Tomorrow's Digital Hardware will be Asynchronous and Verified[1]

Alain J. Martin

Department of Computer Science, California Institute of Technology, Pasadena CA 91125, USA

Abstract Encouraged by the results of almost a decade of research and experimentation, we claim that tomorrow's design methods for digital VLSI will be based on a concurrent programming approach to high-level synthesis, asynchronous techniques, and correctness-preserving program transformations.

1 Introduction

It has become a *cliché* to say that VLSI has revolutionized electronic design. But the most profound transformations in design methods for digital hardware are still to come.

With chip density quadrupling every two years for the last two decades, the quantitative changes brought about by VLSI could not be ignored. Yet, the qualitative changes in the nature of the artifacts embodied in a digital chip have so far not been fully recognized.

A VLSI chip is a highly concurrent computation for the design of which the traditional methods of automata and switching theory are inadequate[13]. Not only is a chip one of the most complex systems technology can produce, but it is also one of the most fragile. Because of the nature of digital computation, a minute design or fabrication error can render the chip inoperable. Unlike software, integrated circuits are not repairable. The development costs are so high that a delay of a few weeks in the completion of an industrial project may account for the difference between profit and loss. To make matters worse, the same rate of technological change that increases the complexity of the products reduces the development time to the point that a manufacturer hardly has time to bring a new product to market before it becomes obsolete.

In view of the size of the problems, it is clear that criteria including correctness by construction, ease of composition and modification, robustness to changing or unpredictable physical parameters are going to determine future design methods. The main thesis in this paper is that it is possible to achieve these goals, *without sacrificing efficiency*, with a method that combines three aspects: a concurrent programming approach to high-level synthesis, asynchronous techniques for digital VLSI, and correctness-preserving program transformations.

Lest the reader would immediately dismiss this claim as yet another beautiful theory

[1]This is a revised version of an invited paper published in the proceedings of the IFIP Congress 1992: Information Processing 92, Volume I, Elsevier Science Publishers B.V.

waiting to be killed by an ugly little fact, let me mention without delay that the views expressed here are strongly supported by almost a decade of research, experiments, and fabricated designs. The results of these experiments, conducted by my research group at Caltech, have been positive beyond our most optimistic expectations.

The method indeed produces correct and efficient circuits. It has been applied to a series of difficult problems, such as distributed mutual exclusion, arbitration, routing automata, stack and queues, multipliers, and a suite of components for a complete computer system comprising a pipelined microprocessor, static RAM, and memory management unit. Apart from the memory management unit, which was simulated but not fabricated, all chips have been fabricated in CMOS, (some of them also in GaAs). All CMOS chips have been found functional "on first silicon."

The discussion can best be partitioned into several research themes. The first one is the application of concurrent programming techniques to the design of VLSI circuits. The second issue is that of asynchronous techniques for digital circuits. The third issue is that of correctness by construction and program transformations. We will then discuss the expected influence of asynchronous VLSI and high-level synthesis on the architecture of future computing systems. Finally, we will discuss the influence of this approach on our understanding of concurrency and digital computation.

1.1 VLSI Design as Concurrent Computing

A VLSI system is a highly concurrent computation and therefore any approach to VLSI design should be a concurrent computing approach. Also, communication in VLSI is becoming increasingly expensive, compared to switching, as the size of the wires determines both the switching costs and the area of a chip. A concurrent computation model for VLSI should reflect those cost ratios, and a model in which communication is explicit is more appropriate to control the cost of communication. Hence, we opted for a notation based on the notion of concurrent processes communicating by explicit message-passing and assignments to variables.

The program notation that best matches those requirements is C.A.R. Hoare's CSP[3]. I will review how CSP was modified to fit our purposes. More generally, I will try to assess the differences and similarities between programming in VLSI and traditional programming for stored-program computers, and show how these differences are reflected in the notation and programming style. A related question is: To which extent is it possible to capture a VLSI designer's choices toward a solution for a VLSI computation at the level of a CSP program?

