Asynchronous Circuits

Maitham Shams
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ont, CANADA

Jo C. Ebergen
Sun Microsystems Laboratories
Mountain View, CA, USA

Mohamed I. Elmasry
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ont, CANADA

Digital VLSI circuits are usually classified into synchronous and asynchronous circuits. Synchronous circuits are generally controlled by global synchronization signals provided by a clock. Asynchronous circuits, on the other hand, do not use such global synchronization signals. Between these extremes there are various hybrids. Digital circuits in today's commercial products are almost exclusively synchronous. Despite this big difference in popularity, there are a number of reasons why asynchronous circuits are of interest.

In this article, we present a brief overview of asynchronous circuits. First we address some of the motivations for designing asynchronous circuits. Then, we discuss different classes of asynchronous circuits and briefly explain some asynchronous design methodologies. Finally, we present a typical asynchronous design in detail.

1 Motivations for Asynchronous Circuits

Throughout the years researchers have had various reasons for studying and building asynchronous circuits. Some of the often mentioned advantages of asynchronous circuits are speed, low energy dissipation, modular design, immunity to metastable behavior, freedom from clock skew, and low generation of and low susceptibility to electromagnetic interference. We elaborate here on some of these potentials and indicate when they have been demonstrated through comparative case studies.

1.1 Speed

Speed has always been a motivation for designing asynchronous circuits. The main reasoning behind this advantage is that synchronous circuits exhibit worst-case behavior, whereas asynchronous circuits exhibit average-case behavior. The speed of a synchronous circuit is governed by its clock frequency. The clock period should be large enough to accommodate the worst-case propagation delay in the critical path of the circuit, the maximum clock skew, and a safety factor due to fluctuations in the chip fabrication process, operating temperature, and supply voltage. Thus, synchronous circuits exhibit worst-case performance. This worst-case behavior is dictated by the global clock and, in spite of the fact that the worst-case propagation in many circuits, particularly arithmetic units, is improbable and may be much longer than the average-case propagation.

Many asynchronous circuits are controlled by local communications and are based on the principle of initiating a computation, waiting for its completion, and then initiating the next one. When a computation is completed early, the next computation can start early. For this reason, the speed of asynchronous circuits equipped with completion-detection mechanisms depend on the computation time of the data being processed, not the worst-case timing. Accordingly, such asynchronous circuits exhibit average-case performance. An example of an asynchronous circuit where the average-case potential is nicely exploited is reported in [1], an asynchronous divider that is twice as fast as its synchronous counterpart. Nevertheless, to date, there are few concrete examples demonstrating that the average-case performance of asynchronous circuits is higher than that of synchronous circuits performing similar functions. The reason is that the average-case performance advantage is often counterbalanced by the overhead in control circuitry and completion-detection mechanisms.

Besides demonstrating the average-case potential, there are case studies in which the speed of an asynchronous design is compared to the speed of a corresponding synchronous version. Molnar et al. report a case study of an asynchronous FIFO that is every bit as fast as any synchronous FIFO using the same data latches [2]. Furthermore, the asynchronous FIFO has the additional benefits that it operates under local control and is easily expandable. At the end of this article we give an example of a FIFO with a slightly different control circuit.

1.2 Immunity to Metastable Behavior

Any circuit with a number of stable states also has metastable states. When such a circuit gets into a metastable state, it can remain there for an indefinite period of time before

resolving into a stable state [3, 4]. Metastable behavior occurs, for example, in circuit primitives that realize mutual exclusion between processes, called *arbiters*, and components that synchronize independent signals of a system, called *synchronizers*. Although the probability that metastable behavior lasts longer than period t decreases exponentially with t, it is possible that metastable behavior in a synchronous circuit lasts longer than one clock period. Consequently, when metastable behavior occurs in a synchronous circuit, erroneous data may be sampled at the the computation time of the clock pulses. An asynchronous circuit deals gracefully with metastable behavior by simply delaying the computation until the metastable behavior has disappeared and the element has resolved into a stable state.

1.3 Modularity

Modularity in design is an advantage exploited by many asynchronous design styles. The basic idea is that an asynchronous system is composed of functional modules communicating along well-defined interfaces. Composing asynchronous systems is simply a matter of connecting the proper modules with matching interfacial specifications. The interfacial specifications describe only the sequences of events that can take place and do not specify any restrictions on the timing of these events. This characteristic reduces the design time and complexity of an asynchronous circuit, because the designer does not have to worry about the delays incurred in individual modules or the delays inserted by connection wires. Designers of synchronous circuits, on the other hand, often pay considerable attention to satisfying the detailed interfacial timing specifications.

Besides ease of composability, modular design also has the potential for better technology migration, ease of incremental improvement, and reuse of modules [5]. Here the idea is that an asynchronous system adapts itself more easily to the advances in technology. The obsolete parts of an asynchronous system can be replaced with new parts to improve system performance. Synchronous systems cannot take advantage of new parts as easily, because they must be operated with the old clock frequency or other modules must be redesigned to operate at the new clock frequency.

One of the earliest projects that exploited modularity in designing asynchronous circuits is the Macromodules project [6]. Another nice example where modular design has been demonstrated is the TANGRAM compiler developed at Philips Research Laboratories [7].

1.4 Low Power

Due to rapid growth in the use of portable equipment and the trend in high-performance processors towards unmanageable power dissipation, energy efficiency has become crucial in VLSI design. Asynchronous circuits are attractive for energy-efficient designs, mainly because of the elimination of the clock. In systems with a global clock, all of the latches and registers operate and consume dynamic energy during each clock pulse, in spite of the fact that many of those latches and registers might not have new data to store. There is no such waste of energy in asynchronous circuits, because computations are initiated only when they need to be done.

Two notable examples that demonstrated the potential of asynchronous circuits when in energy-efficient design are the work done at Philips Research Laboratories and at Manchester University. The Philips group designed a fully asynchronous digital compact-cassette (DCC) error detector which consumed 80% less energy than a similar synchronous version [8]. The AMULET group at Manchester University successfully implemented an asynchronous version of the ARM microprocessor, one of the most energy-efficient synchronous microprocessors. The asynchronous version achieved a power dissipation comparable to the fourth generation of ARM, around 150 mW [9], in a similar technology.

Recently, power management techniques are being used in synchronous systems to turn the clock on and off conditionally. However, these techniques are only worthwhile implementing at the level of functional units or higher. Besides, the components that monitor the environment for switching the clock continue dissipating energy.

