Practical Advances in Asynchronous Design

Erik Brunvand
Dept. of Computer Science
University of Utah
SLC, Utah 84112

Steven Nowick
Dept. of Computer Science
Columbia University
New York, NY 10027

Kenneth Yun
Department of ECE
University of California
San Diego, CA 92093

Abstract

Recent practical advances in asynchronous circuit and system design have resulted in renewed interest by circuit designers. Asynchronous systems are being viewed as an increasingly viable alternative to globally synchronous system organization. This tutorial will present the current state of the art in asynchronous circuit and system design in three different areas. The first section details asynchronous control systems. The second describes a variety of approaches to asynchronous datapaths. The third section is on asynchronous and self-timed circuits applied to the design of general purpose processors

1. Asynchronous Control *(Steven Nowick)*

The earliest asynchronous state machine implementations were *Huffman machines*. These machines have combinational logic, primary inputs, primary outputs and fed-back state variables. There are no clocked latches or flip-flops. Instead, state is simply stored on feedback loops. Typically a *fundamental mode* assumption is required, to insure correct operation: once an input change occurs, no new inputs may arrive until the machine has stabilized. Much of the basic theory on asynchronous state machines was developed by Huffman, Unger, and McCluskey, and is described well in [59].

Hazards, or the potential for glitches, are an important consideration in any asynchronous design [59]. Unlike synchronous machines, asynchronous machines have no global clock to filter out the effect of glitches. Therefore, any glitch may be interpreted as a valid signal change, and may cause the machine to malfunction. The classic theory of combinational hazards was developed by Huffman, Unger and McCluskey for *single-input change (SIC)* state machines [59], and was later extended to *multiple-input change (MIC)* machines [19, 5]. In addition, *critical races* and *essential hazards* were identified, and techniques to eliminate them were proposed.

While this early work laid the foundations of asynchronous controller synthesis, it had a number of limitations. First, many of these controllers had highly constrained behavior, and thus could not be used in concurrent environments. Second, a number of the controllers had poor performance, due to the use of inertial delays to filter out hazards [59]. Finally, little work was done on CAD optimization algorithms and tools, which are needed for industrial applications.

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

1.1. Asynchronous State Machines.

Much of the recent work on asynchronous state machines is centered on *burst-mode machines*. These specifications were introduced to allow more more concurrency than traditional SIC and MIC machines. Burst-mode specifications grew out of early work by Davis [14]. Davis developed 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 contribution is that, unlike classic 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. This work was later extended by Davis, Coates and Stevens [15].

Nowick and Dill [47, 45] formalized and modified the data-driven specifications into the final form called *burst-mode* [47, 45]. They also proposed a new self-synchronized design style called a *locally-clocked state machine* [47, 45], which was the first burst-mode synthesis method to guarantee a hazard-free implementation. The synthesis method has been automated and applied to a number of designs: a high-performance second-level cache controller [46], a DRAM controller and a SCSI controller for example.

Yun and Dill [66] proposed an alternative implementa-

tion style for burst-mode machines, called a *3D machine*. Unlike locally-clocked machines, these are Huffman machines, with no local clock or latches. The synthesis method has been automated and applied to several significant designs, including an experimental SCSI controller at AMD Corporation [68]. Yun, Dill and Nowick [69, 67] later generalized burst-mode specifications into *extended burst-mode* specifications, to allow greater concurrency and flexibility. 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.

A number of burst-mode optimization algorithms and CAD tools have been developed, such as for: state assignment [20], hazard-free 2-level logic minimization [48, 58], multi-level logic optimization [33, 36], and technology mapping [55]. A high-level synthesis package, called *ACK* [32], incorporates several burst-mode tools. Testability techniques have also been developed [49].

Burst-mode tools have been applied to several industrial designs, including an experimental routing chip [15] and low-power infrared communications chip [37] at HP Laboratories, and a high-performance experimental instruction decoder at Intel Corporation.

1.2. 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 is a directed bipartite graph which can describe 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.

Several early synthesis methods use a constrained class of Petri net called a *marked graph*, which can model concurrency, but not choice. A more general class of Petri net, called a *Signal Transition Graph (STG)*, was used by Chu [10] and Meng [42]. Alternatively, some researchers are using *state graphs* for specifications, as an alternative to Petri nets [62, 3, 31]. State graphs allow the direct specification of interleaved behavior, avoiding some of the structural complexity of Petri nets.

Many synthesis algorithms have been developed, for both STG and graph-based synthesis. Algorithms have been developed for state minimization and assignment [34]. A comprehensive algorithm for hazard-free logic decomposition was introduced by Burns [8]. More recently, the *theory of regions* has been used as a powerful tool in developing efficient STG algorithms [13].

