Recent Advances in Asynchronous Design Methodologies

Kenneth Y. Yun
Dept. of Electrical and Computer Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0407 USA
E-mail: kyy@ucsd.edu

Abstract

This tutorial surveys recent advances in asynchronous design methodologies. Whenever possible, actual design examples are used to present the methods. Three primary topics are discussed: (1) asynchronous controllers, (2) asynchronous datapaths, and (3) asynchronous systems. For asynchronous controllers, basic design styles and tools are presented. For asynchronous datapaths, basic issues in timed vs. self-timed, matched delay vs. completion sensing, and power vs. performance tradeoff are discussed. For asynchronous systems, recently fabricated system design examples in processors, co-processors, and DSP’s are described.

Starting in the mid 1980’s, a number of controller synthesis methods were introduced, to address these limitations. These methods fall roughly into 3 categories: (i) translation methods; (ii) graph-based methods; and (iii) state machines.

1.1 Translation Methods

Several design styles use high-level languages of concurrency as their user-level specification formalisms. Martin’s method [32] is based on Hoare’s CSP; Brunvand [3] uses occam; Ebergen’s method [13] is derived from trace theory. The synthesis procedures typically involve syntax-driven translation [4] or algebraic transformation [27] of the specifications into delay-insensitive (DI) or speed-independent (SI) circuits.

An advantage of the compilation methods over other methods is that complex concurrent systems can be described elegantly and concisely in high level constructs without low-level timing concerns, which makes it easier to modify and verify the system behavior. However, because it is difficult to utilize global optimization techniques during the translation process, the automated synthesis often produces inefficient results. In general, the circuits generated using the compilation methods tend to incur considerably more area than those synthesized by other methods. Recently, some efforts have been made to address the optimization problems. Gopalakrishnan et al. [23] uses peephole optimization: i.e., translates a group of DI modules into burst-mode specifications and resynthesizes using the 3D tool [57].

There has been a concerted effort to synthesize commercial-scale circuits to demonstrate the practicality of these methods, such as the DCC error corrector chip developed at Philips Research Laboratories by van Berkel et al. [47] using a synthesis tool called Tangram compiler [48].
1.2 Graph-Based Methods

Almost all of the graph-based methods use the Petri net or a restricted form of the Petri net as the specification formalism. A Petri net is a graph model used for describing concurrent systems. Chu [6] introduced a restricted form of the Petri net called Signal Transitions Graph (STG) to specify asynchronous circuits. Chu’s initial definition of STG (interpreted free-choice Petri net), however, allowed only a limited mechanism to select alternative responses of the circuit. Meng [36] extended Chu’s work and developed an automatic synthesis tool. More recent works in STG-based synthesis include methods of Lavagno [30], Vanbekbergen [50], and Ykman-Couvreur et al. [56].

In general, the strong suit of the STG is its ability to express concurrency. Its main weakness is the awkwardness in specifying input choices. That is, the mechanisms to guide the responses of the machine is limited. In a free-choice STG specification, the machine selects the course of its future behavior solely based on input transitions. The machine cannot handle choices based on input levels. Vanbekbergen [50] introduced the generalized STG which allows input choices based on signal levels. However, his synthesis method cannot guarantee to generate hazard-free circuits. Cortadella at al [7] extended the STG to handle internal conflicts, i.e., arbitration. Some graph-based methods, such as Varshavsky’s [51], Beerel’s [2], and Kondratyev et al.’s [29], use state graphs to avoid syntactic problems associated with STGs.

Although most graph-based synthesis methods generate SI circuits, some methods use bounded wire delay model, a circuit model in which the delay of each gate and wire has a lower and an upper bound. Lavagno’s method inserts fixed delay elements to avoid internal hazards, though it makes no assumptions about the circuit’s environment. Myers’s method [37] uses an STG-like specification formalism called Event-Rule Systems [4]. His tool, ATACS, synthesizes very compact area-efficient generalized C-element circuits by exploiting all known delays, both internal and external.

1.3 Asynchronous Finite State Machines

AFSMs have been around for the past 30 years. The work on AFSMs was pioneered by Huffman and others. Early AFSMs [26, 46] assumed that the environment operates in fundamental mode, that is, the environment generates a single input change and waits for the machine to stabilize before it generates the next input change. Recent work in AFSMs allows the multiple-input change fundamental mode operations. This tutorial focuses on a recently introduced multiple-input change machine called burst-mode machine. Burst-mode asynchronous finite state machines were first introduced by Davis et al. [9] and formalized by Nowick and Dill [41, 38]. Burst-mode machines have been implemented using a method developed at HP Laboratories called MEAT [8], the locally clocked method [38], the 3D method [57], and the ULOCK method [39].

