Asynchronous Processor Survey

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.

Tony Werner
Venkatesh Akella
University of California, Davis

Virtually all computers today are synchronous, thanks to an internal timing device that regulates processing. As systems grow increasingly large and complex, however, this little device—a clock—can cause big problems with clock skew, a timing delay that can create havoc with the overall design. It can also increase the circuit silicon and power dissipation, which can affect overheating and power supplies.

In seeking to overcome such limitations, computer architecture researchers are actively considering asynchronous processor design. By their very nature, for example, asynchronous architectures permit modular design. Each subsystem or functional block can be optimized without being synchronized to a global clock, which simplifies interfacing. Moreover, an asynchronous system exhibits the average performance of all the individual components, rather than the synchronous system's worst-case performance of a single component. Furthermore, asynchronous processors may yet prove to offer reduced power dissipation by inherently shutting down unused portions of the circuit.

This article examines the key architecture issues that concern designers and compares six developmental asynchronous architectures.

ARCHITECTURE

Computer architecture researchers evaluate key areas—including pipeline organization, instruction issue, branching, and exception handling—when considering asynchronous and synchronous design and implementation trade-offs. Our discussions concern only pipelined, scalar, load/store machines, such as Silicon Graphics' MIPS R3000.

Pipeline organization

Pipelines—the means by which instructions are physically processed—typically include one or more stages for each instruction fetch, decode, execution, memory access, and register update function. Instructions do not necessarily pass through all stages. For instance, a branch instruction requires neither memory access nor a register update stage, nor possibly the ALU (arithmetic logic unit) stage if the processor has the hardware to support branch instructions.

A branch instruction can be discarded after the instruction is decoded, although doing so creates a *bubble* (a nonutilized stage in the pipeline). Synchronous pipelines eliminate the bubble by adding hardware that "fills" each available stage on the next clock pulse, to achieve effective data throughput. Asynchronous pipelines, on the other hand, actually depend on bubbles for effective throughput: Because they consider these stages to be unoccupied, ensuing instructions fill the stage, and there is no need for additional hardware. Improving pipeline performance generally involves adding depth by creating multiple, simpler pipeline stages. This has a twofold impact on clock skew in synchronous systems: First, the additional stages increase the capacitive loading on the global clock signal, thereby increasing clock skew. Second, the simpler pipeline stages permit a faster clock cycle, which further exacerbates the clock skew problem.

Asynchronous systems enjoy improved performance through increased pipeline depth but without the problems associated with clock skew.

Instruction issue

Instructions are usually processed in a certain order: fetch, operand access, and either computation or memory access. The processor then posts the result back to the register file. Thus an instruction that requires the result of an earlier instruction can potentially read an operand before the processor posts the result, a data hazard called read-after-write (RAW). A data hazard is a type of processing error that can lead to incorrect results. In processors where instructions are allowed to complete out of order, write-after-read (WAR) and write-after-write (WAW) data hazards are also possible. However, in a processor with in-order completion, WAR and WAW data hazards are avoided because the operands are always read or written before succeeding instructions write.

In synchronous processors with in-order completion, designers can avoid RAW hazards either by inserting blank instructions, called NOPS (for "no operations"), between the independent instructions or by stalling the instruction until the hazard is eliminated. Asynchronous processors, on the other hand, easily eliminate RAW hazards because, after each

