A Case for Asynchronous Computer Architecture

Rajit Manohar

Abstract— **We present a summary of the state-of-the art in asynchronous VLSI and architecture. We present several reasons for adopting an asynchronous approach to computer architecture, including lower design complexity, reduced energy-complexity, and average-case performance. In particular, we describe why formal synthesis techniques for asynchronous design dramatically reduce design time, and present some examples that describe how asynchronous techniques can lead to a reduction in the energy-complexity of VLSI systems.**

Keywords— **asynchronous VLSI; formal synthesis; correctness by construction; low-energy architectures; average-case optimization; energy-complexity.**

I. Introduction

The past decade has seen tremendous progress in the design of computer systems. Just ten years ago any major computing task required expensive, custom hardware consisting of high-end processors connected via custom interconnects. Today, it is possible to network a cluster of PCs with commodity interconnect running a free operating system and use it for most major computing tasks. This dramatic improvement in both price and performance has been due to improvements in both processor design and understanding of interconnect architectures.

Today we are faced with new challenges in designing high-performance architectures. The power dissipation in modern processors is becoming a cause for concern. The Alpha 21164 dissipates about 50W of power operating at 3.3V [1], and the new Alpha 21264 will dissipate an estimated 72W of power at 2.0V [5]. Reduced power consumption is also becoming more important as we embed computing devices in portable electronic devices and appliances. Every bit of saved power extends the battery life for portable devices.

The time and staff-hours taken to validate a modern processor design project are also growing. In 1993, the validation of the R4000 processor took about 25% of the design time [7]. More recently, pre and post silicon validation of the Pentium Pro processor required a total of 300 staff-years [37]. The infamous Pentium division bug and `FOOF` bug shows that current design and hardware verification methods are inadequate. As technology evolves toward smaller feature sizes and larger die areas for chips, the complexity of designs continues to grow—making the task of design validation even more difficult.

We present the case for *asynchronous* computer architecture as a design methodology that addresses these issues. In an asynchronous system, the time taken by an operation is not artificially synchronized to a global clock signal. As a result, adopting an asynchronous methodology permits us to explore a larger, richer design space. The techniques we describe address the following challenges facing modern computer architects:

Design Complexity. Verification and design validation is becoming a larger and larger fraction of the design cycle. We believe that the *formal synthesis approach* to system design is the way to ensure correctness of hardware on first silicon. Asynchronous design lends itself to this methodology, because we can separate correctness and timing issues during the design. Asynchronous architectures are highly modular, and therefore can be designed in reusable components—components for which we can reuse correctness proofs as well.

Energy Complexity. The parts of an asynchronous circuit that do not contribute to the computation being performed have no switching activity. As a result, asynchronous circuits have an advantage in terms of power consumption when compared with clocked systems. We describe the techniques available for high-level power estimation in asynchronous systems, and how one can estimate the energy-complexity of an asynchronous computation.

Average-Case Performance. Asynchronous systems exhibit average-case timing behavior as opposed to the clocked systems that have to account for worst-case timing. We believe that it is possible to trade performance for energy per operation, and we show some examples of how joint energy/performance optimization can be used to explore this tradeoff.

In this paper, we describe asynchronous computations and architectures using a notation known as CHP (Communicating Hardware Processes), a language that is based on Hoare's CSP [8] and Dijkstra's guarded command notation [4]. A summary of the notation is provided in the appendix. This notation and the formal synthesis methodology we describe was first proposed by Martin [21]. In this paper, we use "asynchronous circuits" to mean circuits designed using Martin's formal synthesis approach. The synthesis results in *quasi delay-insensitive* (QDI) circuits. QDI circuits are defined to be those that function correctly under the assumption that gates and wires have arbitrary finite delay, except for some special wires known as isochronic forks [22].

II. Design Complexity

A natural approach to managing design complexity is to describe VLSI computations using a high-level language and then synthesizing VLSI implementations from the high-level description. Adopting a high-level synthesis approach requires finding the correct balance between physical considerations (transistor delay, wire delay, analog

Rajit Manohar <rajit@csl.cornell.edu> is with the Computer Systems Laboratory in the School of Electrical and Computer Engineering at Cornell University, Ithaca NY 14853, U.S.A.

considerations, etc) and logical considerations (data hazards, pipeline structure, etc). Asynchronous QDI VLSI circuits [22] are an ideal match for high-level synthesis approaches, because of the separation of performance issues from correctness issues. In this section we elaborate on the state-of-the-art in the design of provably correct complex asynchronous VLSI systems. We would like to emphasize that the approach we advocate has been repeatedly validated, and in each case the performance of our final implementations have been beyond our initial expectations. Two major designs that have been completed using this approach include the first asynchronous microprocessor designed in 1989 by Martin, Burns, Lee, et al. [19], and an asynchronous MIPS processor designed by Martin, Lines, Manohar, Nyström et al. [20] Both processors were functional on first silicon.

Formal Synthesis. The formal synthesis approach to asynchronous VLSI begins with a simple, sequential description of the specification to be implemented. The end result of the synthesis is a highly complex concurrent system that is a valid implementation of the original sequential specification. The transformation of a sequential specification to the final concurrent system is done using semantics-preserving transformations; therefore, if we know that the original sequential specification is correct, the final concurrent implementation is correct as well [21], [23].

When a synthesis-driven design methodology is used, the only verification necessary is to ensure that the rules that justify the transformations used during synthesis were correctly applied. This tends to be significantly less complex than complete verification because the set of rules is small, and most rules are syntactic. In addition, synthesis techniques preserve intermediate steps thereby simplifying any automated validation methods.

A. The First Asynchronous Microprocessor

The first asynchronous microprocessor was designed by Martin, Burns, Lee, et al. in 1989 [19]. It was a 16-bit integer processor with a load/store architecture. The sequential high-level description of the processor was a single page of CHP, which meant that the high-level description was easy to check. The final processor had many concurrently operating parts that implemented the original, simple sequential specification. The processor was fabricated in a $1.6\mu m$ CMOS process, and then refabricated in a $1.2\mu m$ CMOS process. The same design was also fabricated in GaAs. All three designs were functional on first "silicon."

Some of the transformations used in the design of the first asynchronous microprocessor are described in the appendix. The transformations shown together with similar transformations for control/data decomposition and a special transformation for register file locking were all the transformations necessary to design the first asynchronous microprocessor [19]. The transformations all share the following properties: they are simple, can be applied locally, and are purely syntactic making them easy to check.

