

25 YEARS AGO: THE FIRST ASYNCHRONOUS MICROPROCESSOR

Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125, USA

January 27, 2014

Twenty-five years ago, in December 1988, my research group at Caltech submitted the world's first asynchronous ("clockless") microprocessor design for fabrication to MOSIS. We received the chips in early 1989; testing started in February 1989. The chips were found fully functional on first silicon. The results were presented at the Decennial Caltech VLSI Conference in March of the same year. The first entirely asynchronous microprocessor had been designed and successfully fabricated. As the technology finally reaches industry, and with the benefit of a quarter-century hindsight, here is a recollection of this landmark project.

1 No, You Can't

A year earlier, I had approached our DARPA program manager asking him to support my plan to design and fabricate an asynchronous processor. His immediate answer was to reject the request as ludicrous. "This has been tried before: it doesn't work," he informed me. After I insisted, pointing to several modest experiments that showed promise, he finally said (I quote

from memory): "OK, Alain. This is a bad idea, but you are a good guy. I will let you do it. You will try and fail. And you will be cured of this nonsense!"

DARPA program managers always do the right thing in the end, sometimes for the wrong reason. To his defense, I have to admit that his skepticism was unanimously shared by all experts of the time.

2 Can We?

As a computer scientist, my interest in VLSI was triggered by Mead & Conway's *Introduction to VLSI Systems*. Their revolutionary text established the link between VLSI design and distributed computing. A new field of computing was born, in which computations were implemented directly in a physical medium without the interface of a computer. Another revelation of the text was that shrinking of the devices' physical dimensions would continue unabated for a long time, allowing larger and larger systems to be built, with the corollary that mastering the systems' complexity would be the main issue.

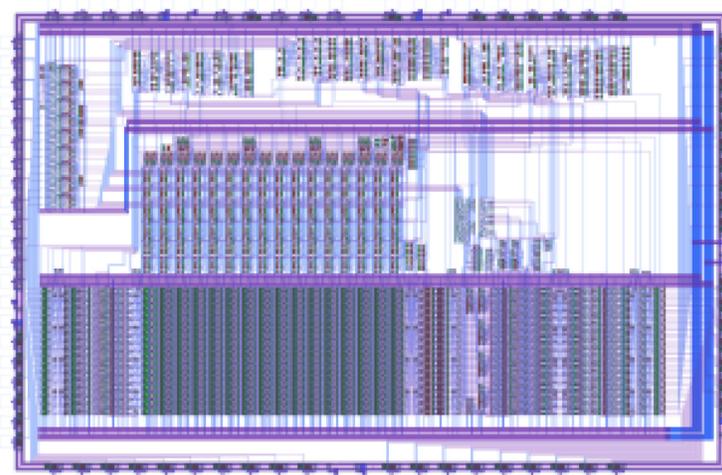


Figure 1: Layout of the Caltech Asynchronous Microprocessor

Yet another revelation of the book was Chuck Seitz’s chapter on “System Timing,” where asynchronous logic, called “self-timed” in the text, was described as a potentially superior approach owing to its independence of timing in the face of expected increasing timing variations.

Those were compelling arguments for a computer scientist, and I decided to apply my experience in distributed computing to the synthesis of asynchronous VLSI systems. In blissful ignorance of the difficulty ahead, I chose as a first exercise the asynchronous implementation of a distributed mutual-exclusion algorithm I had invented. At the end of a painful and solitary effort, I found myself not only with a good circuit but also with the prototype of a synthesis method. The result was presented at the 1985 MIT VLSI conference.

3 A New Synthesis Method

Unlike anything else being tried at the time, the approach relied on three levels of digital repre-

sentation before the transistor level: (1) a high-level description in terms of a novel message-passing programming language (CHP)¹; (2) an intermediate representation, *the handshaking expansion* (HSE) derived from CHP by replacing variables by their boolean expansions; arithmetic and boolean operations on those variables were replaced by boolean operations, and communications by *handshake protocols*; (3) a low-level notation which contains only one operation, *the production rule*, and one composition mechanism, *the production-rule set* (PRS). All production rules of a set are executed in parallel, the required sequencing being enforced by conditional executions. Conveniently, a production rule has an almost direct implementation as a “pull-up” or “pull-down” circuit in CMOS.