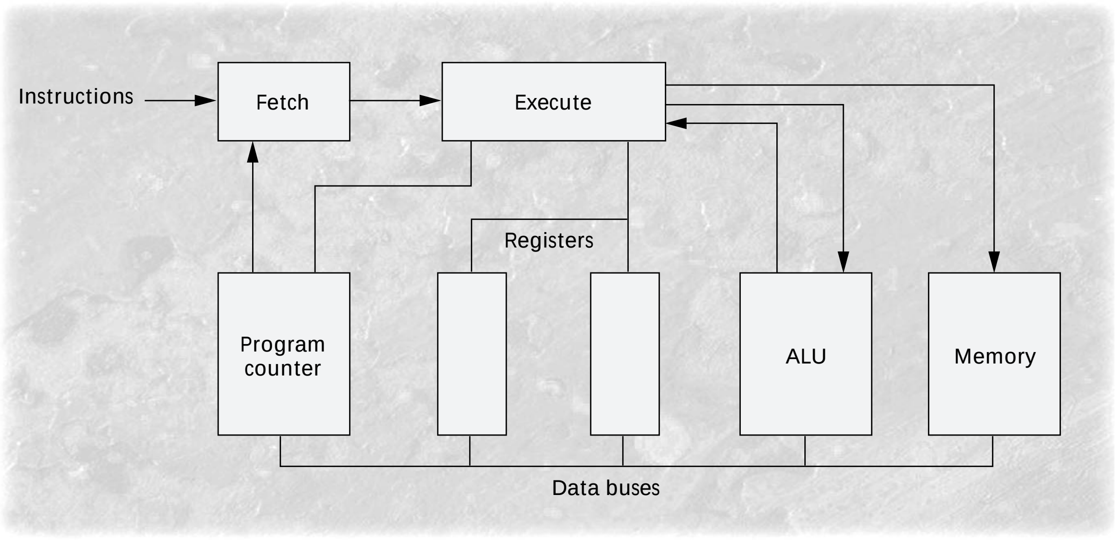

Figure 1. The Caltech Asynchronous Processor structure.

instruction is decoded, the processor sets a flag on the destination register. Subsequent instructions that depend on this result will stall during operand access. The inherent nature of asynchronous communication protocols makes it straightforward to stall the instruction: Because the communication protocol requires an acknowledgment before the instruction can continue, the register suspends any acknowledgment until the flag is removed. Asynchronous systems require no additional hardware or blank instructions.

Branching

Branch instructions alter program flow. Processors typically predict the outcome of branch instructions to continue instruction processing. A misprediction incurs a branch penalty, which is the number of processing cycles needed for the processor to recover from the mispredicted branch. Designers can use one of five techniques to reduce pipeline branch penalties: locking the pipeline, predict not-taken, predict taken, a branch prediction algorithm, or delay slots.[1]

Generally speaking, synchronous and asynchronous processors handle branches similarly, with comparable implementation complexity. Both processors can use these techniques.

When a processor encounters a branch instruction, it predicts the direction of the branch and then fetches and executes additional instructions in the direction of the prediction. If the prediction turns out to be incorrect, the processor must identify which instructions in the pipeline are associated with the wrong prediction.

In a synchronous system the branch delay is fixed and known, so the processor simply invalidates all pipeline stages that contain instructions beyond the mispredicted branch. In an asynchronous system, processors do not know how many instructions were fetched after the mispredicted branch and therefore cannot distinguish invalid from valid instructions.

One technique to solve this problem is to include a bit with each instruction that represents a color.[2] All instructions in the pipeline have the same color until a branch instruction is encountered, at which point a color bit is toggled and subsequent instructions have a different color. If the branch direction is mispredicted, instructions of the wrong color are effectively removed from the pipeline.

Exception or interrupt handling

Exceptions and interrupts are a fact of processor life. Hardware interrupts occur at random with internal control operations of the processor and, therefore, pose a metastability problem for both synchronous and asynchronous processors.

Typically, the metastability is nearly eliminated for hardware interrupts in synchronous systems through a series of flip-flops that the global clock regulates. The possibility for metastability, however, is never entirely eliminated from synchronous systems. In fact, the statistical possibility increases with clock frequency.

An asynchronous processor arbitrates between the hardware interrupts and the instruction fetch procedure. In general, asynchronous systems arbitrate between any two uncorrelated signals. Arbitration—resolving the signals—is a common procedure in asynchronous systems and has specialized circuits for performing the function. Thus, asynchronous processors inherently eliminate problems with metastability caused by hardware interrupts or any other unsynchronized signals.

ASYNCHRONOUS PROCESSORS

To spur more research in asynchronous processing, we reviewed six distinctly different asynchronous processors—all that existed at the time we surveyed them, actually—and report our findings here. 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.

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.[3] 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.

Figure 1 shows the processor's basic structure, which resembles a limited pipeline approach. 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.[4]

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,[5] shown in Figure 2. FAM uses a four-phase communication protocol with dual-rail data transfer.

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.

FAM's pipeline stages are instruction fetch, memory, instruction decode, and instruction execution. Note that the fourth stage includes both the ALU and the register file: In many RISC architectures, the ALU and the register file are accessed separately. FAM uses a combined instruction and data cache; therefore, the external cache memory is accessed by both the first stage, for instruction acquisition, and the fourth stage, for load/store instruction completion. Thus, the processor requires arbitration to resolve any memory conflicts that may occur between the two stages.