A number of CAD packages are now available. Lavagno developed an influential CAD system for STG synthesis, which has been incorporated into the Berkeley *SIS* tool package [35]. A synthesis method which focuses on *timed circuits*, called ATACS, was introduced by Myers [44]. Finally, several tools have been developed for *speed-independent* circuits, which operate correctly regardless of the actual gate delays [62, 3, 31].

1.3. Translation Methods.

In a translation method, 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*, such as *occam* or *trace theory*. The program is then transformed, by a series of steps, into a low-level program which maps directly to a circuit.[1]

Ebergen [18] introduced a synthesis method using specifications called *commands*. The command is then *decomposed* in a series of steps into an equivalent network of components, using a "calculus of decomposition". An alternative approach was proposed by Udding and Josephs [29].

While these methods use algebraic calculi to derive asynchronous circuits, other methods rely on compiler-oriented techniques. Martin [38] specifies an asynchronous system as a set of concurrent processes which communicate on channels. His specification language uses communication constructs from Hoare's CSP, and sequential constructs from Dijkstra's guarded command language. The specification is then translated into a collection of gates and components which communicate on wires. The synthesis method was automated by Burns [9] and applied to many substantial examples. This work was extended by Akella and Gopalakrishnan [1] to allow global shared variables. Brunvand and Sproull [7] developed an alternative compiler based on occam specifications.

A different approach was developed by van Berkel, Rem and others [60, 61], using the *Tangram* language. A Tangram specification is compiled by syntax-directed translation into an intermediate representation called a *handshake circuit*, which is then mapped to a VLSI implementation. The Tangram compiler has been successfully used at Philips Research Laboratories for many experimental designs, including an error corrector for a digital compact cassette player [61].

2. Datapath *(Kenneth Yun)*

This section describes some of the recent advances in self-timed datapath design. We begin by describing two of the most widely used datapath building blocks: adders and multipliers. Then we discuss self-timed memory architecture. We conclude with our views on some of the

[1] Some of these methods synthesize both datapath and control.

advantages and disadvantages of applying self-timed techniques to pipelined and non-pipelined datapaths. Although asynchronous circuits are used for many other reasons, in this section we will concentrate 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 [40]. An alternative approach to detect completion is based on current sensing [17, 28], i.e., monitoring the current flow in the power supply line. Another interesting recent technique is *speculative completion* [50]: speculative reporting of early completion which can be aborted safely.

We describe below some of the recent techniques to improve the average-case performance of self-timed datapath elements 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 [26] and Kinnement [30] 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. It is known 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 [71]. Their techniques resulted in 2.8ns average-case delay for a 32-bit carry bypass adder fabricated in 0.6μm 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 [50]. 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 0.6μm 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 [64]. 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 [65] as a part of a floating-point divider. This design was incorporated in a commercial microprocessor design [63]. It can be shown that this technique is generally applicable to any iterative structure in which the latency needs to be optimized. Con-

sequently, this technique has been applied to other academic and industrial designs, such as a division and square root unit design by Matsubara and Ide [41].

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 [71].

2.1.3 Memory

Garside et al. demonstrated a self-timed cache system for AMULET2e which exhibits average-case delay [27]. 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 has been a tremendous amount of work done in asynchronous pipelines, starting with the classical work by Sutherland [57]. There are micropipelines with two-phase control [70, 2] as well as with four-phase control [16, 25, 22]. Micropipelines have been applied to many asynchronous system designs. The most famous ones are AMULET1 [21] and AMULET2e [24] 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. Our conjecture is that the average-case throughput (taking into account data dependency only, not operating conditions) of a *deeply pipelined* asynchronous circuit would be closer to the worst-case throughput, 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 [71], Williams's divider ring [65], van Berkel et al.'s DCC error corrector [4], 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 Processors *(Erik Brunvand)*

Although asynchronous techniques are applicable across a wide range of circuits, this section of this special session is dedicated to the design of asynchronous processors. Microprocessors are, perhaps, the most highly developed digital circuits, and therefore the most demanding application for asynchronous techniques. Furthermore, the organizations used in modern microprocessors to achieve very high performance are firmly rooted around synchronous pipelines, and many techniques do not transfer readily to the asynchronous domain. Therefore designers of asynchronous processors have to take a step back and view the problem from a different perspective, often developing new techniques and sometimes finding approaches which benefit from the more flexible design environment offered by asynchronous control.

There have been relatively few asynchronous processors reported in the literature. Early work in asynchronous computer architecture includes the Macromodule project during the early 70's at Washington University [11, 12] and the self-timed dataflow machines built at the University of Utah in the late 70's [14].