A priori, it looked preposterous to ask the designer to work with three languages when many of them at the time had trouble with a single hardware description language, but the rewards

¹CHP was originally inspired by C.A.R.Hoare’s CSP. But the two languages are vastly different.

were worth the effort. Each level of representation permitted a class of transformations unique to that level. At the CHP level, the original specification of the system could be decomposed into a collection of components (called “processes” in CHP) communicating by messages. The goal was both to simplify each component and to introduce concurrency. As fine-grain communication would require more efficient use of a channel beyond a simple connecting pipe, CHP was the first language where channels and ports were “first-class” components, with new constructs such as the *probe* and *value probe* allowing to perform some computations directly on ports.

At the HSE level, the main issue was the overhead of handshake protocols. I wanted to manipulate the handshake sequences so as to reduce this overhead. The HSE representation lent itself to another class of transformations, called *reshufflings*. A legitimate reshuffling would reorder the sequence of boolean transitions and waits without compromising the logic or introducing deadlock. It is through the analysis of all legitimate reshufflings that we discovered the most efficient solutions for control circuits.

At the PRS level, not only, as we already mentioned, could a production-rule set be directly implemented as a CMOS circuit, but the notation also lent itself to a simple and efficient event-driven simulator, *prsim*.

A systematic application of the synthesis method quickly led to the discovery of new building blocks for asynchronous systems, discoveries which I think would have been impossible by a “seat-of-the-pants” approach. With several students joining my lab, learning the method, and applying it to various examples, a complete set of basic components was developed including control circuits, adders, and registers.

4 QDI Logic

A new class of asynchronous circuits emerged from our experiments, that we called *quasi delay-insensitive* or QDI. At first, we (other researchers and I) were intent on designing digital circuits not only without clock but even without any reliance on delays. Such circuits were called *delay insensitive* or DI. But all circuits we designed as DI contained some delay assumptions that were justified as “optimizations”. After a while, it dawned on me that we were fooling ourselves: we simply could not produce solutions that were really DI. Instead, I proved that the class of DI circuits was very limited and that most circuits of interest fell outside this class. The issue related to *forks* in the circuit, where a fork is a circuit node that is an input of more than one gate. For anything but very simple functions, at least one timing assumption was required: certain forks relied on some assumption about the delays in the branches of the fork. I called them *isochronic forks*. Isolating the timing issues in asynchronous logic to the isochronic fork was a great step forward in understanding the limits of asynchrony, and it took a while for the community to absorb the new concept. (We recently weakened the isochronicity requirement somewhat.²)

That became the QDI approach we settled on: the circuits could tolerate any amount of delays in operators and wires except in isochronic forks. The delay assumption needed for the correct behavior of an isochronic fork is a one-sided in-

²The original isochronicity condition bounds the difference of delays on the two branches of the fork. It is sufficient but not necessary. A weaker necessary and sufficient condition states that the delay along a specific path of transitions, called “the adversary path,” is longer than the delay in a branch of the fork.

equality between delays that can always be satisfied by making one of the paths longer.

At that time, it was customary for the few practitioners of asynchronous-logic design to make liberal use of delay elements. The QDI discipline essentially forbade the practice: the completion of a function execution (when it needed to be known) had to be explicitly calculated: each communication was implemented with *four-phase, return-to-zero*, handshake protocol³, and each data word was coded with a *dual-rail* or *one-of-n code* (or *one-hot* code). Altogether, the overhead compared to a few delay elements here and there seemed exorbitant, and I was soon branded as a dangerous fanatic... It is for me a vindication to observe that the whole community today is quietly converging toward some form of QDI as people realize that it is the approach most likely to deliver functioning, robust, non-trivial, clockless circuits.