While the first asynchronous microprocessor was relatively simple, it is a testament to the design methodology

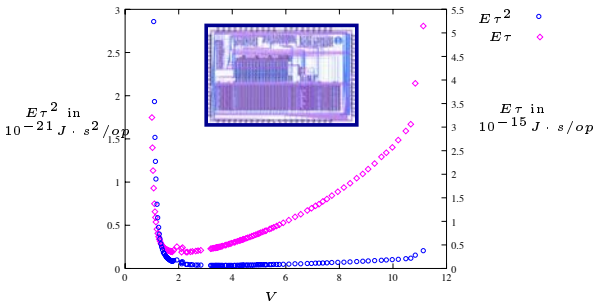

Fig. 1. Measured joint energy/cycle time metrics for the first asynchronous microprocessor over a range of voltages.

that it was completed in 6 months from the start of the project to tape-out by a small group. Figure 1 shows the layout and the lab results from the first asynchronous microprocessor designed at Caltech. It shows the measured $E\tau^2$ and $E\tau$ figures over a range of voltages, where E is the energy per operation and τ is the cycle time.

Testability. As part of the first microprocessor project, techniques were also developed for testing asynchronous circuits. By testing we mean fault testing of a chip to check for fabrication errors. It has been shown that every output stuck-at fault is fully testable in an asynchronous QDI circuit [24]. Contrary to popular belief, input stuck-at faults in asynchronous circuits need not be fully testable. In particular, input stuck-at faults on isochronic forks may not be testable. These stuck-at faults can be made testable by introducing additional test points [24]. Hazewindus has developed techniques for the identification of gates in an asynchronous QDI design that are not fully testable and for inserting test points and a test queue that makes the design testable [11].

B. The MiniMIPS Asynchronous Processor

The MiniMIPS processor is a 32-bit integer processor that implements most of the MIPS-I ISA. The processor has on chip caches and implements precise exceptions. The processor was fabricated in HP's $0.5\mu m$ CMOS process available through MOSIS, and was functional on first silicon.

The MiniMIPS project provided significant extensions to Martin's design methodology for asynchronous VLSI systems by providing a semantic framework for analyzing the correctness of highly pipelined asynchronous systems [20]. The main problem with the transformations outlined in the appendix is that they do not decouple the execution of one process from another. In the example with the counter shown in the appendix, for instance, an increment request forces both the counter and its environment to synchronize, even though the actual increment operation can be overlapped with other parts of the computation. Providing formal transformations that justify the decoupling of such operations using the new semantic framework was one of the contributions of the MiniMIPS project.

Slack Elasticity and Projection. The augmented design methodology is based on a concept known as *slack elas-*

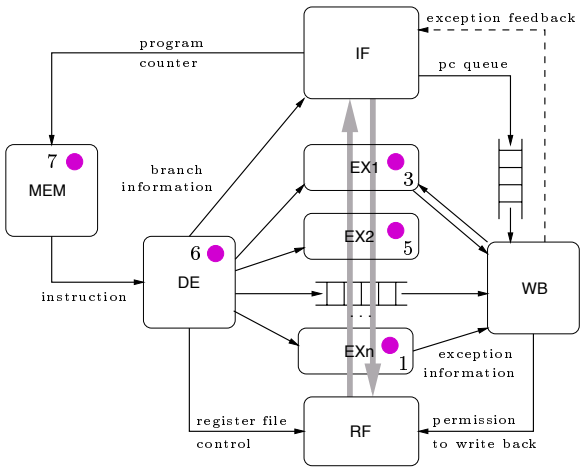

Fig. 2. The MiniMIPS pipeline structure.

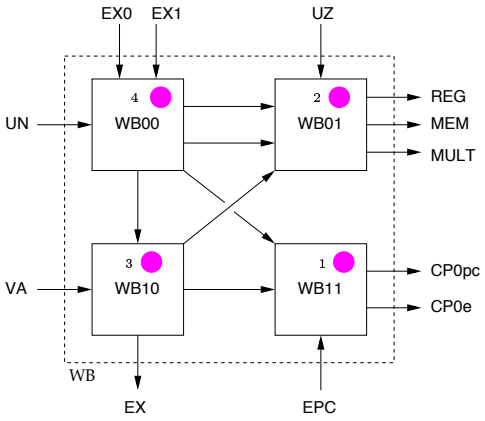

Fig. 3. The MiniMIPS writeback.

ticity [16]. The analysis provides conditions under which powerful transformations can be applied to improve the performance of an asynchronous VLSI system while preserving correctness. We created a general-purpose synthesis technique using the slack elasticity framework known as *projection* [15]. The following examples illustrate some of the important features of projection.

EXAMPLE: Consider the process $*[L?x; R!x; P?y; Q!y]$. This process repeated does the following: receive value on channel L into variable x; send this value out on R; receive value on channel P into variable y; send this value out on Q. The projection transformation decomposes this process into two that are not synchronized:

$$*[\ L?x; R!x\ ]\ \|\ *[\ P?y; Q!y\ ]$$

■

While this may appear to be a straightforward change to the computation, this transformation does not preserve correctness in general. It may introduce deadlock, or even cause a circuit to malfunction. The slack elasticity framework provides sufficient conditions under which such transformations can be applied [15]. These conditions were general enough to encompass the entire MiniMIPS processor design [20].

MiniMIPS Pipeline Structure. Figure 2 shows the pipeline structure of the MiniMIPS processor. Each block corresponds to two to six asynchronous pipeline stages [20]. The arrows correspond to flow of information from one concurrent part to another. The instruction fetch *IF* fetches the next instruction from the instruction cache *MEM* and sends it to *DE* to be decoded. *DE* then issues the instruction to an execution unit that is capable of executing the instruction. Since the MiniMIPS was designed to be compared to an R3000, the decode *DE* only issues one instruction per cycle. (Extending the decode to support multiple issue would be a simple task, and would not affect the rest of the execution pipeline.) However, the MiniMIPS pipeline structure can execute instructions out-of-order with respect to each other because instructions that take different times to execute are not artificially synchronized by a clock signal. This is achieved by providing two write ports to the register file. As a result, an instruction that takes longer to execute can write its results back after a later instruction. The example in Figure 2 shows five pending instructions in the pipeline (the instructions are numbered sequentially, with lower numbers meaning earlier instructions). Note that instruction number 2 and 4 have already completed and written their results back to the register file even though earlier instructions have not completed execution. The architecture has some of the benefits of an out-of-order processor without the complex instruction issue logic present in modern clocked processors such as the R10000 [36]. The MiniMIPS is also the first implementation of an asynchronous microprocessor to implement precise exceptions in the presence of multiple function units.

Figure 3 shows an example where we used projection to transform the writeback process *WB* in the MiniMIPS processor into the concurrent composition of four processes. An interesting feature of the transformation illustrated by Figure 3 is that the resulting system can process multiple data items concurrently without violating the correctness of the processor. HSPICE measurements showed that the new system had a cycle time of 2.9ns in HP's 0.5μm process, as opposed to a monolithic writeback which had an estimated cycle time of closer to 9ns. We could apply this transformation because the MiniMIPS was a slack elastic system [20].