It is also worth mentioning that unlike synchronous circuits, most asynchronous circuits do not waste energy on *hazards*, which are spurious changes in a signal. Asynchronous circuits are essentially designed to be hazard-free. Hazards can be responsible for up to 40% of energy loss in synchronous circuits [10].

1.5 Freedom from Clock Skew

Because asynchronous circuits generally do not have clocks, they do not have many of the problems associated with clocks. One such problem is *clock skew*, the technical term for the maximum difference in clock arrival time at different parts of a circuit. In synchronous circuits, it is crucial that all modules operating with a common clock receive this signal simultaneously, that is, within a tolerable period of time. Minimizing clock skew is a difficult problem for large circuits. Various techniques have been proposed to control clock skew, but

generally they are expensive in terms of silicon area and energy dissipation. For instance, the clock distribution network of the DEC Alpha, a 200 MHz microprocessor at a 3.3 V supply, occupies 10% of the chip area and has a 40% share in the total chip power consumption [11]. Although asynchronous circuits do not have the clock skew problem, they have their own set of problems in minimizing the overhead needed for synchronization among the parts.

2 Models and Methodologies

There are many models and methodologies for analyzing and designing asynchronous circuits. Asynchronous circuits can be categorized by the following criteria: signaling protocol and data encoding, underlying delay model, mode of operation, and formalism for specifying and designing circuits. This section presents an informal explanation of these criteria.

2.1 Signaling Protocols and Data Encodings

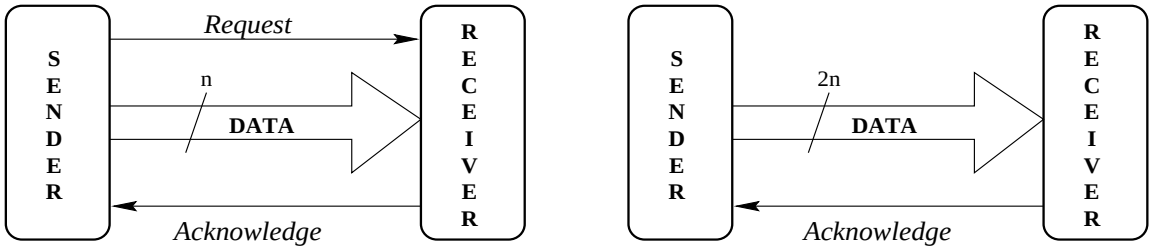
Modules in an asynchronous circuit communicate data with some signaling protocol consisting of request and acknowledgment signals. There are two common signaling protocols for communicating data between a sender and a receiver: the four-phase and the two-phase protocol. In addition to the signaling protocol, there are different ways to encode data. The most common encodings are single-rail and dual-rail encoding. We explain the two signaling protocols first and then discuss the data encodings.

If the sender and receiver communicate through a *two-phase signaling* protocol, then each communication cycle has two distinct phases. The first phase consists of a request initiated by the sender. The second phase consists of an acknowledgment by the receiver. The request and acknowledgment signals are often implemented by voltage transitions on separate wires. No distinction is made between the directions of voltage transitions. Both rising and falling transitions denote a signaling event.

The *four-phase signaling protocol* consists of four phases: a request followed by an acknowledgment, followed by a second request, and finally a second acknowledgment. If the request and acknowledgment are implemented by voltage transitions, then at the end of every four phases, the signaling wires return to the same voltage levels as at the start of the four phases. Because the initial voltage is usually zero, this type of signaling is also called *return-to-zero signaling*. Other names for two-phase and four-phase signaling are *two-cycle* and *four-cycle signaling*, respectively, or *transition* and *level signaling*, respectively.

Both signaling protocols can be used with single and dual-rail data encodings. In *single-rail data encoding* each bit is encoded with one wire, whereas in *dual-rail encoding*, each bit is encoded with two wires.

In single-rail encoding, the value of the bit is represented by the voltage on the data wire. When communicating n data bits with a single-rail encoding, during periods where the data wires are guaranteed to remain stable, we say that the data are *valid*. During periods where the data wires are possibly changing, we say the data are *invalid*. A two-phase or four-phase signaling protocol is used to tell the receiver when data are valid or invalid . The sender informs the receiver about the validity of the data through the request signal, and the receiver, in turn, informs the sender of the receipt of the data through the acknowledgment signal. Therefore, to communicate n bits of data, a total number of $(n+2)$ wires are necessary between the sender and the receiver. The connection pattern for single-rail encoding and two or four-phase signaling is depicted in Figure 1(a).



(a) Bundled Data Convention (b) Dual-Rail Data Encoding

Figure 1: Two different data communication schemes

Figure 2(a) shows the sequence of events in a two-phase signaling protocol. The events include the times when the data become valid and invalid. The transparent bars indicate the period when data are valid, during the other periods, data are invalid. Notice that a request signal occurs only after data become valid. This is an important timing restriction associated with these communication protocols, namely, the request signal that indicates that data are valid should always arrive at the receiver *after* all data wires have attained their proper value. The restriction is referred to as the *bundling constraint*. For this reason the communication protocol is often called the *bundled data protocol*. Figure 2(b) shows a sequence of events in a four-phase protocol and single-rail data encoding. Other sequences are also applicable for the four-phase protocol.

The dual-rail encoding scheme uses two wires for every data bit. There are several dual-rail encoding schemes. All combine the data encoding and signaling protocol. There is no explicit request signal, and the dual-rail encoding schemes all require $(2n + 1)$ wires as

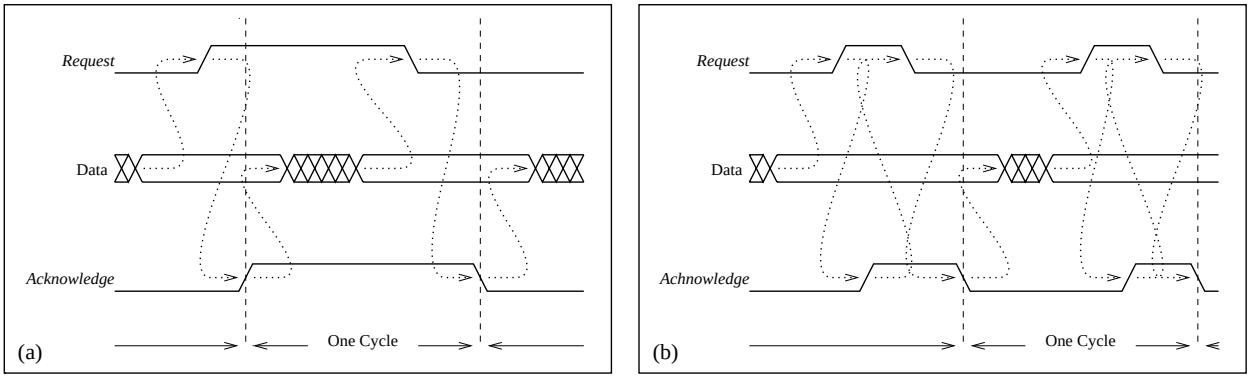


Figure 2: Data transfer in two-phase signaling (a), and four-phase signaling (b)