The FAM design employs both computation and interconnection blocks.[5] An interconnection block separates each pipeline stage or computation block and provides intermediate storage to decouple the computation blocks, which simplifies their control.

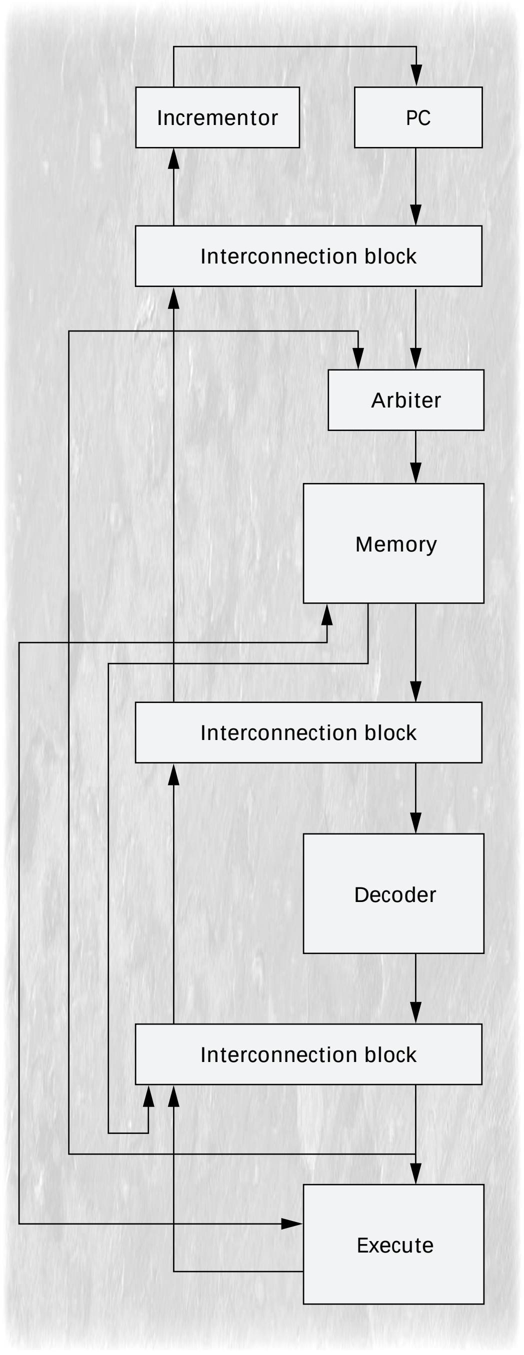

Figure 2. The pipelined architecture of the Fully Asynchronous Microprocessor.

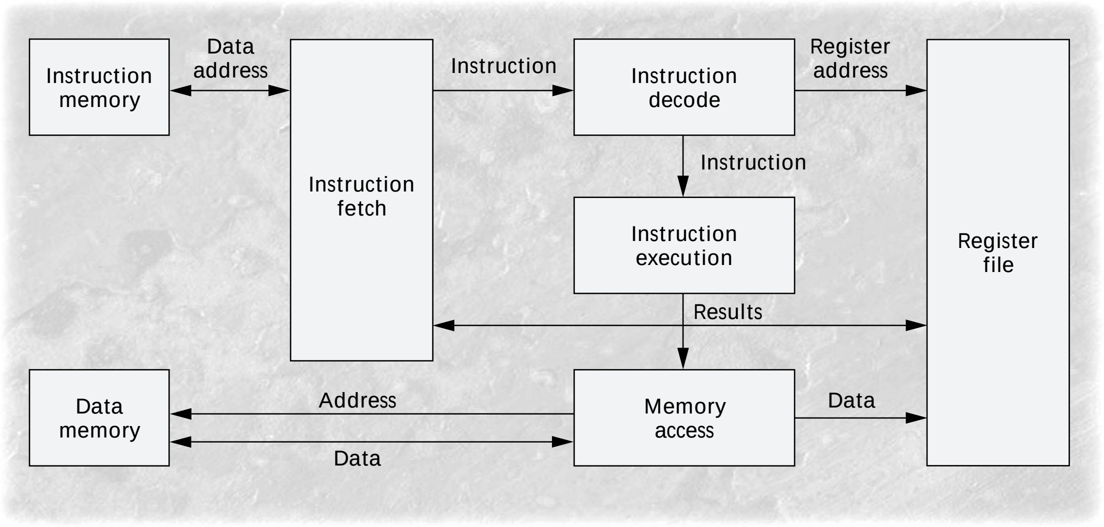

Figure 3. Nonsynchronous RISC architecture.