Physical v/s Logical Pipelining. Asynchronous systems permit the separation between *physical* pipelining and *logical* pipelining. In a clocked system, the introduction of a new physical pipeline stage typically results in additional data hazards; the introduction of an asynchronous pipeline stage does not have this effect [16]. Architectural features such as branch delay slots that we consider to be the result of logical pipelining (since they change the ISA) must be embedded in the specification of the initial asynchronous system to be present in the final, concurrent version. This separation of physical and logical pipelining is what per-

mits the separation of correctness concerns from performance concerns. Physical asynchronous pipeline stages can be inserted or removed (almost always [16]) to improve the throughput of a computation without affecting its correctness. The ratio between physical and logical pipeline stages is dictated by performance considerations. Performance analysis frameworks can be used to accurately determine this ratio [3].

Test Results. Figure 4 show the physical layout and the lab results for the MiniMIPS processor. The processor has over 2M transistors, and the physical design was full-custom. The results are for HP's 0.5μm process (the 1998 version) available through MOSIS. Little time was spent on processor verification, since the design methodology guaranteed correctness by construction. No logical verification was required; the only verification necessary was to eliminate elementary errors such as crossed wires, mislabeled nodes, etc. The final concurrent MiniMIPS processor design contains over 10,000 interacting concurrent processes. The design was tractable only because of the formal, modular, design methodology used.

When we received first silicon from MOSIS, we observed that the performance of the processor was lower than that predicted by HSPICE. We discovered a long polysilicon wire with several resistive contacts that had escaped unnoticed during circuit simulations. It is a testament to the design methodology that the processor functioned correctly even though a few processes in the complete concurrent system ran slower than anticipated. A second problem was that HP's proprietary parameters for the process did not match the measured process parameters (in particular, the threshold voltages were higher than those specified by the process parameters). This problem did not affect correctness either. In summary, in spite of problems that would have caused other design methodologies to result in malfunctioning chips, the MiniMIPS processor was correct on first silicon.

III. Energy Complexity

The parts of an asynchronous circuit that do not contribute to the computation have no switching activity. As a result, those circuit components do not dissipate any power other than that due to leakage currents. In addition, the design methodology used eliminates all switching hazards. The combination of these two techniques results in reduced power dissipation for asynchronous architectures. If one is used to thinking about clock gating, asynchronous computations have "gated clocks" down to individual gates. In what follows, we will use the term power/energy to refer to the switching power/energy (i.e., we will ignore leakage currents).

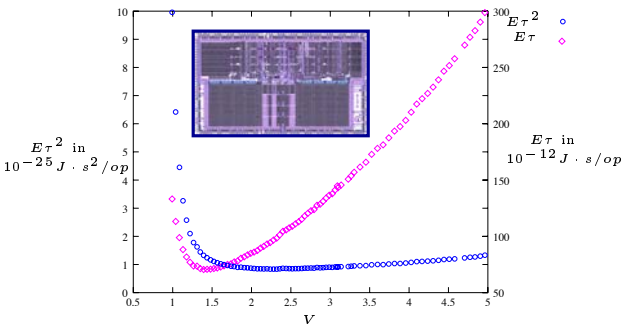

Fig. 4. Measured joint energy/cycle time metrics for the MiniMIPS asynchronous microprocessor over a range of voltages.

The figure of merit we will focus on when evaluating asynchronous architectures for power efficiency is the *energy per operation*. Energy per operation is an appropriate metric for asynchronous architectures for two reasons. First, the idle parts of an asynchronous circuit do not dissipate any switching power. As a result, the energy consumption of different operations can vary widely. Second, a metric that does not depend on the time taken by the operation (unlike power, for instance) allows us to compare different architectures that perform the same computation independent of timing considerations [32]. Tierno derived an elegant method by which we can estimate the energy per operation of the high-level specification of an asynchronous circuit [32]. We also showed how the energy per operation of the specification of a circuit can be related to the information-theoretic entropy of its specification [33]. We also derived lower bounds on the energy per operation by designing optimal coding techniques to specify the traces of the computation [12]. Other work has also shown that the entropy of the specification can be used to estimate the energy per operation [27].

The *energy-index* of a CHP computation is an estimate of the amount of switching capacitance for a given operation [33]. This measure can be determined from the syntax of the CHP computation, along with statistical information about the probabilities of different input cases. The reason this measure is accurate is two-fold: Asynchronous computations only dissipate energy when the CHP computation makes forward progress; the VLSI implementation of a CHP program can be determined from the syntax of the program itself. This energy index model has been used in determining energy-efficient memory configurations for asynchronous systems [34].

Example: Consider a simplified view of the decode phase of a microprocessor. A decoder can be visualized as a selection tree, with the leaves corresponding to the different possible symbols and the root corresponding to the instruction that has to be classified. If we consider the possible instructions to be symbols of an alphabet A with some frequency distribution, then the best clocked implementation would use a "flat" decoder—attempt to equalize decoding time so as to maximize clock rate. The resulting tree would have $\lg |A|$ depth, making decoding time as well as energy per operation proportional to $\lg |A|$. An asynchronous implementation could use a Huffman tree classifier, making the average case decoding time and average energy per operation $H(A) \leq \lg |A|$, where $H(A)$ is the entropy of the input distribution. ∎

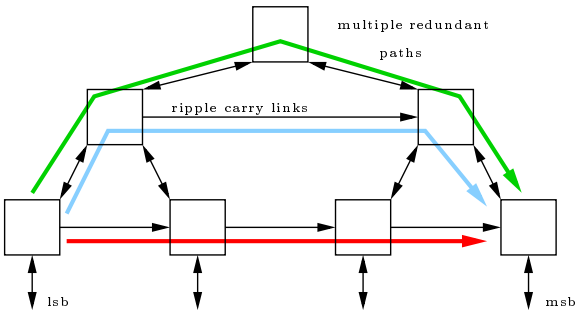

Fig. 5. Architecture of sub-logarithmic adder.

A more intriguing feature of asynchronous systems is that they provide a more continuous tradeoff between energy and performance. This is discussed further in the following section.

IV. Average-Case Performance and Joint Performance/Energy Optimization

A common reason for high energy consumption by modern processors is their use of speculative execution for enhancing performance. For example, a two-way set associative cache typically reads both sets simultaneously even though one of the two sets is guaranteed not to hold the address being read. A conditional-sum adder computes two sums—one for each possible carry-in—and then selects the correct sum once the carry has been computed. In both examples, energy could be saved by serializing some operations by only performing them whenever necessary while increasing the time taken for the computation.