Although these projects were successful in many ways, asynchronous processor design did not progress much in the following years, perhaps because the circuit concepts were a little too far ahead of the available technology. With the advent of easily available custom ASIC technology, either as VLSI or FPGAs, asynchronous processor design is beginning to attract renewed attention. Some recent processor projects include the following:

3.1. The CalTech Asynchronous Microprocessor

The first asynchronous VLSI processor was built by Alain Martin's group at CalTech [39]. It is completely asynchronous, using (mostly) delay-insensitive circuits and dual-rail data encoding. The processor as implemented has a small 16-bit instruction set, uses a simple two-stage fetch-execute pipeline, is not decoupled, and does not handle exceptions. It has been fabricated both in CMOS and GaAs.

3.2. The NSR

The NSR (Non-Synchronous RISC) processor [6] is structured as a five-stage pipeline where each pipe stage operates concurrently and communicates over self-timed data channels in the style of micropipelines. Branches, jumps, and memory accesses are also decoupled through the use of additional FIFO queues which can hide the execution latency of these instructions. The NSR was built using FPGAs. It is pipelined and decoupled, but doesn't handle exceptions. It is a simple 16-bit processor with only sixteen

instructions, since it was built partially as an exercise in using FPGAs for rapid prototyping of self-timed circuits.

3.3. The Amulet

A group at Manchester has built a number of versions of a self-timed micropipelined VLSI implementation of the ARM processor [23, 24] which is an extremely power-efficient commercial microprocessor. The Amulet is a real processor in the sense that it mimics the behavior of an existing commercial processor and it handles simple exceptions. It is more deeply pipelined than the synchronous ARM, but it is not decoupled (although it does allow instruction prefetching), and its precise exception model is a simple one. The Amulet has been designed and fabricated. The performance of the first-generation design is within a factor of two of the commercial version [51]. Later generations of Amulet are closing this gap.

3.4. The Counterflow Pipeline Processor

This is an innovative architecture proposed by a group at Sun Microsystems Labs [56]. It derives its name from its fundamental feature, that instructions and results flow in opposite directions in a pipeline and interact as they pass. The nature of the Counterflow Pipeline is such that it supports in a very natural way a form of hardware register renaming, extensive data forwarding, and speculative execution across control flow changes. It should also be able to support exception processing.

A self-timed micropipeline-style implementation of the CFPP has been proposed. The CFPP is deeply pipelined and partially decoupled, with memory accesses launched and completed at different stages in the pipeline. It can handle exceptions, and a self-timed implementation which mimics a commercial RISC processor's instruction set has been simulated.

3.5. The Fred Architecture

Fred is a self-timed, decoupled, concurrent, pipelined computer architecture [54, 53]. It dynamically reorders instructions to issue out of order using an instruction window to organize the reordering, and allows out-of-order instruction completion. It handles exceptions and interrupts. Several features of the Fred architecture are directly related to its self-timed design, such as the decoupled branch mechanism and exception model.

The major innovation of the Fred architecture, which makes possible many additional features, is the functionally precise exception model [52]. A precise exception model allows the programmer to view the processor state as though the exception occurred at a point exactly between two instructions, such that all instructions before that point have completed while all those after have not yet started. Fred's functionally precise model instead simply presents a snapshot of the current instruction stream, in which some instructions have faulted, some have not yet issued, and any nominally sequential instructions which are not present have completed successfully out of order.

3.6. Rotary Pipeline Processor

The rotary pipeline processor is an architecture for superscalar computing [43]. It is based on a simple and regular pipeline structure which can support several ALUs for efficient dispatching of multiple instructions. In its simplest conceptual form, a rotary pipeline circulates all of the processor's registers around a ring. As the register file flows around this ring, values are inspected by the various function units and results are inserted. Registers are not kept in lock step although they are prevented from lapping each other. During normal operation the control circuits are not on the critical path and performance is limited only by data rates.

References

[1] 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.

[2] S. S. Appleton, S. V. Morton, and M. J. Liebelt. A new method for asynchronous pipeline control. In *7th Great Lakes Symposium on VLSI*, 1997.

[3] P. 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.

[4] 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*, 29(12):1429–1439, Dec. 1994.

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

[6] E. Brunvand. The NSR processor. In *Proceedings of the 26th International Conference on System Sciences*, Maui, Hawaii, January 1993.

[7] 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.

[8] S. 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.

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

[10] 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.

[11] W. A. Clark. Macromodular computer systems. In *Spring Joint Computer Conference*. AFIPS, April 1967.