illustrated in Figure 1(b). In the case of four-phase signaling, there are several encodings that can be used to transmit a data bit. The most common encoding has the following meaning for the four states in which each pair of wires can be in: 00 = reset, 10 = valid 0, 01 = valid 1, and 11 is an unused state. Every pair of wires has to go through the reset state before becoming valid again. In the first phase of the four-phase signaling protocol, every pair of wires leaves the reset state for a valid 0 or 1 state. The receiver detects the arrival of a new set of valid data when all pairs of wires have left the reset state. This detection replaces an explicit request signal. The second phase consists of an acknowledgment to inform the sender that data has been consumed. The third phase consists of the reset of all pairs of wires to the reset state, and the fourth phase is the reset of the acknowledgment.

In a two-phase signaling protocol, a different dual-rail encoding is used. An example of an encoding is as follows. Each pair of wires has one wire associated with a 0 and one wire associated with a 1. A transition on the wire associated with 0 represents the communication of a 0, whereas a transition on the other wire represents a communication of a 1. Thus, a transition on one wire of each pair signals the arrival of a new bit value. A transition on both wires is not allowed. In the first phase of the two-phase signaling protocol, every pair of wires communicates a 0 or a 1. The second phase is an acknowledgment sent by the receiver.

Of all data encodings and signaling protocols, the most popular are the single-rail encoding and four-phase signaling protocol. The main advantages of these protocols are the small number of connection wires and the simplicity of the encoding, which allows using conventional techniques for implementing data operations. The disadvantages of these protocols are the bundling constraints that must be satisfied and the extra energy and time wasted in the additional two phases compared with two-phase signaling. Dual-rail data encodings have been used to communicate data in asynchronous circuits free of any timing constraints. Dual-rail encodings, however, are expensive in practice, because of the many interconnec-

tion wires, the extra circuitry to detect completion of a transfer, and the difficulty in data processing.

2.2 Delay Models

An important characteristic distinguishing different asynchronous circuit styles is the delay model on which they are based. For each circuit primitive, gate or wire, a *delay model* stipulates the sort of delay it imposes and the range of the delays. Delay models are needed to analyze all possible behavior of a circuit for various correctness conditions, like the absence of hazards.

A circuit is composed of gates and interconnection wires, all of which impose delays on the signals propagating through them. The delay models are categorized into two classes: pure delay models and inertial delay models. In a *pure delay* model, the delay associated with a circuit component produces only a time shift in the voltage transitions. In reality, a circuit component may shift the signals and also filter out pulses of small width. A delay model which captures this fact is called an *inertial delay* model. Both classes of delay models can have several ranges for the delay shifts. We distinguish the *zero-delay*, *fixed-delay*, *bounded-delay*, and *unbounded-delay* models. In the zero-delay model, the values of the delays are zero. In the fixed-delay model, the values of the delays are constant, whereas in the bounded-delay model the values of the delays vary within a bounded range. The unbounded-delay model does not impose any restriction on the value of the delays except that they cannot be infinite. Sometimes two different delay models are assumed for the wires and the gates in an asynchronous circuit. For example, the operation of a class of asynchronous circuits is based on the zero-delay model for wires and the unbounded-delay model for gates. Formal definitions of the various delay models are given in [12].

A concept closely related to the delay model of a circuit is its *mode of operation*. The mode of operation characterizes the interaction between a circuit and its environment. Classical asynchronous circuits operate in the *fundamental mode* [13, 14], which assumes that the environment changes only one input signal and waits until the circuit reaches a stable state. Then the environment is allowed to apply the next change to one of the input signals. Many modern asynchronous circuits operate in the *input-output* mode. In contrast to the fundamental mode, the input-output mode allows for input changes immediately after receiving an appropriate response to a previous input change, even if the entire circuit has not yet stabilized. The fundamental mode was introduced in the sixties to simplify the analysis and design of gate circuits with Boolean algebra. The input-output mode evolved in the eighties

from event-based formalisms to describe modular design methods that abstracted from the internal operation of a circuit.

2.3 Formalisms

Just as in any other design discipline, designers of asynchronous circuits use various formalisms to master the complexities in the design and analysis of their artifacts. The formalisms used in asynchronous circuit design can be categorized into two classes: formalisms based on Boolean algebra and formalisms based on sequences of events. Most design methodologies in asynchronous circuits use some mixture of both formalisms.

The design of many asynchronous circuits is based on Boolean algebra or its derivative switching theory. Such circuits often use the fundamental mode of operation, the bounded-delay model, and have, as primitive elements, gates that correspond to the basic logic functions, like AND, OR, and inversion. These formalisms are convenient for implementing logic functions, analyzing circuits for the presence of hazards, and synthesizing fundamental-mode circuits [12, 14].

Event-based formalisms deal with sequences of events rather than binary logic variables. Circuits designed with an event-based formalism operate in the input-output mode, under an unbounded-delay model, and have, as primitive elements, the JOIN, the TOGGLE, and the MERGE, for example. Event-based formalisms are particularly convenient for designing asynchronous circuits when a high degree of concurrency is involved. Several tools have been generated for the automatic verification of asynchronous circuits with event-based formalisms [15, 16]. Examples of event-based formalisms are Trace Theory [17–19], DI Algebra [20], Petri nets, and Signal Transition Graphs [21, 22].

3 Design Techniques

This section introduces the most popular types of asynchronous circuits and briefly describes some of their design techniques.

3.1 Types of Asynchronous Circuits

There are special types of asynchronous circuits for which formal and informal specifications have been given. Here are brief informal descriptions of some of them in a historical context.

There are two types of logic circuits: *combinational* and *sequential*. The output of a combinational circuit depends only on the current inputs, whereas the output of a sequential circuit depends also on the previous sequences of the inputs. With this definition of a sequential circuit, almost all asynchronous circuit styles fall into this category. However, the term *asynchronous sequential* circuits or machines generally refers to those asynchronous circuits based on *finite state machines* similar to those in synchronous sequential circuits [14, 23].

Muller was the first to give a rigorous formalization of a special type of circuits for which he coined the name *speed-independent* circuits. An account of this formalization is given in [24, 25]. Informally, a speed-independent circuit is a network of gates that satisfies its specification irrespective of any gate delays.