By the mid 1980s we had developed a complete and robust approach for QDI circuit synthesis. A component of the method was the systematic separation of the control part and the datapath of a design directly from its high-level CHP specification. The datapath contained QDI implementations of registers and standard arithmetic and boolean functions—adders, comparators, incrementors, etc. The control part contained the control signals that enforced the sequencing of actions between the various components of a datapath. Basic circuit building blocks had been invented, in particular the *D-element*

³Objection to the use of a four-phase, return-to-zero, protocol continues to this day: Why insist on a 4-transition protocol when a non-return-to-zero, 2-transition protocol exists? The answer is that it is very costly to do arithmetic and logical operations when one has to alternate the sense of the signals according to the parity of the communication.

for sequencing two bare handshake protocols, the *write-acknowledge* for detecting the completion of a write to a boolean register, the *completion tree*, dual-rail ripple-carry adders, *channel arbiters*, etc.

By 1987, the approach had been successfully tested on a number of non-trivial examples that had been actually implemented through MOSIS, and we were ready to tackle the design of a small microprocessor.

5 Let's Try!

According to my lab book, the project started in earnest in July 1988. Beside myself, the team was strong of four graduate students: Steve Burns, who would take a leading role in the project, Drazen Borkovic, Tony Lee, and Pieter Hazewindus. Pieter joined the project later in the fall. The students had convinced me that we should design our own instruction set, which they did earlier on. It was a small ISA, typical of a microcontroller of today, without specific IO instructions.

The processor had a 16-bit, RISC-type instruction set, 16 registers, four buses, an ALU, and two adders—one for program-counter calculations and one for memory addresses. Instruction and data memories were separate. The chip size turned out to be approximately 20,000 transistors.

The originally targeted technology was HP 2.0- μm SCMOS, a two-metal-layer technology with N-well, 5V nominal Vdd, and 1V threshold voltage. We ended up fabricating two versions, the CAM20 in 2.0- μm SCMOS, and the CAM16 in 1.6- μm SCMOS. The 1.6- μm HP CMOS 40 technology was also a two-metal-layer N-well scalable CMOS technology with 5V nominal Vdd and 0.75V threshold voltages.

A few days after the CAM20 had been submitted for fabrication, as the team was resting a little, Tony Lee was looking contentedly at his layout when he discovered with horror that he had forgotten to connect some wells to the power rail. Convinced that the chip would fail, we checked the MOSIS run schedule and saw that a run in 1.6- μm was coming up in a few weeks. As the technology was truly scalable in those palmy days, we could fix the error, make a few changes and submit the CAM16 for fabrication in time. Hence the two versions.

But let's go back to the beginning of the summer... The design proceeded on two fronts. One was the top-down synthesis of the processor, starting from a simple CHP program executing instructions sequentially. (This program was then used as the specification from which the final concurrent CHP version was derived.) The other was the design of the datapath. Thanks to the aforementioned separation of control and datapath in the synthesis, we could design the datapath early before choosing the pipeline structure and before the control part was fixed. The only important decision we had to make early was the number of buses that would access the register file. In order to maximize concurrent access to the registers, and thus between instruction executions, we introduced four buses; in retrospect, the number of buses was probably an overkill.

By the end of the summer, the datapath was laid out and tested. (Layout of the datapath was done manually using Magic.⁴) However, we would keep modifying the control until a few

⁴From the website: "Magic is a venerable VLSI layout tool, written in the 1980s at Berkeley by John Ousterhout... Due largely to its liberal Berkeley open-source license, Magic has remained popular with universities and small companies."

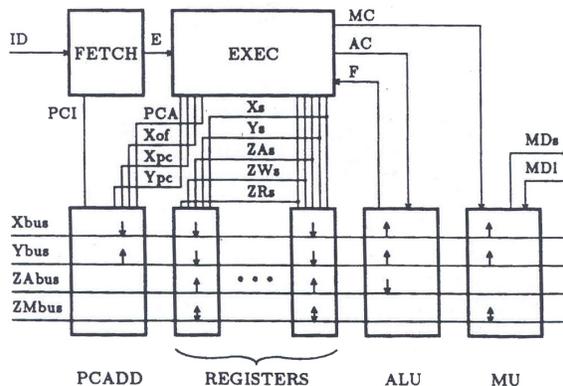


Figure 2: The CHP processes and communication channels of the CAM

weeks before tape out. The CHP processes and their communication channels are shown in Figure 2. (What is called the EXEC in the figure is in fact the Decode process.) The datapath consisted of a register file (16 16-bit registers), a PCADD unit to update the program counter, a memory unit to transfer data between the registers and the data memory, and an ALU. The ALU was used for arithmetic and logical operations: By manipulation of the KPG codes, any logical instruction was turned into an addition.