FAM development focused on 2-AND logic,[6] 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.[5]

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 even then the FAM provides only 18 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.[7] In the NSR, shown in Figure 3, 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.

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. For example, NSR implements only 16 load/store-type instructions and contains only sixteen 16-bit registers. 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.[8] 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.

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.

Figure 4 shows a sample CFPP configuration.

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. For example, suppose a dependent memory instruction follows an `add` instruction. In the basic, identical-stage configuration, the compiler would not have to reschedule independent instructions between these two instructions. Simply, after the `add` completes, the results would be sent backward to the awaiting `store` instruction, which is assumed to be close behind in the pipeline. However, this instruction stream would stall the processor significantly in the configuration shown in Figure 4. The `store` instruction would be halted in the fourth stage of the processor while the `add` continues to propagate to the ninth stage. Some time later, the results of `add` are delivered to the 11th stage.

The processor now has six empty stages between the Add Recover stage and the Memory Launch stage.

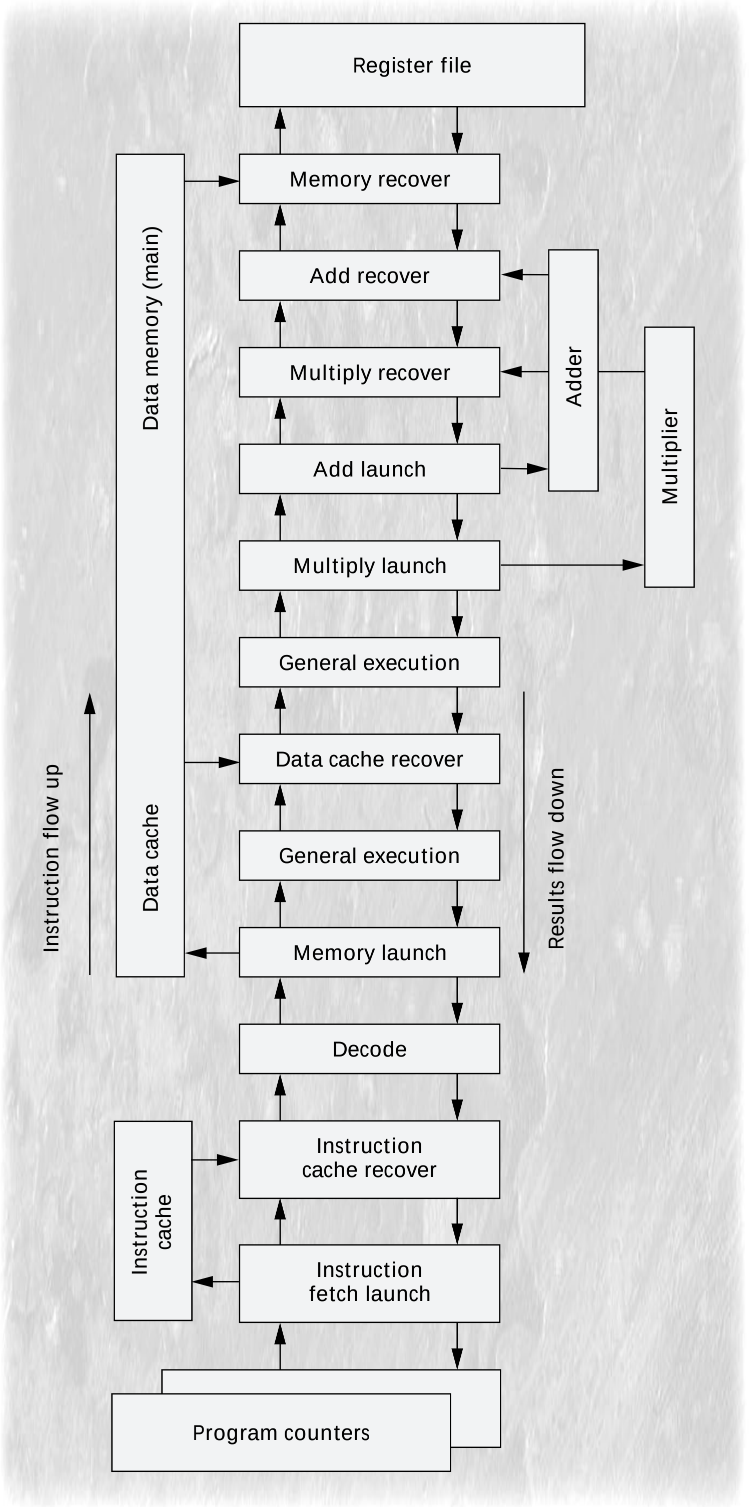

Figure 4. A Counterflow Pipeline Processor with auxiliary sidings to handle long-latency instructions.

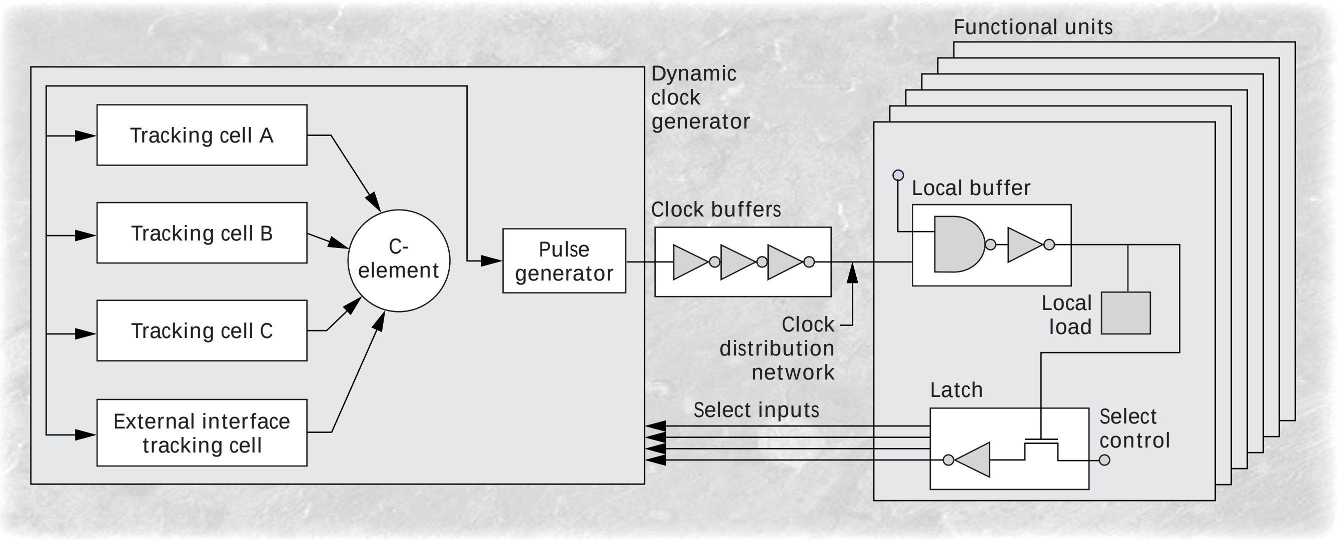

Figure 5. Dynamic clocking structure of the Strip processor.

Furthermore, when results from `add` enter the pipeline, they must propagate through these six stages to reach the awaiting `store` instruction. Meanwhile, the processor has been stalled behind `store` while `add` propagates to the Add Launch stage, executes, and the results propagate back through six stages to the Memory Launch stage. Arbitration is required between each stage, further slowing the results propagation. The compiler can attempt to reschedule independent instructions between these two instructions, but statistically, it is unlikely that the compiler will locate even two independent instructions.[1]

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,[9] 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.