From a design discipline that was developed as part of the Macromodules project [6] at Washington University in St. Louis, the concept of another type of asynchronous circuits evolved, which was given the name *delay-insensitive* circuit, that is, a network of modules that satisfies its specification irrespective of any element *and* wire delays. It was realized that proper formalization of this concept was needed to specify and design such circuits in a well-defined manner. Such a formalization was given by Udding [26].

Another name frequently used in designing asynchronous circuits is *self-timed systems*. This name was introduced by Seitz [27]. A self-timed system is described recursively as either a self-timed element or a legal connection of self-timed systems. The idea is that self-timed elements can be implemented with their own timing discipline, and some may even have synchronous implementations. In composing self-timed systems from self-timed elements, however, no reference to the timing of events is made; only the sequence of events is relevant. In other words, the elements "keep time to themselves."

Some have found the unbounded gate-and-wire delay assumption, on which the concept of a delay-insensitive circuit is based, to be too restrictive in practice. For example, the unbounded gate-and-wire delay assumption implies that a signal sent to multiple recipients by a fork can incur a different unbounded delay for each of the recipients. They proposed to relax this delay assumption slightly by using *isochronic forks* [28]. An isochronic fork is a fork whose difference in the delays of its branches is negligible compared with the delays in the element to which it is connected. A delay-insensitive circuit that uses isochronic forks is called a *quasi-delay-insensitive* circuit [17, 28]. Although the use of isochronic forks gives more design freedom in exchange for less delay insensitivity, care has to be taken with its implementation [29].

3.2 Asynchronous Sequential Machines

The design of asynchronous sequential finite state machines was initiated with the pioneering work of Huffman [23]. He proposed a structure similar to that of synchronous sequential circuits consisting of a combinational logic circuit, inputs, outputs, and state variables [14]. *Huffman circuits*, however, store the state variables in feedback loops containing *delay elements*, instead of in latches or flip-flops, as synchronous sequential circuits do. The design procedure begins with creating a *flow table* and reducing it through some *state minimization* technique. After a *state assignment*, the procedure obtains the Boolean expressions and implements them in combinational logic with the aid of a *logic minimization* program. To guarantee a hazard-free operation, Huffman circuits adopt the restrictive single-input-change fundamental mode, that is, the environment changes only one input and waits until the circuit becomes stable before changing another input. This requirement can substantially degrade the circuit performance. Hollaar realized this fact and introduced a new structure in which the fundamental mode assumption is relaxed [30]. In his implementation, the state variables are stored in NAND latches, so that inputs are allowed to change earlier than the fundamental mode would allow. Although Hollaar's method improves the performance, it suffers from the danger of producing hazards. Besides, neither technique seem to be adequate for designing concurrent systems. Models and algorithms for the analysis of asynchronous sequential circuits have been developed by Brzozowski and Seger [12].

The quest for more concurrency, higher performance, and hazard-free operation, resulted in the formulation of a new generation of asynchronous sequential circuits known as *burst-mode* machines [31,32]. A burst-mode circuit does not react until the environment performs a number of input changes called an *input burst*. The environment, in turn, is not allowed to introduce the next input burst until the circuit produces a number of outputs called an *output burst*. A state graph is used to specify the transitions caused by the input and output bursts. Two synthesis methods have been proposed and automated for implementing burst-mode circuits. The first method employs a locally generated clock to avoid some hazards [33]. The second method uses three-dimensional flow tables and is based on Huffman circuits [34]. One limitation of burst mode circuits is that they restrict concurrency within a burst.

3.3 Speed-Independent Circuits and STG synthesis

Speed-independent circuits are usually designed by a form of Petri nets [35]. A popular version of Petri nets, *signal transition graphs* (STG), was introduced by Chu. He also developed a synthesis technique for transforming STGs into speed-independent circuits [21]. Chu's work

was extended by Meng, who produced an STG-based tool for synthesizing speed-independent circuits from high-level specifications [36]. In this technique, a circuit is composed of computational blocks and interconnection blocks. Computational blocks range from a simple shifter module to more complicated ones, such as ALUs, RAMs, and ROMs. Interconnection blocks synchronize the operation of computational blocks by producing appropriate control signals. Computational blocks generate *completion signals* after their output data become valid. The interconnection blocks use the completion signals to generate four-phase handshake protocols.

3.4 Delay-Insensitive Circuits and Compilation

Several researchers have proposed techniques for designing delay-insensitive circuits. Ebergen [37] has developed a synthesis method based on the formalism of *Trace Theory*. The method consists of specifying a component by a program and then transforming this program into a delay-insensitive network of basic elements. The program notation allows specifying parallel behavior. Ebergen's method has been applied to the design of small components like stacks, various counters, and arbiters [18].

Martin proposes a method [28] that starts with a specification of an asynchronous circuit in a high-level programming language similar to Hoare's *Communicating Sequential Processes* (CSP) [38]. An asynchronous circuit is specified as a group of processes communicating over channels. After various transformations, the program is mapped into a network of gates. This method led to the design of an asynchronous microprocessor [39] in 1989. Martin's method yields quasi-delay-insensitive circuits.

Van Berkel [17] has designed a compiler based on a high-level language called *Tangram*. A Tangram program also specifies a set of processes communicating over channels. A Tangram program is first translated into a *handshake circuit*. Then these handshake circuits are mapped into various target architectures, depending on the data-encoding techniques or standard-cell libraries used. The translation is syntax-directed, which means that every operation occurring in a Tangram program corresponds to a primitive in the translated handshake circuit. This property is exploited by various tools that quickly estimate the area, performance, and energy dissipation of the final design by analyzing the Tangram program. Van Berkel's method also yields quasi-delay-insensitive circuits.

Other translation methods from a CSP-like language to a (quasi-) delay-insensitive circuit can be found in [40, 41].

4 A Typical Asynchronous Design

In this section we present a typical asynchronous design, a *micropipeline* [42]. The circuit uses single-rail encoding with the two-phase signaling protocol to communicate data between stages of the pipeline. The control circuit for the pipeline is a delay-insensitive circuit. First we present the primitives for the control circuit, then we present the latches that store the data, and finally we present the complete design.

4.1 The Control Primitives

Figure 3 shows a few simple primitives used in event-based design styles. The schematic symbol for each primitive is depicted opposite its name.