1.2 Concurrent Computing and Asynchronous VLSI

A high-level synthesis approach to VLSI design requires finding an interface that provides a good separation of the physical and algorithmic concerns. In synchronous techniques, clocks are used to implement sequencing, and thus knowledge on the duration of each computation step has to be used. Since this knowledge is derived from the physical parameters of the circuit, those techniques are detrimental to the use of high-level methods. Furthermore, with the increasing size of circuits, it becomes more and more difficult, and costly in area, delay, and power consumption, to distribute a clock signal across a chip. Finally, the restrictions attached to wire lengths in order to maintain certain timing properties add extra complication to the already difficult layout problem.

For these reasons, asynchronous techniques, and among them, *delay-insensitive* techniques, are particularly attractive for high-level VLSI synthesis. A circuit is delay-insensitive when its correct operation is independent of any assumption on delays in operators and wires except that the delays are finite. [2] Obviously, delay-insensitive circuits don't use a clock and are therefore asynchronous. Sequencing is enforced entirely by communication mechanisms.

Hence, the second aspect of the method is the design of asynchronous VLSI circuits using techniques from concurrent computation. Although it has been known for a long time that asynchronous techniques were potentially superior to the standard synchronous ones, they have been largely ignored so far because they were too difficult to master. In particular, no good method was known to avoid the synchronization errors resulting in the malfunctionings called hazards. As a consequence, the circuits produced, when correct at all, were both too large and too slow.

The problem of hazards was easy to solve once it was addressed as a concurrent programming one. Knowledge of concurrent computing was essential to raise asynchronous design from an interesting curiosity to that of the design technique of the future. Conversely, asynchronous VLSI design poses fundamental questions related to the nature of concurrent computing, and challenges conventional approaches to computer architecture.

1.3 Correctness by Program Transformations

The issue of correctness is central to all research in programming methodology. After two decades of intense activity and impressive advances, the impact of the research on the software industry is still disappointing. Perhaps, the main reason is that both software technology and, more importantly, software users are just too malleable and forgiving. It is technically possible—although very costly and dangerous—and socially acceptable to debug large pieces of software by intensive testing and modifications. But, as a VLSI designer puts it, "if FORTRAN compilation cost $40,000 and took 6-8 weeks, significantly more effort would go into the pre-compilation verification of FORTRAN programs"[4]. Because of the drastically different cost structures between hardware and software, I believe that designing systems that are correct by construction will happen in hardware earlier than in software. *Tomorrow's digital hardware will be asynchronous and verified.*

The approach to designing correct VLSI circuits I advocate is that of program transformations. A circuit is first constructed as a concurrent program. This program is proved to meet the specifications. Often, the program is simple enough to be considered the specification itself. The production of the final circuit is then a matter of applying a series of semantics-preserving transformations. Each transformation replaces a program with a semantically equivalent one, until a version is obtained that can be directly implemented in VLSI. We never need to leave the algorithmic domain for other forms of description, like

[2]We have proved that the class of entirely delay-insensitive circuits is very limited, and that some compromise to delay-insensitivity has to be made in most circuits. See [10] for a discussion of these issues.

finite-state machines or state-transition graphs, hence eliminating an important source of errors introduced during the informal translation from one representation to another.

Entirely automatic compilation of programs into circuits ("silicon compilation") is possible with our method: We have demonstrated the possibility by writing a compiler[1]. But the resulting circuits are unnecessarily inefficient compared to what can be achieved by what we call "designer-assisted compilation." In such an approach, the designer uses a set programs that perform the transformations automatically—avoiding the clerical errors that humans excel at. But the designer can choose which transformations to apply by using global knowledge (invariants) of the system that is not available to an automatic compiler. This approach gives excellent results (the circuits obtained are simpler and more efficient than those produced by a "seat-of-the-pants" approach), hence refuting the often accepted "fatality" that formal methods necessarily lead to inefficient designs.

2 An Asynchronous Microprocessor

As an example, I will briefly describe the (quasi) delay-insensitive microprocessor my students and I designed in the fall of 1988 [6]. It is the first asynchronous microprocessor ever designed. The chips were found fully functional on "first silicon."

The processor was first specified as a sequential program, which was then transformed into a concurrent program so as to pipeline instruction execution. The circuits were derived from the concurrent program by semantics-preserving program transformation. It took five persons, some of us working part-time, less than five months to complete the design from scratch.