A burst-mode specification is a variation of a Mealy machine that allows multiple-input changes in a burst fashion—in a given state, when all of a specified set of input edges appear, the machine generates a set of output changes and moves to a new state. The specified input edges can appear in arbitrary order, thus allowing input concurrency, and the outputs are generated concurrently. The advantages of a burst-mode specification over STG specifications are that it is similar to the synchronous Mealy machine with which designers are familiar, that the input choice is more flexible than that of the STG, and that the state encoding is more flexible in the implementations. Burst-mode specifications have been very useful in specifying large, practical controllers, such as an asynchronous high performance cache controller [40], and asynchronous communications controllers [31]. Its main practical disadvantage is that it does not allow input changes to be concurrent with output changes. The input choice mechanism is more flexible than the STG but still primitive. For example, it cannot handle choices between two sets of concurrent events if one set is a subset of the other.

Yun and Dill [61, 57] extended the burst-mode design style by adding two new features: directed don’t cares and conditionals. Directed don’t cares allow an input signal to change concurrently with output signals, and conditionals allow control flow to depend on the input signal levels, in the same way synchronous state machines regulate control flow. Thus the extended burst-mode design style not only supports burst-mode multiple-input change asynchronous designs with added input/output concurrency, also allows the automatic synthesis of any synchronous Moore machine, in which the synchronous inputs are represented as conditional signals, and the clock is the only non-conditional signal. Moreover, this design style covers a wide range of circuits between burst-mode and fully synchronous. There have been several chips designed and fabricated, which incorporate communicating extended burst-mode controllers synthesized by the 3D tool: a high-performance differential equation solver chip [59]; a high-performance SCSI controller [62]; a significant portion of control circuitry for Intel’s Asynchronous Instruction Length Decoder chip.

2 Datapath

This section describes some of the recent advances in self-timed datapath design. First, two most widely used datapath building blocks, adders and multipliers, are discussed. Second, some recent advances in self-timed mem-
ory architecture are described. Third, some of the advantages and disadvantages of applying self-timed techniques to pipelined and non-pipelined datapaths are discussed. Although asynchronous circuits are used for many other reasons, this section concentrates on pros and cons of asynchronous circuits related to performance issues only.

2.1 Datapath building blocks

One feature that sets an asynchronous datapath element apart from its synchronous counterpart is its ability to report the completion of computation. This completion reporting mechanism enables a self-timed datapath to perform back-to-back operations without significant dead time between the operations. On the other hand, a synchronous datapath must reserve a sufficient amount of time to complete every operation in the worst-case scenario. Thus the performance of a synchronous circuit is limited by its worst-case behavior, whereas that of an asynchronous circuit is governed by its average-case behavior.

The most conventional technique used to detect and report the completion of computation is the use of a completion detection circuit based on dual-rail logic, as used in [33]. An alternative approach to detect completion is based on current sensing [12, 24], i.e., monitoring the current flow in the power supply line. Another interesting recent technique is speculative completion [42]: speculative reporting of early completion which can be aborted safely.

Below some of the recent techniques to improve the average-case performance of self-timed datapath elements are discussed in the context of adder and multiplier designs.

2.1.1 Adders

It is well-known that the delay of a ripple carry addition corresponds to the length of the longest carry chain (carry propagation). Von Neumann proved that for random input statistics the average length of the longest carry chain is \( \log_2 n \) for \( n \)-bit addition. Thus, for random inputs, the average delay of 32-bit addition is equal to the worst-case delay of 5-bit addition. It is an enormously difficult task to design a 32-bit adder with a worst-case delay that even approaches the worst-case delay of a 5-bit ripple carry adder. Hence, the use of the ripple carry scheme was justified for self-timed adders.

However, in most applications input statistics are not random. In fact, a large fraction of carry chains tends to be long in most applications, which makes the average-case delay to be skewed much closer to the worst-case. The analyses by Garside [20] and Kinnement [28] using the actual input operand statistics from an ARM-6 simulator demonstrate that the average-case delay of self-timed ripple carry addition is worse than the worst-case delay of an adder with a simple mechanism to reduce the worst-case delay, such as carry bypass or carry lookahead. Thus the recent high-performance self-timed designs incorporate worst-case delay reduction features.