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

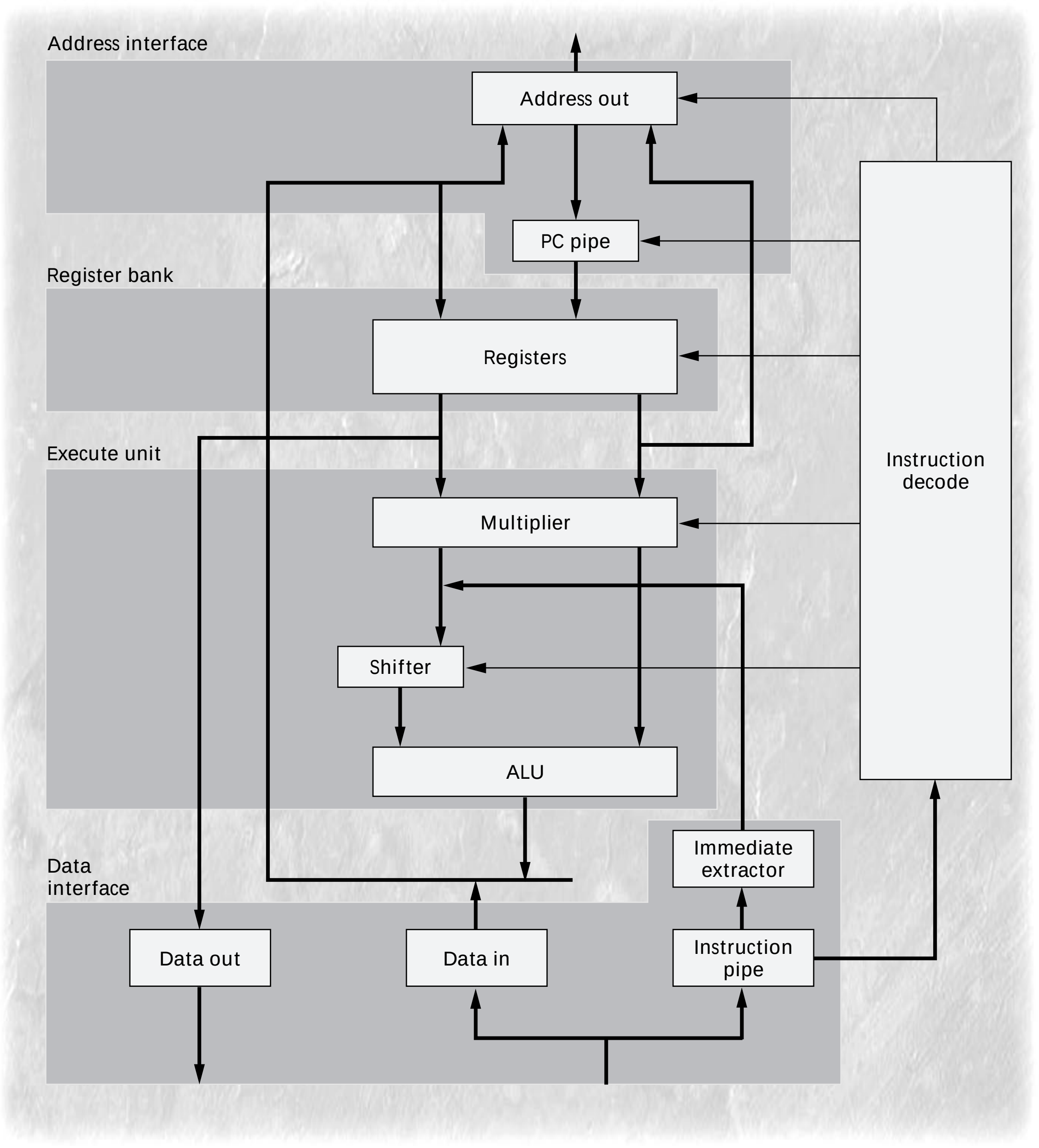

Figure 6. Amulet1 processor organization.

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-μm 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, shown in Figure 6,[2] 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

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

University of Manchester. It uses a two-phase communication protocol, with bounded-delay data transmission.

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 nonmultiply 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 register bank and execute unit utilize dynamic logic to reduce physical size and transistor count.[2] Because of charge leakage present in the dynamic structures, a latch between each stage stores the result. The intermediate latches serve as an output latch of one stage for storing the result and as the input latch of the subsequent stage. Because of charge leakage, the output latch of a stage must be empty before the computation begins. Also, the input stage must maintain the data throughout the computation. Consequently, every active stage requires control of both the input and output latches. At best, only half of the pipeline can be active at any one time.

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.

COMPARISON

Table 1 compares the performance of five of the processors we surveyed. We did not include CFPP because, as a proposed architecture, no simulations were performed. At the time of our survey, however, it had created quite a stir in the asynchronous community.

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 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 precharged in preparation for the next calculation. The major difference is that the dynamic logic, used by the Amulet1 processor, suffers from charge leakage.[2]

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.