The processor has a 16-bit, RISC-like instruction set. It has sixteen registers, four buses, an ALU, and two adders. Instruction and data memories are separate. The chip size is about 20,000 transistors. Two versions have been fabricated: one in $2\mu m$ MOSIS SCMOS, and one in $1.6\mu m$ MOSIS SCMOS. With the exception of *isochronic forks*[10] and the interfaces with the memories, the chips are entirely delay-insensitive.

The $2\mu m$ version runs at 12 MIPS. The $1.6\mu m$ version runs at 18 MIPS. (These performance figures are based on measurements from sequences of ALU instructions without carry. They do not take advantage of the overlap between ALU and memory instructions.)

We have tested the chips under a wide range of *VDD* voltage values. At room temperature, the $2\mu m$ version is functional in a voltage range from 7V down to 0.35V! And it reaches 15 MIPS at 7V. We have also tested the chips cooled in liquid nitrogen. The $2\mu m$ version reaches 20 MIPS at 5V and 30 MIPS at 12V. The $1.6\mu m$ version reaches 30 MIPS at 5V. For the $2\mu m$ version, the power consumption is 145mW at 5V and 6.7mW at 2V. For the $1.6\mu m$ version, it is 200mW at 5V and 7.6mW at 2V.

3 Power Consumption

The relevant measure of performance for microprocessors and other general-purpose devices that I propose is the speed-to-power ratio, for instance expressed in MIPS per Watt. Usual RISC microprocessors deliver less than 10 MIPS/W at 5V. The fastest RISC microprocessor announced at the moment of writing, the 32-bit DEC Alpha, is supposed to run at 300MIPS and consume 30W, which amounts to 10MIPS/W at 3.3V. For our microprocessor, the performances range from 60MIPS/W at 5V for the 1.6μm version to 600MIPS/W at 2V for the 2μm version. Even with a 1 to 2 ratio in word lengths, the discrepancy is worth some attention! The power advantage of asynchronous circuits, which is suggested by this experiment, is still a matter of controversy. In the absence of identical designs implemented as both asynchronous and clocked circuits, it is difficult to compare the power performances of both implementation techniques.

An argument advanced *against* asynchronous design with respect to power consumption is the use of special data encoding techniques, like for instance dual-rail, which require that a larger number of data wires switch for each data transmission than with usual clocked datapaths. In standard four-phase dual-rail encoding, which is the data transmission scheme used in the microprocessor, for each transmission of an n-bit word, n wires change voltages twice.

However, we believe that the following factors contribute to a lower power consumption in asynchronous circuits. First, the absence of a clock circuitry removes the main power sink. (It has been said that half of the power consumed in the DEC Alpha is dissipated by the clock.) Secondly, because of the *reactive* nature of asynchronous circuits, no power is drawn by a part of a circuit when that part is not used. Thirdly, because of the requirement that signals should change monotonically, no voltage oscillations are allowed, hence eliminating spurious switching of gates.

4 Programming in VLSI

Programming in VLSI requires overcoming prejudices from both hardware and software designers. VLSI designers experience the greatest difficulties with the idea that their circuits can be conceived entirely as programs. Computer scientists designing circuits as programs have the tendency to carry over to VLSI programs the cost assumptions that are valid for stored-program implementations but not for a direct hardware implementation.

The main difference between software programming and VLSI programming is that in VLSI, concurrency is free and sequencing is costly. Concurrency is implemented by mere juxtaposition of circuits. Sequencing requires synchronization. We therefore avoid sequencing as much as possible, and implement it as a restricted form of concurrency.

The program notation we use (called CHP, for Communicating Hardware Processes) is not a hardware description language. It is inspired by C.A.R. Hoare's CSP[3] and E.W. Dijkstra's guarded commands[2]. Compared to CSP, CHP contains both restrictions and extensions. The restrictions are required by the limitations of hardware to boolean logic, and by the impossibility to create resources during execution of the computation. "Dynamic" objects, like certain data types and general recursion are excluded.

The only basic data type is the boolean. An integer is a collection of booleans. An "integer of length n" is a predefined record type consisting of n boolean components. Any operation on a data type other than boolean is a shorthand notation or function call for the sequence of operations on boolean variables that will implement it.