5.1 A New Pipeline Architecture

Good ideas in design often have an *a posteriori* simplicity and obviousness that obfuscate the fact that arriving there was anything but obvious. That was the case for the general pipeline structure that we converged on after multiple iterations, as testified by my lab notes.

What emerged from the exercise was a simple and general architecture that would become, *mutatis mutandis*, the archetype of future asynchronous processor pipelines. The pipeline had two main connected parts: a *fetch loop* and an

execution loop. The fetch loop was a ring of processes consisting of the Fetch process computing the next program counter (pc) and sending it to the instruction memory IMEM; the instruction memory receiving a pc and sending the corresponding instruction to the Decode; the Decode decoding the instruction and sending information about the next instruction to the Fetch.

The Decode was also part of the execution loop. As such, it would concurrently send pieces of a decoded instruction to the register file (the address(es) of the register(s) involved) and the instruction opcode to the proper execution unit (ALU or memory unit). The execution unit would typically receive the parameters of the instruction from the register file and send the result back to the register file. Execution units and the register file would exchange parameters and result via the buses.⁵

Unlike in a synchronous architecture, the execution units were not aligned on the pipeline, and therefore out-of-order execution was automatic, provided the use of the registers allowed it.

The architecture also exposed the main performance bottlenecks in an asynchronous processor pipeline. The first was the fetch loop: one data “token” was circulating around the loop and therefore the latency of the token around the fetch loop was a critical factor in the throughput of the whole system. Two *critical cycles* were identified as limiting the throughput: the fetch loop and the cycle including sending parameters to an execution unit and returning the result to the register file, with the fetch loop usually the most critical. Another difficulty was the reser-

vation of the registers for an instruction in view of the concurrent execution of several instructions. Preventing read/write hazards on register access was the only restriction to the concurrent use of registers by execution units. However, in the absence of a global time reference, it became a tricky synchronization problem. This issue will remain critical in all subsequent designs with several different solutions being proposed.

Later asynchronous microprocessors with more complex instruction sets, like the Caltech MiniMIPS, a QDI clone of MIPS R3000 microprocessor, will have considerably more involved pipelines. But the basic structure introduced in the CAM will obtain.

5.2 CAD Tools

Our use of CAD tools was very modest. We did use Magic for hand layout and SPICE for electrical simulation, but no other commercial tool. Steve Burns, with the help of Pieter Hazewindus and Andrew Fyfe, wrote an excellent place-and-route tool, *VGladys*, that would take production rules as input, match them with standard operators for which we had built a standard cell library, and place and route the cells. It was very useful for the control part which went through many last-minute modifications. Another useful tool was the production-rule simulator *prsim* which allowed excellent low-level digital simulation, and gave an estimate of the critical cycles.

What about high-level synthesis? Based on my early high-level synthesis experiments, Steve Burns had written a rather sophisticated syntax-directed compiler to generate gate-level implementation of CHP programs; and our original intention was to use it for the CAM. But the first attempts indicated that the performance of the generated circuit would not be sufficient to

⁵In the CAM, a bus was a many-to-many communication channel. We abandoned this approach in later systems in favor of an implementation as a network of split and merge components with only point-to-point channels.

make a convincing demonstration of QDI logic. We abandoned syntax-directed translation and never used it much afterward.

6 Yes, We Can!

Testing started on February 23rd 1989. Only two of the 12 CAM20 chips passed all tests, probably due to the floating wells, but they turned out to be the most robust. Out of the 50 CAM16 chips, 34 were found to be entirely functional. The chips were tested in various settings: over the widest range of operating voltages, at different temperatures, in particular in liquid nitrogen, and with a variety of test programs.

