

Figure 1: 4-cycle Asynchronous Signaling Protocol

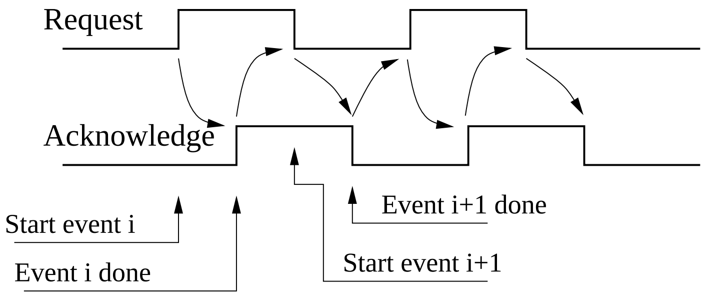

Figure 2: 2-cycle Asynchronous Signaling Protocol

Two choices have been so pervasive that they will be described here to illustrate the concept. One common choice is the *4-cycle* protocol shown in Figure 1. Other names for this protocol are also in common use: *RZ* (return to zero), *4-phase*, and *level-signaling*. In Figure 1, the waveforms appear periodic for convenience but they do not need to be so in practice. The curved arrows indicate the required before/after sequence of events. There is no implicit assumption about the delay between successive events. Note that in this protocol there are typically 4 transitions (2 on the request and 2 on the acknowledge) required to complete a particular event transaction. Proponents of this scheme argue that typically 4-cycle circuits are smaller than they are for 2-cycle signaling, and that the time required for the falling transitions on the request on acknowledge lines do not usually cause a performance degradation. This is because falling transitions can happen in parallel with other circuit operations, or are can be used to control the transmission of the answer data back to the requester.

The other common choice is *2-cycle* signaling shown in Figure 2, also called *transition*, *2-phase*, or *NRZ* (non-return to zero) signaling. In this case the waveforms are the same as for 4-cycle signaling with the exception that every transition on the request wire, both falling and rising, indicates a new request. The same is true for transitions on the acknowledge wire. 2-cycle signaling is particularly useful for high-speed micropipelines, as pointed out by Ivan Sutherland in his Turing Award paper [200].

2-cycle proponents argue that that 2-cycle signaling is better from both a power and a performance standpoint, since every transition represents a meaningful event and no transitions or power are consumed in returning to zero, since there is no resetting of the handshake link. While in principle this is true, it is also the case that most 2-cycle interface implementations require more logic than their 4-cycle equivalents. The increased logic complexity may consume more power than is saved by the reduced control transitions. This was shown to be the case in the two versions of the low-power asynchronous ARM processor produced by researchers at the University of Manchester. ARM1 [70] was a 2-cycle design. The lack of a distinct low-power advantage in ARM1, led to an improved ARM2 [73] 4-cycle design which demonstrated both a performance and low-power improvement over the ARM1. Some of this improvement can certainly be attributed to increased design expertise, but the experience provides compelling evidence that power and performance arguments can not be based solely on counting the number of control transitions per event.

4-cycle proponents argue that the falling (return to zero) transitions are often easily hidden by overlapping them with other actions in the circuit. Another approach, called *early acknowledge*, is to design 4-cycle circuits to indicate event completion with the reset transition on the acknowledge wire rather than by acknowledge assertion. Since the sender can then deassert the request, the implication is that the receiver must latch the incoming transaction prior to completing the requested action. The result is an asynchronous pipeline structure similar to synchronous pipeline circuits. The goal of all pipeline circuits is the increase in throughput performance. Still, most designers would agree that both 2- and 4-cycle protocols have advantages over the other in particular circuits. Certain design styles [48] and designs [199] show that the 2-cycle protocols can coexist in the same system, albeit on different interfaces. Numerous 2-cycle to 4-cycle (and vice versa) conversion circuits exist and can be used for interfaces where performance is not critical, since the circuits do add some latency to the interface operation.

Other interface protocols, based on similar sequencing rules, exist for 3 or more module interfaces. A particularly common design requirement is to conjoin 2 or more requests to provide a single outgoing request, or conversely to provide a conjunction of acknowledge signals. A commonly used asynchronous element is the *C-element*, which can be viewed as a protocol-preserving conjunctive gate. Note that this element is equally useful for both 2- and 4-cycle protocols. The description here will consider a 2-input C-element for simplicity. The common logic symbol and a positive-logic, gate-level implementation are shown in Figure 3. From an initial state where inputs x and y are both low, the output z is low. When both x and y go high, then the output z will go high. Similarly when both inputs go low, then the output will go low. The C-element effectively merges two requests into a single request and permits 3 subsystems to communicate in a protocol-preserving 2- or 4-cycle manner. Many consider C-elements to be as fundamental as a NAND gate in asynchronous circuits, and they will appear repeatedly in many of the basic circuits that will be subsequently presented here. The feedback signal from the output of the C-element to two of the 2-input NAND gates indicates that the C element is itself a form of latch. It therefore acts as a synchronization point which is necessary for protocol preservation. However, excess synchronization reduces performance and there are numerous asynchronous circuits which have been designed with too many C-elements and their performance has suffered. An AND (or NAND) gate is a conjunction of only the low to high input signal trajectories whereas the C-element is conjunctive for both rising and falling trajectories. The key is to properly understand the conjunction requirements of the circuit and not use C-elements where some form of AND gate will suffice. It is rare that large asynchronous circuits can be built using no C-elements, but the existence of many C-elements in the circuit is often an indication that the performance of the circuit will be reduced.

So far, the discussion has only addressed control signals. There are also choices for how to encode data. A common choice is the use of a *bundled* protocol with either 2- or 4-cycle signaling. In this

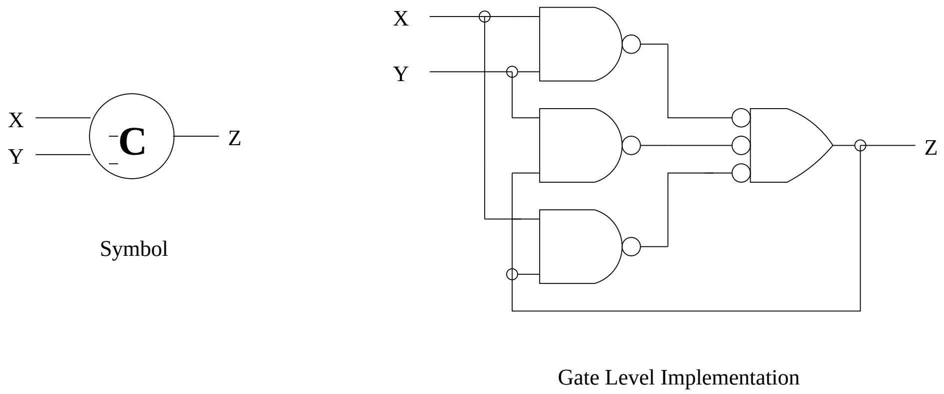

Figure 3: The C-element

case, for an n-bit data value to be passed from the sender to the receiver, n+2 wires will be required (n bits of data, 1 request bit, and 1 acknowledge bit). While this choice is conservative in terms of wires, it does contain an implied timing assumption. Namely the assumption is that the propagation times of the control and data lines are either equal or that the control propagates slower than the data signals. A sending module will assert the data wires and when they are valid will assert the request. It is important that the same relationship of data being valid prior to request assertion be observed at the receiving side. If this were not the case, the receiver could initiate the requested action with incorrect data values. This requirement is often simply called the *bundling constraint.* Most asynchronous circuits have been designed with bundled data protocols because the logic and wires required to implement bundled data circuits is significantly less than with non-bundled approaches. However, in order for bundled data asynchronous circuits to work properly, the bundling constraint must be met. Antagonists of this approach note that these timing assumptions, while local to a particular interface, are similar to those made for synchronous circuit design.

The common alternative to the bundled data approach is *dual rail* encoding. In this case, data and control signals are not separated onto distinct wire paths. Instead, using the dual rail approach, a bit of data is encoded with its own request onto 2 wires. A typical dual rail encoding has four states:

1. 00 - Idle, data is not valid
2. 10 - Valid 0
3. 01 - Valid 1
4. 11 - Illegal

In this case, for an n-bit data value, the link between sender and receiver must contain 3n wires: 2 wires for each bit of data and the associated request plus another bit for the acknowledge. An improvement on this protocol is possible when n-bits of data are considered to be associated in every transaction, as is the case when the circuit operates on bytes or words. In this case it is convenient to combine the acknowledges into a single wire. The resulting wiring complexity is then reduced to 2n+1 wires: 2n for the data and requests plus an additional acknowledge signal. In a four cycle variant of this dual rail protocol, sending a bit requires the transition from the Idle state to either the valid 0 or valid 1 state and then, after receiving the acknowledge, it must transition back to the idle state. The acknowledge wire must be reset prior to a subsequent assertion of a valid 0 or 1. The illegal state is not used. If recognized by the receiver, it should cause an error.

A 2-cycle dual-rail protocol would signal a valid 0 by a single transition of the left bit, while a valid 1 would be signaled by a transition on the right bit. Concurrent transitions on both the left and right bits are illegal. Sending a 0 or a 1 must be followed by a transition on the acknowledge wire before another bit can be transmitted. Alternative encoding schemes have been proposed as well [218, 56]. Dual rail signaling is insensitive to the delays on any wire and therefore is more robust when assumptions like the bundling constraint cannot be guaranteed. The receiver will need to check for validity of all n-bits before using the data or asserting the acknowledge. The downside of the dual rail approach is often the increased complexity in both wiring and logic.

3.2 Completion Signals

One of the added complexities of asynchronous circuits is the need to generate completion signals that directly or indirectly control the acknowledge signal in a signaling protocol. There are many methods, none of which is universally satisfactory. One approach is to design an asynchronous module in a manner that is similar to a synchronous circuit. Namely, the arrival of the request starts the modules internal clock generator and after a certain number of internal clocks: the circuit is done, the clock is stopped, and an acknowledge is generated. The idea was originally suggested by Chuck Seitz and was used during the construction of the first dataflow computer, DDM1 [49]. This technique works well when the size of the module is large, but when the module is small, the additional logic required for the internal clock generator represents an overhead that is too costly. Also, the technique does not lend itself well to high performance designs, due to the increased circuit complexity and the delay associated with starting the clock generator. The result is that this approach is seldom used today, since modules represent relatively small pieces of an integrated circuit.

Another choice for completion signal generation is the use of a *model delay.* In this case, conventional synchronous timing analysis of the datapath is used to determine how long the circuit will take to compute a valid result after the request has been received. A delay element, such as an inverter chain, is then used to turn the request into the appropriately delayed acknowledge signal. Note that this method works equally well for both 2- and 4-cycle signaling protocols.

Special functions often have unique opportunities. For example, arithmetic circuits can be built to generate completion signals based on carry propagation patterns [88]. Other functions can independently compute both F and $\overline{F}$ and use the exclusive-OR of their outputs to generate the acknowledge signal. Note that this technique will only work directly in a 4-cycle signaling protocol. If used with a 2-cycle protocol, additional logic such as a T flip-flop will be required.

A novel technique was proposed by Mark Dean [55] where completion detection was performed by observing the power consumption of the circuit. When activated the circuit consumes power, and when it is done the power consumption falls below a particular threshold.

The study of completion signal generation methods in asynchronous circuits could be the topic of an entire book. For now, it is only necessary to realize that some method must be chosen, and that the need for completion signals and related signaling protocols is a necessary overhead of asynchronous circuit design. Many modern memory chips are integrated into synchronous systems using this same technique. For example, memories are not inherently synchronous, but do have specified access latencies. These specifications are essentially model delays. Systems which use these chips assume that the access is complete after a certain number of clock cycles which correspond to a delay that is no smaller than the access latency of the device.

4 Delay Models and Hazards

4.1 Delay Models, Circuits and Environments

There is a wide spectrum of asynchronous designs. One way to distinguish among them is to understand the different underlying models of delay and operation. Every physical circuit has inherent delay. However, since synchronous circuits process inputs between fixed clock ticks, they can often be regarded as instantaneous operators, computing a new result in each clock cycle. On the other hand, since asynchronous circuits have no clock, they are best regarded as computing dynamically through time. Therefore, a delay model is critical in defining the dynamic behavior of an asynchronous circuit.

There are two fundamental models of delay: the *pure delay* model and the *inertial delay* model [206, 182]. A pure delay can delay the propagation of a waveform, but does not otherwise alter it. An inertial delay can alter the shape of a waveform by attenuating short glitches. More formally, an inertial delay has a threshold period, δ. Pulses of duration less than δ are filtered out.

Delays are also characterized by their timing models. In a *fixed delay* model, a delay is assumed to have a fixed value. In a *bounded delay* model, a delay may have any value in a given time interval. In an *unbounded delay* model, a delay may take on any finite value.

An entire circuit's behavior can be modeled on the basis of its component models. In a *simple-gate*, or *gate-level*, model, each gate and primitive component in the circuit has a corresponding delay. In a *complex-gate* model, an entire sub-network of gates is modeled by a single delay; that is, the network is assumed to behave as a single operator, with no internal delays. Wires between gates are also modeled by delays. A *circuit model* is thus defined in terms of the delay models for the individual wires and components. Typically, the functionality of a gate is modeled by an instantaneous operator with an attached delay.

Given a circuit model, it is also important to characterize the interaction of the circuit with its *environment*. The circuit and environment together form a closed system, called a *complete circuit* (see Muller in [132]). If the environment is allowed to respond to a circuit's outputs without any timing constraints, the two interact in *input/output mode* [24]. Otherwise, environmental timing constraints are assumed. The most common example is *fundamental mode* [126, 206] where the environment must wait for a circuit to stabilize before responding to circuit outputs. Such a requirement can be seen as the hold time for a simple latch or flipflop [127].

4.2 Classes of Asynchronous Circuits

Given these models for a circuit and its environment, asynchronous circuits can be classified into a hierarchy.

A *delay-insensitive (DI)* circuit is one which is designed to operate correctly regardless of the delays on its gates and wires. That is, an unbounded gate and wire delay model is assumed. The concept of a delay-insensitive circuit grows out of work by Clark and Molnar in the 1960's on *Macromodules* [42].[1] DI systems have been formalized by Udding [205] and Dill [58]. The class of DI circuits built out of simple gates and operators is quite limited. In fact, it has been proven that almost no useful DI circuits can be built if one is restricted to a class of simple gates and operators [121, 25]. However, many practical DI circuits can be built if one allows more complex components [61, 91]. A complex component is constructed out of several simple gates. Internal to the component, timing assumptions must be satisfied; externally, the component operates in a delay-insensitive manner. A C-element is such a component and other examples of DI designs using complex components are described in Section 6.4; see Figure 15.

A *quasi-delay-insensitive (quasi-DI or QDI)* circuit is delay-insensitive except that "isochronic forks" are required [27]. An isochronic fork is a forked wire where all branches have exactly the same delay. In other formulations, a bounded skew is allowed between the different branches of each fork. In contrast, in a DI circuit, delays on the different fork branches are completely independent, and may vary considerably. The motivation of QDI circuits is that they are the weakest compromise to pure delay-insensitivity needed to build practical circuits using simple gates and operators. C-elements are somewhat problematic since they inherently contain an output which is fed back internally to the C element. This case represents the worst form of isochronic fork, since one of the forks is contained within the C element circuit module while the other is exported to outside modules. Martin [122] and van Berkel [211] have used QDI circuits extensively and have described their advantages and disadvantages [121, 214].

A *speed-independent (SI)* circuit is one which operates correctly regardless of gate delays; wires are assumed to have zero or negligible delay. SI circuits were introduced by David Muller in the 1950's (see [132]). Muller's formulation only considered deterministic input and output behavior. This class has recently been extended to include circuits with a limited form of non-determinism [12, 100].

A *self-timed* circuit, described by Seitz [129], contains a group of self-timed "elements". Each element is contained in an "equipotential region", where wires have negligible or well-bounded delay. An element itself may be an SI circuit, or a circuit whose correct operation relies on use of local timing assumptions. However, no timing assumptions are made on the communication between regions; that is, communication between regions is delay-insensitive.

Each of the above circuits operate in input/output mode: there are no timing assumptions on

[1] This article contains numerous references to the work of Charles Molnar. The asynchronous circuit discipline lost one of its brightest lights when Charlie passed away in December, 1996. Charlie's influence on the field was profound. He inspired many of the people who are today considered to be pioneers and senior statesmen of the field. His inventions are numerous as both his publications and patents attest. The difficult aspect of Charlie's influence for people to grasp, with the exception of the few who had the privilege to know and work with Charlie over the years, is the depth and creativity of his thinking. Charlie's work has provided both a solid foundation for the field as well as an inspiration to continue. At the time of his death, he was one of the creative leaders of the asynchronous circuits group at Sun Microsystems Laboratories, Inc. [195]. His influence will be sorely missed.

when the environment responds to the circuit. The most general category is an *asynchronous circuit* [206]. These circuits contain no global clock. However, they may make use of timing assumptions both within the circuit and in the interaction between circuit and environment. Latches and flip-flops, with setup and hold times, belong to this class. Other examples include *timed circuits* [141], where both internal and environmental bounded-delay assumptions are used to optimize the designs.

4.3 Hazards

A fundamental difference between synchronous and asynchronous circuits is in their treatment of *hazards*. In a synchronous system, computation occurs between clock ticks. Glitches on wires during a clock cycle are usually not a problem. The system operates correctly as long as a stable and valid result is produced before the next clock tick, when the result is sampled. In contrast, in an asynchronous system, there is no global clock; computation is no longer sampled at discrete intervals. As a result, any glitch may be treated by the system as a real change in value, and may cause the system to malfunction.