All additions to CSP are motivated by efficiency. Because of the extensive use of concurrency and communication, we have refined the communication mechanism with the probe[5], (which allows complete symmetry between input and output), multiple channels (buses), and other direct manipulations of ports, like the assignment of an input to an output: $R!(L?)$.

Also motivated by efficiency are the availability of a restricted form of shared variables, and of both deterministic and non-deterministic choices. A variable can be written by one process and read by several other processes. As we design at several levels of refinements, shared variables are introduced during the course of the transformations—for instance during process decomposition. We have also found cases when shared variables were useful at the communicating processes level.

It is very difficult, if at all possible, to determine at "compile-time" which choices require arbitration. Since arbitration is expensive, we introduce two sets of control structures, a deterministic set and a non-deterministic set, and let the programmer explicitly indicate where arbitration is needed.

Justifying the adequacy of this notation for circuit construction would require at least to show a series of convincing examples. Due to space limitation, we refer the reader to the literature, in particular the papers describing the processor. See [7], [8], [9], [11], [12], [14].

5 Program Transformations

CHP is ideally suited for designing the control and synchronization parts of a computation. But, usually data manipulation and arithmetic are represented as integer operations in CHP, and therefore an important refinement to the solution consists of replacing the integer operations with their implementations as boolean operations . Rather than going in one step from program to circuit, the designer applies a series of transformations to the original CHP program. At each step, some part of the algorithm is refined and some algebraic transformations can be applied leading to important optimizations.

The general justification of this approach is that the task of designing a correct VLSI system is much more manageable if one starts with a simple, abstract, solution the correctness of which is easy to establish. The solution is then refined by repeated applications of a set of transformations the correctness of which has been established once and for all.

We are using three types of transformations. A CHP-to-CHP transformation can be applied first to increase concurrency. This transformation is part of the high-level design more than the "compilation." An example is the derivation of the pipelined version of the processor from a sequential version.

A second CHP-to-CHP transformation, called *process decomposition*, is used to simplify the structure of the processes. It is a syntax-directed transformation that is applied repeatedly until the structure of each process is either a sequence of communication actions of the form: $*[C_0; C_1; ...]$ or "reactive process" of the form:

$$*[[G_1 \rightarrow S_1 \| \ldots \| G_n \rightarrow S_n]] \ .$$

The third type of transformations to be applied are the real "compilation" transformations: implementation of communication and arithmetic, and of sequencing.

6 The Object Code: Production Rules

The notation for the object code provides the weakest possible form of control structure and the smallest number of program constructs. In fact, it contains exactly one construct, the *production rule* (PR), and one control structure, the *production rule set*.

The production-rule notation is the canonical representation of a digital circuit. It can be decomposed into several equivalent networks of digital operators, depending on the set of building blocks used, or even depending on the technology (e.g., CMOS or GaAs) used, but the production-rule set represents the circuit independently of the chosen physical implementation.

A production rule (PR) is a construct of the form $G \mapsto S$, where S is a simple assignment, i.e. an assignment of the constant **true** or **false** to a boolean variable, and G is a boolean expression called the guard of the PR. For example, the *NAND*-gate with inputs x and y and output z has the production rules:

$$\begin{aligned} x \wedge y &\mapsto z\downarrow \\ \neg x \vee \neg y &\mapsto z\uparrow \end{aligned}$$

The semantics of a PR are defined only if the PR is *stable*: A PR $G \mapsto S$ is said to be stable in a given computation, if, at any point of the computation, G either is **false** or remains invariantly **true** until the completion of S. Stability is not guaranteed by the implementation. It has to be enforced by the synthesis procedure.

An execution of the stable PR $G \mapsto S$ is an unbounded sequence of *firings*. A firing of $G \mapsto S$ with G **true** amounts to the execution of S. A firing of $G \mapsto S$ with G **false** amounts to a **skip**.

A PR set is the concurrent composition of all PRs of the set. The only composition operation on two PR sets is the set union. The implementation of two concurrent processes is the set union of the two PR sets implementing the processes and of the PR sets implementing the channels between the processes, if any. PRs are *complementary* when they are of the type $G1 \mapsto x\uparrow$ and $G2 \mapsto x\downarrow$. We require that complementary PRs be *non-interfering*.