One interesting experiment was using a hair dryer to heat up the chip. As the temperature rose, one could see the oscillation period stretch on the oscilloscope, and then shrink again as the dryer was removed.

6.1 Testing for Stuck-at Faults

It has been claimed that “self-timed is self-testing.” We debunked that claim. In an asynchronous system, in absence of a global time reference, the knowledge that a transition has completed can only be derived from its causing another transition to take place. We say that the second transition *acknowledges* the first one. If all transitions are acknowledged, then a transition being “stuck” will cause the system to deadlock, and the stuck-at fault will be detected, by running normal computations. This is the idea behind self-testing. But only in an entirely DI system are all transitions acknowledged. In all other cases, some transitions are not. And therefore not all stuck-at faults lead to deadlock. In a QDI system, it is the isochronic fork that causes certain transitions to be unacknowledged.

Detecting all stuck-at faults requires adding extra circuitry, which we decided not to do in this experimental design. Instead, we used a program written by Pieter Hazewindus that could test the entire chip for detectable stuck-at faults in less than 700 instructions.

6.2 Performance

The results reported here are for the chips running a sequence of ADD instructions. Since the performance does not suffer from the branch delays, we consider it the peak performance.

Performance can be summarized as follows: at 2V, the CAM16 runs at 5MIPS drawing 5.2mA of current; at nominal 5V, it runs at 18MIPS drawing 45mA, and at 10V it runs at 26MIPS drawing 105mA. The CAM20 runs at 11MIPS at 5V, and reaches 15MIPS at 7V. The self-regulating power supply of asynchronous circuits allowed us to operate the processor from just about any power source that is capable of providing approximately $50\mu\text{W}$ of power at 0.8V. The CAM16 performance over the whole voltage range is shown in Figure 3.

Once again in total ignorance of the expected difficulties, we undertook to test our chips across the widest possible range of operating voltage, starting from **deep subthreshold** (around 0.3V). The chips would stop functioning around 0.3V, but starting from 0.4V for the CAM20 and 0.5V for the CAM16, most chips would work flawlessly.

Figure 4 is a photograph of a page from our lab book showing the first performance measurements of a CAM20 chip in subthreshold. The frequency goes from 3Hz at 0.4V to 680KHz at 1.1V.

The chips were also tested at liquid nitrogen temperature (77K) by pouring liquid nitrogen in

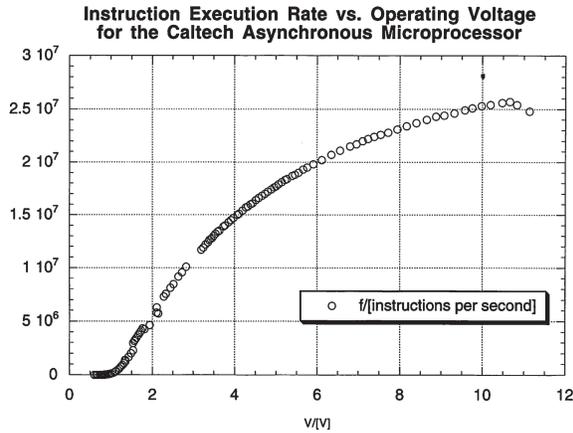


Figure 3: Frequency as a function of voltage for the CAM16

a styrofoam cup fixed on top of the die. The experiment was a little tricky, but we were able to measure the performance. The CAM20 version reached 20MIPS at 5V, and 30MIPS at 12V. The CAM16 version reached 30MIPS at 5V. Figure 5 shows all measurements of frequency against voltage. The solid line in the figure represents the CAM20 performance in liquid nitrogen. Observe the significant jump in performance compared to the bottom line representing the room temperature frequencies. Also worth noticing is the upward shift of the threshold voltage at low temperature.

Figure 6 shows the energy per instruction for a CAM16 chip across the whole voltage range from 0.5V up to 12V with a minimum energy point around 1V, i.e. above threshold voltage.