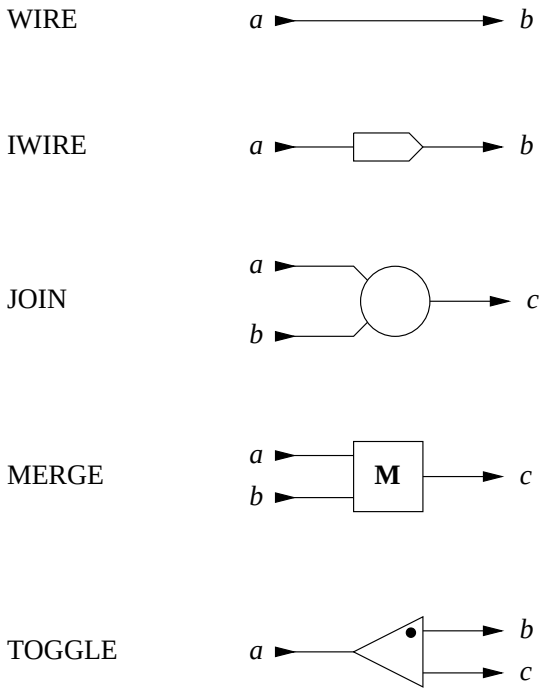


Figure 3: Some delay-insensitive primitives

The simplest primitive is the WIRE, a two-terminal element that produces an output event on its output terminal b after every input event on its input terminal a. Input and output events in a WIRE must alternate. An input event a must be followed by an output event b before another event a occurs. A WIRE is physically realizable with a wire, and events are implemented by voltage transitions. An initialized WIRE, or IWIRE, is very similar to a WIRE, except that it starts by producing an output event b instead of accepting an input event a; after this, its behavior exactly resembles that of a WIRE.

The primitive for synchronization is the JOIN, also called the RENDEZVOUS [6]. A JOIN has two inputs a and b and one output c. The JOIN performs the AND operation of two events a and b. It produces an output event c only after both of its inputs, a *and* b, received an event. The inputs can change again after an output is produced. A JOIN can be implemented by a *Muller C-element*, explained in the next section.

The MERGE component performs the OR operation of two events. If a MERGE component receives an event on either of its inputs, a *or* b, it produces an output event c. After an input event, there must be an output event; successive input events are not allowed. A MERGE can be implemented by a XOR gate.

The TOGGLE has a single input a and two outputs b and c. After an event on input a, an event occurs on output b. The next event on a results in a transition on output c. An input event must be followed by an output event before another input event can occur. Thus, output events alternate or *toggle* after each input event. The dot in the TOGGLE schematic indicates the output which produces the first event.

4.2 The Muller C-Element

The Muller C-element is named for its inventor D. E. Muller [24]. Traditionally, its logical behavior is described as follows. If both inputs are 0 (1), then the output becomes 0 (1); otherwise the output remains the same. For the proper operation of the C-element, it is also assumed that, once both inputs become 0 (1), they will not change again until the output changes. A state diagram is given in Figure 4. The behavior of the output c of the C-element

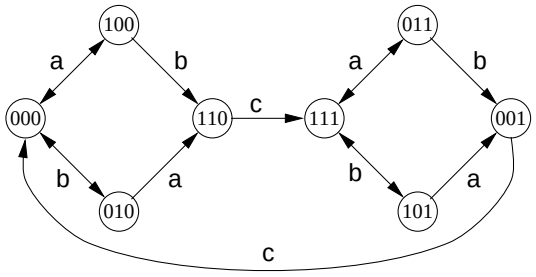


Figure 4: State diagram of the C-element

is expressed in terms of the inputs a and b and the previous state of the output $\hat{c}$ by the following Boolean function

$$c = [\hat{c} \cdot (a + b)] + (a \cdot b) \tag{1}$$

The C-element can be used for implementing the JOIN, which has a slightly more restrictive environment behavior in the sense that an input is not allowed to change twice in succession. A state graph for the JOIN is produced by replacing the bidirectional arcs by unidirectional arcs.

There are many implementations of the C-element. We have given two popular CMOS implementations in Figure 5. Implementation (a) is a conventional pull-up pull-down implementation suggested by Sutherland [42]. Implementation (b) is suggested by Van Berkel [29]. Each implementation has its own characteristics. Implementation (b) is the best choice for speed and energy efficiency [43].

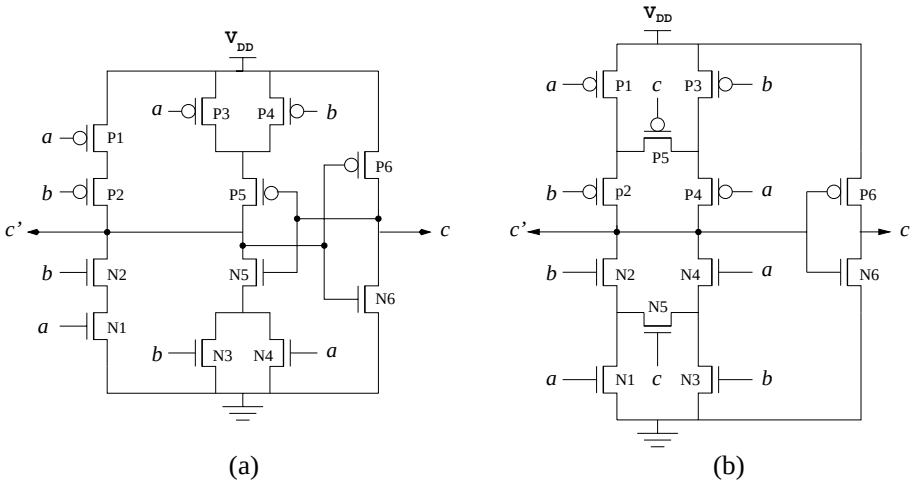


Figure 5: Two CMOS implementations of the C-element: (a) conventional and (b) symmetric

4.3 Storage Primitives

Now we discuss two event-controlled latches due to Sutherland [42], as depicted in Figure 6. Their operation is managed through two input control signals: capture and pass, labeled c and p respectively. They also have two output control signals: capture done, cd, and pass done, pd. The input data is labeled D, and the output data is labeled Q. Implementation (a) is composed of three so-called *double-throw* switches. Implementation (b) includes a MERGE, a TOGGLE, and a level-controlled latch consisting of a double-throw switch and an inverter. A double-throw switch is schematically represented by an inverter and a switching tail. The tail toggles between two positions based on the logic value of a controlling signal. A double-throw switch, in fact, is a two-input multiplexer that produces an inverted version of its selected input. A CMOS implementation of the double-throw switch is shown in Figure 7 [42]. The position of the switch corresponds to the state where c is low.