Two complementary PRs are non-interfering when $\neg G1 \vee \neg G2$ holds invariantly. It can be proven that, under the stability of each PR and non-interference among complementary PRs, the concurrent execution of the PRs of a set is equivalent to the following sequential execution:

$$*[\textit{select a PR with a true guard};\ \textit{fire the PR}]$$

where the selection is weakly fair (each PR is selected infinitely often).

Hence, any valid execution of a production-rule set in which non-interference and stability are fulfilled is equivalent to a non-deterministic sequential execution of the production-rule set. This equivalence facilitates the analysis of production-rule sets. It also establishes the connection between our definition of concurrency as set union and the more traditional definition based on interleaving of atomic actions. Observe that our semantic model does not require the notion of atomic actions.

6.1 A Simple Example of Program Transformation

A common form of a process is the so-called “one-place buffer”: $*[L?a; R!f(a)]$. The process receives a parameter a on input port L, and sends the result of the function evaluation $f(a)$ on the output port R. All pipeline stages in the microprocessor are variations on this basic theme. The first transformation is the process decomposition that separates the control part of the process—the part that implements the sequencing—from the datapath—the part that manipulates data. The control part is simply the skeleton process: $*[L; R]$.

The next transformation is called “handshaking expansion.” It replaces the bare communication actions with an implementation called “handshaking,” which is a synchronization protocol using two boolean variables for each port: li and lo for L, and ri and ro for R. The handshaking expansion gives:

$$*[[\neg li];\ lo\uparrow;\ [li];\ lo\downarrow;\ [ri];\ ro\uparrow;\ [\neg ri];\ ro\downarrow]\ .$$

In order to perform the next transformation, the “production-rule expansion,” we need to introduce a state variable x:

$$*[[\neg li];\ lo\uparrow;\ x\uparrow;\ [x];\ [li];\ lo\downarrow;\ [ri];\ ro\uparrow;\ x\downarrow;\ [\neg x];\ [\neg ri];\ ro\downarrow]\ .$$

The production rule expansion is:

$$\begin{aligned}
\neg x \wedge \neg li \wedge \neg ro &\mapsto lo\uparrow\\
lo &\mapsto x\uparrow\\
x \wedge li &\mapsto lo\downarrow\\
x \wedge \neg lo \wedge ri &\mapsto ro\uparrow\\
ro &\mapsto x\downarrow\\
\neg x \wedge \neg ri &\mapsto ro\downarrow
\end{aligned}$$

This production-rule set can be directly implemented in hardware. Observe that none of the binary operators represented by a pair of production rules setting and resetting the same variable, is a standard gate. The method is particularly efficient precisely because one is not required to map an implementation onto a particular set of standard gates. It is also a true synthesis method: The circuits are derived by pure symbolic manipulation without any preconception of the result.

7 Asynchronous vs. Synchronous Designs

The tradeoffs between asynchronous and synchronous designs can be described in terms of the advantages of ignorance versus the advantages of knowledge.

An asynchronous implementation—or more precisely, a delay-insensitive one—ignores all information about timing. Conversely, a synchronous implementation exploits all available knowledge about timing. The advantage of ignorance is that the implementation has to be correct independently of timing, therefore gaining qualities of robustness to variations of physical parameters.

The price paid for ignorance is that the completions of all actions have to be detected (computed) locally. The so-called "completion detection" mechanism is the most costly asynchronous technique. For instance, detecting that a value has been written in a one bit-register requires 6 transistors in CMOS. Hence, an entirely delay-insensitive static RAM in CMOS will have an overhead of 6 transistors per bit compared to an implementation in which delays are (assumed to be) known.

The reward for knowledge is efficiency: If the duration of all actions is known precisely, sequencing of actions can be implemented efficiently with a global clock, since a single clock signal is enough to signify the end of a computation step and the start of the next one. The price for knowledge is paid in several ways. First, knowledge of timing relies on knowledge of the physical parameters of the design, and therefore creates an obligation to comply with the assumed values that limits the robustness to variations of these parameters.

An even higher price paid for knowledge is...doubt! Designers are aware that their knowledge of both the physical properties of the devices and the runtime behavior of the circuits is imperfect. Consequently, they have to lengthen the clock period to take into account an error margin in the evaluation of the duration of a computation step. This error margin is becoming prohibitive as technology advances.