The robustness of the chip to voltage variations, and its ability to operate at very low voltage invited Mika Nyström to try what he called the first *Potato-Chip Experiment*: He used a potato as power supply! The Potato Processor runs at around 300KHz at 0.75V. (See Figure 7.)

2/24/89

Speed Test : Instruction = Add R0,R0,R0 (D000)

Chip #2	Voltage	Frequency
	0.427 V	
	0.440 V	
	0.426 V	2.95 Hz
	0.449 V	59.5 Hz
	0.461 V	84.6 84.6 Hz
	0.472 V	109.2 Hz
	0.483 V	141.0 Hz
	0.494 V	182.5 Hz
	0.506 V	236.1 Hz
	0.517 V	305.8 Hz
	0.529 V	402.9 Hz
	0.540 V	514.2 Hz
	0.574 V	1.110 KHz
	0.608 V	2.355 KHz
	0.665 V	7.715 KHz
	0.688 V	12.19 KHz
	0.733 V	27.69 KHz
	0.710 V	18.60 KHz
	0.744 V	33.41 KHz
	0.800 V	74.69 KHz
	0.902 V	207.7 KHz
	1.034 V	482.6 KHz
	1.112 V	682.0 KHz

<Continue Next Page>

Figure 4: First measurements of frequency in deep subthreshold voltage for a CAM20 chip, taken by Steve Burns on February 24th, 1989

To give the reader an idea of where the state of the art (in clocked VLSI) stood at the time, I consulted the *Berkeley Hardware Prototypes* site, where Berkeley had the excellent idea of archiving their chip designs. In 1988, the **SPUR** chip set, containing a RISC microprocessor, was fabricated in a similar (or perhaps identical) technology, 1.6- μ m CMOS. It operated at 10MHz. In 1990, the **VLSI-BAM**, a RISC microprocessor fabricated in 1.3- μ m CMOS ran at 20MHz and consumed 1W at 5V. Our, admittedly more

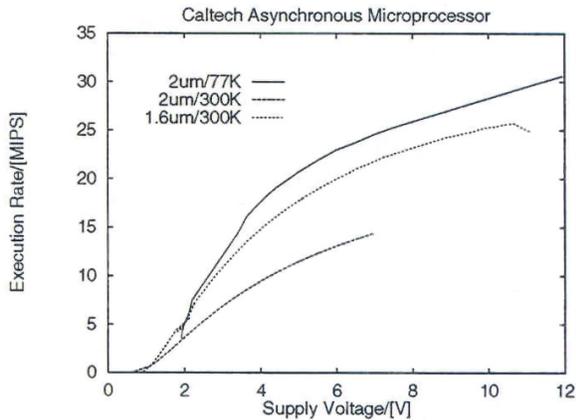


Figure 5: All frequency measurements for CAM16 and CAM20 at room temperature and CAM20 at liquid nitrogen temperature

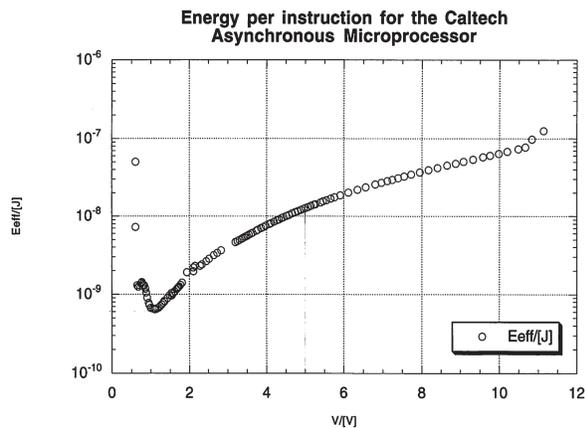


Figure 6: Energy per instruction for CAM16 across the whole voltage range from 0.5V up to 12V with a minimum energy point around 1V

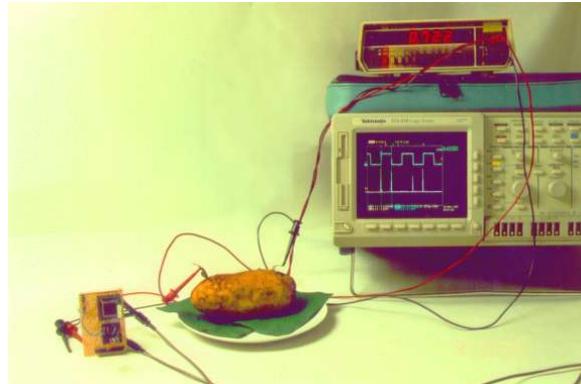


Figure 7: The potato-chip experiment: the CAM runs with a potato as power supply

modest, microprocessor ran at 18MHz at 5V and consumed less than a quarter of a watt!

