Asynchronous Processors

Nicolae-Andrei Vasile

Faculty of Computer Science and Automatic Control

POLITEHNICA University of Bucharest

Bucharest, Romania

Abstract – Synchronous processors, dependent on a clock, are not necessarily the perfect computing solution. As this look at several experimental approaches indicates, asynchronous processors may one day offer improvements over present system performance.

I. INTRODUCTION

The recent advances in solid-state technology have provided computer designers with powerful functional capabilities at low cost. This enables computer designers to isolate the functions of a computing element and add intelligence to functional units that can optimize their circuits locally. Such an approach eliminates the cost involved in time and band-width to submit these local events for a central judgment in real time. This trend has led to the emergence of distributed systems.

Furthermore, the recent growth in microprocessor architectures, their capabilities, and their low cost, have motivated system designers to design computer systems as distributed networks of microprocessors. By using multiple processing elements, system throughput can be improved and processing requirements and capabilities unobtainable by uniprocessors can be satisfied. However, the success of multiple processor systems greatly depends on the effectiveness of the synchronization among the processing elements [1].

II. ASYNCHRONOUS MODEL

Several asynchronous microprocessors were developed in the late 80's, 90's and early 00's, but none of them reported the full advantages promised. They were a test of the potential for the design method at the time, and one which only showed a modest improvement for a high cost. With current design advances providing less and less benefit for their costs, it may be time to look at them again.

There are five main advantages to asynchronous circuits: low power, low Electromagnetic Interference (EMI), high speed, high tolerance to some types of errors, and modularity. As will be shown, these advantages are not guaranteed in microprocessors. Careful designing is required to get even some of these, and often the design will cost one or more advantage to realize the others. In particular, the processors shown in this paper tended to function at a slightly lower power than their synchronous equivalents at the cost of at best matching the throughput.

There are three significant drawbacks to asynchronous circuits: they are difficult to design (made worse by a lack of design tools), they are impossible to fully test without additional design for test (DFT) circuitry, and they are more susceptible to some types of errors. The limits of design tools can lead to poor performance, even when the design works. The DFT circuitry can cost up to a 50% increase in area to ensure full stuck-at testability.

III. ASYNCHRONOUS PROCESSORS

To spur more research in asynchronous processing, six distinctly different asynchronous processors were reviewed. We wanted to see the performance the processors had achieved or, in a couple cases where the processor had not yet been fabricated, to see what the potential performance might be [2].

CAP (Caltech Asynchronous Processor)

The earliest known asynchronous processor, CAP, was designed in the late 1980s by Alain Martin's group at the California Institute of Technology [2]. CAP was never intended to be a formal processor design. It was intended for circuit design experimentation and for development of method testing based on program transformations. Initially, Martin and his colleagues designed the individual processor functions as eight concurrent programs: Program execution thus enabled functional testing. The programs were then individually compiled to model electronic circuits and interconnected to form the processor.

CAP uses a four-phase communication protocol with dual rail data transfer (each data bit is represented by two wires). This arrangement works efficiently with dynamic logic families.

The designers estimated performance of 15 MIPS (the processor was not fabricated when we surveyed it). The performance analysis method (estimation of critical path) did not include the potential benefits of interleaving ALU and memory instructions; nor the effects of branching, cache misses, and other interrupts; nor the delay that is created by additional circuit loading from omitted (yet necessary) functionality. More recently, Martin's group has built a gallium arsenide version of the basic CAP architecture, exhibiting 100-MIPS performance [2].

FAM (Fully Asynchronous Microprocessor)

Developed in the early 1990s by Kyoung Rok Cho from the Korean Institute of Science and Technology and Kazum Okura and Kunijiro Asada from the Tokyo Institute of Technology, FAM models a load/store, four-stage pipelined RISC architecture. FAM uses a four-phase communication protocol with dual-rail data transfer [2].

FAM, like CAP, is experimental. Though the processor contains a 32-bit data path and thirty-two 32-bit general registers, the processor's capability is severely limited by having only 18 available instructions. This effort nonetheless demonstrates a method to design asynchronous data and control paths.

The FAM design employs both computation and interconnection blocks. An interconnection block separates each pipeline stage or computation block and provides intermediate storage to decouple the computation blocks, which simplifies their control.

FAM development focused on 2-AND logic, which uses at most two transistors in series to both set and reset the flip-flops, depicting the system state. Of these two transistors, one is always controlled by a system input and the other, by a flip-flop internal to the system. The advantage of 2-AND logic is that it is faster and more compact than C-elements, gate-level components for synchronizing two events.