Several sources of errors have to be accounted for. Miniaturization, "scaling-down," of the devices introduces more and more variations in the geometry of the devices and their processing, which cause variations in their electrical parameters. But more importantly, the duration of a computation step may vary significantly with the values of the data. For instance, the addition of two integer numbers using a ripple-carry adder varies in time with the length of the carry-chain. The clock period of a synchronous implementation has to be adjusted for the worst case, and thus a synchronous ripple-carry adder takes a time proportional to the number of bits of the operands. On the other hand, an asynchronous ripple-carry adder takes a time on the average proportional to the logarithm of the number of bits[12].

Global knowledge requires global information in the form of a clock signal. Distributing a clock signal across a chip with the requirement that the signal arrive at the different locations on the chip "at the same time" is becoming more and more problematic as the size and the speed of the chips increase. The skew of the clock signal across the chip needs to be absorbed by an added delay in the clock period. But more simply it will soon become very difficult, if not impossible, to distribute a clock signal across a chip and meet the timing requirements of modern technology like Gallium Arsenide, or superconducting devices.

Let us summarize the trade-offs. In an asynchronous implementation, a time penalty t_a is paid for generating the completion signals ("completion detection") and for the handshaking mechanism in the control. In a synchronous implementation a time penalty t_s is paid for the inaccuracy of clock signal distribution ("clock skew") and for the variations due to fabrication defects (geometry defect, ion implantation, etc.). We claim that at the moment, t_a and t_s are about equal for CMOS designs.

With these two drawbacks cancelling each other, we are left with the following alternatives. For complex computations with data dependencies, asynchronous design has the advantage of exploiting the best-case delay, whereas synchronous solutions have to adjust to the worst case. For small and "data-regular" designs, synchronous solutions have the advantage of both speed and size.

But in the long run, it is at the level of a complete system design that an asynchronous and concurrent computing approach will win.

8 Asynchronous Systems

The extent of the influence of the clock mechanism on the architecture of computing systems is more pervasive than designers realize.

The choice of an instruction set is severely restricted by the requirement that the execution time of an instruction be equal to a multiple of the clock period. One may wonder how relevant the "CISC" vs. "RISC" debate would be if the duration of an instruction execution were entirely flexible. It is quite possible to envision a choice of instruction set in which the most frequently used instructions are very short but yet some instructions are included whose execution time may be long. Such an instruction set would most likely mix the best attributes of both "RISC" and "CISC" types of instructions.

We have learned from the designs of the different versions of the microprocessor that the most significant optimizations are done at the highest level, i.e. at the level of the concurrent computation description of the circuit.

By removing the tight lockstep constraint imposed by the clock upon the concurrent activity inside a chip or an ensemble of chips, the method allows the designer to exploit concurrency to an extent difficult to achieve with synchronous techniques. A whole array of communication and synchronization techniques from the field of concurrent computation and parallel algorithms are available which open up completely new architectural possibilities.

It has been said that all large systems that work have evolved from small systems that worked. By providing an interface between components of a system that is independent of the physical properties of the different components, asynchronous techniques make it possible to refine and "evolve" a family of designs without starting each new version from scratch. A small example from the design of the microprocessor may illustrate this point. A week or so before we sent the second version to fabrication, we decided to replace the ALU process with a two-process version. The purpose was to pipeline the execution of an ALU instruction and the storing of the result. We were able to do the replacement and include the necessary changes in the other process involved without modifying the rest of the design.

Another, we hope, even more spectacular, example is the design of a GaAs version of the microprocessor that we have completed a few months ago. We wanted to show that the almost perfect interface between logical design and physical implementation that the method provides makes it possible to port a design from one technology (CMOS) to another, very different, one (GaAs) with practically no design changes. In about half a year, we were able to design an entirely new logic family (the standard implementation of a set of operators) in GaAs, and map the set of production rules defining the processor into a network of GaAs transistors. Our first choice of logic family was extremely conservative (we were concerned about noise immunity), and as a result the first version was functional but ran at only 70MIPS and consumed 4W. We are now redesigning the chips with another choice of logic family. We expect this version to run at 200MIPS and consume 2W.

More than anything else, the modularity of this approach will revolutionize hardware design by drastically reducing the design time and simplifying the interfaces.