The potential for a glitch in an asynchronous design is called a *hazard* [206]. Hazards were first studied in the context of asynchronous state machines, and much of the original work focused on combinational logic. Sequential hazards are also possible in asynchronous state machines; these are called *critical races* or *essential hazards*, and will be discussed later.

Several approaches have been used to eliminate combinational hazards. First, inertial delays may be used to attenuate undesired "spikes"; much of the early work in asynchronous synthesis relied on use of inertial delays (see Unger [206]). Second, if a bounded delay model is assumed, hazards may be "fixed" by adding appropriate delays to slow down certain paths in a circuit. Third, hazards are sometimes tolerated where they will do no harm; this approach was also used in some early work. Finally, and most importantly, synthesis methods can be used to produce circuits with no hazards, *i.e., hazard-free* circuits.

In the remainder of this section, the basics of hazard-free combinational synthesis are presented. Two traditional classes of combinational hazards are defined: *SIC* and *MIC* hazards. Classic techniques to eliminate both SIC and MIC hazards in 2-level circuits are introduced, illustrated by some simple examples. The section concludes with a description of recent work on hazard-free minimization. Throughout this section, a conservative circuit model is used: the combinational circuit is assumed to have unknown gate and wire delays. That is, an unbounded gate and wire delay model is assumed.

4.3.1 SIC Hazards

Hazards are temporal phenomena: they are manifest during the dynamic operation of a circuit. As an example, consider the Karnaugh map ("K-map") [127] in Figure 4(a), defining a Boolean function with 3 inputs: A, B, C. A minimum-cost sum-of-products realization, or cover, is given by expression $f = A'B + AC$; the corresponding AND-OR circuit is shown in the figure. Consider the behavior of the circuit during the *single-input change* (*SIC*) from $ABC = 011$ to $ABC = 111$. In this transition, only a single input, A, changes value. Initially, AND-gate $A'B$ is 1, AND-gate AC is 0, and the OR-gate has output 1. When A changes, AND-gate $A'B$ goes to 0 and AC goes to 1. However, if AND-gate AC

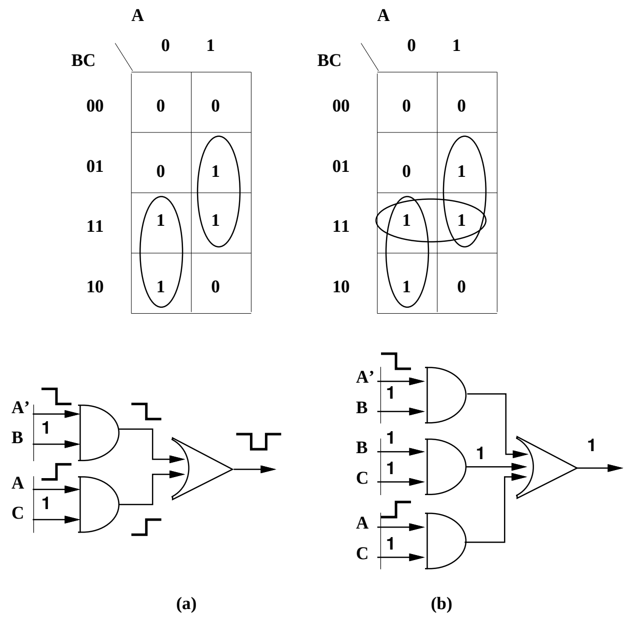

Figure 4: Combinational hazard example: SIC transition

is slower than $A'B$, the result is a glitch on the OR-gate output: $1 \to 0 \to 1$. Therefore, the circuit has a hazard for this transition.

The Karnaugh map in Figure 4(b) shows an alternative *hazard-free* realization of the function. A third product, BC, has been added to the cover. For the same SIC transition, AND-gate BC holds its value at 1, and the OR-gate output remains at 1 without any glitches. Therefore, the new circuit is hazard-free for the transition. This new product, BC, is redundant in terms of function f, but is necessary to eliminate the hazard. This product is used to cover the K-map transition, $ABC: 011 \to 111$.

The original theory of combinational hazards for SIC transitions was developed by Huffman, Unger and McCluskey (see [206]). The above example indicates how to eliminate an SIC *static-1* hazard, that is, for an input change where the function makes a $1 \to 1$ transition. In this case, *some* product must cover (*i.e.,* completely contain) the entire transition. There are 3 remaining types of transitions, where the output makes a $0 \to 0$, $0 \to 1$, or $1 \to 0$ transition. It has been shown that, given an arbitrary AND-OR implementation, *no* hazard will occur for any of these 3 transitions [206]. [2] That is, only static-1 SIC hazards must be avoided during synthesis of an AND-OR circuit; other SIC transitions will be hazard-free.

4.3.2 MIC Hazards

The case of a *multiple-input change* (*MIC*) is much more complex: both static and dynamic hazards must be eliminated. An MIC transition has a start input value, M, and a destination input value, N, where *several* inputs change monotonically between M and N.

[2] More precisely, these realizations will be hazard-free as long as no AND gate contains a pair of complementary literals.

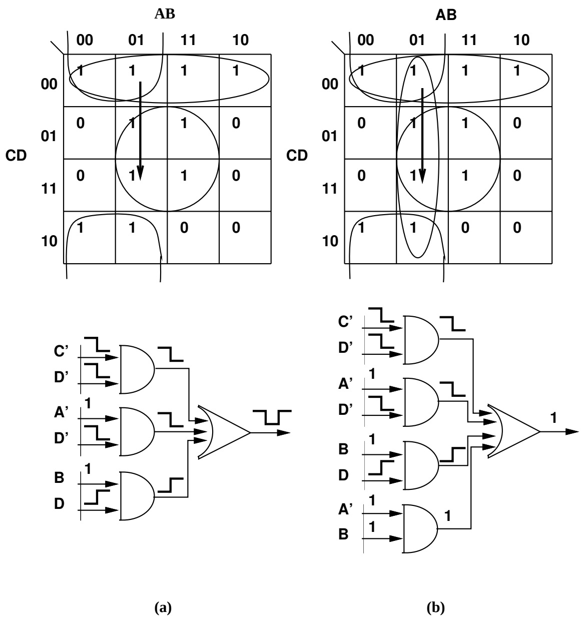

Figure 5: Combinational hazard example: static MIC transition

The first problem which arises when considering MIC transitions is that of *function hazards* [206]. Consider the Karnaugh map in Figure 5(a). An example of an MIC transition has start point $ABCD = 0010$ and end point $ABCD = 0111$, and two changing inputs: B and D. During this MIC transition, the function itself is not monotonic; that is, it can change value several times. To see this, consider the change in the function's value if input D first goes to 1, followed by input B. Initially, at $ABCD = 0010$, the function is 1. When D goes high, the function changes to 0. When B then goes high, the function then changes to 1. Therefore, the function itself changes value more than once. Such an MIC transition is said to have a *function hazard*.

It has been shown that, assuming gates and wires may have arbitrary delay, there is no guaranteed method to synthesize a circuit which is hazard-free for a transition with a function-hazard [206]. Intuitively, a function hazard is a glitch that is inherent in the Karnaugh map specification itself. If input D changes much later than input B, there is no way to prevent the function output from glitching.[3]

In summary, function hazards cannot be avoided. Therefore, classic synthesis methods focus only on MIC transitions which are already *function-hazard-free*. Examples a function-hazard-free transitions are shown in Figure 5(a). The transition from $ABCD = 0100$ to $ABCD = 0111$ is *static*, since the output remains at a single value (1) throughout the transition. A *dynamic* transition is shown from $ABCD = 0111$ to $ABCD = 1110$; this transition is function-hazard-free, since the output changes *exactly once*, from 1 to 0, on all direct paths from the start point to the end point.

Given a function-hazard-free MIC transition, the goal of hazard-free synthesis is to produce an AND-OR circuit which is glitch-free for the transition. If a glitch can occur, the transition is said to

[3]Alternatively, even if B and D change simultaneously, there are always delay values for the given gates to force the function to drop low before going high.

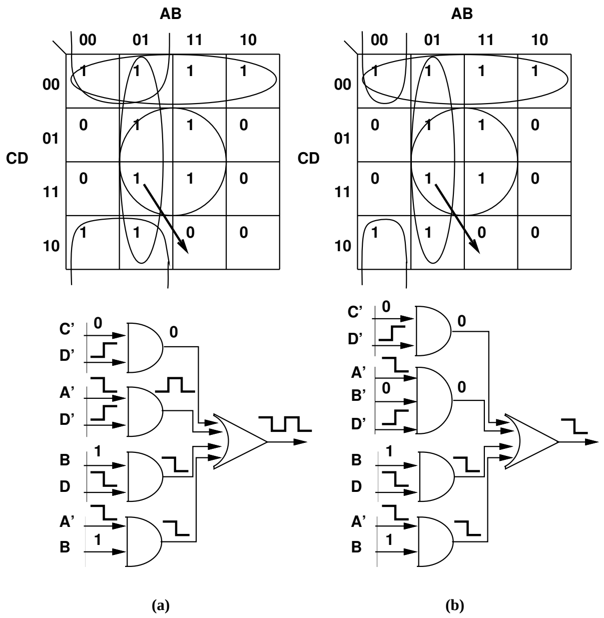

Figure 6: Combinational hazard example: dynamic MIC transition

have a *logic hazard.* If no glitches are possible, the transition is *logic-hazard-free.*

Static-1 logic hazards (*i.e.,* hazards during a $1 \rightarrow 1$ transition) can be avoided in an AND-OR implementation by using an approach similar to the SIC case [63]. As an example, consider the Karnaugh map in Figure 5(a). A minimum-cost sum-of-products realization is: $f = C'D' + A'D' + BD$. Consider the MIC transition from $ABCD = 0100$ to $ABCD = 0111$, indicated by an arrow. AND-gates $C'D'$ and $A'D'$ each make a $1 \rightarrow 0$ transition, and AND-gate BD makes a $0 \rightarrow 1$ transition. If BD is slow, the result is a glitch on the OR-gate output: $1 \rightarrow 0 \rightarrow 1$. Therefore, the implementation has a static logic hazard. An alternative hazard-free implementation is shown in Figure 5(b). The hazard is eliminated by adding a fourth product term, $A'B$, which holds its value at 1 throughout the transition. This product covers the entire transition, $ABCD: \; 0100 \rightarrow 0111$.

The next problem is to eliminate static-0 logic hazards. These hazards are easily handled. In fact, it has been shown that, given any MIC $0 \rightarrow 0$ transition which is already function-hazard-free, the transition is *guaranteed* to be free of logic hazards in any AND-OR implementation [63]. That is, no special care need be taken during 2-level synthesis to avoid static-0 logic hazards.

A more difficult problem is to eliminate MIC dynamic logic hazards. Figure 6(a) contains the same Karnaugh map as in Figure 5(b), but with a new MIC transition: from $ABCD = 0111$ to $ABCD = 1110$. This is a dynamic function-hazard-free transition; the function makes a $1 \rightarrow 0$ transition. The implementation has a dynamic logic hazard. AND-gates BD and $A'B$ each make a $1 \rightarrow 0$ transition. At the same time, AND-gate $A'D'$ has inputs changing from 10 to 01, and therefore may glitch: $0 \rightarrow \; 1 \rightarrow \; 0$. If $A'D'$ is slow, this glitch will propagate to the OR-gate *after* the other AND-gates have gone to 0, and the OR-gate output will glitch: $1 \rightarrow \; 0 \rightarrow \; 1 \rightarrow \; 0$.

To prevent a dynamic MIC hazard, *no* AND-gate may temporarily turn on during the transition. In the example, $A'D'$ becomes enabled, then disabled, as inputs A and D changed value. This phenomenon is apparent in the Karnaugh map: product $A'D'$ intersects the transition from $ABCD : 0111 \rightarrow 1110$, but intersects *neither* the start point ($ABCD = 0111$) *nor* the end point ($ABCD = 1110$) of the transition [20]. A solution to this problem was proposed by Beister[15]: product $A'D'$ is *reduced* to a smaller product $A'B'D'$ which no longer intersects the transition. Note that this product is non-prime. The final cover is shown in Figure 6(b). AND-gate $A'B'D$ remains at 0 throughout the transition, and the dynamic MIC transition is hazard-free.

4.3.3 Hazard-Free Minimization

The above examples indicate how to eliminate hazards for any one MIC transition. Hazard elimination can be viewed as a covering problem on a Karnaugh map. For the $1 \rightarrow 1$ case, the entire transition must be covered by some product. For the $1 \rightarrow 0$ and $0 \rightarrow 1$ cases, every product which intersects the transition must also contain its start or end point. For the remaining case, $0 \rightarrow 0$, no hazard will occur in any AND-OR realization [206]. These conditions suffice to eliminate any single MIC hazard. Unfortunately, when attempting to eliminate hazards for *several* MIC transitions simultaneously, these covering conditions may be unsatisfiable. That is, for a given set of MIC transitions, a hazard-free cover may not exist [206, 66, 153].

An exact hazard-free two-level minimization algorithm was developed by Nowick and Dill [153]. The algorithm finds an exactly minimum-cost cover which is hazard-free for a set of MIC transitions, if a solution exists. A heuristic hazard-free two-level minimization algorithm has also been developed [202].

There is a rich literature on *multi-level* hazard-free circuits as well, and several synthesis methods have been proposed. One approach is to start with a hazard-free circuit (for example, a two-level circuit), and apply *hazard-non-increasing* multi-level transformations [206, 19, 103]. These transformations transform a hazard-free two-level circuit into a hazard-free multi-level circuit. Alternatively, a hazard-free multi-level circuit can be synthesized directly, using binary decision diagrams *(BDDs)* [110]. Other algorithms have been developed for the hazard-free *technology mapping* of circuits to arbitrary cell libraries [191, 102, 14].

4.3.4 An Alternative View of Hazards

The above discussion follows a classical framework, focusing on combinational hazards separately from sequential hazards. This distinction has been quite useful for synthesis of asynchronous state machines. However, for other synthesis styles, a uniform treatment of hazards is more natural. In this latter approach, each gate and sequential component is assigned a specified "legal" behavior, describing the correct operation of the component. As components are combined into a circuit, their composite behavior is formally determined. If a component may produce an output which cannot legally be accepted by another component, then a violation occurs. This notion has been formalized, in different contexts, as: *computation interference* [61], *stability violation* [122] and *choking* [58].

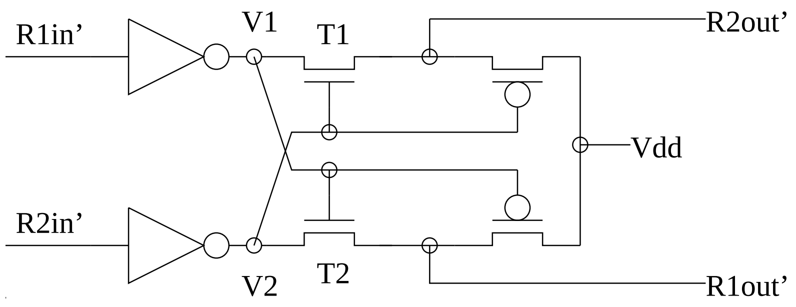

Figure 7: The Mutual Exclusion Element

5 Arbitration

In order to avoid non-deterministic behavior, asynchronous circuits must be hazard-free under some circuit delay model. As discussed above, certain forms of MIC behavior can be tolerated; but the most general forms of signal concurrency must be controlled by arbitration in order to avoid unrestricted MIC behaviors that result in circuit hazards. For example, if some circuit is to react one way if it sees a transition on signal A and react differently for a transition on signal B, then some guarantee must be provided that this circuit will see mutually exclusive transitions on inputs A and B. Nondeterministic behavior will occur if this guarantee cannot be provided. Such mutually exclusive signal conditioning is usually provided by *arbitration.*

Latches and flip-flops cannot be used for arbitration due to the inherent possibility that they may enter their metastable regions [34, 33]. Arbiter circuits are typically constructed to adhere to a particular signaling protocol and therefore vary somewhat. However all arbiters rely on a *mutual exclusion*, or *ME element*, to separate possible concurrent signal transitions. The ME element is essentially a latch with an analog metastability detector on its outputs. If sufficient signal separation exists between the two inputs then the first one wins. However, if both inputs occur within a device-specific time, then the latch will go metastable, but the metastability detector will prevent the outputs of the ME circuit from changing until the metastability condition is resolved. The duration of metastability is unbounded but normally persists for a very short time. It has been experimentally confirmed [34, 33] that the metastability duration is an exponentially decaying probability which depends somewhat on the particular latch properties. The result is that in the case of a tie, exactly one side will win the arbitration. The additional implication is that the distinction of which side wins does not matter.

Chuck Seitz proposed an ME circuit that is particularly useful for MOS based designs, shown in Figure 7. The cross-coupled inverters form the usual SR latch. The outputs of the latch are connected to a pair of transistors which form the metastability detector. When the latch is in its metastable region, V1 and V2 will differ by less than the threshold voltage of the N-type transistors. In this case, both T1 and T2 will be off, since the gate-to-source voltage will be less than the threshold. If T1 and T2 are off then the outputs of the ME circuit will remain high. When V1 and V2 differ by more than the threshold voltage, then the latch will stabilize into either of its stable states. At this point, either T1 or T2 will turn on and the respective output will fall to its asserted level.