The designers calculated an estimated (as the FAM was not fabricated) 300 MIPS from the average instruction execution time. For this MIPS rating to be truly relevant, "real" programs must execute with an equal distribution of instructions and must avoid stalls from load latencies, branching, interrupts, and so forth. Still, the reported average instruction cycle time of 3.5 ns is noteworthy, even for the 0.5-mm CMOS process technology. In addition, the FAM demonstrates what could be an optimal organization for an asynchronous processor: computation blocks that make use of fast dynamic logic separated by static interconnection blocks implemented with 2-AND logic.

NSR (Nonsynchronous RISC)

NSR, developed by Erik Brunvand at the University of Utah in 1993, is basically a collection of self-timed blocks. In the NSR, the processor essentially comprises five concurrent blocks, analogous to the standard synchronous pipeline functions of instruction fetch, decode, execute, memory access, and “write-back” or register file. Also, the NSR has added FIFO queues between the concurrent blocks (not shown) to minimize stalls caused by slower instructions [2].

Unlike the CAP and the FAM, the NSR uses a two-phase communication protocol and a bounded-delay data transmission, a method that adds delay elements to the control path as needed to ensure validity in the data path. Using this protocol, each processor stage or concurrent block accepts data for processing and passes the result to the next stage by way of the FIFO. Potentially, the FIFOs could greatly increase the penalties associated with memory access, branching, and so on; however, the instructions will pass through only those stages required for completion.

Like CAP and FAM, NSR is experimental. A prototype NSR was implemented with Actel field programmable gate arrays (FPGAs). To permit testing, each processor stage or block includes a switch that can block an outgoing request. Since other stages never see the request, no acknowledge signals are sent and the stage is effectively blocked.

With the FPGA implementation, the designer reported a performance of 1.3 MIPS, based on best case operating speed. Since FPGAs are inherently slower than standard CMOS logic circuits, it is difficult to compare the FAM performance with that of other processors, but a 10-times performance increase (or more) for a full-custom CMOS implementation would not be unreasonable.

CFPP (Counterflow Pipeline Processor)

The CFPP was developed in 1994 at Sun Microsystems by Ivan Sutherland, Robert Sproull, and Charles Molnar. In this architecture, as instructions flow through the pipeline in one direction, the instructions generate data that flows through the pipeline in the opposite direction. This particular CFPP was not fabricated [2].

The CFPP’s basic structure places the program counter at one end of a multiple-stage pipeline and the register file at the other. The processor inserts instruction packets, consisting of the opcode, source and destination register bindings, and possibly the corresponding program counter value, into the pipeline stage next to the program counter. The packets then proceed toward the register file. The processor reads the source operands and inserts them into the pipeline stage adjacent to the register file before they proceed toward the program counter. In general, each stage of the pipeline is identical, capable of executing any instruction. The only requirement is that the instruction packet must first rendezvous with its source operands as they flow opposite each other. After completion, the instruction continues to flow toward the register file, in which the processor posts the result. In addition, it inserts the result into the opposite-flowing result pipeline. Now a dependent instruction that might follow will not have to wait to receive its source operands after they have been posted to the register file. Instead, it will receive the source operands from the results pipeline, possibly before the register file is even updated. This architecture, therefore, naturally provides register renaming.

It handles interrupts and wrongly predicted branches by inserting an identifier into the results pipeline. Instructions that precede the identifier will continue to flow toward the register file and post their results as desired. All instructions following the identifier, however, are marked invalid and prevented from altering the register file. When the identifier reaches the program counter, the processor either enters an interrupt routine or loads the correct branch destination into the program counter, depending on the action required. In theory, the architecture supports precise interrupts and can recover easily from erroneous branch predictions.

In practice, the CFPP stages are not identical and cannot execute all possible instructions. Therefore an unexecuted instruction must be prevented from passing beyond the last stage capable of executing it. Also, the CFPP may use auxiliary stages or "sidings" to execute instructions with long computation delays. Results from these long-latency instructions are recovered later.

No optimal CFPP configuration yet exists. Furthermore, because the CFPP was not fabricated when we surveyed it, we can discuss the CFPP's performance only in qualitative terms. One advantage cited by the designers is the pipeline's regular structure, which facilitates layout. However, both the sidings and the fact that not all stages are identical seem to prohibit any regularity. Figure 4 shows a long, 12-stage pipeline between the program counter and the register file. After including the sidings, building the CFPP may require an exorbitant amount of physical area.

The CFPP architecture does provide register renaming, data forwarding, and a simple, efficient implementation for handling interrupts and branching. Unfortunately, the mechanism that provides these features may also cripple the CFPP's performance.