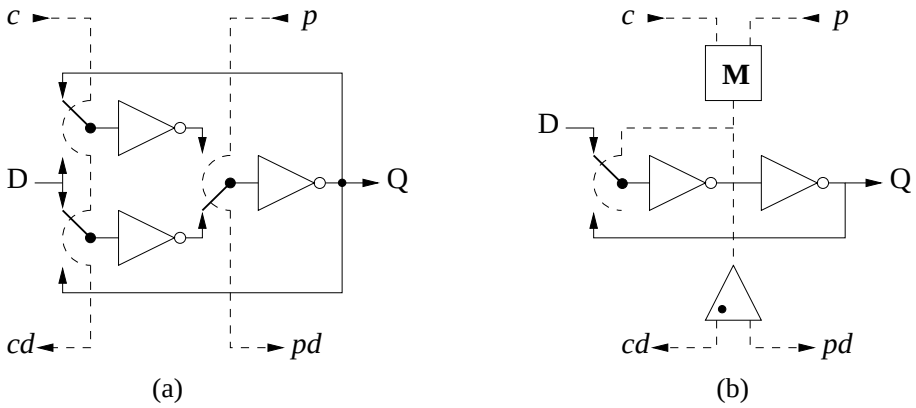


Figure 6: Two event-driven latch implementations

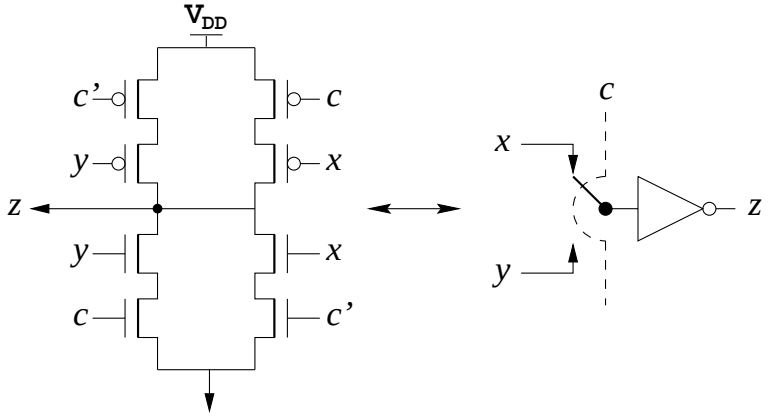


Figure 7: A CMOS implementation of a double-throw switch

An event-controlled latch can assume two states: *transparent* and *opaque*. In the transparent state no data is latched, but the output replicates the input, because a path of two inverting stages exists between the input and the output. In the opaque state, this path is disconnected so that the input data can change without affecting the output; the current data at the output, however, is latched. Implementations in Figures 6(a) and 6(b) are both shown in their initial transparent states. The capture and pass signals in an event-controlled latch always alternate. Upon a transition on c, the latch captures the current input data and becomes opaque. The following transition on cd is an acknowledgment to the data provider that the current data has been captured and that the input data can be changed safely. A subsequent transition on p returns the latch back to its transparent state to pass the next data to its output. The p signal is acknowledged by a transition on pd. Notice that in implementation (a) of Figure 6, signals cd and pd are merely delayed and possibly amplified versions of c and p, respectively.

A group of event-controlled latches, similar to implementation (a) of Figure 6, can be

connected, sharing a capture wire and a pass wire, to form an event-controlled register of arbitrary data width. Implementation (b) of Figure 6 can be generalized similarly into a register by inserting additional level-controlled latches between the MERGE and the TOGGLE. A comparison of different micropipeline latches is reported in [44] and later in [45].

4.4 Pipelining

Pipelining is a powerful technique for constructing high-performance processors. Micropipelines are elegant asynchronous circuits that have gained much attention in the asynchronous community. Many VLSI circuits based on micropipelines have been successfully fabricated. The AMULET microprocessor [9] is one example.

The simplest form of a micropipeline is a FIFO. A four-stage FIFO is shown in Figure 8. It has a control circuit composed solely of interconnected JOINs and a data path of event-controlled registers. The control signals are indicated by dashed lines. The thick arrows show the direction of data flow. Data is implemented with single-rail encoding, and the data path is as wide as the registers can accommodate. Adjacent stages of the FIFO communicate through a two-phase, bundled-data signaling protocol. This means that a request arrive at the next stage only when the data for that stage becomes valid. A bubble at the input of a JOIN is a shorthand for a JOIN with an IWIRE on that input. It implies that, initially, an event has already occurred on the input with the bubble, and the JOIN can produce an output event immediately upon receiving an event on the other input.

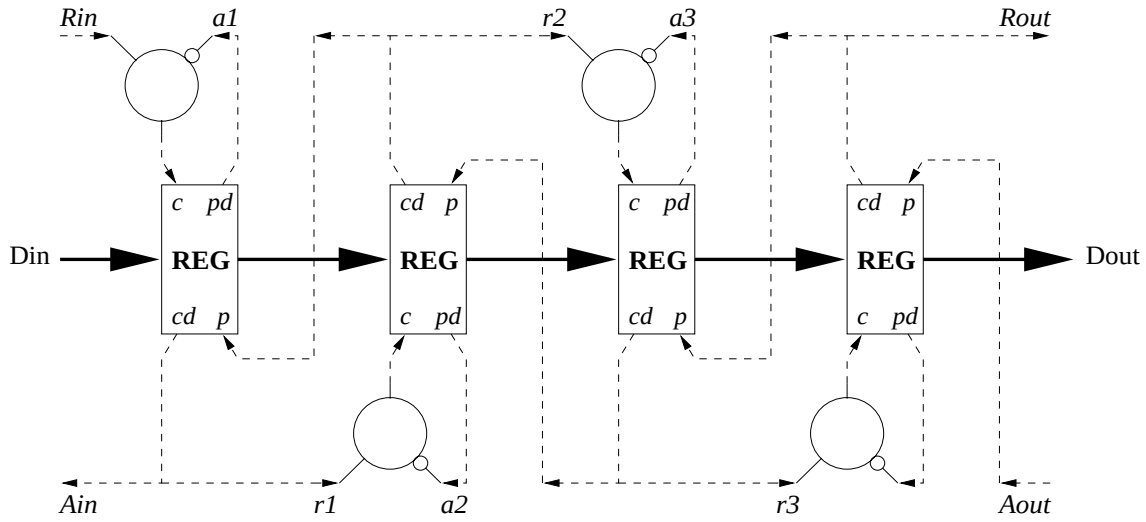


Figure 8: A four-stage micropipeline FIFO structure

Initially, all control wires of the FIFO are at low voltage, and the data in the registers are