Once the mutually exclusive resolution of the input race has been provided by an ME element,

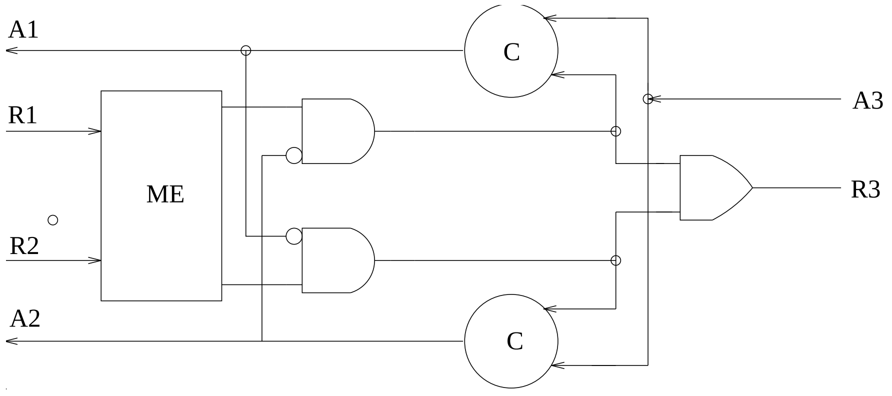

Figure 8: A 4-cycle Arbiter

constructing the rest of an arbiter circuit to conform to a particular signaling protocol is relatively straightforward. An example of a 4-cycle arbiter originally proposed by David Dill and Ed Clarke is shown in Figure 8. Four-cycle arbitration is relatively simple since the input race must only be detected for both signal trajectories going in the same single direction (typically a low to high transition). The use of the C-elements in the arbiter prevents another pending request from passing through the arbiter until after the active request cycle has cleared.

Two cycle arbitration is somewhat more complex, since the inputs of the arbiter may race in all possible combinations of signal trajectories. Ebergen [24], for example, has reported on a particular 2-cycle arbiter known as the *RGD* (Request, Grant, Done) arbiter.

Another interesting arbitration problem was posed by Davis and Stevens during the development of the Post Office chip [52]. One potential performance difficulty with asynchronous signaling protocols is that waiting for the next event is the normal mode of operation. Hence if two requesters want to share some resource, the loser must wait until the winner is finished before access to that resource can be granted. However, if the loser wants to do something else if it does not win arbitration, then the previously discussed arbitration methods will be insufficient. The need is for a NACK'ing arbiter which provides the requester with an acknowledge if the resource is available, and a negative acknowledge or *NACK* if the resource is busy. Several versions of this NACK'ing arbiter have been designed. The version used in the Post Office design used 4 ME elements [51]. Each ME element resolved one of the 4 possible race trajectories. The remaining protocol control was provided by an asynchronous finite state machine.

Arbiters for more than 2 inputs allow numerous implementation options. The simplest case is to create a binary tree of 2-input arbiters of the appropriate size (see [58]) . The tree may be balanced or unbalanced. Balanced trees are *fair* in that they give equal priority to all of the leaf inputs. Unbalanced trees inherently provide higher priority for inputs which enter the structure closest to the root (in this case the output) of the arbitration tree. The problem with tree-structured N-way arbiters is that they contain many C-elements and therefore suffer from decreased performance. Another approach is to use redundant ME elements to provide mutually exclusive assertion of 1 of the N input signals. This approach was also used by Ken Stevens in the design of the Post Office chip [52], and several variants of the multiple ME element theme have been investigated by Charles Molnar at Sun Laboratories for the counterflow pipeline processor [195].

Perhaps some of the best work on cascaded arbiters and nacking arbiters has been performed by Robert Shapiro and Hartmann Genrich [190, 189]. Sadly this work has not been published in an available forum. Their work started as a formal effort to prove the cascaded arbiter properties of the arbiter circuits used in the aforementioned Post Office chip, after a defect was discovered during testing of an arbiter fragment chip. Shapiro and Genrich used Petri Net based models to create behavioral traces of both the basic arbitration modules and their properties when cascaded. Their analysis found what turned out to be a simple design flaw which had caused the problem. The more important aspect of their work is that they then found that the arbitration circuits were overconstrained in terms of C-element synchronization. They produced a series of 4-cycle designs which contained few C-elements. The C-elements were replaced by NAND gates. Another interesting result of their work is that arbiters can be made to be faster than those containing C-elements in either the forward (requesting) direction or in the backward (acknowledge) direction but not both.

Another interesting approach to the low-latency cascaded arbitration problem has been taken by Yakovlev, Petrov, and Lavagno [222]. Their circuits are speed-independent and have an improved response delay at the input request-grant handshake link due to two factors. First, request propagation is performed in parallel with the start of arbitration. The arrival of any request at a stage can trigger an immediate request to the next higher stage, prior to arbitration resolution in the lower stage. Second, resetting the request-grant handshakes is done concurrently in different cascades of the request-grant propagation chain.

The field of arbiter design is as diverse as the circuits for which the arbiters are being designed. While the methods are diverse, there is little doubt that the design of efficient arbitration structures is a key aspect of any high-performance asynchronous system design. In fact, organizing the system so that the arbitration requirements become minimal is viewed by many designers as the key factor in achieving high performance.

6 An Overview of Prior Work

6.1 Pioneering Efforts

In the mid 1950's asynchronous circuits were first studied by analyzing the nature of input restrictions on sequential circuits. These efforts were part of the general interest in switching theory. Huffman postulated [84, 85] that there must be a *minimum* time between input changes in order for a sequential circuit to be able to recognize them as being distinct. There must then be two critical periods, δ_1 and δ_2, where $\delta_1 < \delta_2$. Signals which occur within a time that is less than or equal to δ_1 cannot be distinguished as being separate events. Signals which are separated by a time of δ_2 or greater are distinguishable as a sequence of separate events. Signal events separated by a time between δ_1 and δ_2 cause nondeterministic sequential circuit behavior. This led to a class of circuits that became known as *Huffman* circuits. This work was extended in the 1950's and 60's by the fundamental contributions of Unger, McCluskey and others.

Muller [138, 139] proposed a different class of circuits which are more closely related to modern asynchronous circuits. In particular, he proposed the use of a *ready* signal. Input signals to Muller circuits were only permitted when the ready signal was asserted. In some sense, the concept is similar to that of a simple 4-cycle circuit. The unasserted acknowledge serves as a ready indication. When the circuit is not ready to accept additional input then it can merely hold its acknowledge to indicate that no further requests can be tolerated.

The efforts of Muller and Huffman spurred considerable theoretical debate in the switching circuit literature. The next notable event from a modern perspective was the seminal work by Stephen Unger that resulted in the publication of his classic text [206]. In this book, Unger provided a detailed method for synthesizing single-input change asynchronous sequential switching circuits. He provided a partial view of what would be required for the larger domain of multiple-input change circuits. This textbook had a significant influence on much of the practical work that followed in the next decade. For example, the subsequent work of both of these authors was heavily influenced by Unger's work. Additionally, several early mainframe computers were constructed as entirely asynchronous systems, notably the MU-5 and Atlas computers.

Another noteworthy effort, the *Macromodule Project* [42], conducted at Washington University in St. Louis, provided an early demonstration of the composition benefits of asynchronous circuit modules. This project created a digital "Lego" kit of modules. These modules could (and were) rapidly used to configure special-purpose computing engines, as well as general-purpose computers. The project took a significant step forward and provided a sound foundation for the numerous macromodular synthesis approaches being investigated today [23, 210, 61].

Yet another noteworthy pioneer was Chuck Seitz, whose MIT dissertation [186] introduced a Petri Net like formalism which proved to be extremely useful in the design and analysis of asynchronous circuits. In his subsequent academic career, Prof. Seitz taught numerous courses at the University of Utah and then later at CalTech where he infected a large number of students with what proved to be an incurable interest in asynchronous circuits. His influence directly resulted in the asynchronous implementation of the first operational dataflow computer [49] and the first commercial graphics system, the Evans & Sutherland LDS-1. Professor Seitz's role as an educator is also significant in that his courses on asynchronous circuits, starting as early as 1970, inspired many of the field's current researchers.

The influence of these pioneering efforts is still seen in most of the asynchronous circuit work that is in progress today.

6.2 Asynchronous Finite State Machine Synthesis

The most traditional approach to specifying and synthesizing asynchronous controllers is to view them as finite state machines. This view of computation is state-based: a machine is in some state, it receives inputs, generates outputs, and moves to a new state. Such specifications are naturally described by a *flow table* or *state table* [206]. These tables define the behavior of outputs and next state as a function of the inputs and current state. Current and next states are described symbolically (see Figure 9).

The earliest asynchronous state machine implementations were *Huffman machines* (see [206]).

Next State, Outputs X, Y — inputs a b c

State	000	001	011	010	110	111	101	100
A	A,00	--	--	A,00	B,11	--	--	A,00
B	--	--	--	--	B,11	C,01	--	--
C	--	--	--	--	D,10	C,01	--	--
D	--	--	--	--	D,10	--	--	E,01
E	A,00	--	--	--	--	--	--	E,01

Figure 9: An asynchronous flow table

These machines consist of combinational logic, primary inputs, primary outputs and fed-back state variables. No latches or flip-flops are used: state is stored on feedback loops, which may have added delay elements. A block diagram of a Huffman machine is shown in Figure 10.

Synthesis methods for asynchronous state machines usually follow the same general outline as synchronous methods [127]. A flow table is reduced through *state minimization*. Symbolic states are assigned binary codes using *state assignment*. Finally, the resulting Boolean functions are implemented in combinational logic using *logic minimization*.

There are several possible *operating modes* for an asynchronous state machine. Unger [207] proposed a hierarchy, based on the kinds of input changes that a machine can accept. In a *single-input change (SIC)* machine, only one input may change at a time. Once the input has changed, no further inputs may change until the machine has stabilized. This operating mode is highly restrictive, but simplifies the elimination of hazards. A summary of SIC asynchronous state machines can be found in Unger [206].

A *multiple-input change (MIC)* machine allows several inputs to change concurrently. Once the inputs change, no further inputs may change until the machine has stabilized. This approach allows greater concurrency, but it is still quite restricted. In particular, MIC machines have the added constraint that the multiple-input change is *almost simultaneous*. More formally, all inputs must change within some narrow time period, δ. This constraint helps to simplify hazard elimination, which is still more complicated than in the SIC case.

MIC designs were proposed by Friedman and Menon [67] and Mago [115]. These designs require the use of delays on inputs or outputs, special “delay boxes”, and careful timing requirements. The usefulness of these designs in a concurrent environment is limited, since input changes are required to be near-simultaneous.

Finally, an *unrestricted-input change (UIC)* machine allows arbitrary input changes, as long as no one input changes more than once in some given time interval, δ. This behavior is quite general, but hazard elimination is problematic. UIC designs were first proposed by Unger [207]. These designs are

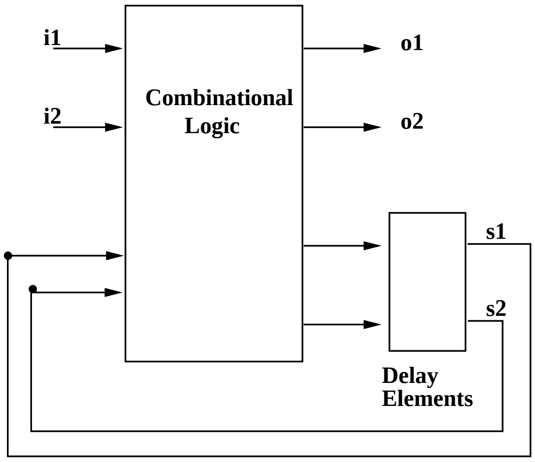

Figure 10: Block diagram of a Huffman machine

not currently practical: they require the use of large inertial delays and have not been proven to avoid metastability problems.

In any asynchronous state machine, the problem of hazards must be addressed. First is the problem of combinational hazards. The difficulty of combinational hazard elimination depends on whether the machine operates in SIC or MIC mode. As mentioned earlier, SIC hazards are easier to eliminate. Hazards are eliminated by hazard-free synthesis or by using inertial delays to filter out glitches. Alternatively, many traditional synthesis methods ignore hazards on outputs, and only eliminate hazards in the next-state logic. Such machines are called *S-proper* or *properly-realizable* [206].

Second, since asynchronous state machines have state, sequential hazards must be addressed. When a state machine goes from one state to another, several state bits may change. If the machine may stabilize incorrectly in a transient state, a *critical race* occurs. Critical races are eliminated using specialized state encodings, such as *one-hot* [206], *one-shot* [206], Liu [112] or Tracey [204] *critical race-free* codes. These codes often require extra bits. A second type of sequential problem is an *essential hazard* [206]. Essential hazards arise if a machine has not fully absorbed an input change at the time the next-state begins to change. In effect, the machine sees the new state before the combinational logic has stabilized from the input change. Essential hazards are avoided by adding delays to the feedback path or, in some cases, using special logic factoring [6].

Because of the complexity of building correct Huffman machines, an alternative approach was proposed, called *self-synchronized machines*. These machines are similar to Huffman machines, but have a local self-synchronization unit which acts like a clock on the machine's latches or flip-flops. Unlike a synchronous design, the clock is *aperiodic*, being generated as needed for the given computations. A block diagram of a self-synchronized machine is shown in Figure 11. Both SIC [82, 201] and MIC [3, 41, 169, 208] self-synchronized machines have been proposed. In a related approach, the local clock is replaced by an explicit external completion signal [105]. Other researchers have developed hybrid *mixed-operation*

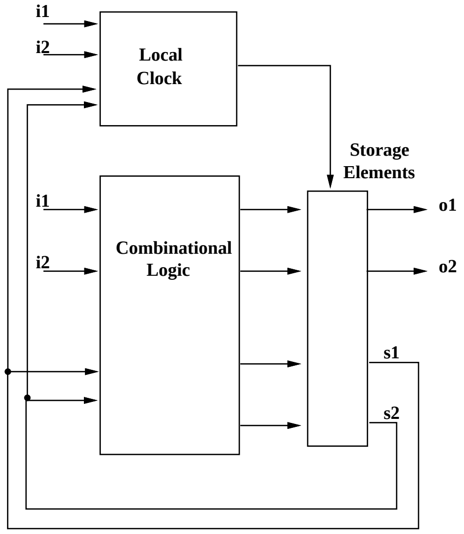

Figure 11: Block diagram of self-synchronized machine

mode machines [224, 37]. Self-synchronized machines tend to have a simpler construction but a greater overhead than Huffman machines.

In general, asynchronous state machines offer a number of attractive features. First, input-to-output latency is often low: if no delays are added to inputs or outputs, the delay is combinational. Second, since the machines are state-based, many sequential and combinational optimization algorithms can be used, similar to those which have been effective in the synchronous domain. However, asynchronous state machine design is subtle: it is difficult to design hazard-free implementations which (i) allow reasonable concurrency and (ii) have high-performance.

Much of the recent work on asynchronous state machines is centered on *burst-mode machines*. These specifications were introduced to allow more much more concurrency than traditional SIC machines, and therefore to be more effective in building concurrent systems. At the same time, burst-mode implementations are guaranteed hazard-free, while maintaining high-performance. Burst-mode specifications are based on the work of Davis on the *DDM Machine* [50]. In this dataflow machine, Davis used state machines which would wait for a collection of input changes ("input burst"), and then respond with a collection of output changes ("output burst"). The key difference between this data-driven style and MIC-mode is that, unlike MIC machines, inputs within a burst could be *uncorrelated*: arriving in any order and at any time. As a result, these machines could operate more flexibly in a concurrent environment.

More recently, Davis, Coates and Stevens implemented this approach in the MEAT synthesis system at Hewlett-Packard Laboratories [48]. The synthesis method was applied to the design of controllers for the Post Office routing chip for the Mayfly project. However, although it produced high-performance implementations, it relied on a verifier to insure hazard-free designs. An example of a burst-mode specification is shown in Figure 12. Each transition is labeled with an input burst followed by an output burst. Input and output bursts are separated by a slash, /. A rising transition is indicated by a "+"

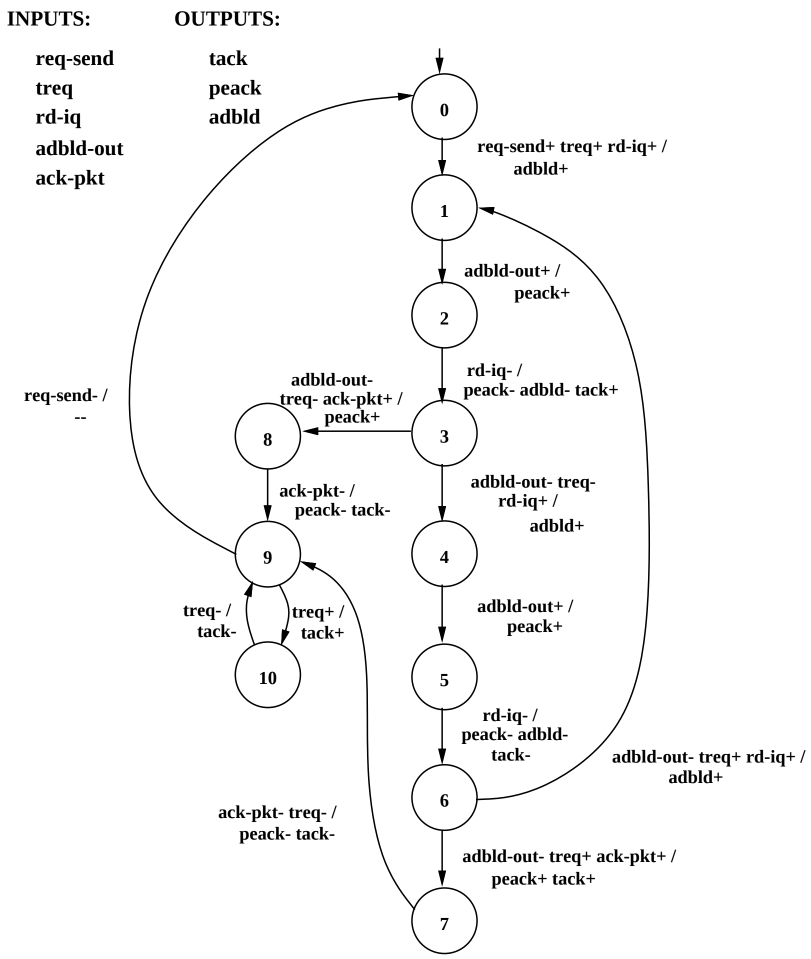

Figure 12: Burst-mode specification for HP controller *pe-send-ifc*

and a falling transition is indicated by a "–". The specification describes a controller, *pe-send-ifc*, which has been implemented in the Post Office chip.