The advantage of a simple interface can be exploited also at the level of system architecture. Future advances in computer technology will no longer be due entirely to the speed increase of the semiconductors, but also, and mainly, to the ability to assemble very large collections of chips. Parallel supercomputers all exploit this principle. Unless the interface mechanism between the chips is flexible enough to accommodate a diversity of designs due to different fabrications or to different solutions, the reliability of such architectures is very questionable. My experience with the design of the AMETEK multicomputer [15] provides an illustration of this principle. I was able to convince the designers that (at least) the mesh routing network should be asynchronous. Consequently, it was possible, later in the development of the system, to mix two different families of routing chips—different designs and speed but same interface—inside the same routing network without any noticeable effect.

9 Conclusion

The results already achieved indicate that the type of approach to VLSI design we have described is very promising. Based on these results, we will venture a vision of the future that we hope is only slightly optimistic.

Uniform notations and methods will be used across the hardware/software boundary. Concurrent computation paradigms will replace traditional ones (switching and automata theory) in the design of VLSI systems. Logic verification issues will become critical; verification tools and methods will be widely used. Correctness by construction and high-level synthesis will become routine. Asynchronous circuits will be the preferred implementation both for methodological reasons and for efficiency reasons (power consumption, robustness, speed). For large systems, designs using synthesis tools will outperform hand designs both in reliability and performance. However, these new methods will require a new generation of designers.

Acknowledgements

The research described in this paper would not have been possible without the contributions of my present and recent students, Dražen Borković, Steve Burns (now at the University of Washington), Marcel van der Goot, Pieter Hazewindus, Tony Lee, Christian Nielsen, and José Tierno. The insights, help, and encouragement of my Caltech colleague Charles L. Seitz are deeply appreciated. The research was sponsored by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745.

References

[1] Steven M. Burns and Alain J. Martin. Syntax-directed Translation of Concurrent Programs into Self-timed Circuits. *Proc. Fifth MIT Conference on Advanced Research in VLSI*, ed. J. Allen and F. Leighton, MIT Press, 35-40, 1988.

[2] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs NJ, 1976.

[3] C.A.R. Hoare. Communicating Sequential Processes. *Comm. ACM* 21,8, 666-677, 1978.

[4] David L. Johannsen. Silicon Compilation. *Decennial Caltech Conference on VLSI*, ed. C.L. Seitz, MIT Press, 17-36, 1989.

[5] Alain J. Martin. The Probe: An Addition to Communication Primitives. *Information Processing letters* 20, pp 125-130, 1985.

[6] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, P.J. Hazewindus. The Design of an Asynchronous Microprocessor. *Decennial Caltech Conference on VLSI*, ed. C.L. Seitz, MIT Press, 351-273, 1989.

[7] Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, 1,(4), 1986.

[8] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. *UT Year of Programming Institute on Concurrent Programming*, ed. C.A.R. Hoare, Addison-Wesley, Reading MA, 1989.

[9] Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. *Formal Methods for VLSI Design*, ed. J. Staunstrup, North-Holland, 1990.

[10] Alain J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. *Sixth MIT Conference on Advanced Research in VLSI*, ed. W.J. Dally, MIT Press, 1990.

[11] Alain J. Martin and Pieter J. Hazewindus. Testing Delay-Insensitive Circuits. *Proc. 1991 University of Santa Cruz Conference on Advanced Research in VLSI*, ed. Carlo H. Séquin, MIT Press, 118-132, 1991.

[12] Alain J. Martin. Asynchronous Datapaths and the Design of an Asynchronous Adder. *Formal Methods in System Design*, 1:1, Kluwer, 117-137, 1992.

[13] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*, Addison-Wesley, Reading MA, 1980.

[14] Christian D. Nielsen and Alain J. Martin. A Delay-Insensitive Multiply-Accumulate Unit. Caltech Technical Report CS-TR-92-03, Computer Science Department, California Institute of Technology, 1992.

[15] Seitz, C.L., Athas, W.C., Flaig, C.M., Martin, A.J., Seizovic, J., Steele, C.S., and Su, W.-K. The Architecture and Programming of the Ametek Series 2010 Multicomputer. *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, ACM Press, New York, 1988.