[12] W. A. Clark and C. A. Molnar. Macromodular system design. Technical Report 23, Computer Systems Laboratory, Washington University, April 1973.

[13] 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.

[14] A. Davis. The architecture and system method for DDM1: A recursively structured data-driven machine. In *5th Annual Symp. on Computer Architecture*, April 1978.

[15] 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.

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

[17] M. E. Dean, D. L. Dill, and M. Horowitz. Self-timed logic using current-sensing completion detection (CSCD). *Journal of VLSI Signal Processing*, 7(1/2):7–16, Feb. 1994.

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

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

[20] R. Fuhrer, B. Lin, and S. 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.

[21] S. Furber. Computing without clocks: Micropipelining the ARM processor. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 211–262. Springer-Verlag, 1995.

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

[23] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In *Proceedings of VLSI93*, Grenoble, France, 1993.

[24] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. Paver. AMULET2e: An asynchronous embedded controller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Apr. 1997.

[25] S. B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Mar. 1996.

[26] J. D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 181–207. Elsevier Science Publishers, 1993.

[27] J. D. Garside, S. Temple, and R. Mehra. The AMULET2e cache systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Mar. 1996.

[28] E. Grass, R. C. S. Morling, and I. Kale. Activity monitoring completion detection (AMCD): A new single rail approach to achieve self-timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Mar. 1996.

[29] M. Josephs and J. 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.

[30] D. J. Kinnement. An evaluation of asynchronous addition. *IEEE Transactions on VLSI Systems*, 4(1):137–140, Mar. 1996.

[31] 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.

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

[33] D. 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.

[34] L. Lavagno, C. Moon, R. 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.

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

[36] 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.

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

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

[39] A. Martin, S. Burns, T. Lee, D. Borkovic, and P. Hazewindus. The design of an asynchronous microprocessor. In *Proc. Cal Tech Conference on VLSI*, 1989.

[40] A. J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119–137, July 1992.

[41] G. Matsubara and N. Ide. A low power zero-overhead self-timed division and square root unit combining a single-rail static circuit with a dual-rail dynamic circuit. In *Proc. International Symposium on Advanced Research in Asynchronous*

Circuits and Systems. IEEE Computer Society Press, Apr. 1997.

[42] T. H.-Y. Meng, R. Brodersen, and D. 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.

[43] S. Moore, P. Robinson, and S. Wilcox. Rotary pipeline processors. *IEE Proceedings, Computers and Digital Techniques*, 143(5), September 1996.

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

[45] S. 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).

[46] S. Nowick, M. Dean, D. 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.

[47] S. Nowick and D. 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.

[48] S. Nowick and D. 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.

[49] S. Nowick, N. 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.

[50] S. M. Nowick, K. Y. Yun, and P. A. Beerel. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Apr. 1997.

[51] N. C. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, University of Manchester, 1994.

[52] W. F. Richardson and E. Brunvand. Precise exception handling for a self-timed processor. In *1995 International Conference on Computer Design: VLSI in Computers & Processors*, pages 32–37, Los Alamitos, CA, October 1995. IEEE Computer Society Press.

[53] W. F. Richardson and E. Brunvand. Architectural considerations for a self-timed decoupled processor. *IEE Proceedings, Computers and Digital Techniques*, 143(5), September 1996.

[54] W. F. Richardson and E. Brunvand. Fred: An architecture for a self-timed decoupled computer. In *Advanced Research in Asynchronous Circuits and Systems (ASYNC96)*, Aizu-Wakamatsu, Japan, 1996.

[55] 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.

[56] R. E. Sproull, I. E. Sutherland, and C. E. Molnar. The counterflow pipeline processor architecture. *IEEE Design and Test of Computers*, 11(3), 1994.

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

[58] M. Theobald, S. 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.

[59] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, 1969.

[60] C. van Berkel and R. 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.

[61] 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.

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

[63] T. Williams, N. Patkar, and G. Shen. SPARC64: A 64-b 64-active-instruction out-of-order-execution MCM processor. *IEEE Journal of Solid-State Circuits*, 30(11):1215–1226, Nov. 1995.

[64] T. E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, June 1991.

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

[66] K. Yun and D. 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.

[67] K. Yun and D. 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.

[68] K. Yun and D. 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.

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

[70] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance two-phase micropipeline building blocks: double edge-triggered latches and burst-mode select and toggle circuits. *IEE Proceedings, Circuits, Devices and Systems*, 143(5):282–288, Oct. 1996.

[71] K. Y. Yun, A. E. Dooply, J. Arceo, P. A. Beerel, and V. Vakilotojar. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Apr. 1997.