In a clocked system, the global clock frequency is determined by the maximum worst-case delay of all components in the system.[1] Therefore, sequentializing an operation would be unacceptable if it causes the worst-case delay of the component to exceed the target clock cycle time—a local change would impact global performance through a lower clock rate. In an asynchronous system, it would only affect the delay for the particular operation under consideration—and only when the operation is performed.

Example: The decoder example in the previous section showed how using a Huffman code would reduce both the average energy as well as the average delay. Note that the worst-case energy and delay would probably be worse when using a Huffman coded decoder, and the delay considerations would preclude such an implementation in a clocked system. ∎

Average-case design can lead to reduced latencies, and sometimes the latencies can be asymptotically lower than the corresponding latencies for clocked systems. As early as 1946, von Neumann showed that an asynchronous n-bit ripple-carry adder had an average-case latency of $O(\log n)$ steps [2]. However, the adder had a worst-case latency of $O(n)$ steps since it was a ripple-carry adder. Recently, we designed an asynchronous n-bit adder with an average-case latency of $O(\log \log n)$ steps and a worst-case latency of $O(\log n)$ steps, showing how asynchronous techniques can be used to improve the latency of operations by exploiting data-dependent delays [18]. Figure 5 shows the architecture of the sub-logarithmic adder. It contains the normal *kpg* tree as well as the ripple-carry chain. The adder stages wait for the carry-in to arrive from either the *kpg* computation or the ripple-carry chain and then produce their output. This optimization results in multiple paths that can be used for carry propagation, and analysis shows that the average-case latency is sub-logarithmic. We also showed that the adder had the best possible asymptotic average-case latency for *any* input probability distribution [13].

The MiniMIPS processor demonstrates several instances of average-case optimization. The exception mechanism is optimized for the common-case so as to avoid synchronizing the instruction fetch and execution pipelines. The cache core was designed to have a lower cycle time than the rest of the system as an energy-saving measure, because it was determined that this would only slow down system throughput in 2% of the cases. The register file uses a special bypass circuit to handle forwarding. This not only improves performance, but results in reduced energy consumption because a forwarded operand is not read from the register file. These preliminary results are encouraging, and suggest that other structures in modern architectures should be examined and re-optimized for average-case performance by exploiting data-dependent delays.

Techniques for asynchronous systems need to find a compromise between energy optimization and delay optimization. Since asynchronous QDI circuits are robust to variations in voltage, we can vary the voltage to attain different design points in the (E, τ) space, where E is the energy per operation and τ is the cycle time. Since $E \sim V^2$ and $\tau \sim 1/V$, $E\tau^2$ is a good metric for the joint energy/performance optimization of asynchronous systems since it is voltage-independent to first order [20]. Given two designs, the one with a better (lower) $E\tau^2$ value would be able to match the performance of the other while using less energy per operation; it could also match the energy per operation while having better performance. (The only caveat is that the design point might be out of the operating voltage range—higher than the punch through voltage or lower than the threshold voltages.) Figures 1 and 4 show that $E\tau^2$ is relatively voltage-independent over a wide range of voltages for both asynchronous processors discussed in this paper. The same figures show another commonly used metric—the energy delay product—for the same processor. As can be seen, this metric varies linearly with voltage (as expected), and therefore cannot be used to compare VLSI computations that operate correctly over a wide voltage range. Metrics like the energy delay product and the energy can be artificially minimized by running a computation at the lowest permissible supply voltage.

$E\tau^2$ optimization can be applied at both the circuit level as well as the architecture level. The following example shows how one might calculate the pipeline depth in an

[1] Skew-tolerant domino circuits alleviate some local timing problems in clocked circuits [10].

asynchronous system that optimizes $E\tau^2$.

EXAMPLE: Consider a simple linear pipeline being optimized for $E\tau^\alpha$, where α is some constant. Let n be the number of pipeline stages, $E(n)$ be the energy per operation, and $\tau(n)$ the cycle time of the pipeline. A simple model for both energy and delay is that $E(n) = E_c + E_p n$, and $\tau(n) = t_p + t_c/n$, where E_c corresponds to the energy spent computing, E_p is the energy overhead of each additional pipeline stage, t_p is the cycle time overhead of pipelining, and t_c is the time spent computing. A simple calculation shows that when $E\tau^\alpha$ is optimized for pipeline depth,

$$\begin{aligned} E_{opt} &= E_c(1 + (\alpha-1)f/2 + f/2\sqrt{(\alpha-1)^2 + 4\alpha/f}) \\ \tau_{opt} &= t_p(1 + 2/((\alpha-1)^2 + \sqrt{(\alpha-1)^2 + 4\alpha/f})) \end{aligned}$$

where $f = (E_p t_c)/(E_c t_p)$. For $f = 1$, this simplifies to:

$$E_{opt} = E_c(1+\alpha)$$

In other words, if the ratio between the energy overhead for pipelining and the energy used by the computation is the same as the ratio between the cycle time overhead for pipelining to the time spent computing, then the optimal pipeline depth results when the energy spent in pipelining overhead is α times the energy spent computing. For $\alpha = 2$, $E_{opt} = 3E_c$ and $\tau_{opt} = 1.5t_p$. ∎

The robustness of asynchronous design methods to variations in voltage permit a constant $E\tau^2$ scaling, rather than the constant E scaling that would result by a simple scaling in the clock frequency. Constant $E\tau^2$ scaling can also be achieved by clocked systems designed to be robust to voltage variations, and that vary their voltage with clock frequency.

V. Design Tools

While most designers will agree that tools for the design of modern VLSI systems are inadequate in several ways, there has been much more work done in developing tools for clocked circuits. In this section we describe the tools that have been developed for asynchronous systems. These tools were used during the two aforementioned processor projects. To place these tools in context, we provide a brief description of the design flow for asynchronous systems.

A. Design Flow

The initial specification of the circuit to be designed is described in the programming notation CHP. The constructs of the language are described in the appendix, and include primitives for computation and communication.

The next step in the design flow is to decompose the sequential CHP program into a number of concurrently operating CHP programs using program transformations. This is the part of the design flow where most architectural choices are made. Decisions about pipeline structure, degree of pipelining, and control/data partitioning are taken at this step. Transformations such as process decomposition and projection are used to ensure that correctness is preserved when translating a sequential CHP program into a number of concurrent parts.