Nowick and Dill [152, 148] made three main contributions. First, they constrained and formalized the specifications used in MEAT into the final form called *burst-mode* [152, 148]. Second, they introduced a new self-synchronized design style called a *locally-clocked state machine* [152, 148]. This was the first burst-mode synthesis method to guarantee a hazard-free gate-level implementation, given a burst-mode specification. In addition, unlike several previous self-synchronized design methods, this method produces low-latency designs, where the latency of the machine is primarily combinational delay. The design method has been automated and applied to a number of significant designs: a high-performance second-level cache controller [151], a DRAM controller and a SCSI controller [156]. Finally, they developed a hazard-free 2-level minimization algorithm, which produces a minimum-cost hazard-free sum-of-products implementation [153]. The minimizer has been refined into a CAD package called *hfmin* [68], which includes multi-output and multi-valued minimization and uses highly-optimized synchronous tools such as *mincov* [180] to solve substeps. A heuristic hazard-free minimizer, *espresso-hf*, has also been developed [202].

Yun and Dill [229] later proposed an alternative implementation style for burst-mode machines, called a *3D machine*. These machines are named after the 3-dimensional flow table used in their synthesis. Unlike locally-clocked machines, these are Huffman machines, with no local clock or latches. The synthesis method has been fully automated into a CAD tool and applied to several large designs, including an experimental SCSI controller at AMD Corporation [231]. A more recent unclocked burst-mode method, *UCLOCK*, was developed by Nowick and Coates [150].

The burst-mode approach allows greater concurrency than MIC designs, but it still has two main limitations. First, it requires strictly alternating bursts of inputs and outputs: concurrency occurs only within a burst. Second, as in many asynchronous design styles, there is no notion of "sampling"signal levels which may or may not change. Yun, Dill and Nowick [232, 230] introduced *extended burst-mode* specifications to eliminate these two restrictions. These generalized specifications allow a limited form of intermingled input and output changes, and provide greater concurrency. These designs also allow the sampling of level signals. Yun has extended his 3D synthesis algorithms and tools to handle extended burst-mode specifications [230]. His work includes performance-oriented optimizations targetted to multi-level implementations [233]. A novelty of Yun's method is that it can be used to synthesize controllers for mixed synchronous/asynchronous systems, where the global clock is one of the controller inputs [230].

A number of CAD optimization algorithms have been developed, which have been used in burst-mode synthesis. These include: optimal state assignment [68]; hazard-free 2-level logic minimization, both exact (*hfmin* [153, 68]) and heuristic (*espresso-hf* [202]); hazard-free multi-level logic optimization [103, 110]; and hazard non-increasing technology mapping [191, 102, 14], which enables more modern standard cell methodologies to be utilized.

Davis, Marshall, Coates and Siegel [117] have built a CAD framework to incorporate all of the burst-mode synthesis methods. The framework includes tools for simulation and layout as well. Their tools have been applied to several significant designs, including an low-power infrared communications chip for portable communication, developed at Hewlett-Packard Laboratories and Stanford University. An experimental chip has been fabricated; the measured current consumption of the core receiver (without pads) is less than 1 mA at 5 volts when the receiver is actually receiving data, and less than 1 μA when it is waiting for data.

Beerel and Yun have recently used burst-mode synthesis tools at Intel Corporation, including *3D* [229, 230] and *hfmin* [153, 68], to design of an experimental high-performance instruction decoder. Gopalakrishnan *et al.* have developed a high-level asynchronous synthesis tool, called *ACK* [101], which incorporates burst-mode CAD tools to synthesize controllers.

6.3 Petri-net and Graph-based Methods

Petri nets and other graphical notations are a widely-used alternative to specify and synthesize asynchronous circuits. In this model, an asynchronous system is viewed not as state-based, but rather as a partially-ordered sequence of events. A Petri net [163] is a directed bipartite graph which can describe

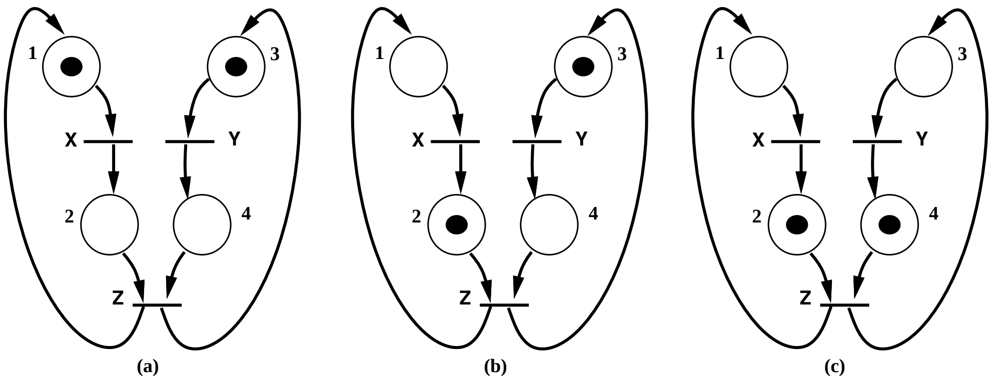

Figure 13: Petri-net example

both concurrency and choice. The net consists of two kinds of vertices: *places* and *transitions*. *Tokens* are assigned to the various places in the net. An assignment of tokens is called a *marking*, which captures the state of the concurrent system. Numerous semantics have been associated with Petri nets. A useful introductory view is that a marked place is an indication that a condition is true, and a transition specifies an action. When all of the conditions preceding a transition are true the action may *fire* which removes the tokens from the preceding places and marks the successor places. Hence, starting from an initial marking, tokens flow through the net, transforming the system from one marking to another. As tokens flow, they fire transitions in their path according to certain *firing rules*. Since the firing of a transition in a Petri net corresponds to the execution of an event, each such simulation or *token game* describes a different possible interleaved execution of the system.

A Petri net is shown in Figure 13(a). Places are drawn as circles, and transitions as bars. The initial marking is indicated by black dots in two of the places. If a place is connected by an arc to a transition, the former is called an *input place* of the transition. Likewise, if a transition is connected by an arc to a place, the latter is called an *output place* of the transition. In this example, transition X has input place 1 and output place 2; transition Y has input place 3 and output place 4.

Two transitions are enabled in the figure: X and Y. Each transition is enabled because there is a token in each input place. An enabled transition may fire at any time, removing a token from each input place and moving one to each output place. The result of firing transition X is shown in Figure 13(b). The firing of a transition corresponds to the occurrence of an event. In this example, events X and Y can occur concurrently: both transitions are enabled and may fire in any order. Figure 13(c) indicates the result of firing transition Y after X. After both events have fired, transition Z is enabled and may fire.

Patil [157] proposed the synthesis of Petri nets into *asynchronous logic arrays*. In this approach, the structure of the Petri net is mapped directly into hardware. Many modern synthesis methods use a Petri net as a behavioral specification only, not as a structural specification. Using *reachability analysis*, the Petri net is typically transformed into a *state graph*, which describes the explicit sequencing behavior of the net. An asynchronous circuit is then derived from the state graph.

Several approaches use a constrained class of Petri net called a *marked graph* [43]. Marked graphs are used to model concurrency, but not choice. That is, a marked graph cannot model that one of several possible inputs (or outputs) may change in some state. Examples include Seitz's *M-Nets* [185] and Rosenblum and Yakovlev's *Signal Graphs* [179]. Vanbekbergen *et al.* [215] introduced the notion of a *lock class* to synthesize designs from marked graphs.

More general classes of Petri nets include Molnar *et al.*'s *I-Nets* [134] and Chu's *Signal Transition Graphs* or *STGs* [38, 39]. These nets allow both concurrency and a limited form of choice. Chu developed a synthesis method which transforms an STG into a speed-independent circuit, and applied the method to a number of examples, such as an A-to-D controller and a resource locking module. This work was extended by Meng [130], who produced an automated synthesis tool for speed-independent designs from STGs. Meng also explored design tradeoffs to allow greater concurrency in the resulting circuits.

Recent work on Petri-net and graph-based asynchronous synthesis is proceeding in three major directions: (i) extending specifications; (ii) optimizing synthesis algorithms; and (iii) improving hazard elimination.

Several extensions have been proposed to describe more general behavior than is possible with the original STG's. These include the use of "epsilon" and "dummy" transitions [38], "don't-care" and "toggle" transitions [136], *OR-causality* [223] and semaphore transitions [46]. Sutherland and Sproull have introduced a notation for composite Petri nets called "snippets". Others allow timing constraints for specification and synthesis, using a related *Event-Rule* formalism [141].

In addition, some researchers are using state graphs for specifications, as an alternative to Petri nets [217, 12, 100]. State graphs allow the direct specification of interleaved behavior, avoiding some of the structural complexity of Petri nets. The target designs are usually speed-independent gate-level implementations. Originally, this work focused on determinate specifications, having no input or output choice, based on Muller's semi-modular lattice formulations (see [132]). More recent research allows generalized behavior with choice.

A number of optimized synthesis algorithms have been developed. Lavagno *et al.* [107], Vanbekbergen *et al.* [216], Chu *et al.* [40] and Puri and Gu [165] have each developed algorithms for state minimization and state assignment from STG specifications. A partitioning algorithm for STG-based specifications was proposed by Puri and Gu [166]. Lin and Lin [111] have developed algorithms which avoid expensive intermediate representations during synthesis, instead performing synthesis directly on an STG representation, for a limited class of STGs. More recently, the *theory of regions* has been used as a powerful tool in developing efficient STG algorithms, including state minimization and assignment (see Cortadella *et al.* [44, 45]). A region is a set of states in the state graph corresponding to a place in the associated STG. The theory of regions allows synthesis steps to be performed directly on the STG, without the need to generate a complete state graph.

Recent STG methods are also addressing the problem of gate-level hazards. Early STG synthesis methods typically assumed a *complex-gate model*, where an entire combinational circuit is treated as a monolithic block, rather than a collection of separate gates with individual delays. These methods could not be used to synthesize large circuits, where blocks are mapped to a *network* of gates, since the resulting network could have hazards. Several recent methods address this problem, using a *simple-gate model* which can model hazards due to actual delays in a collection of individual gates and wires.

Moon *et al.* [136] and Yu and Subrahmanyam [227] proposed heuristic techniques for gate-level hazard elimination for speed-independent design. Lavagno *et al.* [106] used logic synthesis algorithms, hazard analysis and added delays to avoid hazards, assuming bounded gate delays. Lavagno has developed an influential CAD system for STG synthesis, which has been incorporated into the Berkeley *SIS* tool package [108, 188].

Several speed-independent synthesis methods have been developed, which insure hazard-freedom at the gate-level. Much of this work has been pursued by Kishinevsky, Kondratyev, Taubin and Varshavsky [97, 217], by Beerel and Meng [12], and by Cortadella, Lavagno, Lin, Vanbekbergen, Yakovlev and others [100, 44]. These methods have been effectively applied to a number of designs. The sustained research effort of Kishinevsky *et al.*, pursued over many years in Russia and Japan, has been especially noteworthy, resulting in a collection of algorithms and tools which are making SI design practical. A general asynchronous CAD system, including speed-independent tools, has also been developed at IMEC Laboratory [225]. A comprehensive solution to the problem of hazard-free decomposition complex-gates into simpler gates, under a speed-independent model, has been developed by Burns [28].

6.4 Transformation Methods

While STG-based methods view computation as partially-ordered sequences of events, a different approach is to view an asynchronous system as a collection of communicating processes. A system is specified as a program in a high-level language of concurrency. Typically, the program is based on a variant of Hoare's *CSP* [83], such as *occam* or *trace theory* [167]. The program is then transformed, by a series of steps, into a low-level program which maps directly to a circuit. Such transformation methods use algebraic or compiler techniques to carry out the translation. Some of these methods treat datapath and control uniformly during synthesis.

Ebergen [61] introduced a synthesis method for delay-insensitive circuits using specifications called *commands*. A command is a concise program notation to describe concurrent computation based on trace theory. Several operations are used to construct a complex command from simpler commands, such as *concatenation*, *repetition* and *weave*.

Figure 14 illustrates commands for several basic DI components. A *wire* is a component with one input, $a?$, and one output, $b!$. The symbol "?" indicates an input to the wire, and "!" indicates an output of the wire. In a delay-insensitive system, a wire may have arbitrary finite delay. As a result, if two successive changes occur on input $a?$, the output behavior is unpredictable: $b!$ may glitch. To insure correct operation, input and output events must strictly alternate: once input $a?$ changes value, no further change on $a?$ is permitted until output event $b!$ occurs. A command for *wire* is given in Figure 14(a). The notation $a?; b!$ indicates that input event $a?$ must be followed by output event $b!$; ";" is the *concatenation* operator. No distinction is made between a *rising* or *falling* event on a wire; $a?$ simply means a change in value on the wire. An asterisk ($*$) indicates *repetition*: $a?$ and $b!$ may alternate any number of times. Finally "pref" is the *prefix-closure* operator, indicating that any prefix of a permitted behavior is also permitted. The final command describes the permitted interaction of a wire and its environment when it is properly used.

Figure 14(b) illustrates a more complex component called a *toggle*. A toggle has one input, $a?$,

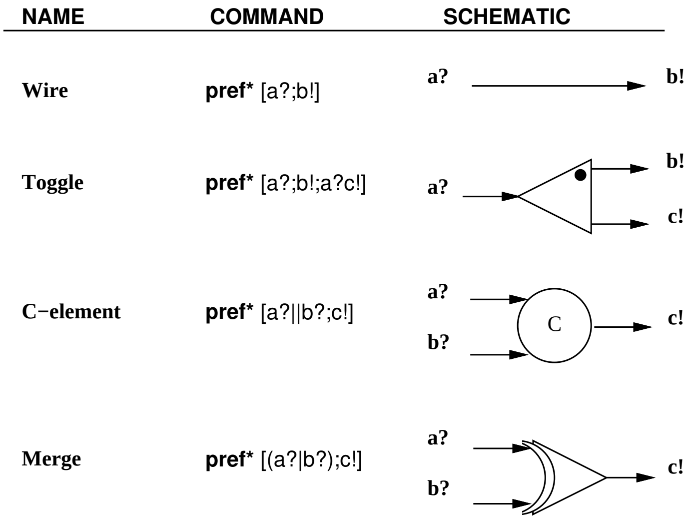

Figure 14: Commands for some simple components

and two outputs, $b!$ and $c!$. Each input event, $a?$, results in exactly one output event. Output events alternate or toggle: the first input event $a?$ results in output event $b!$ (as indicated by the black dot); the next input event results in output event $c!$; and so on. The resulting command is shown in the figure.

Another important component is a *C-element*, shown in Figure 14(c) (also known as a Muller C-element, DI C-element, rendezvous, or join element). The component has two inputs, $a?$ and $b?$, and one output, $c!$. The component waits for events on *both* inputs. When both inputs arrive, the component produces a single event on output $c!$. Each input may change only once between output events, but the input events $a?$ and $b?$ may occur in any order. Such parallel behavior is described in a command by the *weave operator*: $a? \mid\mid b?$. The final command for a C-element, allowing repeated behavior, is shown in the figure.

A final component, called a *merge*, is shown in Figure 14(d). The component is basically an *exclusive-or* gate, but its operation is restricted so that no glitching occurs. The component has two inputs, $a?$ and $b?$, and one output, $c!$. The component waits for *exactly one* input event: either $a?$ or $b?$. Once an input event occurs, the component responds with output event, $c!$. The component can be thought of as "joining" two input streams to a single output stream, where only one input stream is active at a time. Such an exclusive choice between inputs is described in a command by the *union operator*: $a? \mid b?$. The final command for a join element, allowing for repeated behavior, is shown in the figure.

A command can be used to specify a complex circuit or system. The command is then *decomposed* in a series of steps into an equivalent network of components, using a "calculus of decomposition". As an example, a *modulo-3 counter* can be specified by the following command [61]:

$$MOD3 = \mathbf{pref}{*}[a?; q!; a?; q!; a?; p!]$$

This command describes a counter with one input, $a?$, and two outputs, $p!$ and $q!$. The counter receives

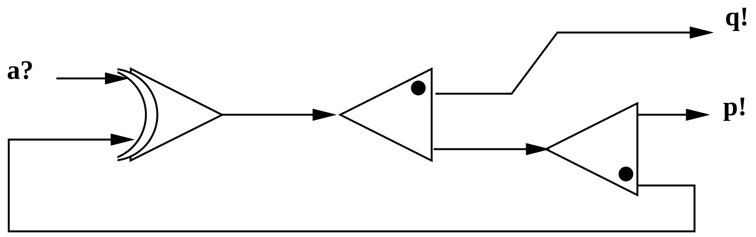

Figure 15: Ebergen's modulo-3 counter

events on input $a?$. Each input event must be acknowledged by one output event before the next input event can occur. The first and second input events are acknowledged on $q!$, while the third input event is acknowledged on $p!$. This behavior repeats, hence the command describes a modulo-3 counter. Using techniques for delay-insensitive decomposition, this command can be decomposed into a network of 2 toggles and 1 merge which implements equivalent behavior, as shown in Figure 15. Ebergen has applied his decomposition method to a number of designs, including modulo-n counters, stacks, committee schedulers [17] and token ring arbiters.