Dynamic scheduling may provide additional instructions; however, the complexity of detecting WAW and WAR data hazards in both the register file and the large results pipeline is likely to diminish throughput.

Strip (A Self-Timed RISC Processor)

The Strip architecture is unique in that it is essentially a synchronous processor with an adjustable clock, thanks to its dynamic clocking communication protocol. Developed by Mark Dean and based on the synchronous Stanford MIPS-X processor, it has five pipeline stages: instruction fetch, register fetch and instruction decode, ALU execution, data memory access, and the register write-back. Strip uses bounded-delay data transmission [2].

Because the clock period is determined by the current clock cycle's slowest critical path, every pipeline stage and functional unit must be optimized. This contrasts with the clock cycle of a synchronous or globally clocked implementation, which is determined by the slowest operation, currently active or not, that ever takes place.

The dynamic clock generator's design is crucial to Strip's performance and functionality: Increase clock delay too much and you degrade performance; increase it too little and you generate computation errors. To accomplish dynamic clocking, Strip uses a set of tracking cells, a C-element, and a simple pulse generator. Each tracking cell approximates the propagation delay of a particular critical path—it is key to identifying dominant critical paths. The tracking cells attain accurate tracking and optimal performance by incorporating the same types of gates, signal wires, and loading that is present in the corresponding critical path.9 At the beginning of each clock cycle, the processor triggers the tracking cells and immediately forces those tracking cells not required for the current clock cycle to their next state. Thus, the clock cycle depends on only active critical paths. Each tracking cell provides an input to a common C-element. Once all tracking cells complete, the C-element transitions to the next state. This transition restarts the tracking cells and activates the pulse generator. Figure 5 shows the dynamic clocking structure. Though the Strip follows a synchronous design methodology, it can, like asynchronous processors, still benefit from favorable environmental conditions such as temperature and voltage.

External interfacing is performed by the bus interface unit. Although the Strip architecture communicates internally with a global dynamic clock, the external interface operates on a four-phase, dual-rail protocol. It has been implemented in this manner to support efficient communication with devices of different operating speeds.

In the Strip architecture, an instruction cannot change the processor's state until the write-back stage. Therefore, the Strip architecture supports precise interrupts. When an interrupt occurs, pipeline instructions are not allowed to complete, regardless of where the interrupt occurred. Furthermore, the program counter is immediately set to zero to begin the exception handling, and the processor saves the addresses of the register fetch and instruction decode, ALU, and data memory access stages. Recovering from the interrupt requires restarting the instructions that occupied these three stages.

In addition to dynamic clocking, Strip's overall performance depends on memory access time. By removing the memory access time from the critical logic path, the true benefit of dynamic clocking can be realized. Otherwise, the instruction memory access would always be the dominant critical path. If one critical path is always dominant, then dynamic clocking provides little improvement over a synchronous architecture. To minimize the memory access latency, this processor places small memory buffers in each of the instruction and data memory paths. Each buffer holds 256 bytes and is reported to have one-half the access latency of the 8-KByte first-level cache. Obviously, a buffer of only 256 bytes will have a high miss rate, which nullifies reduced latency. The processor circumvents the high miss rate, however, by implementing predictive prefetching from the first-level cache. With predictive prefetching, the designers determined a miss rate of 12.75 percent.

Overall, predictive prefetching produced the same memory performance as a pipelined cache scheme, but predictive prefetching does not increase the load and branch penalty.

The designers determined the processor's performance through Spice analysis of the dynamic clocking structure, and they determined overall system performance by simulating the clock cycle created by individual tracking cells. They combined these cycle times with the percentage of clock cycles during which the particular critical path is dominant.

The end result is a weighted average that describes the effective clock frequency. For a 2-mm CMOS process, the designers reported a 63-MHz effective clock frequency and a 62.5-MIPS performance rating, based on measuring average instruction speed.

Amulet1

The Amulet1 processor is the first fully functional asynchronous processor and thus is the most fully developed of those we surveyed. Amulet1 is a code-compatible asynchronous version of the Advanced RISC Machine (ARM) processor developed by Steve Furber and his colleagues at the University of Manchester. It uses a two-phase communication protocol, with bounded-delay data transmission [2].

The RISC-like processor comprises the address interface, register bank, execution unit, and data interface. In addition, the processor supports two levels of interrupts and supports the exceptions generated by a virtual memory system.

Because Amulet1's asynchronous nature prohibits a known correlation between the current program counter value and that of the instruction entering the execution stage, the Amulet1 processor keeps a record of the program counter values in a FIFO structure, called a pc pipe. As each instruction enters the execute unit, the pc pipe contains the program counter value for that instruction at the top of the FIFO. If needed, this value is transferred along with the instruction to the execute unit. Otherwise, the value is discarded.