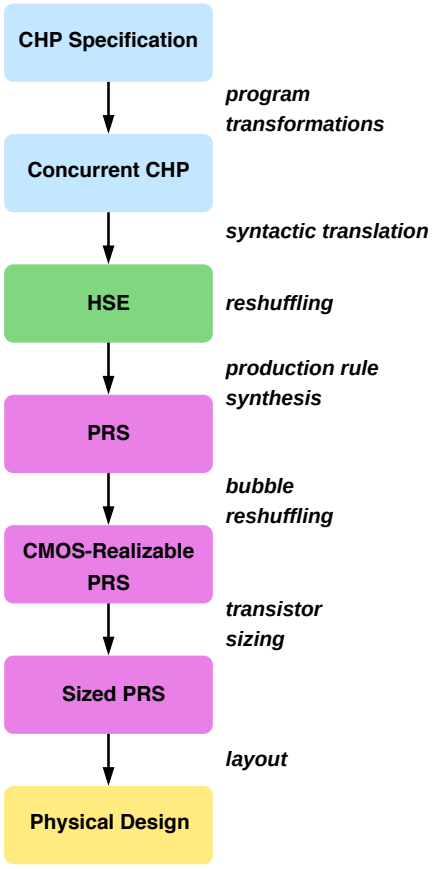

Fig. 6. Top-down design flow. If the design does not meet the required performance/energy/area target, design steps may have to be iterated.

The next step in the design flow is to translate each CHP process into *handshaking expansions* (HSE). Handshaking expansions can be thought of as a restricted subset of CHP, where all variables are Boolean-valued and all communication actions are replaced with synchronization protocols (handshake protocols). This transformation is syntax-directed, since each CHP action can be locally replaced with the appropriate HSE. Once the HSE for a process is obtained, it can be *reshuffled* for performance optimization.

The next step in the design flow is to translate HSE into a *production rule set* (PRS), a digital abstraction for CMOS circuits. A production rule is of the form $G \mapsto x\uparrow$ or $G \mapsto x\downarrow$, where G is a Boolean expression and x is a variable. The rule $G \mapsto x\uparrow$ corresponds to a pull-up network, and $G \mapsto x\downarrow$ corresponds to a pull-down network. The translation from HSE to PRS guarantees that the resulting PRS satisfies two key properties: stability and non-interference. Stability and non-interference are necessary and sufficient for hazard-free circuit behavior [17]. Once production rules are obtained, they must be made CMOS-implementable—rules of the form $G \mapsto x\downarrow$ must be implementable with n-transistors, and rules of the form $G \mapsto x\uparrow$ must be implementable with p-transistors. This transfor-

mation is known as *bubble reshuffling*, because it involves the insertion or movement of inverter "bubbles" in the circuit.

The next step in the design flow is to take the bubble-reshuffled PRS and determine the appropriate transistor sizing to optimize the desired performance metric. Once transistor sizes are chosen, we can implement the circuit in CMOS. For QDI circuits, transistor sizing does not affect correctness except for the isochronic fork assumption. The entire design flow is summarized in Figure 6.

B. Simulation and Synthesis Tools

In the first asynchronous processor and the MiniMIPS design, high-level CHP transformations were applied manually. We are currently developing tools that will assist in the CHP-level design of asynchronous systems. Developing good algorithms for CHP-level design is a topic of current research. We do not expect that these transformations can be completely automated in the general case, because these transformations determine both the architecture and pipeline structure of the final implementation.

We have several CHP-level simulators currently available. The first one, `mcc`, is a compiler for an extension of the C language that supports the CHP constructs described in the appendix. This simulator was used when running test programs through the high-level description of the MiniMIPS processor. More recently, we have developed a simulator that supports the syntax of the CHP language and automatically generates (approximate) timing and energy information.

Handshaking expansions can be syntactically derived from CHP, and they were manually generated for the first microprocessor project. We have developed a program that can translate CHP programs into handshaking expansions. Reshuffling handshaking expansions is another task that is performed manually. In the MiniMIPS project, we determined a small set of reshuffled handshaking expansions that led to good circuit implementations, and each CHP process was translated into a "canonical" handshaking expansion [20]. This greatly simplified the HSE part of the design.

A program, `he2prs`, that transforms handshaking expansions to production rules in a purely syntactic manner supports simulation of handshaking expansions using PRS simulators. While this program generates production rules, the resulting rules are inefficient. `prgen` is a program that generates good production rules for a common subset of handshaking expansions that arise in control processes. `hse` and `hse2prs` were programs that were developed (and are currently being improved) to handle general handshaking expansions. The restricted handshaking expansions chosen for the MiniMIPS project made the translation of HSE to PRS a straightforward task [20].

When production rules are written by hand, the program `prlint` can check that they are stable and non-interfering. Another program `prchk` performs the same function except it uses OBDDs to represent the state space.

We have several production rule simulators available. The first, `prsim`, has a number of features that makes it suitable for quickly testing a set of production rules. For instance, `prsim` can simulate the environment of a CHP process internally and can be used to generate a stream of input values for a CHP process. This simulator was used for most of the design of the MiniMIPS project. A simulator with less functionality but which could simulate much larger circuits in a small memory footprint and with greater speed, `csim`, was used for the complete digital switch-level simulation of the MiniMIPS. These simulators can report accurate timing information. A third simulator, `nsim`, is currently under development. In addition to the functionality of `csim`, it also reports accurate energy estimates and supports mixed-mode simulation of CHP and PRS. All these production rule simulators report errors if they detect a switching hazard during simulation. This can only occur if a designer decides to bypass the synthesis procedure and does not run `prlint` or `prchk` on hand-generated production rules.

Bubble-reshuffling can be done by using the program `bubble`. Once bubble-reshuffling is performed, `ergen` can be used for transistor sizing. `ergen` uses a performance analysis framework to determine the cycle time of the asynchronous circuit and then attempts to optimize it by adjusting the transistor widths of different gates. Once sized production rules are obtained, we can proceed with physical design.

During the course of the MiniMIPS project, two tools to support analog simulation were developed. `aspice` is a mixed analog-digital simulator that was used to simulate the full MiniMIPS design. Different units could be simulated at the analog level while most of the chip was simulated at the digital level. `alint` was developed to analyze the output of `aspice` to identify a number of different analog problems including poor slew rates, critical path analysis, and charge-sharing. We also used MetaSoft's HSPICE for analog simulations.

C. Physical Design Tools

The QDI model for asynchronous VLSI assumes that gates in the final implementation may have arbitrary positive delays. Therefore, the correctness of the system does not depend on the actual delays of the gates. This eliminates the need for timing verification of QDI circuits. Timing analysis is solely used to optimize the performance of the design, and is not required to ensure that the design functions correctly.

A sized production rule set can be syntactically transformed into a netlist with transistor width specifications. At this point we are faced with the same physical design problem as the one faced by clocked circuit designers: determine the best possible placement for each gate and route each netlist.

The first asynchronous microprocessor was designed using full-custom layout for the datapath and automated layout for the control. Full-custom layout was done with the Magic VLSI layout editor. Two programs were used for the control design: `cellgen` and `Vgladys`. `cellgen` is a pro-

gram that takes a sized production rule set and generates a standard cell style layout for it. `Vgladys` is a placement and routing tool that uses simulated annealing for placement and a channel router.