not valid. The FIFO is activated by a rising transition on R_{in}, which indicates that input data is valid. Subsequently, the first-stage JOIN produces a rising output transition. This signal is a request to the first-stage register to capture the data and become opaque. After capturing the data, the register produces a rising transition on its *cd* output terminal. This causes a transition on A_{in} and a transition on $r1$, which is a request to the second stage of the FIFO. Meanwhile, the data has proceeded to the second-stage register and has arrived there before the transition on $r1$ occurs. If the environment does not send any new data, the first stage remains idle, and the data and the request signals propagate further to the right. Notice that each time the data is captured by a stage, an acknowledgment is sent back to the previous stage which causes its latch to become transparent again. When the data has propagated to the last register, it is stored and a request signal R_{out} is forwarded to the consumer of the FIFO. At this point, all control signals are at high voltage except for A_{out}. If the data is not removed out of the FIFO, that is, A_{out} remains low, the next data coming from the producer advance only up to the third-stage register, because the fourth-stage JOIN cannot produce an output. Finally, A_{out} also becomess high when the consumer acknowledges receipt of the data. Further data storage and removal follows the same pattern. The operation of each JOIN can be interpreted as follows. If the previous stage has sent a request for data capture *and* the present stage is empty, then send a signal to capture the data in the present stage.

The FIFO can be modified easily to include data processing. A four-stage micropipeline, in its general form, is illustrated in Figure 9. Now the data path consists of alternately positioned event-driven registers and combinational logic circuits. The event-driven registers store the input and output data of the combinational circuits, and the combinational circuits perform the necessary data processing. To satisfy the data bundling constraint, delay elements may occasionally be required to slow down the propagation of the request signals. A delay element must at least match the delay through its corresponding combinational logic circuit, either by some completion detection mechanism or through the insertion of a worst-case delay.

A micropipeline FIFO is flexible in the number of data items it buffers. There is no restriction on the rate at which data enters or exits the micropipeline, except for the delays imposed by the circuit elements. That is why this FIFO and micropipelines generally, are termed *elastic*. In contrast, in an ordinary synchronous pipeline, the rates at which data enter and exit the pipeline are the same, dictated by the external clock signal. A micropipeline is also flexible in the amount of energy it dissipates, which is proportional to the number of data movements. A clocked pipeline, however, continuously dissipates energy as if all

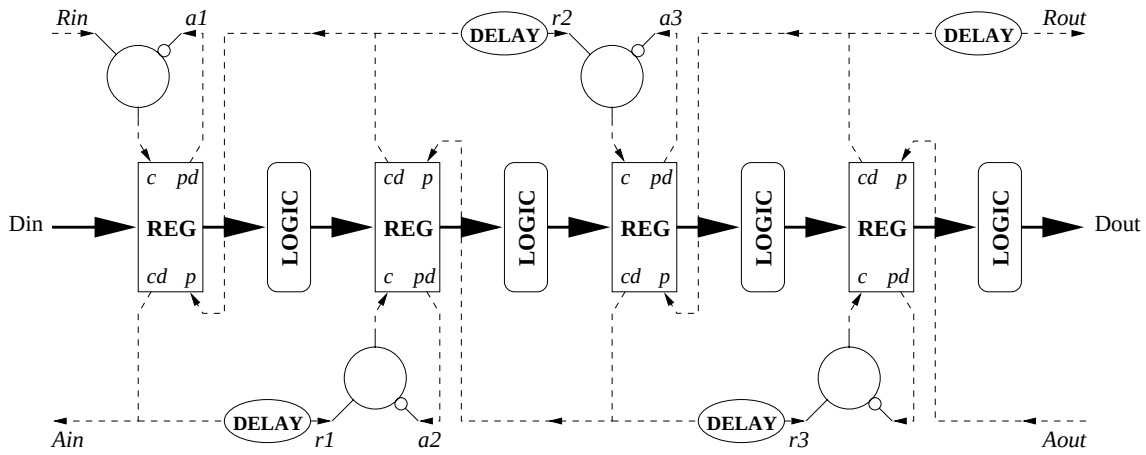


Figure 9: A general four-stage micropipeline structure

stages of the pipeline capture and pass data all the time. Another attractive feature of a micropipeline is that it automatically shuts off when there is no activity. A clocked pipeline, on the other hand, requires a special clock management mechanism to implement this feature. This sensing mechanism, however, constantly consumes energy, because it should never go idle.

5 Concluding Remarks

We have touched only on a few topics relevant to the area of asynchronous circuits and omitted many others. Among the topics omitted are the important areas of verification, testing, and performance analysis of asynchronous circuits. We hope, however, that within the scope of these pages we have provided enough information for further readings. For more information on asynchronous circuits, please see [46], [12], or [47]. A comprehensive bibliography of asynchronous circuits can be found in [48]. Up-to-date information on research in asynchronous circuit design can be found at [49].

The authors wish to thank Bill Coates for his generous criticisms of a previous draft of this article.

References

[1] T. E. Williams and M. A. Horowitz, "A zero-overhead self-timed 160ns 54b CMOS divider," *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 1651–1661, Nov. 1991.

[2] C. E. Molnar, I. W. Jones, B. Coates, and J. Lexau, "A FIFO ring oscillator performance experiment," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press, Apr. 1997.

[3] T. J. Chaney and C. E. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," *IEEE Transactions on Computers*, vol. C-22, pp. 421–422, Apr. 1973.

[4] L. R. Marino, "General theory of metastable operation," *IEEE Transactions on Computers*, vol. C-30, pp. 107–115, Feb. 1981.

[5] R. F. Sproull and I. E. Sutherland, *Asynchronous Systems*. Palo Alto: Sutherland, Sproull and Associates, 1986. Vol. I: Introduction, Vol. II: Logical effort and asynchronous modules, Vol. III: Case studies.

[6] W. A. Clark and C. E. Molnar, "Macromodular computer systems," in *Computers in Biomedical Research* (R. W. Stacy and B. D. Waxman, eds.), vol. IV, ch. 3, pp. 45–85, Academic Press, 1974.

[7] K. v. Berkel and M. Rem, "VLSI programming of asynchronous circuits for low power," in *Asynchronous Digital Circuit Design* (G. Birtwistle and A. Davis, eds.), Workshops in Computing, pp. 152–210, Springer-Verlag, 1995.

[8] K. v. Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij, "A fully-asynchronous low-power error corrector for the DCC player," *IEEE Journal of Solid-State Circuits*, vol. 29, pp. 1429–1439, Dec. 1994.