A related algebraic approach was proposed by Udding and Josephs [205, 91]. Their method is based on a *delay-insensitive algebra* which formally characterizes a delay-insensitive system. Using axioms and lemmas, a specification is transformed into a provably correct delay-insensitive circuit. An alternative *speed-independent algebra* has also been proposed [89]. Proof methods for recursively-defined DI specifications have been formally justified [116]. The DI synthesis method has been used to design a stack, a routing chip, an up-down counter, and a polynomial divider [90]. Lucassen and Udding [113] have used DI algebra to design, and prove correct, a stage in the Counterflow Pipeline Processor developed at Sun Laboratories. In related work, Patra and Fussell [158] have proposed a "basis set" of DI components. They have shown that any DI circuit can be constructed using only components from the set, and that the set is minimal.

While the above methods use algebraic calculi to derive asynchronous circuits, other transformation methods rely on compiler-oriented techniques. An elegant and influential method for QDI synthesis has been developed by Martin and his students at Caltech [120, 122]. Martin specifies an asynchronous system as a set of concurrent processes which communicate on channels, using a CSP-like language. The language uses communication constructs from Hoare's CSP, sequential constructs from Dijkstra's guarded command language, and new constructs such as the *probe* (see [122]). The specification is then translated into a collection of gates and components which communicate on wires.

Martin's translation process is accomplished in several steps: (i) in *process decomposition*, a process is refined into an equivalent collection of interacting simpler processes; (ii) in *handshaking expansion*, each "communication channel" between processes is replaced by a pair of wires, and each atomic "communication action" is replaced by a handshaking protocol on the wires; (iii) in *production-rule expansion*, each handshaking expansion is replaced by a set of "production rules (PRs)", where each rule has a "guard" that insures it is activated (*i.e.,* "fires") under the same semantics as specified by the earlier handshaking expansion; and, finally, (iv) in *operator reduction*, PRs are grouped into clusters, and each cluster is then mapped to a basic hardware component. These steps include several optimizations and sub-steps, such as reshuffling, in step (ii); and state assignment, guard strengthening, and guard symmetrization, in step (iii). In most designs, a four-phase handshaking protocol is used (step (ii)), although two-phase handshaking can be used as well.

Martin's synthesis method has been automated by Burns [29, 26] and applied to many substantial examples, including a distributed mutual exclusion element [119, 26], a stack [122], and a multiply-accumulate unit [145]. The compiler includes algorithms for optimal transistor sizing [27]. (Designs for other datapath components, and for a microprocessor, using this method are described in the next two sections.) Martin's work has been extended by Akella and Gopalakrishnan in a system called *SHILPA* [4]. This method allows global shared variables, and uses flow analysis techniques to optimize resource allocation.

A different compiler-based approach was developed by van Berkel, Rem and others [210, 211, 161, 212] at Philips Research Laboratories and Eindhoven University of Technology, using the *Tangram* language. Tangram, based on CSP, is a specification language for concurrent systems. A system is specified by a Tangram program, which is then compiled by syntax-directed translation into an intermediate representation called a *handshake circuit*. A handshake circuit consists of a network of *handshake processes*, or *components*, which communicate asynchronously on channels using handshaking protocols. The circuit is then improved using peephole optimization and, finally, components are mapped to VLSI implementations.

As an example, the following is a Tangram program for a 1-place buffer, *BUF1*:

$$(a?W \& b!W) \cdot \mid [x : \mathbf{var}\ W \mid \#[a?x; b!x]] \mid$$

The buffer accepts input data on a and produces output data on b. The expression in parentheses is a declaration of the external ports of the module. The buffer has an *input port*, a, and an *output port*, b, handling data of some type, W. The remainder of the program, structured as a block, is called a *command*. A local variable x is defined for internal storage of data. The statement $\#[a?x; b!x]$ indicates that data is received on port a and stored in internal variable x; this data is then sent out on port b. The ";" operator indicates sequencing, and "#" indicates infinite repetition.

This Tangram program is translated into the handshake circuit of Figure 16. Each circle represents a handshake process or component. Each arc represents a channel, which connects an *active port* (indicated by a black dot) to a *passive port* (indicated by a white dot). Communication on a channel is by handshaking: an active port initiates a request and a passive port returns an acknowledgment.

In this example, port $\triangleright$ is the top-level port for the circuit, called *go*. The environment activates the buffer by an initial request on this passive port. This port is connected to a *repeater* process, which implements the repetition operator, "#". This process repeatedly initiates handshaking on channel c. Channel c is connected to a *sequencer* process, which implements the ";" operator. The sequencer first performs handshaking on channel d. When handshaking is complete, it then performs handshaking on channel e.

Channels d and e in turn are each connected to *transferrers*, labelled T. When the sequencer process initiates a request on channel d, the corresponding transferrer actively fetches data on input channel a and then transfers it to storage element x. Once the transfer is complete, the sequencer

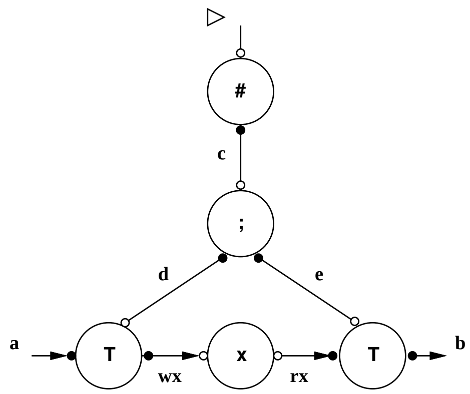

Figure 16: Handshake circuit for BUF1 example

initiates a request on channel e, causing the second transferrer to fetch the data from x and transfer it to output channel b.

A more complex example is 2-place buffer, *BUF2*, which can be described in terms of two 1-place buffers:

$$(a?W \& c!W) \cdot \mid [b : \textbf{chan } W \mid (BUF_1(a,b) \parallel BUF_2(b,c))] \mid$$

The program defines the buffer by the parallel composition, $\|$, of the two 1-place buffers, which are connected by an internal channel, b. The corresponding handshake circuit is shown in Figure 17. A *parallel* component implements the composition operator, $\|$. An initial request on its passive *go* port, results in parallel communication on channels l and r. These channels are both connected to a 1-place buffer B (indicated by double circles). The two buffers communicate through a *synchronizer* process (indicated by a black dot). If active requests arrive on both of its channels, lb and rb, the synchronizer first performs handshaking on channel b, then returns parallel acknowledgments on channels lb and rb. The attached *run* process is used to hide channel b; it simply acknowledges every request it receives.

The Tangram compiler has been successfully used at Philips for several experimental DSP designs for portable electronics, including a systolic RSA Converter, counters, decoders, image generators, and an error corrector for a digital compact cassette player [213]. A major goal this work is rapid turnaround time and low-power implementation.

Even though some peephole optimizations have been developed, Tangram is basically a syntax-directed translation method. Recently, two *resynthesis* methods have been proposed, by Pena/Cortadella [162] and Kolks *et al.* [99], which use aggressive peephole techniques to further optimize the resulting Tangram circuits. In each approach, handshaking components are clustered, formally specified as a single block, then resynthesized using STG techniques. A different approach has been proposed, which uses burst-mode techniques for the resynthesis step [78].

Brunvand and Sproull [21, 23] introduced an alternative compiler using occam specifications. Unlike the approaches of Martin and van Berkel, communication between processes is through two-phase handshaking, or transition-signaling. In their method, an occam specification is first compiled

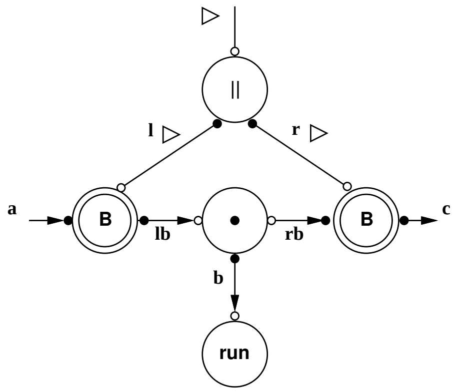

Figure 17: Handshake circuit for BUF2 example

into an unoptimized circuit using syntax-directed translation. Peephole optimization techniques are then applied to improve the resulting circuits. The circuits are then mapped to a library of transition-signaling components.

6.5 Timed Methods

While all asynchronous synthesis methods make some timing assumptions, much of the discipline is focused on minimizing these timing assumptions or at least localizing them into low-level modules. Myers [141, 142] contends that this approach often leads to additional time and space being spent in the circuit to deal with contingencies which never occur. In his timed state space method, rather than using timing analysis for post-synthesis-based optimizations to remove the unnecessary circuitry, timing information is used *during* synthesis to avoid generating unnecessary circuitry. His method is based on *timed event rule (ER) structures* [140], which can be automatically generated from high-level language representations such as CSP or VHDL. ER structures and Petri Nets use a similar representational semantics, but ER structures have a more concise syntax.

A *rule set* represents a causal dependence between events. It is notated as a 4-tuple, $\langle e, f, l, u \rangle$, consisting of an enabling event e, an enabled event f, a lower timing bound l, and an upper timing bound u. The rule is considered to be *satisfied* for the time between l and u after an enabling event has occurred. The use of this timing bound restricts the possible state space to events which can actually occur. A disjunction operator is used to permit both choice and exclusive-OR causality behaviors, although there is currently no provision for true OR behaviors. Myers' gate-level synthesis method permits larger circuits to be synthesized more efficiently as a result of the state space reduction.

A new set of timing analysis algorithms, based on a theory of geometric regions [173], allows a large number of discrete timed states to be condensed into a single region. The worst-case complexity of the algorithms is actually worse than discrete methods, but it has been shown that the region based approach works well in practice [174, 143, 16]. The method is automated in a tool called **ATACS**, which has been used to design a number of practical circuits. The tool has also been used at Intel Corporation in the design of an experimental high-performance instruction decoder.

Results show that, in the best case, circuits can be up to 40% smaller and 50% faster than those synthesized using methods which do not eliminate temporally unreachable states. Perhaps the most interesting aspect of this work is that it treats synchronous circuits as a subset of timed circuits and therefore provides a method for treating hybrid circuit structures consisting of both synchronous and asynchronous modules.

6.6 Design of Asynchronous Datapaths

As in synchronous design, different techniques and structures are used when designing datapaths and controllers.

Modern datapaths are often built using pipelines. However, the operation of synchronous and asynchronous pipelines are fundamentally different. In a synchronous pipeline, data advances at a fixed clock rate, in lock-step, through the pipe. Since function blocks in the stages may have different delays, the clock cycle must be set to the slowest stage. Furthermore, a stage's latency varies with the actual temperature, voltage, process and data inputs; therefore, additional delay margin is typically added to the clock cycle. Finally, the clock speed must be further reduced to avoid clock skew problems. As a result, under typical operating conditions, a synchronous pipeline may operate far slower than its potential performance.

In contrast, an asynchronous pipeline is not globally clocked. Therefore, in principle, each stage may pass data to its neighbor whenever the stage is done and the neighbor is free. Such *elastic* pipelines promise improved performance: different stages may operate at different speeds, and stages may complete early depending on the actual data. Of course, new overhead may be introduced, since each stage must now tell its neighbor when it is ready.

Sutherland introduced an elegant and influential approach to building asynchronous pipelines, which he called *micropipelines* [200]. A micropipeline has alternating computation stages separated by storage elements and control circuitry. This approach uses transition-signaling for control along with bundled data. Sutherland describes several designs for the storage elements, called "event-controlled registers", which respond symmetrically to rising and falling transitions on inputs. Such pipelines have been used by several researchers in the design of asynchronous microprocessors. Sutherland, Sproull, Molnar, and others at Sun Labs have recently designed a "counterflow microprocessor" based on micropipelines [195]. Micropipelines also form the basis for the Manchester ARM microprocessors, developed by Furber and the AMULET group [70, 71, 159].

Figure 18 illustrates the operation of a micropipeline with 4 stages. For simplicity, only the control is indicated. In practice, a bundled datapath is also used, along with event-controlled registers to store the data as it propagates down the pipe. A control stage of the pipeline consists of a C-element (described above). A C-element with two inputs and one output behaves as follows. If both inputs are 1, the output is 1; if both inputs are 0, the output is 0. Otherwise, if inputs have different values, the output holds its current value. The C-elements in the micropipeline behave similarly, except that each has one inverted input.

Initially, all wires in the micropipeline are at 0, as shown in Figure 18(a). When new data

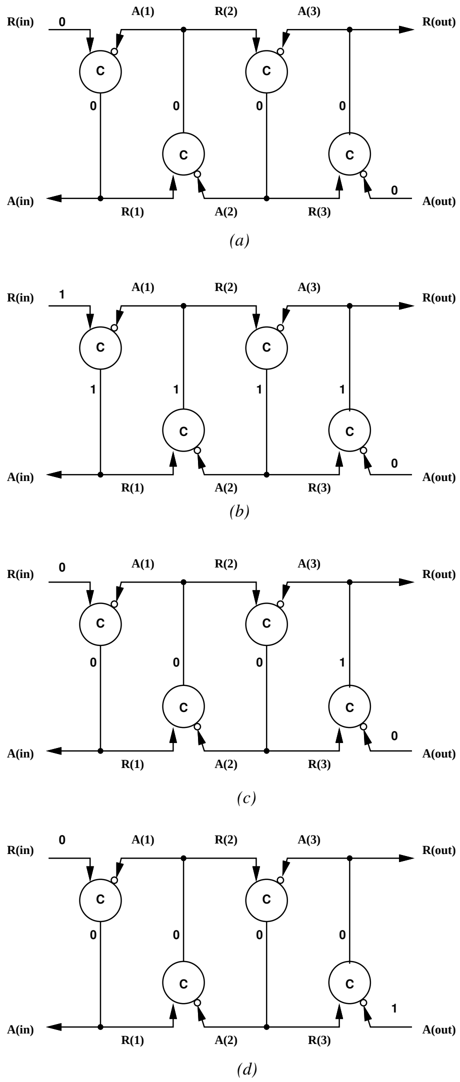

Figure 18: Micropipeline example

arrives, a request *R(in)* is asserted (*R(in)* goes to 1). The first C-element, C_1, becomes enabled, and its output makes a transition (to 1). This event has two consequences: the request is acknowledged on the left interface (transition on *A(in)*), and the request is forwarded to the right interface (transition on *R(1)*). The same behavior is repeated at the second stage: the request is acknowledged on the left interface (transition on *A(1)*), and the request is forwarded to the right interface (transition on *R(2)*). This process continues through stages 3 and 4, and a final request appears on the rightmost interface, *R(out)*. Effectively, the initial request propagates to the right through the pipe, and acknowledges are generated to the left. The resulting micropipeline configuration is shown in Figure 18(b). Note that since transition-signaling is used, only one request and one acknowledge are generated between each pair of stages. In contrast, a 4-phase (RZ) protocol would have required a second request/acknowledge sequence to reset the wires to their original values.

Since the initial request was acknowledged at the leftmost interface, *A(in)*, new data may now arrive and a second request, *R(in)*, can occur. Since *R(in)* is currently 1, a request is asserted by changing *R(in)* to 0. This request propagates through the micropipeline as before. The left interface is acknowledged (*A(in)* goes to 0), and the request is forwarded to the right interface (*R(1)* goes to 0). This process repeats at the 2nd and 3rd stages. However, once the second stage is acknowledged (*A(2)* goes to 0) and the request is made to the 4th stage (*R(3)* goes to 0), the propagation halts. Although *R(3)* made a transition (request from stage 3), stage 4 still contains the earlier data that was entered into the pipe. This data was not removed, since no *A(out)* transition has occurred. Since *A(out)* is still 0, no new transition can occur on *R(out)*. Instead, the new data is held in the 3rd stage. The resulting micropipeline configuration is shown in Figure 18(c). The micropipeline now contains data in the 3rd and 4th stages.

Finally, the original data in the 4th stage can be removed from the right interface, which then issues an acknowledge (*A(out)* goes to 1). At this point, the 4th stage issues a new *R(out)* transition (since *R(3)* is 0 and *A(out)* is 1), as data in stage 3 is moved to stage 4. The 3rd stage is acknowledged as well (*A(3)* goes to 0). The resulting configuration is shown in Figure 18(d). In practice, more complicated scenarios are possible, since data may be added and removed from the pipeline concurrently.

Although micropipelines use transition-signaling, other signaling conventions have been used in asynchronous pipelines as well. Williams [220], Martin [123] and van Berkel [211] have used 4-phase handshaking (the "return-to-zero" protocol) between stages. An alternative two-phase signaling scheme, called *LEDR* (level-encoded dual-rail), was introduced to combine advantages of both transition-signaling and four-phase [56].

Asynchronous pipelines have been designed for numerous applications: multiplication [200, 145], division [221], and DSP [211, 131]. The Williams and Horowitz self-timed divider [221] is especially impressive: the fabricated chip was twice as fast as comparable synchronous designs.

Research on asynchronous pipelines and datapaths is now proceeding in many directions. Several new asynchronous pipelining schemes have been proposed. Some emphasize low-power [65, 72] while others emphasize high-performance [53, 228, 72, 135]. In addition, several generalizations to asynchronous pipeline structures have been proposed: *rings* [220], *multi-rings* [194] and *2-dimensional micropipelines* [76]. Techniques to reduce the communication overhead between stages have been developed [220, 55, 79]. Liebchen and Gopalakrishnan have proposed a *reordering pipeline* [109] which allows the freezing and dynamic reordering of data within the pipe using "LockC" elements. Finally, low-power micropipeline structures have been introduced using adaptive scaling of supply voltage [146].

While pipelining is fundamental to high-performance systems, *sequencing* is the basic control operation in low-performance non-pipelined systems. A number of sequencer designs have been proposed [122, 211, 209, 161, 8, 64, 164].