The MiniMIPS processor was designed using full-custom layout done with a version of Magic augmented with a simple production rule layout capability. A programming language was added to the user interface to make the layout editor extensible, and the language was used to add several features to Magic. These changes have been incorporated into the standard Magic release, and the current version is available through CVS from Cornell. In addition, a program called `lvs` was developed to validate the physical design against a production rule description. Apart from checking that production rules are correctly implemented, `lvs` performs checks for analog problems such as charge sharing and capacitive coupling. The program was also used for checking clocked designs in Caltech's class on VLSI design.

D. Summary

With the current design tools, a typical asynchronous chip would be designed as follows:

- Initial CHP specification and concurrent CHP design done manually. Architectural, pipelining, and other major design decisions made. The design can be simulated using the CHP simulator, and timing/energy feedback can be used to identify and eliminate architectural bottlenecks.
- The concurrent CHP is translated into reshuffled HSE using the circuit templates developed for the MiniMIPS project. Non-standard HSE reshufflings can also be used if necessary for achieving the desired throughput/energy/area target.
- HSE translated to PRS using `hse2prs` or `hse`, and the result can be bubble-reshuffled using `bubble`. The production rules can be simulated using `prsim`. If the design is too large for `prsim` to handle, `nsim` can be used for digital simulation. If production rules are generated by hand, `prlint` may be used for production rule validation.
- Transistor sizing is performed using `ergen`.
- The design is floorplanned, and the CMOS implementation of the production rules are placed and routed. Any manual design using Magic can be validated against the production rule description using `lvs`. The physical design can be extracted and simulated using HSPICE, or a mixed-mode simulation can be performed using `aspice`. In either case, `alint` can be used to quickly identify analog problems such as charge-sharing or capacitive coupling.

E. Comparison with Clocked Designs

The design flow for asynchronous systems is similar to the flow for clocked systems, but the architectural tradeoffs differ because of the asynchronous nature of the implementation. The initial architectural decisions in a clocked system are made by the designers, much like the high-level CHP design in an asynchronous system. Once the RTL description is finalized, the design is implemented using a hardware description language such as Verilog or VHDL. The design can be simulated at this level, and contains a complete description of the architecture of the clocked system.

Verilog/VHDL design is followed by logic synthesis, where the hardware description language is translated into logic gates by synthesis tools or by hand. This phase roughly corresponds to the translation of CHP to production rules.

The mapped logic is then simulated, and timing validation is performed. While timing validation is not necessary for QDI circuits, production rule simulation and timing analysis are also performed for performance tuning.

Finally, the design is floorplanned, and the synthesized logic gates are placed and routed. This step is analogous to the one for asynchronous designs.

VI. Related and Future Work

A. Historical Perspective

Asynchronous switching circuits have been in use since the 1940's. The Illiac, designed by the University of Illinois in the 1950's, is an example of a computer that contained both synchronous and asynchronous switching circuits [26]. Early concepts in the design of asynchronous circuits were contributed by D.A. Huffman in the 1950's [26]. As complex asynchronous circuits became difficult to design because of the problem of hazards (glitches) in switching signals, they were replaced by synchronous circuits.

A theory of *speed-independent* asynchronous switching circuits was developed by D.E. Muller in the early 1960's as an attempt to abstract from the difficulties of designing circuits that depended heavily on their precise physical implementation. His model assumed that transistor networks may have arbitrary delay, and that the propagation delay through wires is negligible compared to the delay through the network. However, his design methodology was limited to simple sequential circuits.

Modern asynchronous circuit design probably began when concerns arose regarding problems with the physical realization of large-scale synchronous systems. In 1979, Seitz proposed a design methodology for *self-timed* circuits wherein the circuit was to be decomposed into equipotential regions—regions where delays in wires could be considered negligible, with explicit modeling of signal propagation delay between such regions [30].

The main problem with all these design methodologies is that they attempted to design asynchronous circuits in a bottom-up manner by removing switching hazards as they were encountered, thereby making the design method cumbersome and error-prone. The resulting circuits were very sensitive to gate delays, making timing validation a critical part of circuit verification. The techniques we refer to in this paper are top-down and synthesis based, and make the minimum possible timing assumptions.

The first method for the synthesis of asynchronous circuits whose correct functioning did not depend on the delays of gates and which permitted multiple concurrent switching signals was introduced by Alain Martin [21]. His

formal synthesis approach is inspired by the observation that a VLSI chip is a fine-grained concurrent computation. VLSI computations are modeled using CHP programs that describe their behavior algorithmically. Asynchronous *quasi delay-insensitive* (QDI) circuits are synthesized from these programs using semantics-preserving transformations. The circuit design methodology assumes that gates have arbitrary delay, and only makes relative timing assumptions on the propagation delay of some signals that fanout to multiple gates. The result is a methodology that produces circuits robust to variations in device parameters, voltage, and temperature.

Martin's synthesis methodology results in asynchronous circuits that are *guaranteed* to have no switching hazards. The methodology generates circuits that satisfy two key properties—stability and non-interference—that are both necessary and sufficient for hazard-free circuit behavior [17].

B. Current Asynchronous Architectures

Existing asynchronous processors have either used a RISC instruction set in the interests of simplicity, or an instruction set based on one for a clocked processor [6], [28], [29], [20]. The AMULET is a self-timed ARM processor. It uses the same pipeline structure as its clocked counterpart, and it has only one execution unit. The next version of this processor is planned to be dual-issue. The Fred architecture uses a Motorola 88100-based instruction set [35], and a decoupled architecture similar to out-of-order processors like the R10000 [36]. The asynchronous MiniMIPS design has a better throughput and $E\tau^2$ compared to these processors.[2] The counterflow pipeline architecture uses a pipeline where operands and instructions flow in opposite directions with arbitration being used to resolve dependencies [31]. In the clocked world, skew-tolerant domino pipes [10] permit local time borrowing to alleviate some timing issues in latched-based clocked designs. Asynchronous circuits have the added benefit that they support the execution of computations even when they exceed the "borrowing limit" from an adjacent pipeline stage, while still being optimized for higher average-case throughput.

A comparison of the MiniMIPS processor with other clocked architectures can be found in [20]. The comparison is difficult to make because modern high-performance processors have many additional features not supported by the MiniMIPS (such as floating-point), and most clocked processors with a feature set that match the MiniMIPS are not designed for performance. The next-generation asynchronous processor under development will have several additional features, and we plan to make a comparison when the design is relatively complete.