[9] S. Furber, "Computing without clocks: Micropipelining the ARM processor," in *Asynchronous Digital Circuit Design* (G. Birtwistle and A. Davis, eds.), Workshops in Computing, pp. 211–262, Springer-Verlag, 1995.

[10] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 473–484, Apr. 1992.

[11] D. W. Dobberpuhl and et. al., "A 200-mhz 64-b dual-issue cmos microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 1555–1568, Nov. 1992.

[12] J. A. Brzozowski and C.-J. H. Seger, *Asynchronous Circuits*. Springer-Verlag, 1995.

[13] E. J. McCluskey, "Fundamental mode and pulse mode sequential circuits," in *Proc. of IFIP Congress 62*, pp. 725–730, North-Holland, 1963.

[14] S. H. Unger, *Asynchronous Sequential Switching Circuits.* New York: Wiley-Interscience, John Wiley & Sons, Inc., 1969.

[15] D. L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations, MIT Press, 1989.

[16] J. Ebergen and S. Gingras, "A verifier for network decompositions of command-based specifications," in *Proc. Hawaii International Conf. System Sciences*, vol. I, IEEE Computer Society Press, Jan. 1993.

[17] K. v. Berkel, *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, vol. 5 of *International Series on Parallel Computation.* Cambridge University Press, 1993.

[18] J. C. Ebergen, J. Segers, and I. Benko, "Parallel program and asynchronous circuit design," in *Asynchronous Digital Circuit Design* (G. Birtwistle and A. Davis, eds.), Workshops in Computing, pp. 51–103, Springer-Verlag, 1995.

[19] T. Verhoeff, *A Theory of Delay-Insensitive Systems.* PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1994.

[20] M. B. Josephs and J. T. Udding, "An overview of DI algebra," in *Proc. Hawaii International Conf. System Sciences*, vol. I, IEEE Computer Society Press, Jan. 1993.

[21] T.-A. Chu, *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, MIT Laboratory for Computer Science, June 1987.

[22] T. H.-Y. Meng, *Asynchronous Design for Digital Signal Processing Architectures.* PhD thesis, UC Berkely, 1988.

[23] D. A. Huffman, "The synthesis of sequential switching circuits," *IRE Transactions on Electronic Computers*, vol. 257, no. 3 & 4, 1954.

[24] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," in *Proceedings of an International Symposium on the Theory of Switching*, pp. 204–243, Harvard University Press, Apr. 1959.

[25] R. E. Miller, *Sequential Circuits and Machines*, vol. 2 of *Switching Theory.* John Wiley & Sons, 1965.

[26] J. T. Udding, *Classification and Composition of Delay-Insensitive Circuits.* PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1984.

[27] C. L. Seitz, "System timing," in *Introduction to VLSI Systems* (C. A. Mead and L. A. Conway, eds.), ch. 7, Addison-Wesley, 1980.

[28] A. J. Martin, "Programming in VLSI: From communicating processes to delay-insensitive circuits," in *Developments in Concurrency and Communication* (C. A. R. Hoare, ed.), UT Year of Programming Series, pp. 1–64, Addison-Wesley, 1990.

[29] K. v. Berkel, "Beware the isochronic fork," *Integration, the VLSI journal*, vol. 13, pp. 103–128, June 1992.

[30] L. A. Hollaar, "Direct implementation of asynchronous control units," *IEEE Transactions on Computers*, vol. C-31, pp. 1133–1141, Dec. 1982.

[31] B. Coates, A. Davis, and K. Stevens, "The Post Office experience: Designing a large asynchronous chip," *Integration, the VLSI journal*, vol. 15, pp. 341–366, Oct. 1993.

[32] A. Davis, "Synthesizing asynchronous circuits: Practice and experience," in *Asynchronous Digital Circuit Design* (G. Birtwistle and A. Davis, eds.), Workshops in Computing, pp. 104–150, Springer-Verlag, 1995.

[33] S. M. Nowick and D. L. Dill, "Automatic synthesis of locally-clocked asynchronous state machines," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 318–321, IEEE Computer Society Press, Nov. 1991.

[34] K. Y. Yun and D. L. Dill, "Automatic synthesis of 3D asynchronous state machines," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 576–580, IEEE Computer Society Press, Nov. 1992.

[35] J. L. Peterson, "Petri nets," *Computing Surveys*, vol. 9, pp. 223–252, Sept. 1977.

[36] 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*, vol. 8, pp. 1185–1205, Nov. 1989.

[37] J. C. Ebergen, *Translating Programs into Delay-Insensitive Circuits*, vol. 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.

[38] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[39] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus, "The design of an asynchronous microprocessor," in *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI* (C. L. Seitz, ed.), pp. 351–373, MIT Press, 1989.

[40] E. Brunvand and R. F. Sproull, "Translating concurrent programs into delay-insensitive circuits," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 262–265, IEEE Computer Society Press, Nov. 1989.

[41] S. Weber, B. Bloom, and G. Brown, "Compiling Joy to silicon," in *Proceedings of Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems* (T. Knight and J. Savage, eds.), pp. 79–98, MIT Press, Mar. 1992.

[42] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, pp. 720–738, June 1989.

[43] M. Shams, J. Ebergen, and M. Elmasry, "A comparison of CMOS implementations of an asynchronous circuits primitive: the C-element," in *International Symposium on Low Power Electronics and Design*, pp. 93–96, Aug. 1996.

[44] P. Day and J. V. Woods, "Investigation into micropipeline latch design styles," *IEEE Transactions on VLSI Systems*, vol. 3, pp. 264–272, June 1995.

[45] K. Y. Yun, P. A. Beerel, and J. Arceo, "High-performance asynchronous pipeline circuits," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press, Mar. 1996.

[46] A. Davis and S. M. Nowick, "Asynchronous circuit design: Motivation, background, and methods," in *Asynchronous Digital Circuit Design* (G. Birtwistle and A. Davis, eds.), Workshops in Computing, pp. 1–49, Springer-Verlag, 1995.

[47] S. Hauck, "Asynchronous design methodologies: An overview," *Proceedings of the IEEE*, vol. 83, Jan. 1995.

[48] A. Peeters, "The 'Asynchronous' Bibliography (BibTeX) database file `async.bib`." `ftp://ftp.win.tue.nl/pub/tex/async.bib.Z`. Corresponding e-mail address: `async-bib@win.tue.nl`.

[49] J. Garside, "The Asynchronous Logic Homepage." `http://www.cs.man.ac.uk/amulet/async/`.