There has been much recent research on asynchronous datapaths, beyond the above work on pipelines and sequencers. Much of this work is focused on *low-power design*, including designs for a digital compact cassette (DCC) error corrector [213], an infrared communications chip [117], an FFT [137], a FIR Filter bank [147], and cache [74], microprocessor [123, 69] and memory designs [203]. Others have developed techniques which use novel low-power devices, such as RSFQ [114].

Nowick introduced a method for *high-performance design*, called *speculative completion*, which uses a single-rail bundled datapath but also allows early completion [149]. The method uses a multi-slotted matched delay, where several of the delays are faster than the worst-case. These speculative delays allow early completion, and are disabled for worst-case data. The method has been applied to a high-performance Brent-Kung adder [155]; SPICE results indicate a 19-29% performance improvement over a comparable synchronous design.

Other datapath research has focused on architectures and protocols for *chip-to-chip communication*, including recent methods by Greenstreet [80] and Roiene [172]. An architecture for communication between synchronous and asynchronous chips has been developed by Chappel *et al.* [35].

6.7 Asynchronous Processor Design

Perhaps the greatest challenge in large-scale asynchronous design to date has been to combine the techniques for asynchronous controller and datapath synthesis, and build asynchronous processors. Asynchrony has in fact been present in processors from the early days when there was little distinction between synchronous and asynchronous circuits. Some level of persistent asynchrony has always been present since memory systems have been typically asynchronous. With the advent of virtual memory and cache memory systems, there is significant uncertainty about when a memory access will be resolved. In today's systems: a cache hit takes a few cycles, a cache miss requires approximately a hundred cycles, and a page miss takes 500,000 or more cycles to resolve. The result is that the processor-memory interface effectively uses a model delay and a 4-cycle protocol where the request is associated with the presentation of an address and a read or write command. The acknowledge is a ready signal indicating the requested operation has been completed. In a more direct fashion, machines such as the MU-5 and DDM1 (previously cited) have been constructed along more purely asynchronous lines.

However, it is only recently that such techniques have been applied systematically to the design of asynchronous microprocessors. The first asynchronous microprocessor-like device was developed in the early 1980's at Caltech by Chuck Seitz as class projects for his courses in VLSI and asynchronous circuit design. This annual effort went through several iterations and eventually became the CalTech Mosaic processor [184] that was subsequently used in the Cosmic Cube multiprocessor [183]. This binary N-cube connected multicomputer spawned a significant amount of activity in the parallel processing industry and was the forerunner of similarly architected machines vended by NCube corporation and Intel [196]. The Caltech tradition has subsequently been kept alive by Alain Martin and his students who are pursuing a more formally based design approach while continuing to in their efforts to use these techniques in the design of asynchronous microprocessors. The first QDI asynchronous microprocessor was developed by Martin *et al.* [123] in the late 1980's. The 16-bit design is almost fully quasi-delay-insensitive except for the memory interface. A 2μ CMOS version consumed 145mW at 5V and 6.7mW at 2V. A 1.6μCMOS version consumed 200mW at 5V and 7.6mW at 2V. The architecture was later re-implemented in GaAs. These early efforts were not competitive with synchronous microprocessor designs of the day in an architectural sense, however they did exhibit some of the benefits of asynchronous design methodology and provided realistic complexity for the design effort.

Subsequently, Martin and his students then became engaged in a much more architecturally realistic design effort to implement the asynchronous equivalent of a MIPS R3000 processor. While the R3000 is somewhat antiquated in terms of 1997 complexity when compared to commercial processors such as the MIPS R10000, DEC's Alpha 21164, and the HP 8000 or 8200, the R3000 does provide advanced architectural features such as on-chip caches, precise exceptions, register bypassing, branch prediction, and branch delay slot issues. The implementation was not simply a refabrication of the MIPS R3000 using asynchronous techniques but rather an attempt to implement the R3000 instruction set while exploiting architecture and logic design opportunities that are available to the asynchronous circuit designer. The result was a deeply pipelined design of a simplified R3000 which demonstrated an interesting trade-off between performance and low-power operation. By varying the supply voltage, the device could achieve better then expected performance at higher voltages, or low power operation at lower voltages. This flexibility is not inherently available from synchronous design methods but essentially is a free side effect using the quasi-delay insensitive CalTech design style. The inherent robustness of asynchronous design was also exploited to demonstrate increased pipeline elasticity over what could be expected from synchronous designs, as well as extending the pipeline model into the cache design as well. Of course today's microprocessors also utilize pipelined caches for CPU's which tolerate multiple outstanding misses without stalling. The result of the CalTech R3000 "MiniMIPS" experiment [118] is expected to run at 280 MIPS and dissipate 7 watts (at 3.3V and 75 degrees Celsius) or run at 150 MIPS dissipating 1 watt (at 2.0V and 75 degrees Celsius).

Recently, Furber and the AMULET group at Manchester University have fabricated two asynchronous implementations of the ARM microprocessor [70, 160, 71, 159]. The designs are based on micropipelined datapaths, and are part of a large-scale investigation of low-power techniques. The project addresses issues such as caching, exceptions and architectural optimization which are critical to the development of production-quality asynchronous machines. The Amulet1 used 2-cycle protocols and was disappointing in terms of both power and performance in comparison to its synchronous ARM equivalent. This is not surprising since the Amulet1 was the first significant asynchronous design attempted by the Manchester group and the design experience gave them a significant data point for analysis. The result was that, even though 2-cycle signaling is conceptually elegant, it resulted in circuits which were too large and slow, and which consumed too much power for their intended ARM application.

This Amulet1 results forced the design team to explore other protocol options for the subsequent Amulet2 effort. A version of the Amulet2 called the Amulet2e [69] has been fabricated. The Amulet2e is an Amulet2 processor core (93,000 transistors) coupled with 4K bytes of memory, in a 128-pin package containing 454,000 transistors. It was fabricated in a .5 micron CMOS technology and operates at 3.3 volts. The Amulet2e is intended for embedded controller applications. A number of architectural enhancements were made, including a jump target buffer and a flexible external interface called the *funnel*. The funnel permits 8, 16, and 32 bit external devices to be attached to the controller, as well as a DRAM based main memory system. Another key difference is that the Amulet2e uses 4-cycle signaling protocols, which results in improved performance, power consumption, and circuit areas. While the power consumption data has not yet been released, the performance of the Amulet2e is 38 Dhrystone MIPS, which is faster than the synchronous ARM710 but about half that of the recently announced ARM810 which uses the same process technology.

Sutherland, Sproull, Molnar, and others at Sun Labs have been developing an asynchronous *Counterflow Pipeline Processor* [195]. The architecture is based on a novel looped micropipeline, which synchronizes instructions and data flowing in opposite directions. The processor makes careful use of arbiters to regulate the synchronization.

Brunvand developed the *NSR* RISC microprocessor [22] at the University of Utah, using transition-signaling for control, bundled data, and a micropipelined datapath. The NSR was implemented using commercially available FPGA technology. The result of the NSR effort led to a more aggressive architecture called FRED [171, 170] which was implemented to the level of structural VHDL and subsequently analyzed. Fred is perhaps the closest attempt to create an equivalent to the modern microprocessor in that it provided speculative execution and precise interrupts while utilizing a novel architecture that was inspired, or perhaps constrained, by asynchronous design techniques. Fred was an architectural study and therefore was not actually fabricated. Other micropipelined-based RISC designs have been proposed by David *et al.* [47] and Ginosar and Michell [75].

A delay-insensitive microprocessor, *TITAC*, has been developed by Nanya *et al.* at Tokyo Institute of Technology [144]. The designers introduce several optimizations to improve performance. A different approach was proposed by Unger at Columbia University [209]. His "computers without clocks" use traditional asynchronous state machines for control logic, and a building block approach to design rather than compilation schemes. This approach requires a spectrum of timing assumptions to insure correct designs.

Finally, Dean's *STRiP* (self-timed RISC) processor at Stanford University combines synchronous and asynchronous features [54]. The design uses synchronous functional units in a globally-clocked pipeline. However, the clock rate may change dynamically based on the current contents of the pipeline, using a technique called *dynamic clocking*. The clock also is suspended during off-chip operations, such as input/output or access to a second-level cache. Using careful simulation, the design was shown to be almost twice as fast as a comparable synchronous design due solely to its asynchronous features.

6.8 Formal Verification and Timing Analysis

The above survey indicates an impressive surge of activity in the design of asynchronous controllers, datapaths and processors. However, design techniques alone cannot make asynchronous circuits commercially viable. In synchronous design, many ancillary technology components are needed to insure the correctness of designs, including verification, timing analysis and testing. These techniques are especially critical for asynchronous design because of their inherent subtlety. This subsequent sections briefly sketch some of the recent work on validation of asynchronous designs.

Due to the large variety of asynchronous design approaches, it is difficult to find a unified approach to the analysis and verification of all asynchronous circuits. For speed-independent and delay-insensitive systems, though, Hoare's *CSP* [83] and Milner's *CCS* [133] have been especially effective as formal underpinnings.

Rem, Snepscheut and Udding's *trace theory* [167], based on CSP, has been used both for specification and formal verification. In trace theory, the behavior of a concurrent system is described by the set of possible *traces*, or sequences of events, which may be observed. Each trace describes one possible interleaved behavior of the system. The traces are combined into a set, which defines the observable behavior of the system. Dill [58, 59] and Ebergen [62] have built effective verification tools for SI and DI circuits based on trace theory. In Dill's theory, an implementation and specification are each modeled by trace sets. These sets are compared using a formal relation called *conformance*, which defines precisely when an implementation meets its specification. Dill has uncovered bugs in published circuits using the verifier. More efficient algorithms for approximate verification (allowing occasional false negatives) have been developed by Beerel *et al.* [11].

Dill's verifier effectively checks for safety violations (where a design has incorrect behavior), but does not check for liveness violations (where a design has deadlock or livelock). Dill also introduced a theory of *complete trace structures* [58], based on Buchi automata, which can model general liveness properties. Although these general verification algorithms may be too expensive to apply in practice, a verifier has been developed for a constrained class of specifications [197]. Other methods use a restricted notion of liveness that can be easily checked [62, 77, 226]. A method which uses Signal Graphs for verification of properties of speed-independent circuits has been proposed by Kishinevsky *et al.* [96]. Another approach, by Kol, Ginosar and Samuel [98], uses *state charts* to verify both safety and liveness properties.

An alternative verification method based on CCS has been proposed by Birtwistle, Stevens, *et al.* [198, 197]. CCS has been successfully used for the specification of several asynchronous designs, including a token ring arbiter and SCSI controller. Specifications can then be checked for deadlock, safety and liveness properties using a modal logic. A substantial specification has been developed for the AMULET processor [18], with detailed models for the different instruction classes.

The above verification techniques handle SI and DI circuits and protocols, and therefore are not concerned with timing. However, timing is critical for the analysis and verification of many asynchronous systems. A general model for timed systems was introduced by Alur and Dill [5]. Timing analysis and verification methods for asynchronous state machines with bounded delays were developed by Devadas *et al.* [57] and Chakraborty *et al.* [31, 30]. Methods using Timed Petri Nets have been developed by Rokicki [173], Semenov *et al.* [187] and Verlind *et al.* [219]. Williams [220] and Burns [27] have introduced methods to analyze the performance of systems. A notion of *timing-reliability* was proposed by Kuwako and Nanya [104]. Timing and hazard analysis tools have been developed by Ashkinazy *et al.* [7]. Other recent work has focused on timing analysis to determine minimum and maximum separation of events in a concurrent circuit or system [128, 87, 86]. Such analysis can aid in both the optimization and verification of asynchronous designs.

6.9 Testing and Synthesis-for-Testability

While formal verification is used to validate designs, testing is needed to validate the correctness of fabricated implementations. Testing and synthesis-for-testability play a major role in the industrial production of synchronous chips. However, the testing of asynchronous circuits is complicated by their special design constraints. For example, asynchronous circuits may use redundant logic to eliminate hazards, but redundant logic makes testing more difficult.

Initial results on the testing of speed-independent circuits include work by Beerel and Meng [10] and Martin and Hazewindus [124]. These papers indicate that certain classes of speed-independent circuits are "self-testing" with respect to stuck-at faults, where certain faults will cause the circuit to halt. Beerel and Meng generalized their approach to handle stuck-at faults in timed control circuits [13].

A general synthesis-for-testability method was proposed by Keutzer, Lavagno and Sangiovanni-Vincentelli [93] which considers both stuck-at and path-delay faults in combinational circuits. The method uses algebraic transformations to produce hazard-free and fully-testable multi-level logic. This work was extended by Nowick, Jha and Cheng [154], to include a richer set of transformations and to handle a more general class of hazards.

Subsequent research has focused on testing of handshaking circuits and micropipelines. Roncken *et al.* [178, 175] at Philips Research Laboratories have developed techniques and tools for partial scan of handshaking circuits. The method is now used in the Tangram synthesis compiler. A novelty of the approach is that testability is insured at the highest-level, *i.e.,* by modifying the Tangram program specification. The method was used in the design of a DCC error corrector, where it led to 99.9% stuck-at output fault coverage at an expense of less than 3% additional area [175, 213]. More realistic fault models, such as for I_{DDQ} testing, have recently been addressed as well [177].

The most prevalent "design-for-test" technique in the synchronous domain has been the use of a serial *scan path* technique, which effectively creates a shift register out of the storage components on the chip. The external interface provides both read and write capability to this shift register. The method works well for synchronous systems, since the concept of state corresponds to the contents of the storage elements in the circuit after a particular clock. However, the inherent *temporally decoupled* nature of asynchronous circuits tends to make the concept of "total system state" counter-productive. The implication is that the design for test methods developed for synchronous circuits are not appropriate for asynchronous systems. However, a surprisingly analogous technique was developed by Khoche [95, 94]. This technique applies to macromodular, micropipelined self-timed circuits. The key idea is that, while the circuits operate asynchronously in normal mode, the scan mode operation is synchronous and the clock propagates in the backward direction along the micropipeline. Khoche demonstrated that the overhead of adding *scanability* to asynchronous circuits is commensurate with the overhead for synchronous circuits. More recent approaches to testing micropipelines have been developed as well [181, 176].

An issue related to testing is *initializability,* which is the process of driving a circuit at power-up to a known state. Initializability is also often required by automatic test pattern generators. Two recent methods for asynchronous initializability have been developed [32, 192].

One interesting area of asynchronous circuit testing that is just beginning to be studied is the issue of hazards. Asynchronous circuits by nature often contain redundant logic to prevent hazards. This is a particularly problematic issue with respect to testing. Namely, if the circuit contains redundant logic to prevent hazards, then how is this redundancy tested at the chip's external pins? Fundamentally, a solution requires that redundancy path analysis connections be exported to the pad ring of the chip. The increase of area, power consumption and packaging costs of the device to support this capability directly is a problem. However, integrating this analysis capability within the scan path is an interesting option. The solution and its complexity remain open research issues.

7 Conclusions

This monograph has provided an introduction to the current (1997) state of the art of asynchronous circuit design. The focus has been on motivations, fundamental concepts, design methods, and the physical artifacts and results that have been the result of these design styles.

The current status of asynchronous circuits is that it is a growing research area that has yet to have significant impact on the design of commercial integrated circuits. However there are significant signs that asynchronous circuits may become more of a mainstream discipline in the future. Serious asynchronous circuit efforts exist in corporate research labs, namely at Sun Microsystems, Philips, and Intel. The goals of these groups vary from increased performance at Sun and Intel, to reduced power consumption at Philips.

Another promising sign is the rapid restructuring of the semiconductor design and electronic design automation (EDA) companies. The industry leaders have joined together to form the **VSI Alliance.** VSI stands for virtual socket interface, and the purpose of the alliance is to create standards for design technology to be exchanged and reused easily. The goal is to prevent the need to redesign circuits that already exist in another company. Rapid changes in the marketplace result in rapidly decreasing design cycle requirements. The result is that companies simply do not have the time to design each new product from scratch. It is becoming more cost effective to buy macrocells and processor cores and combine them to create a new design. The high level of competition however requires that new products take full advantage of the latest integrated circuit technology. The new technology is faster but creates a new set of timing problems which must be managed carefully in a synchronous design. The inherent ease of asynchronous module composition nicely fits the requirements of this new industry model. The composability advantage is directly due to the fact that asynchronous circuits explicitly export their timing requirements at their interfaces via their signaling protocols.

More fundamental motivations come from basic integrated circuit technology trends. As transistor sizes shrink and as chips become significantly larger, the cost of distributing increasingly faster clock frequencies with minimal skew becomes too expensive. The expense comes both in terms of reduced performance and increased power consumption. The basis for this belief is that the speed improvement of wires versus transistors is disparate. Gate delays improve by 150% while wire delays improve by only 20% in each new integrated circuit process generation. Improvements in wire technology are to be expected but the disparity is equally likely to remain. The result is that given the trend of 10% growth in the physical dies size per generation, less than 10% of the die area will be reachable in a single clock period when the feature size reaches .06μm [125]. The result is that it is highly unlikely that one billion transistor chips will be both cost effective and synchronous. This monograph has provided a number of options that may well be the basis for a future solution to this critical problem.

References

[1] W. B. Ackerman and J. B. Dennis. VAL - A Value-Oriented Algorithmic Language Preliminary Reference Manual. Technical Report LCS/TR-218, Massachusetts Institute Technology, Computer Science Department, 1979.

[2] M. Afhahi and C. Svensson. Performance of Synchronous and Asynchronous Schemes for VLSI Systems. *IEEE Transactions on Computers*, 41(7):858–872, July 1992.

[3] F. Aghdasi. Synthesis of asynchronous sequential machines for VLSI applications. In *Proceedings of the 1991 International Conference on Concurrent Engineering and Electronic Design Automation (CEEDA)*, pages 55–59, March 1991.