The register bank consists of 30 general-purpose, 32-bit-wide registers. Only 16 are accessible at one time; the 15th register contains the program counter. The register bank provides two read ports and one write port. The Amulet1 eliminates register hazards with a mechanism called a lock FIFO. Each word of the FIFO stores the register destination of a pending write. Since the Amulet1 completes all instructions in order, the FIFO structure maintains the proper order of multiple outstanding writes. An operand can be tested for "pending write" by examining the lock FIFO. RAW data hazards are eliminated.

The execution unit has three stages. For non-multiply instructions, the asynchronous operation of pipeline stages lets operands simply pass through the multiplier stage, with none of the complex bypassing required in synchronous designs. When a multiply instruction is encountered, however, the multiplier will produce the partial product and carries, which are added in the ALU stage.

Though the multiply stage will terminate execution as soon as the input operands allow, the stage may be active for several cycles, thereby stalling subsequent instructions. The final execution unit (ALU) stage executes all remaining logic and arithmetic functions.

Lastly, both the execution unit and the data interface write results to the “write bus,” but because the units are not synchronized with each other, the processor arbitrates access to the write bus.

The designers measured the Amulet1’s performance at 9K Dhrystones. The Dhrystone test suite, which approximates real programs, is a better, more realistic test measurement than a simplistic MIPS rating.

IV. COMPARISON

Figure 1 compares the performance of five of the surveyed processors.

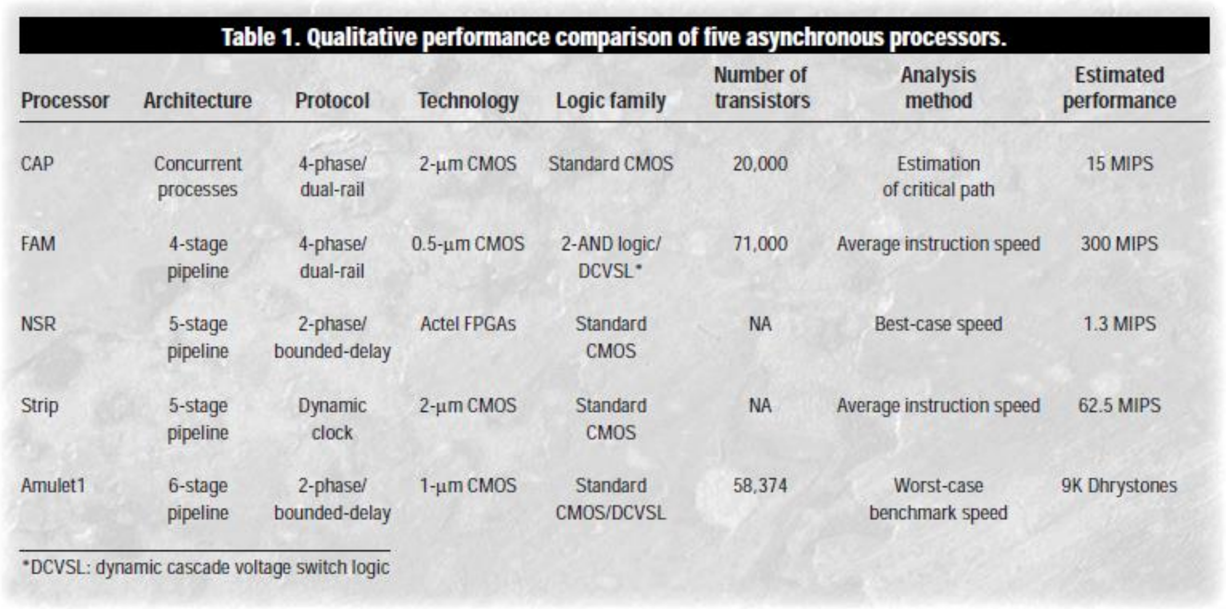

Table 1. Qualitative performance comparison of five asynchronous processors.