C. Future Work

In future work, we plan to explore the design of a low-energy asynchronous processor that incorporates the ideas that we have discussed in this paper. The MiniMIPS processor project was not focused on energy optimization, and there are several instances of optimizations that would significantly reduce the energy per operation of the MiniMIPS while keeping throughput high. As part of this project, we are currently developing new design tools that will be capable of automating several tasks that were done manually during the MiniMIPS project. We also plan to explore the construction of asynchronous memory systems, in particular asynchronous active memories, since asynchronous design techniques enable a number of interesting optimizations in memory design [14].

[2] We are not aware of published energy per operation values for Fred.

VII. Summary

We have presented a case for the design of modern asynchronous systems. We believe that this approach will provide significant advantages in terms of design complexity, energy efficiency, performance optimization for average-case behavior, and system scalability.

VIII. Acknowledgments

The results described here have primarily been the result of over a decade of research in asynchronous VLSI at Caltech under the direction of Alain Martin. Some of the students from his group that have made significant contributions to the design methodology described in this paper include Steve Burns, Marcel van der Goot, Pieter Hazewindus, Tak-Kwan Lee, Andrew Lines, Rajit Manohar, Mika Nyström, and José Tierno.

The people that contributed to the design of the first asynchronous microprocessor include Drazen Borkovic, Steve Burns, Pieter Hazewindus, Tak-Kwan Lee, and Alain Martin. The people that contributed to the design and post-fab test of the MiniMIPS asynchronous processor include Steve Burns, Uri Cummings, Vito Dai, Marcel van der Goot, Matt Hanna, Peter Hofstee, Tak-Kwan Lee, Andrew Lines, Rajit Manohar, Alain Martin, Mark Neidengard, Mika Nyström, Paul Penzes, Marc Renaudin, Robert Southworth, and Cathy Wong.

The design tools described in Section V were authored and/or extensively modified by Steve Burns, James Cook, Marcel van der Goot, Eitan Grinspun, Matt Hanna, Pieter Hazewindus, David Long, Tak-Kwan Lee, Andrew Lines, Rajit Manohar, Mika Nyström, Robert Southworth, and José Tierno. The VLSI layout editor Magic was originally developed at Berkeley by John Ousterhout, and has since been modified by groups at Caltech, Cornell, DEC WRL, Johns Hopkins, Lawrence Livermore National Labs, and Stanford, to name a few of the contributors. The current Magic maintainers are Tim Edwards and Rajit Manohar.

The research described in this report was sponsored by the Defense and Advanced Research Projects Agency as part of the Submicron Systems Architecture Project and the Asynchronous Systems Architecture Project. The research was also supported in part by NSF CAREER award CCR 9984299, and by ONR MURI award N00014-00-1-0564.

The author would like to thank Mark Heinrich for invaluable feedback, and for providing the perspective of a designer of clocked systems. The author would also like to thank the anonymous referees for their comments and suggestions.

References

[1] Gregg Bouchard and Pete Bannon. Design Objectives of the 0.35μm Alpha 21164 Microprocessor. In *Proceedings Notebook for HotChips VIII*, pp. 21–34, August 1996.

[2] A.W. Burks, H.H. Goldstein, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Institute for Advanced Study, Princeton, N.J., June 1946.

[3] Steven M. Burns and Alain J. Martin. Performance analysis and optimization of asynchronous circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference*, pp. 71–86, 1991.

[4] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[5] Daniel W. Dobberpuhl. Circuits and Technology for Digital's StrongARM and ALPHA Microprocessors. *Proceedings of the 17th Conference on Advanced Research in VLSI*, September 1997.

[6] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods. A micropipelined ARM. *Proceedings of the VII Banff Workshop: Asynchronous Hardware Design*, August 1993.

[7] John Hennessy. Plenary session talk, *International Symposium on Computer Architecture*, June 1999.

[8] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, **21**(8):666–677, August 1978.

[9] Marcel van der Goot. Semantics of VLSI Synthesis. Ph.D. thesis, California Institute of Technology, 1995.

[10] David Harris and Mark A. Horowitz. Skew-Tolerant Domino Circuits. *IEEE Journal of Solid-State Circuits*, **32**(11):1702–1711, November 1997.

[11] Pieter Hazewindus. Testing Delay-insensitive circuits. Ph.D thesis., CS-TR-92-14, California Institute of Technology, May 1992.

[12] Rajit Manohar. The Entropy of Traces in Parallel Computation. *IEEE Transactions on Information Theory*, **45**(5):1606–1608, July 1999.

[13] Rajit Manohar. *The Impact of Asynchrony on Computer Architecture*. Ph.D. thesis, CS-TR-98-12, California Institute of Technology, July 1998.

[14] Rajit Manohar and Mark Heinrich. A Case for Asynchronous Active Memories. Cornell Computer Systems Lab Technical Report CSL-TR-2000-1003, May 2000.

[15] Rajit Manohar, Tak-Kwan Lee, and Alain J. Martin. Projection: A Synthesis Technique for Concurrent Systems. *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1999.

[16] Rajit Manohar and Alain J. Martin. Slack Elasticity in Concurrent Computing. *Proceedings of the Fourth International Conference on the Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, pp. 272–285, Springer-Verlag, June 1998.

[17] Rajit Manohar and Alain J. Martin. Quasi-delay-insensitive circuits are Turing-complete. Invited article, *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 1996. Available as Caltech technical report CS-TR-95-11, November 1995.

[18] Rajit Manohar and José A. Tierno. Asynchronous Parallel Prefix Computation. *IEEE Transactions on Computers*, **47**(11):1244–1252, November 1998.

[19] 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, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pp. 351–373, MIT Press, 1991.

[20] A.J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. Cummings, and T.K. Lee. The Design of an Asynchronous MIPS R3000 Processor. *Proceedings of the 17th Conference on Advanced Research in VLSI*. Los Alamitos, Calif.: IEEE Computer Society Press, 1997.

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

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

[23] A.J. Martin. "Synthesis of Asynchronous VLSI Circuits," in *Formal Methods for VLSI Design*, J. Staunstrup, Ed. North-Holland, 1990.

[24] Alain Martin and Pieter Hazewindus. Testing delay-insensitive circuits. *Advanced Research in VLSI, Proceedings of the 1991 UC Santa Cruz Conference*, 1991.

[25] Carver Mead and Lynn Conway. *Introduction to VLSI System Design*. Addison-Wesley, 1979.

[26] Raymond E. Miller. *Switching Theory*, Volumes 1 and 2. John Wiley and Sons, 1965.

[27] Farid Najm. Towards a high-level power estimation capability. *Proceedings 1995 Symposium on Low Power Design*, pp. 87–92. Association for Computing Machinery, April 1995.