[4] V. Akella and G. Gopalakrishnan. SHILPA: a high-level synthesis system for self-timed circuits. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 587–91. IEEE Computer Society Press, November 1992.

[5] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 1(126):183–235, 1994.

[6] D.B. Armstrong, A.D. Friedman, and P.R. Menon. Realization of asynchronous sequential circuits without inserted delay elements. *IEEE Transactions on Computers*, C-17(2):129–134, February 1968.

[7] A. Ashkinazy, D. Edwards, C. Farnsworth, G. Gendel, and S. Sikand. Tools for validating asynchronous digital circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 12–21. IEEE Computer Society Press, November 1994.

[8] A.M. Bailey and M.B. Josephs. Sequencer circuits for VLSI programming. In *Proceedings of the Working Conference on Asynchronous Design Methodologies*, pages 82–90. IEEE Computer Society Press, 1995.

[9] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley, 1990.

[10] P. Beerel and T. Meng. Semi-Modularity and Self-Diagnostic Asynchronous Control Circuits. In Carlo H. Sequin, editor, *Proceedings of the 1991 University of California/Santa Cruz Conference*, pages 103–117. The MIT Press, 1991.

[11] P.A. Beerel, J. Burch, and T. Meng. Efficient verification of determinate speed-independent circuits. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 261–267. IEEE Computer Society Press, November 1993.

[12] P.A. Beerel and T. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 581–586. IEEE Computer Society Press, November 1992.

[13] P.A. Beerel and T. H.-Y. Meng. Testability of asynchronous timed control circuits with delay assumptions. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 446–451. ACM, June 1991.

[14] P.A. Beerel, K.Y. Yun, and W.C. Chou. Optimizing average-case delay in technology mapping of burst-mode circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 244–260. IEEE Computer Society Press, November 1996.

[15] J. Beister. A unified approach to combinational hazards. *IEEE Transactions on Computers*, C-23(6):566–575, June 1974.

[16] W. Belluomini and C.J. Myers. Efficient timing analysis algorithms for timed state space exploration. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async97)*. IEEE Computer Society Press, April 1997.

[17] I. Benko and J.C. Ebergen. Delay-insensitive solutions to the committee problem. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 228–237. IEEE Computer Society Press, November 1994.

[18] G. Birtwistle and Y. Liu. Specification of the Manchester Amulet 1: Top level Specification. Computer Science Department Technical Report, University of Calgary, December 1994.

[19] J.G. Bredeson. Synthesis of multiple-input change hazard-free combinational switching circuits without feedback. *International Journal of Electronics (GB)*, 39(6):615–624, December 1975.

[20] J.G. Bredeson and P.T. Hulina. Elimination of static and dynamic hazards for multiple input changes in combinational switching circuits. *Information and Control*, 20:114–224, 1972.

[21] E. Brunvand. Translating concurrent communicating programs into asynchronous circuits. Technical Report CMU-CS-91-198, Carnegie Mellon University, 1991. Ph.D. Thesis.

[22] E. Brunvand. The NSR processor. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 428–435. IEEE Computer Society Press, January 1993.

[23] E. Brunvand and R. F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 262–265. IEEE Computer Society Press, November 1989.

[24] J.A. Brzozowski and J.C. Ebergen. Recent developments in the design of asynchronous circuits. Technical Report CS-89-18, University of Waterloo, Computer Science Department, 1989.

[25] J.A. Brzozowski and J.C. Ebergen. On the delay-sensitivity of gate networks. *IEEE Transactions on Computers*, 41(11):1349–1360, November 1992.

[26] S. M. Burns. Automated compilation of concurrent programs into self-timed circuits. Technical Report Caltech-CS-TR-88-2, California Institute of Technology, 1987. M.S. Thesis.

[27] S.M. Burns. Performance analysis and optimization of asynchronous circuits. Technical Report Caltech-CS-TR-91-01, California Institute of Technology, 1991. Ph.D. Thesis.

[28] S.M. Burns. General condition for the decomposition of state holding elements. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 48–57. IEEE Computer Society Press, November 1996.

[29] S.M. Burns and A.J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and T.F. Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 35–50. MIT Press, Cambridge, MA, 1988.

[30] S. Chakraborty and D.L. Dill. More accurate polynomial-time min-max timing simulation. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async97)*. IEEE Computer Society Press, April 1997.

[31] S. Chakraborty, D.L. Dill, K.-Y. Chang, and K.Y. Yun. Timing analysis for extended burst-mode circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async97)*. IEEE Computer Society Press, April 1997.

[32] S.T. Chakradhar, S. Banerjee, R.K. Roy, and D.K. Pradhan. Synthesis of initializable asynchronous circuits. *IEEE Transactions on VLSI Systems*, 4(2):254–262, June 1996.

[33] T. J. Chaney and C. E. Molnar. Anomalous Behaviour of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers*, C–22(4):421–422, 1973.

[34] T.J. Chaney, S.M. Ornstein, and W.M. Littlefield. Beware the synchronizer. In *IEEE 6th International Computer Conference*, pages 317–319, 1972.

[35] J.F. Chappel and S.G. Zaky. A delay-controlled phase-locked loop to reduce timing errors in synchronous/asynchronous communication links. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 156–165. IEEE Computer Society Press, November 1994.

[36] V. L. Chi. Salphasic Distribution of Clock Signals for Synchronous Systems. *IEEE Transactions on Computers*, 43(5):597–602, May 1994.

[37] J.-S. Chiang and D. Radhakrishnan. Hazard-free design of mixed operating mode asynchronous sequential circuits. *International Journal of Electronics*, 68(1):23–37, January 1990.

[38] T.-A. Chu. Synthesis of self-timed vlsi circuits from graph-theoretic specifications. Technical Report MIT-LCS-TR-393, Massachusetts Institute of Technology, 1987. Ph.D. Thesis.

[39] T.-A. Chu. Automatic synthesis and verification of hazard-free control circuits from asynchronous finite state machine specifications. In *Proceedings of the IEEE International Conference on Computer Design*, pages 407–413. IEEE Computer Society Press, 1992.

[40] T.-A. Chu, N. Mani, and C.K.C. Leung. An efficient critical race-free state assignment technique for asynchronous finite state machines. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 2–6. ACM, June 1993.

[41] H.Y.H. Chuang and S. 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.

[42] W.A. Clark. Macromodular computer systems. In *Proceedings of the Spring Joint Computer Conference, AFIPS*, April 1967.

[43] F. Commoner, A. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5(5):511–523, October 1971.

[44] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Complete state encoding based on the theory of regions. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 36–47. IEEE Computer Society Press, November 1996.

[45] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Methodology and tools for state encoding in asynchronous circuit synthesis. In *33rd ACM/IEEE Design Automation Conference*, June 1996.

[46] J. Cortadella, L. Lavagno, P. Vanbekbergen, and A. Yakovlev. Designing asynchronous circuits from behavioural specifications with internal conflicts. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 106–115. IEEE Computer Society Press, November 1994.

[47] I. David, R. Ginosar, and M. Yoeli. Self-timed implementation of a reduced instruction set computer. Technical Report 732, Technion and Israel Institute of Technology, October 1989.

[48] A. Davis, B. Coates, and K. Stevens. Automatic synthesis of fast compact self-timed control circuits. In *1993 IFIP Working Conference on Asynchronous Design Methodologies (Manchester, England)*, 1993.

[49] A.L. Davis. The architecture and system method of DDM-1: A recursively-structured data driven machine. In *Proc. Fifth Annual Symposium on Computer Architecture*, 1978.

[50] A.L. Davis. A data-driven machine architecture suitable for VLSI implementation. In C.L. Seitz, editor, *Proceedings of the Caltech Conference on Very Large Scale Integration*, pages 479–494, January 1979.

[51] Al Davis. Synthesizing Asynchronous Circuits: Practice and Experience. In *Asynchronous Digital Circuit Design*, pages 104–150, 1995.

[52] A.L. Davis, B. Coates, and K. Stevens. The post office experience: Designing a large asynchronous chip. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 409–418. IEEE Computer Society Press, January 1993.

[53] P. Day and J.V. Woods. Investigation into micropipeline latch design styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

[54] M.E. Dean. STRiP: A self-timed RISC processor architecture. Technical report, Stanford University, 1992. Ph.D. Thesis.

[55] M.E. Dean, D.L. Dill, and M. Horowitz. Self-timed logic using current-sensing completion detection (CSCD). In *Proceedings of the IEEE International Conference on Computer Design*. IEEE Computer Society Press, October 1991.

[56] M.E. Dean, T.E. Williams, and D.L. Dill. Efficient self-timing with level-encoded 2-phase dual-rail (LEDR). In Carlo Sequin, editor, *Advanced Research in VLSI : Proceedings of the 1991 University of California Santa Cruz Conference*, pages 55–70. The MIT Press, 1991. *ISBN 0-262-19308-6*.

[57] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Verification of asynchronous interface circuits with bounded wire delays. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 188–195. IEEE Computer Society Press, November 1992.

[58] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, Cambridge, MA, 1989.

[59] D.L. Dill, S.M. Nowick, and R.F. Sproull. Specification and automatic verification of self-timed queues. *Formal Methods in System Design*, 1(1):29–60, July 1992.

[60] D. W. Dobberpuhl and et al. A 200-MHz 64-bit Dual-issue CMOS Microprocessor. *Digital Technical Journal*, 4(4):35–50, 1993.

[61] J.C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[62] J.C. Ebergen. A verifier for network decompositions of command-based specifications. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 310–318. IEEE Computer Society Press, January 1993.

[63] E.B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9(2):90–99, 1965.

[64] C. Farnsworth, D.A. Edwards, J. Liu, and S.S. Sikand. A hybrid asynchronous system design environment. In *Proceedings of the Working Conference on Asynchronous Design Methodologies*, pages 91–98. IEEE Computer Society Press, 1995.

[65] C. Farnsworth, D.A. Edwards, and S.S. Sikand. Utilising dynamic logic for low power consumption in asynchronous circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 186–194. IEEE Computer Society Press, November 1994.

[66] J. Frackowiak. Methoden der analyse und synthese von hasardarmen schaltnetzen mit minimalen kosten I. *Elektronische Informationsverarbeitung und Kybernetik*, 10(2/3):149–187, 1974.

[67] A.D. Friedman and P.R. Menon. Synthesis of asynchronous sequential circuits with multiple-input changes. *IEEE Transactions on Computers*, C-17(6):559–566, June 1968.

[68] R.M. Fuhrer, B. Lin, and S.M. Nowick. Symbolic hazard-free minimization and encoding of asynchronous finite state machines. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 604–611, November 1995.

[69] S. Furber, P. Day, J. Garside, N. Paver, and S. Temple. Amulet2e. In *EMSYS96 - OMI Sixth Annual Conference*. IOS Press, 1996. ISBN 90 5199 300 5.

[70] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In *Proceedings of VLSI 93*, pages 5.4.1 –5.4.10, September 1993.

[71] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, S. Temple, and J.V. Woods. The design and evaluation of an asynchronous microprocessor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 217–220. IEEE Computer Society Press, October 1994.

[72] S.B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 11–16. IEEE Computer Society Press, November 1996.

[73] S.B. Furber and P. Woods. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[74] J.D. Garside, S. Temple, and R. Mehra. The AMULET2e cache system. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 208–217. IEEE Computer Society Press, November 1996.

[75] R. Ginosar and N. Michell. On the potential of asynchronous pipelined processors. Technical Report UUCS-90-015, VLSI Systems Research Group, University of Utah, 1990.

[76] G. Gopalakrishnan. Micropipeline wavefront arbiters using lockable C-elements. *IEEE Design and Test*, 11(4):55–64, Winter 1994.

[77] G. Gopalakrishnan, E. Brunvand, N. Michell, and S.M. Nowick. A correctness criterion for asynchronous circuit validation and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(11):1309–1318, November 1994.

[78] G. Gopalakrishnan, P. Kudva, and E. Brunvand. Peephole optimization of asynchronous macromodule networks. In *Proceedings of the IEEE International Conference on Computer Design*, pages 442–446. IEEE Computer Society Press, October 1994.

[79] E. Grass, R.C.S. Morling, and I. Kale. Activity-monitoring completion-detection (AMCD): a new single rail approach to achieve self timing. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 143–149. IEEE Computer Society Press, November 1996.

[80] M. Greenstreet. Implementing a STARI chip. In *Proceedings of the IEEE International Conference on Computer Design*, pages 38–43. IEEE Computer Society Press, October 1995.

[81] S. Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[82] A.B. Hayes. Stored state asynchronous sequential circuits. *IEEE Transactions on Computers*, C-30(8):596–600, August 1981.

[83] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[84] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161–190, March 1954.

[85] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(4):275–303, April 1954.

[86] H. Hulgaard and S.M. Burns. Bounded delay timing analysis of a class of CSP programs with choice. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 2–11. IEEE Computer Society Press, November 1994.

[87] H. Hulgaard, S.M. Burns, T. Amon, and G. Borriello. Practical applications of an efficient time separation of events algorithm. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 146–151. IEEE Computer Society Press, November 1993.

[88] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design.* John Wiley and Sons, 1979.

[89] M.B. Josephs. Receptive process theory. *Acta Informatica*, 29:17–31, 1992.

[90] M.B. Josephs, P.G. Lucassen, J.T. Udding, and T. Verhoeff. Formal design of an asynchronous DSP counterflow pipeline: A case study in handshake algebra. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 206–215. IEEE Computer Society Press, November 1994.

[91] M.B. Josephs and J.T. Udding. An overview of D-I algebra. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 329–338. IEEE Computer Society Press, January 1993.

[92] W. Keister, A. E. Ritchie, and S. H. Washburn. *The Design of Switching Circuits.* Van Nostrand, Princeton, New Jersey, 1951.

[93] K. Keutzer, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis for testability techniques for asynchronous circuits. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 326–329. IEEE Computer Society Press, November 1991.

[94] A. Khoche. *Testing Macro-Module Based Self-Timed Circuits.* PhD thesis, University of Utah, 1996.

[95] A. Khoche and E. Brunvand. Testing micropipelines. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 239–246. IEEE Computer Society Press, November 1994.

[96] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. Analysis and identification of speed-independent circuits on an event model. *Formal Methods in System Design*, 4(1):33–75, January 1994.

[97] M.A. Kishinevsky, A.Y. Kondratyev, A.R. Taubin, and V.I. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* John Wiley and Sons Ltd., 1994.

[98] R. Kol, R. Ginosar, and G. Samuel. Statechart methodology for the design, validation, and synthesis of large scale asynchronous systems. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 164–174. IEEE Computer Society Press, November 1996.

[99] T. Kolks, S. Vercauteren, and B. Lin. Control resynthesis for control-dominated asynchronous designs. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 233–243. IEEE Computer Society Press, November 1996.

[100] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pages 56–62. ACM, June 1994.

[101] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *33rd ACM/IEEE Design Automation Conference*, June 1996.

[102] P. Kudva, G. Gopalakrishnan, H. Jacobson, and S.M. Nowick. Synthesis of hazard-free customized CMOS complex-gate networks under multiple-input changes. In *33rd ACM/IEEE Design Automation Conference*, pages 77–82, June 1996.

[103] D.S. Kung. Hazard-non-increasing gate-level optimization algorithms. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 631–634. IEEE Computer Society Press, November 1992.

[104] M. Kuwako and T. Nanya. Timing-reliability evaluation of asynchronous circuits based on different delay models. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 22–31. IEEE Computer Society Press, November 1994.

[105] M. Ladd and W. P. Birmingham. Synthesis of multiple-input change asynchronous finite state machines. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 309–314. ACM, June 1991.

[106] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 302–308. ACM, June 1991.

[107] L. Lavagno, C.W. Moon, R.K. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proceedings of the 29th IEEE/ACM Design Automation Conference*, pages 568–572. IEEE Computer Society Press, June 1992.

[108] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits.* Kluwer Academic, 1993.

[109] A. Liebchen and G. Gopalakrishnan. Dynamic reordering of high latency transactions using a modified micropipeline. In *Proceedings of the IEEE International Conference on Computer Design*, pages 336–340. IEEE Computer Society Press, 1992.

[110] B. Lin and S. Devadas. Synthesis of hazard-free multi-level logic under multiple-input changes from binary decision diagrams. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 542–549. IEEE Computer Society Press, November 1994.

[111] K.-J. Lin and C.-S. Lin. Automatic synthesis of asynchronous circuits. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 296–301. ACM, June 1991.

[112] C.N. Liu. A state variable assignment method for asynchronous sequential switching circuits. *Journal of the ACM*, 10:209–216, April 1963.

[113] P.G. Lucassen and J.T. Udding. On the correctness of the sproull counterflow pipeline processor. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 112–120. IEEE Computer Society Press, November 1996.

[114] M. Maezawa, I. Kurosawa, Y. Kameda, and T. Nanya. Pulse-driven dual-rail logic gate family based on rapid single-flux-quantum (RSFQ) devices for asynchronous circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 134–142. IEEE Computer Society Press, November 1996.

[115] G. Mago. Realization methods for asynchronous sequential circuits. *IEEE Transactions on Computers*, C-20(3):290–297, March 1971.

[116] W.C. Mallon and J.T. Udding. Using metrics for proof rules for recursively defined delay-insensitive specifications. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async97)*. IEEE Computer Society Press, April 1997.

[117] A. Marshall, B. Coates, and P. Siegel. The design of an asynchronous communications chip. *IEEE Design and Test*, 11(2):8–21, Summer 1994.