6.3 To Be or Not To Be QDI

For the sake of full disclosure, let me add that not all circuits in the CAM were entirely QDI. First, the external memory used as instruction memory was not asynchronous. This required that the acknowledge signal from the memory back to the processor be simulated with an external feedback path including an adjustable delay element matching the latency of an instruction read. Second, each bus implementation contained a large distributed nor-gate. The multi-transistor pull-up was replaced with a single transistor controlled by a signal that required a timing assumption. Finally, the write circuitry for the register file was not QDI.

But all subsequent designs at Caltech were entirely QDI. They were all successful in terms of first-silicon functionality and performance.

A few misconceptions about the CAM found their way into the literature. One relates to

isochronic forks. During testing, some of the chips did behave incorrectly at low voltage. After a long and puzzling research, we concluded that the cause ought to be a failing isochronic fork, a conclusion that I reported in a paper. This bad news about isochronic fork was immediately propagated. But meanwhile, Steve Burns had taken a faculty position at the University of Washington, where he repeated the CAM measurements with a better test board. And the misbehavior disappeared. The culprit was our test board.

As of today, no malfunction caused by an isochronic fork has ever been observed in any of our fabricated chips, including a recent microcontroller in 40nm CMOS⁶.

Another comment that appeared in the literature is that the CAM was just a toy example with hardly any pipelining. Whereas the instruction set was indeed small and simple, the pipeline structure was quite sophisticated and allowed up to 4 instructions in flight concurrently.

6.4 Reject!

In the fall of 1988, even before the chip had been fabricated, I took the gamble of submitting a paper about the CAM to the forthcoming Decennial Caltech VLSI conference in March 1989. The reviewers' comments were scathing and recommended flat rejection. Fortunately, Chuck Seitz was the chair, and he was able to convince the program committee to accept the paper. By the time of the conference, the chips had been tested with the excellent results already mentioned. The presentation was a success, ending with a true ovation from the audience.

⁶However, given the high variability in this technology, in order to guarantee a negligible probability of failure, all adversary paths were made to contain at least 7 gates.

7 Can Improve

It was just a beginning. In spite of its robustness and flexibility, the control/datapath separation carried considerable overhead. It required that the input data of a component be stored into registers before being handed over to the output function. Combined with the four-phase protocol, this resulted in an incompressible critical cycle containing two completion-tree delays. We would soon introduce a fine-grain implementation where the output function could be computed directly from the data on the input ports, without requiring registers. This new approach was successfully used a decade later for the design of the Caltech MIPS R3000 microprocessor.

The CAM was ported to Gallium Arsenide in 1993 by José Tierno with the help of Drazen Borkovic and Tony Lee. **The GaAs version ran at 100MIPS.**

8 Why Can We?

Certainly we did the best we could at all levels of the synthesis to produce a good circuit, but nevertheless we did not really target high performance. For one thing, we hardly sized transistors. I think that the only sizing we did concerned the inverters driving the control signals to all bits of a datapath module. We were therefore very agreeably surprised by the results, in terms of frequency, energy, and robustness. And those excellent results have been confirmed in all following designs. So, the question arose and still stands, why such good results? The answer has several components.

At the circuit level, comparing synchronous and asynchronous technology boils down to arguing about knowledge versus ignorance. You remember the traditional saying by educators: "If

knowledge is expensive, try ignorance” with the implied conclusion that ignorance is even more expensive. But here we turn the slogan around and advice: “When knowledge becomes expensive, it’s time to try ignorance.”