Processor	Architecture	Protocol	Technology	Logic family	Number of transistors	Analysis method	Estimated performance
CAP	Concurrent processes	4-phase/ dual-rail	2-µm CMOS	Standard CMOS	20,000	Estimation of critical path	15 MIPS
FAM	4-stage pipeline	4-phase/ dual-rail	0.5-µm CMOS	2-AND logic/ DCVSL*	71,000	Average instruction speed	300 MIPS
NSR	5-stage pipeline	2-phase/ bounded-delay	Actel FPGAs	Standard CMOS	NA	Best-case speed	1.3 MIPS
Strip	5-stage pipeline	Dynamic clock	2-µm CMOS	Standard CMOS	NA	Average instruction speed	62.5 MIPS
Amulet1	6-stage pipeline	2-phase/ bounded-delay	1-µm CMOS	Standard CMOS/DCVSL	58,374	Worst-case benchmark speed	9K Dhrystones

*DCVSL: dynamic cascade voltage switch logic

Figure 1. Performance comparison of five asynchronous processors

All the processors employ a pipelined architecture to some degree. Ignoring the Strip processor, which implements dynamic clocking, the communication protocol is evenly split between two-phase/bounded delay schemes and four-phase/dual-rail schemes. Except for the NSR processor, all processors were implemented with CMOS; however, the line geometries, which directly influence circuit speed, varied from 0.5 to 2 microns. The variations in the implemented functionality and analysis method prohibited a direct quantitative performance comparison, so we discuss performance in qualitative terms. Of the five, only Amulet1 was fully functional.

We focus chiefly on the FAM, NSR, and Amulet1. Though the CAP proved that an asynchronous processor was possible, the underlying architecture has no real potential. Although simulations show that the Strip processor will operate twice as fast as an equivalent synchronous processor where both are built with the same technology,9 the architecture is susceptible to global clock skew problems and doesn’t offer any savings with power dissipation.

Of the FAM, NSR, and Amulet1, only the FAM uses a four-phase communication protocol, yet its reported propagation delay of only 3.5 ns far exceeds the performance of the other two processors. Admittedly, the FAM is an incomplete design and it has the advantage of 0.5-micron line geometries; however, it challenges the perception that four-phase is inherently slower than two-phase. One reason that four-phase is not inherently slower is that a “bubble” must exist for data to move through an asynchronous pipeline. For a four-phase system, the request and acknowledge signals returning to the inactive state is analogous to the creation of this bubble. Furthermore, two-phase control structures are largely implemented with C-elements and exclusive-OR gates, both of which are slower than the AND/OR gates making up the four-phase control structures.

The NSR, FAM, and Amulet1 configurations are similar because of the FIFO structures existing between computation blocks. The FAM and Amulet1 have only a single register between computation blocks, which can be regarded as a FIFO of depth one. The NSR utilizes the FIFO structures to avoid stalling the entire pipeline by just a single slow instruction. By incorporating the FIFOs between the pipeline stages, the NSR continues to process instructions, storing intermediate results in the FIFOs until the slow instruction can be completed. The NSR architecture avoids the excessive branch penalties and load latencies that would otherwise be created by the FIFOs.

The FAM and Amulet1 use the FIFO for a slightly different reason. Both utilize dynamic logic for computation blocks and require the register to store data while the computation block is pre-charged in preparation for the next calculation. The major difference is that the dynamic logic, used by the Amulet1 processor, suffers from charge leakage.

The Amulet1 requires that the output latch, where the calculation will be stored, be empty before the computation begins. This method removes any risk that the result may become invalid due to charge leakage before it is loaded into the output latch.

In contrast, the FAM uses a dynamic logic family that does not suffer from charge leakage. Though the elimination of the charge leakage likely increases the gate delay, the control logic and the fact that the computation can begin before the output latch is empty allow the FAM to expedite processing.

The FAM architecture is essentially a compromise between the NSR and Amulet1. The FAM lets all computation stages be active yet it contains an identical number of latches between computation stages as the Amulet1. Thus, a slow instruction in the FAM pipeline will still allow some useful work to complete in preceding stages of the pipeline without increasing the branch penalty or load latency.

V. CONCLUSIONS

Though asynchronous processors may not match the performance of synchronous processors now, the condition generating the research into asynchronous processors will grow more prevalent as device geometries continue to shrink. In the meantime, processors may follow a locally synchronous, globally asynchronous approach where individual functional units use a local clock signal but are asynchronous with other functional units on the circuit die. The problem with clock distribution is thereby minimized while the processor retains the advantages of a synchronous system. One possible approach may perform instruction encoding and issue asynchronously, but the instructions themselves will be distributed to synchronous execution units.

In general, asynchronous methodology may be beneficial to those functions that are simplistic to do sequentially but complex to do in parallel. Asynchronous methodology can exploit the simplicity provided by sequential computation while attaining performance benefits by beginning the next computation as soon as the previous one is completed, instead of having to wait for the next clock pulse.

VI. REFERENCES

[1] C. V. Ramamoorthy and G. S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Transactions on Software Engineering*, Vols. SE-6, no. 5, pp. 440-449, 1980.

[2] T. Werner and V. Akella, "Asynchronous Processor Survey," *Computer*, vol. 30, no. 11, pp. 67-76, 1997.