[118] A. J. Martin, A. Lines, R. Manohar, M Nystroem, P. Penzes, R. Southworth, U. Cummings, and T.K. Lee. The Design of an Asynchronous MIPS R3000 Processor. In Richard B. Brown and Alexander T. Ishii, editors, *17th Conference on Advanced Research in VLSI*, pages 164–181. IEEE, IEEE Computer Society Press, 1997.

[119] A.J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration*, pages 245–60. CSP, Inc., 1985.

[120] A.J. Martin. Compiling communicating processes into delay-insensitive vlsi circuits. *Distributed Computing*, 1:226–234, 1986.

[121] A.J. Martin. The limitation to delay-insensitivity in asynchronous circuits. In W.J. Dally, editor, *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, pages 263–278. MIT Press, Cambridge, MA, 1990.

[122] A.J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Institute on Concurrent Programming, pages 1–64. Addison-Wesley, Reading, MA, 1990.

[123] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. The design of an asynchronous microprocessor. In *1989 Caltech Conference on Very Large Scale Integration*, 1989.

[124] A.J. Martin and P.J. Hazewindus. Testing delay-insensitive circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference*, pages 118–132. MIT Press, 1991.

[125] Doug Matzge. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, 30(9):37–39, September 1997.

[126] E.J. McCluskey. *Introduction to the Theory of Switching Circuits*. McGraw-Hill, New York, NY, 1965.

[127] E.J. McCluskey. *Logic Design Principles: with emphasis on testable semicustom circuits*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[128] K. McMillan and D.L. Dill. Algorithms for interface timing verification. In *Proceedings of the IEEE International Conference on Computer Design*, pages 48–51. IEEE Computer Society Press, October 1992.

[129] C. Mead and L. Conway. *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, Reading, MA, 1980. C.L. Seitz, System Timing.

[130] T. H.-Y. Meng, R.W. Brodersen, and D.G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(11):1185–1205, November 1989.

[131] T.H. Meng. *Synchronization Design for Digital Systems.* Kluwer Academic Publishers, Boston, MA, 1991.

[132] R.E. Miller. *Switching Theory. Volume II: Sequential Circuits and Machines.* John Wiley and Sons, New York, NY, 1965.

[133] R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.

[134] C.E. Molnar, T.-P. Fang, and F.U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. CSP, Inc., 1985.

[135] C.E. Molnar, I.W. Jones, B. Coates, and J. Lexau. A fifo ring oscillator performance experiment. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async97).* IEEE Computer Society Press, April 1997.

[136] C.W. Moon, P.R. Stephan, and R.K. Brayton. Synthesis of hazard-free asynchronous circuits from graphical specifications. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 322–325. IEEE Computer Society Press, November 1991.

[137] S.V. Morton, S.S. Appleton, and M.J. Liebelt. An event controlled reconfigurable multi-chip FFT. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 144–153. IEEE Computer Society Press, November 1994.

[138] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits I. Digital Computer Laboratory 75, University of Illinois, November 1956.

[139] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits II. Digital Computer Laboratory 78, University of Illinois, March 1957.

[140] C. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits.* PhD thesis, Stanford University, 1995.

[141] C. Myers and T. Meng. Synthesis of timed asynchronous circuits. In *Proceedings of the IEEE International Conference on Computer Design*, pages 279–284. IEEE Computer Society Press, October 1992.

[142] C. Myers and T. Meng. Synthesis of Timed Asynchronous Circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[143] C.J. Myers, T.G. Rokicki, and T. H.-Y. Meng. Automatic synthesis of gate-level timed circuits with choice. In W.J. Dally, J.W. Poulton, and A.T. Ishii, editors, *Advanced Research in VLSI : Proceedings of the 1995 University of North Carolina Conference*, pages 42–58. IEEE Computer Society Press, 1995.

[144] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: design of a quasi-delay-insensitive microprocessor. *IEEE Design and Test*, 11(2):50–63, Summer 1994.

[145] C.D. Nielsen and A. Martin. The design of a delay-insensitive multiply-accumulate unit. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 379–388. IEEE Computer Society Press, January 1993.

[146] L.S. Nielsen, C. Niessen, J. Sparso, and K. van Berkel. Low-Power Operation Using Self-Timed Circuits and Adaptive Scaling of the Supply Voltage. *IEEE Transactions on VLSI*, 2(4):7, 1994.

[147] L.S. Nielsen and J. Sparso. A low-power asynchronous data path for a fir filter bank. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 197–207. IEEE Computer Society Press, November 1996.

[148] S.M. Nowick. Automatic synthesis of burst-mode asynchronous controllers. Technical report, Stanford University, March 1993. Ph.D. Thesis (available as Stanford University Computer Systems Laboratory technical report, CSL-TR-95-686, Dec. 95).

[149] S.M. Nowick. Design of a low-latency asynchronous adder using speculative completion. *IEE Proceedings - Computers and Digital Techniques*, 143(5):301–307, 1996.

[150] S.M. Nowick and B. Coates. UCLOCK: automated design of high-performance unclocked state machines. In *Proceedings of the IEEE International Conference on Computer Design*, pages 434–441. IEEE Computer Society Press, October 1994.

[151] S.M. Nowick, M.E. Dean, D.L. Dill, and M. Horowitz. The design of a high-performance cache controller: a case study in asynchronous synthesis. *INTEGRATION, the VLSI journal*, 15(3):241–262, October 1993.

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

[153] S.M. Nowick and D.L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(8):986–997, August 1995.

[154] S.M. Nowick, N.K. Jha, and F.-C. Cheng. Synthesis of asynchronous circuits for stuck-at and robust path delay fault testability. In *Proceedings of the Eighth International Conference on VLSI Design (VLSI Design 95)*. IEEE Computer Society Press, January 1995.

[155] S.M. Nowick, K.Y. Yun, P.A. Beerel, and A.E. Dooply. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async97)*. IEEE Computer Society Press, April 1997.

[156] S.M. Nowick, K.Y. Yun, and D.L. Dill. Practical asynchronous controller design. In *Proceedings of the IEEE International Conference on Computer Design*, pages 341–345. IEEE Computer Society Press, October 1992.

[157] S.S. Patil. An Asynchronous Logic Array. Technical Report Technical Memorandom 62, Massachusetts Institute of Technology, Project MAC, 1975.

[158] P. Patra and D.S. Fussell. Efficient building blocks for delay insensitive circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 196–205. IEEE Computer Society Press, November 1994.

[159] N.C. Paver. The design and implementation of an asynchronous microprocessor. Technical report, University of Manchester, June 1994. Ph.D. Thesis.

[160] N.C. Paver, P. Day, S.B. Furber, J.D. Garside, and J.V. Woods. Register locking in an asynchronous microprocessor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 351–355. IEEE Computer Society Press, October 1992.

[161] A. Peeters. Single-rail handshake circuits. Technical report, Eindhoven University of Technology, June 1996. Ph.D. Thesis.

[162] M.A. Pena and J. Cortadella. Combining process algebras and petri nets for the specification and synthesis of asynchronous circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 222–232. IEEE Computer Society Press, November 1996.

[163] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[164] L. Plana and S.M. Nowick. Concurrency-oriented optimization for low-power asynchronous systems. In *IEEE International Symposium on Low-Power Electronics and Design*, pages 151–156. IEEE Computer Society Press, August 1996.

[165] R. Puri and J. Gu. Area efficient synthesis of asynchronous interface circuits. In *Proceedings of the IEEE International Conference on Computer Design*, pages 212–216. IEEE Computer Society Press, October 1994.

[166] R. Puri and J. Gu. A modular partitioning approach for asynchronous circuit synthesis. In *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pages 63–69. ACM, June 1994.

[167] M. Rem, J.L.A. van de Snepscheut, and J.T. Udding. Trace theory and the definition of hierarchical components. In Randal Bryant, editor, *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 225–239. CSP, Inc., 1983.

[168] R. D. Rettberg, W. R. Crowther, P. P. Carvey, and R. S. Tomlinson. The Monarch Parallel Processor Hardware Design. *Computer*, 23(4):18–30, April 1990.

[169] C.A. Rey and J. Vaucher. Self-synchronized asynchronous sequential machines. *IEEE Transactions on Computers*, C-23(12):1306–1311, December 1974.

[170] W. Richardson and E. Brunvand. Fred: An Architecture for a Self-Timed Decoupled Computer. In *Advanced Research in Asynchronous Circuits and Systems*, pages 60–68. IEEE Computer Society Press, 1996.

[171] W. F. Richardson. *Architectural Considerations for a Self-Timed Decoupled Processor*. PhD thesis, University of Utah, March 1996.

[172] P.T. Roeine. A system for asynchronous high-speed chip to chip communication. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 2–10. IEEE Computer Society Press, November 1996.

[173] T. Rokicki. Representing and modeling digital circuits. Technical report, Stanford University, December 1993. Ph.D. Thesis.

[174] T. Rokicki and C. Myers. Automatic Verification of Timed Circuits. In *International Conference on Computer-Aided Verification*, pages 468–480. Springer-Verlag, 1994.

[175] M. Roncken. Partial scan test for asynchronous circuits illustrated on a DCC error corrector. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 247–256. IEEE Computer Society Press, November 1994.

[176] M. Roncken, E. Aarts, and W. Verhaegh. Optimal scan for pipelined testing: an asynchronous foundation. In *Proceedings of the IEEE International Test Conference*. IEEE Computer Society Press, October 1996.

[177] M. Roncken and E. Bruls. Test quality of asynchronous circuits: a defect-oriented evaluation. In *Proceedings of the IEEE International Test Conference*. IEEE Computer Society Press, October 1996.

[178] Marly Roncken and Ronald Saeijs. Linear Test Times for Delay-Insensitive Circuits: a Compilation Strategy. In S. Furber and M. Edwards, editors, *Proceedings of the IFIP WG 10.5 Working Conference on Asynchronous Design Methodologies, Manchester*, pages 13–27. Elsevier Science Publishers B.V., 1993.

[179] L.Y. Rosenblum and A.V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets, Torino, Italy*, pages 199–207. IEEE Computer Society Press, July 1985.

[180] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued optimization for PLA optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, September 1987.

[181] V. Schoeber and T. Kiel. An asynchronous scan path concept for micropipelines using the bundled data convention. In *Proceedings of the IEEE International Test Conference*. IEEE Computer Society Press, October 1996.

[182] C.-J. Seger. A bounded delay race model. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 130–133. IEEE Computer Society Press, November 1989.

[183] Charles L. Seitz. "The Cosmic Cube". *Communications of the ACM*, 28(1):22–33, January 1984.

[184] Charles L. Seitz. The Mosaic C: An Experimental Fine Grain Multicomputer. In Alain Bensoussan and Jean Pierre Verjus, editors, *Future Tendencies in Computer Science*. Springer Verlag Lecture Notes in Computer Science #653, 1992.

[185] C.L. Seitz. Asynchronous machines exhibiting concurrency. In *Conference Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, 1970.

[186] C.L. Seitz. *Graph representations for logical machines*. PhD thesis, MIT, Jan 1971.

[187] A. Semenov and A. Yakovlev. Verification of asynchronous circuits using time petri net unfolding. In *33rd ACM/IEEE Design Automation Conference*, June 1996.

[188] E.M. Sentovich. SIS: a system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, UC Berkeley, May 1992. Dept. of EECS.

[189] Robert Shapiro and Hartmann Genrich. A Design of a Cascadable Nacking Arbiter. MetaSoftware, Inc. Cambridge, MA 02140, 1993 February.

[190] Robert Shapiro and Hartmann Genrich. Formal Verification of an Arbiter Cascade. MetaSoftware, Inc. Cambridge, MA 02140, 1992 January.

[191] P. Siegel, G. De Micheli, and D. Dill. Technology mapping for generalized fundamental-mode asynchronous designs. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 61–67. ACM, June 1993.

[192] M. Singh and S.M. Nowick. Synthesis-for-testability of asynchronous sequential machines. In *Proceedings of the IEEE International Test Conference*. IEEE Computer Society Press, October 1996.

[193] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, 1992.

[194] J. Sparso and J. Staunstrup. Design and performance analysis of delay insensitive multi-ring structures. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 349–358. IEEE Computer Society Press, January 1993.

[195] R.F. Sproull, I.E. Sutherland, and C.E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, 1994.

[196] staff. New Products. *IEEE Computer*, 25(1), 1992.

[197] K. Stevens. Practical Verification and Synthesis of Low Latency Asynchronous Systems. PhD Thesis, Computer Science Department, University of Calgary, 1994.

[198] K. Stevens, J. Aldwinckle, G. Birtwistle, and Y. Liu. Designing parallel specifications in CCS. In *Proceedings of Canadian Conference on Electrical and Computer Engineering*, Vancouver, 1993.

[199] K.S. Stevens, S.V. Robison, and A.L. Davis. The post office - communication support for distributed ensemble architectures. In *Sixth International Conference on Distributed Computing Systems*, 1986.

[200] I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[201] M.A. Tapia. Synthesis of asynchronous sequential systems using boolean calculus. In *14th Asilomar Conference on Circuits, Systems and Computers*, pages 205–209, November 1980.

[202] M. Theobald, S.M. Nowick, and T. Wu. Espresso-HF: a heuristic hazard-free minimizer for two-level logic. In *33rd ACM/IEEE Design Automation Conference*, pages 71–76, June 1996.

[203] J.A. Tierno and A.J. Martin. Low-energy asynchronous memory design. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 176–185. IEEE Computer Society Press, November 1994.

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

[205] J.T. Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1(4):197–204, 1986.

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

[207] S.H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions on Computers*, C-20(12):1437–1444, December 1971.

[208] S.H. Unger. Self-synchronizing circuits and nonfundamental mode operation. *IEEE Transactions on Computers (Correspondence)*, C-26(3):278–281, March 1977.

[209] S.H. Unger. A building block approach to unclocked systems. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 339–348. IEEE Computer Society Press, January 1993.

[210] C.H. van Berkel and R.W.J.J. Saeijs. Compilation of communicating processes into delay-insensitive circuits. In *Proceedings of the IEEE International Conference on Computer Design*, pages 157–162. IEEE Computer Society Press, 1988.

[211] K. van Berkel. *Handshake Circuits. An asynchronous architecture for VLSI programming.* International Series on Parallel Computation 5. Cambridge University Press, 1993.

[212] K. van Berkel and A. Bink. Single-track handshake signaling with application to micropipelines. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 122–133. IEEE Computer Society Press, November 1996.

[213] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. Asynchronous Circuits for Low Power: a DCC Error Corrector. *IEEE Design & Test*, 11(2):22–32, June 1994.

[214] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, 1992.

[215] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 184–187. IEEE Computer Society Press, November 1990.

[216] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A generalized state assignment theory for transformations on signal transition graphs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 112–117. IEEE Computer Society, November 1992.

[217] V.I. Varshavsky, M.A. Kishinevsky, V.B. Marakhovsky, V.A. Peschansky, L.Y. Rosenblum, A.R. Taubin, and B.S. Tzirlin. *Self-timed Control of Concurrent Processes*. Kluwer Academic Publishers, 1990. Russian edition: 1986.

[218] Tom Verhoeff. Delay-insensitive codes – an overview. *Distributed Computing*, 3(1):1–8, 1988.

[219] E. Verlind, G. de Jong, and B. Lin. Efficient partial enumeration for timing analysis of asynchronous systems. In *33rd ACM/IEEE Design Automation Conference*, June 1996.

[220] T.E. Williams. Self-timed rings and their application to division. Technical Report CSL-TR-91-482, Computer Systems Laboratory, Stanford University, 1991. Ph.D. Thesis.

[221] T.E. Williams and M.A. Horowitz. A zero-overhead self-timed 54b 160ns CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.

[222] A. Yakovlev, A. Petrov, and L. Lavagno. A Low Latency Asynchronous Arbitration Circuit. *IEEE Trans. on VLSI Systems*, 2(3):372–377, September 1994.

[223] A.V. Yakovlev. On limitations and extensions of STG model for designing asynchronous control circuits. In *Proceedings of the IEEE International Conference on Computer Design*, pages 396–400. IEEE Computer Society Press, October 1992.

[224] O. Yenersoy. Synthesis of asynchronous machines using mixed-operation mode. *IEEE Transactions on Computers*, C-28(4):325–329, April 1979.

[225] C. Ykman-Couvreur, B. Lin, and H. De Man. ASSASSIN: a synthesis system for asynchronous control circuits. Technical report, IMEC Laboratory, September 1994. User and tutorial manual.

[226] T. Yoneda and T. Yoshikawa. Using partial orders for trace theoretic verification of asynchronous circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 152–163. IEEE Computer Society Press, November 1996.

[227] M.L. Yu and P.A. Subrahmanyam. A path-oriented approach for reducing hazards in asynchronous designs. In *Proceedings of the 29th IEEE/ACM Design Automation Conference*, pages 239–244. IEEE Computer Society Press, June 1992.

[228] K.Y. Yun, P.A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 17–28. IEEE Computer Society Press, November 1996.

[229] K.Y. Yun and D.L. Dill. Automatic synthesis of 3D asynchronous finite-state machines. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society Press, November 1992.

[230] K.Y. Yun and D.L. Dill. Unifying synchronous/asynchronous state machine synthesis. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 255–260. IEEE Computer Society Press, November 1993.

[231] K.Y. Yun and D.L. Dill. A high-performance asynchronous SCSI controller. In *Proceedings of the IEEE International Conference on Computer Design*, pages 44–49. IEEE Computer Society Press, October 1995.

[232] K.Y. Yun, D.L. Dill, and S.M. Nowick. Practical generalizations of asynchronous state machines. In *The 1993 European Conference on Design Automation*, pages 525–530. IEEE Computer Society Press, February 1993.

[233] K.Y. Yun, B. Lin, D.L. Dill, and S. Devadas. Performance-driven synthesis of asynchronous controllers. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, November 1994.