[28] T. Nanya, A. Takamura, M. Kuwako et al. Scalable Delay-Insensitive Design: A high-performance approach to dependable asynchronous systems. *Proc. Int. Symp. on Future of Integrated Electronics*, pp. 531–540, Japan (March 1999)

[29] M. Renaudin, P. Vivet, and F. Robin. ASPRO-216: A Standard-Cell Q.D.I. 16-Bit RISC Asynchronous Microprocessor. *Proceedings of the 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1998.

[30] Charles L. Seitz. System Timing. Chapter 7 in *Introduction to VLSI Systems*, by Carver Mead and Lynn Conway, Addison-Wesley, 1979.

[31] Robert F. Sproull and Ivan E. Sutherland. Counterflow pipeline processor architecture. Technical Report SMLI TR-94-25, Sun Microsystems, April 1994.

[32] José A. Tierno. *An Energy Complexity Model for VLSI Computations*. Ph.D. thesis, California Institute of Technology, 1995.

[33] José A. Tierno, Rajit Manohar, and Alain J. Martin. The Energy and Entropy of VLSI Computations. *Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 1996.

[34] José A. Tierno and Alain J. Martin. Low-Energy Asynchronous Memory Design. *Proceedings of the First International Symposium on Advanced Research in Asynchronous Circuits and Systems*, November 1994.

[35] William F. Richardson. Architectural Considerations in a Self-Timed Processor Design. Ph.D. thesis, Department of Computer Science, University of Utah, 1996.

[36] Ken Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, **16**(2):28–40, April 1996.

[37] Albert Yu. Personal Communication, 1997.

Appendix

I. Notation

The notation we use is based on Hoare's CSP [8]. What follows is a short and informal description of the notation we use. A complete formal semantics can be found in van der Goot's thesis [9].

Simple statements and expressions.

- Skip: **skip**. This statement does nothing.
- Assignment: $x := E$. This statement means "assign the value of E to x." When E is **true**, we abbreviate $x := E$ to $x\uparrow$, and when E is **false** we abbreviate $x := E$ to $x\downarrow$.
- Communication: $X!e$ means send the value of e over channel X; $Y?x$ means receive a value over channel Y and store it in variable x. When we are not communicating data values over a channel, the directionality of the channel is unimportant. In this case, the statement X denotes a synchronization action on port X.
- Probe: The boolean $\overline{X}$ is true if and only if a communication over channel X can complete without suspending.

Compound statements.

• Selection: $[G_1 \to S_1 \;[]\; ... \;[]\; G_n \to S_n]$, where G_i's are boolean expressions (guards) and S_i's are program parts. The execution of this command corresponds to waiting until one of the guards is true, and then executing one of the statements with a true guard. The notation $[G]$ is short-hand for $[G \to \textbf{skip}]$, and denotes waiting for the predicate G to become true. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]."
• Repetition: $*[G_1 \to S_1 \;[]\; ... \;[]\; G_n \to S_n]$. The execution of this command corresponds to choosing one of the true guards and executing the corresponding statement, repeating this until all guards evaluate to false. The notation $*[S]$ is short-hand for $*[\textbf{true} \to S]$. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]."
• Sequential Composition: $S; T$. The semicolon binds tighter than the parallel composition operator $\|$, but weaker than the comma or bullet.
• Parallel Composition: $S \parallel T$ or S, T. The $\|$ operator binds weaker than the bullet or semicolon. The comma binds tighter than the semicolon but weaker than the bullet.
• Simultaneous Composition: $S \bullet T$ (read "S bullet T") means that the actions S and T complete simultaneously. Typically, the two actions are communication actions only, and the implementation of the bullet corresponds to replacing S by $S; S$ and T by $T; T$ and then picking an interleaving of the "doubled" actions, like $S; T; S; T$. The operator binds tighter than the semicolon and parallel composition.

The concurrent execution of a collection of CHP processes is assumed to be *weakly fair*—every continuously enabled action will be given a chance to execute eventually. The choice operator in the selection statement is assumed to be demonic, and therefore the choice is *not* fair. Consider the following four processes:

$$*[\; X!0 \;] \parallel *[\; Y!1 \;] \parallel *[[\overline{X} \longrightarrow X?x \mid \overline{Y} \longrightarrow Y?x \;]; \; Z!x \;] \parallel *[\; W!2 \;]$$

Since the selection statement is not fair, Z is permitted to output an infinite sequence of zeros. However, both $Z!x$ and $W!2$ will execute eventually, since parallel composition is assumed to be weakly fair.

II. Program Transformations

The following describes some of the program transformations used in the first asynchronous microprocessor.

Example: Consider a computation described as the infinite repetition of three actions S, T, and U. The CHP for this computation is shown below:

$$*[\; S; T; U \;]$$

(The $*[$ and $]$ denote an infinite repetition.) One of the synthesis transformations, known as *process decomposition* [23], breaks up a sequential computation into two parts in a structured manner. An instance of process decomposition would be to replace the computation shown above by:

$$*[\; S; X!; U \;] \parallel *[[\overline{X} \longrightarrow T; X?]]$$

In this decomposition, after S executes, a communication action $X!$ (send) is attempted. The second process waits until a communication on X is attempted, and then executes T. Once T completes, $X?$ (a receive) is executed, permitting $X!$ to complete. U can now execute, and the entire computation can be repeated.

What should be noted is that the transformation applied to convert $*[S; T; U]$ into $*[S; X!; U] \| *[[\overline{X} \to T; X?]]$ clearly preserves the semantics of the computation. Also, the transformation can be easily checked if we specify the program fragment to which the transformation was applied. This is a recurring theme in the transformations used for asynchronous VLSI design. ∎

Process decomposition does not introduce concurrency into the system. We can do so by attempting to overlap different parts of the computation that are clearly not data-dependent. The following example demonstrates such a transformation.

Example: Consider a counter that stores an integer x, and can supports operations that increment x by 1, reset x to zero, or read x. This is described as follows:

$$\begin{array}{l} *[[\overline{inc} \longrightarrow x := x + 1; inc? \\ \;\;[]\overline{reset} \longrightarrow x := 0; reset? \\ \;\;[]\overline{read} \longrightarrow read!x \\ \;]] \end{array}$$

The process waits until one of the three requests: *inc*, *reset*, or *read* is **true**, performs the specified operation, and completes the communication action. Since x is a local variable, we can replace this process with the following without changing the correctness of the computation:

$$\begin{array}{l} *[[\overline{inc} \longrightarrow inc?; x := x + 1 \\ \;\;[]\overline{reset} \longrightarrow reset?; x := 0 \\ \;\;[]\overline{read} \longrightarrow read!x \\ \;]] \end{array}$$

Now when the environment requests an increment operation, the request is completed before the increment is performed.

Once again, we have shown a local transformation that introduces concurrency between the counter and its environment without affecting correctness. The transformation is syntactic, local, and easy to check. ∎