In adders, the order of sum bit generation cannot be known a priori; thus completion detection circuits must detect the completion of every sum bit. If dual-rail logic is used to detect the completion of each bit, the completion of an \( n \)-bit addition requires an \( n \)-bit OR-AND function in general. Yun et al. [60] observed that it is sufficient to detect the completion of carry bits because the delay from a carry to the corresponding sum bit is always smaller than the minimum delay through the completion logic in practical circuits. In order to further minimize the completion sensing overhead for the worst-case computation, Yun et al placed late arriving signals (for the worst-case computation) closer to the output in the domino logic implementation of completion detection circuit. Their techniques resulted in 2.8ns average-case delay for a 32-bit carry bypass adder fabricated in 0.6\( \mu \)m CMOS process, with only 20% completion sensing overhead on average.

Nowick et al. used the speculative completion technique to speed up the reporting of completion for Brent-Kung style binary carry lookahead adders [42]. This technique partitions the delay of a datapath element in several regions. For example, the delay of an adder with 4ns worst-case delay may be discretized into 2 regions: less than 2ns and between 2ns and 4ns. The circuit then reports that the delay is 2ns if the actual delay is less than 2ns, and 4ns if the actual delay is between 2ns and 4ns. This technique requires a special auxiliary circuit called "abort detection circuit", which operates in parallel with the datapath element itself, to compute using data operands whether the actual delay may exceed the initial prediction. Therefore the circuit assumes that the delay is less than 2ns initially. However, if the "abort detection circuit" computes that the actual delay may take longer than 2ns, then the reporting of early completion (2ns delay) is aborted and the circuit reports its completion at 4ns. The application of this technique to a 32-bit Brent-Kung adder resulted in the simulated average-case delay to be less than 2ns in 0.6\( \mu \)m CMOS process.

2.1.2 Multipliers and dividers

Perhaps the most significant advance in self-timed iterative structures is the development of zero-overhead self-timed ring technique by Williams [53]. Williams showed that a self-timed ring can be designed in dual-rail domino logic with essentially zero overhead. He applied this technique to a self-timed 160ns 54-bit mantissa divider [54] as a part of a floating-point divider. This design was incorporated in a commercial microprocessor design [52]. It can be shown that this technique is generally applicable to any iterative
structure in which the latency needs to be optimized. Consequently, this technique has been applied to other academic and industrial designs, such as a division and square root unit design by Matsubara and Ide [35].

The self-timed dual-rail domino circuit technique can be applied to non-iterative structures to optimize the latency. This technique is used in many array multipliers (both synchronous and self-timed) to speed up carry save addition. Yun et al. designed a 32-bit 8ns self-timed multiplier, which is optimized to produce the least significant 32 bits of the products fast, using this technique in [60].

2.1.3 Memory

Garside et al. demonstrated a self-timed cache system for AMULET2e which exhibits average-case delay [21]. Although there are many interesting aspects of this design, the most important feature is its ability to exploit the sequential nature of memory references. For example, if a cache read access is a hit and the data is from the same cache line as the previous cache access, then the data can be read out quickly without timing-consuming precharge-discharge cycle.

2.2 Pipelined and non-pipelined datapath

Datapaths are built by combining the elements described above. There are two types of datapaths: pipelined and non-pipelined. There have been a tremendous amount of work done in asynchronous pipelines, starting with the classical work by Sutherland [45]. There are micropipelines with two-phase control [58, 1] as well as with four-phase control [10, 19, 16]. Micropipelines have been applied to many asynchronous system designs. The most famous ones are AMULET1 [15] and AMULET2 [18] by Furber’s Manchester group. All of these designs strive to obtain an average-case throughput that is superior to the worst-case throughput of comparable synchronous circuits. However, most have failed to show any real advantages. One possible explanation is that the average-case throughput (taking into account data dependency only, not operating conditions) of a deeply pipelined asynchronous circuit is fairly close to the worst-case throughput in most systems. Thus it is likely that shorter pipelines exhibit much better average-case behavior.

Examples of non-pipelined datapaths are Yun et al.’s differential equation solver [60], Williams’s divider ring [54], van Berkel et al.’s DCC error corrector [47], etc. The performance advantage of asynchronous circuits is much more pronounced in non-pipelined datapaths because the latency is simply the sum of all datapath element delays in the critical path. Thus the average-case latency is determined roughly by the sum of the average-case delay of individual elements.

3 Asynchronous Systems

Recently, researchers have identified many systems applications for which asynchronous implementations would bring out significant, tangible benefits. Some have speculated that it would be easier to obtain true “average case” performance in asynchronous systems than in synchronous counterparts. Some have asserted that it is easier and more natural to design low-power asynchronous systems than synchronous counterparts. Still others have argued that asynchronous circuits are best suited for bridging locally synchronous modules in globally asynchronous locally synchronous (GALS) systems [5]. This tutorial surveys examples from all three camps: systems with high average-case performance, low power/EM systems, and interface circuits for GALS systems.