Synchronous logic relies on the knowledge of delays. If all delays are known accurately, then the clock period can be set accurately with a minimum of added timing margin to absorb timing uncertainties (let’s call this “padding”). The resulting circuits can be very efficient. But the less accurate the knowledge about delays, the more padding is needed. As technology advances, parameter variation increases, and so does uncertainty about delays, requiring an increase in the padding—apparently up to 50% of the clock period in today’s technology. The cost of knowledge increases as technology advances.

On the other hand, the cost of ignorance (of delays), which is the cost of QDI logic, is high but remains practically constant. At some point in the evolution of the technology, the two cost figures are crossing, with the cost of knowledge exceeding the cost of ignorance. I believe that, as of 2013, we have passed this point. But this was not the case in 1988. Therefore, there are additional reasons to the good performance of the CAM. They have to be found in the efficiency of the synthesis method.

First, the use of an advanced message-passing language for high-level design provides a total liberty in introducing concurrency in the most efficient way. For instance, as we have mentioned, the execution units’ not being aligned allows natural out-of-order execution. All subtleties of concurrent computing can be put to bear without the restrictions imposed by a global synchronization. Second, the three distinct levels of synthesis give the designer the opportunity to apply different types of optimizations while

never losing the semantic equivalence between the various representations.

Another important reward of the approach is that the verification phase of a design is always minimal, with very few *a-posteriori* modifications (if any). **When applied systematically, it is truly a correctness-by-construction approach.**

9 People

The success of this project could not be explained without mentioning the quality of the students involved. I have been blessed with generations of excellent graduate students and with the unique Caltech environment of the 1980s in the wake of the Mead & Conway revolution. VLSI education at Caltech was based on a principle described by Carver Mead as the “thin tall man”⁷: a VLSI designer ought to know just enough (“thin”) at all levels (“tall”) of the process to be able to deal with all aspects of an integrated circuit conception from a high-level description down to layout.

I stretched the tall man upward: I added a solid education in concurrent computing, using the CHP language, and I taught QDI design. As a result, the same students could be talking, one moment, about deadlock and trace semantics, and the next moment about charge sharing and transistor sizing.

But there is more to first-rate research collaborators than good education. New uncharted endeavors require character as well. I will be forever grateful to my students for following me down this path in the face of widespread skepticism. The greatest satisfaction for me was to see

⁷In this context, “man” means a person of either gender.

them adopt the method I had developed, add to it, taking it into new directions, and finally mastering it better than I did!

10 Further Reading

Publications can be accessed from the group website at

<http://www.async.caltech.edu/publications.html>

Reference [2] is the original conference publication. Because the paper was submitted before the chips were tested, it doesn't contain the performance results. The results can be found in a small paper ([3]) that we published as soon as the tests were done. The GaAs version is described in [4]. A rather complete description of the QDI synthesis approach is in [5]. A definition of CHP can be found in [6]. Marcel van der Goot also contributed to the design of CHP.

Acknowledgments

I thank Mathieu Desbrun, Mika Nyström, Sean Keller, and Siddharth Bhargav for their comments on the manuscript.

References

- [1] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*. Reading, Mass.: Addison-Wesley, 1980.
- [2] Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The Design of an Asynchronous Microprocessor. In Charles L. Seitz, ed., *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, 351–373. Cambridge, Mass.: MIT Press, 1989.
- [3] Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The First Asynchronous Microprocessor: The Test Results. *Computer Architecture News*, 17(4),95–110, 1989.
- [4] José Tierno, Alain J. Martin, Drazen Borkovic, Tak-Kwan Lee. A 100MIPS Asynchronous Microprocessor in Gallium Arsenide. *IEEE Design & Test of Computers*, 11(2),43–49, 1994.
- [5] Alain J. Martin, Mika Nyström. Asynchronous Techniques for System-on-Chip Design. *Proceedings of the IEEE*, 94(6), 1089–1120, 2006.
- [6] Alain J. Martin, Christopher D. Moore, CHP and CHPsim: A Language and Simulator for Fine-Grain Distributed Computation. *Caltech Technical Report CS-TR-1-2011*, 2011.