3.1 High-Performance Systems

It is well known that asynchronous circuits can achieve average-case delay while synchronous circuits achieve a fixed delay determined by their clock frequency that must be preset according to the worst-case conditions. The difference between the average and worst-case conditions stems from both data-dependency and environmental factors. In synchronous systems, clock frequencies are set, assuming that the chip is running at a temperature as high as 100°C, even though the chip usually operates at a much lower temperature. This difference can lead to significant average-performance advantage for asynchronous circuits [11].

Yun et al. [60] demonstrated a high-performance asynchronous differential equation solver chip, whose average-case speed (tested at 22°C and 3.3V) is 48% faster than comparable synchronous designs (designed to operate at 100°C and 3V for the slow process corner). This speedup is attributed to two factors: (i) inherent margins that must be built in synchronous systems to accommodate worst-case timing behavior but are not required for asynchronous systems; (ii) drastic reduction in control overhead associated with typical asynchronous systems.

In certain applications in which there is a large variation in processing delays between common and rare cases, asynchronous designs tend to fare much better than synchronous designs. A research group at Intel demonstrated this with their asynchronous instruction length decoder design called RAPPID (“Revolving Asynchronous Pentium Processor® Instruction Decoder”). The RAPPID’s length decoding out-performs, by a factor of 3, the same function inside a 400MHz Pentium II fabricated in the identical 0.25µm CMOS process. This speedup is primarily attributed to optimizations for common, short-length instructions and self-timed techniques enabling these optimizations.
3.2 Low-Power Systems

Many researchers sought high-performance asynchronous alternatives to conventional synchronous microprocessors, advocating modular "plug and play" speed-independent designs that avoid clock skew problems. However, what they discovered instead, by the time the first generation Alpha© microprocessors, which minimized clock skews by electrically shorting large clock buffers, were introduced, was that designers would pay almost any price for high performance. The focus of asynchronous microprocessor designers then shifted to low power aspects of asynchronous design. It was obvious to many that asynchronous systems are very good at doing nothing without dissipating power, whereas synchronous designs must employ complex clock gating techniques and power-down circuitry to reduce power during standby. It is difficult to assess how much power advantage asynchronous designs have, compared to synchronous designs employing aggressive clocking gating, such as StrongARM© microprocessors. However, it is easy to claim that it is much easier to save power using asynchronous designs. Furthermore, some researchers claim that asynchronous systems have low noise emission, compared to synchronous counterparts.

The first asynchronous VLSI microprocessor was introduced by Martin et al. [34]. This was a 16-bit RISC microprocessor design, which operated correctly under a wide range of temperature and voltage. The 2μm CMOS version dissipated only 6.7 mW at 2 V and 145 mW at 5 V. The design was mostly speed-independent (except for the memory interface).

The AMULET group headed by Furber at Manchester University have designed three ARM©-compatible microprocessors: AMULET1 [55], AMULET2e [18], and AMULET3 [17]. The first two have been fabricated and tested successfully. Although AMULET1 did not fare very well compared to the synchronous version, AMULET2e demonstrated competitive performance and power-efficiency compared to its synchronous counterpart (ARM-710).

There have been other recent microprocessor designs which boast low power or low noise emission: ASPRO-16 [44], a 16-bit microprocessor by French Telecom which dissipates 0.5 W at 200 peak MIPS; a self-timed DSP [43] by Cogency with low noise and power profiles; an asynchronous low-power implementation of 80C51 microcontroller [49] by Philips Research.

3.3 Asynchronous/Synchronous Interface

It is clear that there at least exist interesting niche applications that can take advantage of asynchronous techniques. However, a vast majority of systems are and will continue to be synchronous. The question then is how to utilize some of the proven benefits of asynchronous circuits in a largely synchronous environment.

Some have suggested that communication between modules should be asynchronous (although the modules themselves are synchronous) because the cost of global synchrony is prohibitively high in large-scale VLSI systems. Chaplii first suggested the idea of GALS system in [5]. Yun and Donohue demonstrated a prototype GALS system with a mixture of asynchronous and synchronous modules in [63]. In this chip, asynchronous modules were equipped with "pausable clocking control" to prevent synchronization failures.

Yet others have argued that maintaining precise frequency reference in a globally synchronous environment is not too difficult. The real problem is the uncertainty in clock phases. Ginosar and Kol [22] suggested an adaptive synchronization scheme to remedy this problem. Furthermore, some synchronous systems [25, 64] are moving closer to asynchronous by allowing significant time borrowing to overcome clock skew and jitter problems.

References