A slow instruction in the Amulet1, such as a `multiply` instruction or a data cache miss, stalls two pipeline stages (the stage with the slow instruction and the preceding stage, which is waiting for the output latch that is currently occupied by the slow instruction). If the Amulet1 applied the NSR's FIFO strategy, it could avoid this pipeline stall, at least until the FIFO became full. Unlike the NSR, however, the Amulet1's application of this method would produce excessive branch penalties and load latencies.

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.

Newer processors built since we concluded our survey are the Titac[10] and Fred,[11] which had its roots in the NSR. For more information on Titac, see http://www.hal.rcast.u-tokyo.ac.jp/titac2/index.html. To read more about Fred, see http://www.cs.utah.edu/~willrich/async/mypapers.html.

The growing interest in this field is encouraging. Several universities are actively investigating new asynchronous architectures, and both Sun Microsystems and Intel have started asynchronous research groups. Progress has been slow thus far, as asynchronous processors have significant obstacles to overcome. Nonetheless, as an example, Amulet1's latest successor, Amulet 2e, has achieved 66K Dhrystones with a 0.5-micron process. This represents a 30 percent improvement over Amulet1 after factoring in

Several universities are investigating new asynchronous architectures, and both Sun Microsystems and Intel have started asynchronous research groups.

process effects. To read about Amulet, see http://www.cs.man.ac.uk/amulet/index.html.

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. Data hazard detection in a superscalar processor is a possible application; another is decoding variable-length data words, such as those found with Huffman decoding. ❖

Acknowledgment

We thank Erik Brunvand for his comments and assistance.

References

1. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, Calif., 1990.
2. N.C. Paver, *The Design and Implementation of an Asynchronous Microprocessor*, doctoral dissertation, Dept. of Computer Science, Univ. of Manchester, 1994.
3. A.J. Martin et al., *The Design of an Asynchronous Microprocessor*, Tech. Report Caltech-CS-TR-89-2, California Inst. of Technology, Pasadena, Calif., 1989.
4. J.A. Tierno et al., “A 100 MIPS GaAs Asynchronous Microprocessor,” *IEEE Design and Test of Computers*, Summer 1994, pp. 43-49.
5. K.-R. Cho, K. Okura, and K. Asada, “Design of a 32-bit Fully Asynchronous Microprocessor (FAM),” *Proc. 35th Midwest Symp. Circuits and Systems*, IEEE Press, Piscataway, N.J., 1992, pp. 1,500-1,503.
6. K.-Rok Cho and K. Asada, “VLSI Oriented Design Method of Asynchronous Sequential Circuits Based on One-Hot State Code and Two-Transistor AND Logic,” *Proc. Int’l Symp. Computers and Systems*, IEEE Press, Piscataway, N.J., 1991, pp. 1,793-1,796.
7. E. Brunvand, “The NSR Processor,” *Proc. 26th Hawaii Int’l Conf. System Sciences*, Vol. 1, T.N. Mudge, V. Milutinovic, and L. Hunter, eds., IEEE Press, Piscataway, N.J., 1993, pp. 428-435.
8. R.F. Sproull, I.E. Sutherland, and C.A. Molnar, *Counterflow Pipeline Processor Architecture*, Tech. Report SMLI TR-94-25, Sun Microsystems Laboratories, Mountain View, Calif., Apr. 1994.
9. M.E. Dean, *STRIP: A Self-Timed RISC Processor*, Tech. Report CSL-TR-92-543, Stanford Univ., Stanford, Calif., July 1992.
10. T. Nanya et al., “TITAC: Design of a Quasi-Delay-Insensitive Microprocessor,” *IEEE Design and Test of Computers*, Summer 1994, pp. 50-53.
11. W.F. Richardson and E. Brunvand, “Fred: An Architecture for a Self-Timed Decoupled Computer,” *Proc. Second Int’l Symp. Advanced Research in Asynchronous Circuits and Systems*, IEEE Press, Piscataway, N.J., 1996, pp. 60-68.

Tony Werner is a researcher in the Semiconductor Research Laboratory in the R&D Division of Hitachi America Ltd. His research interests include RISC architecture, dynamic instruction scheduling, and asynchronous logic design. Werner received a BS in computer engineering from the University of Illinois at Urbana-Champaign and an MS in electrical engineering from the University of Arizona. He is a PhD candidate in curriculum at the University of California, Davis.

Venkatesh Akella is an assistant professor of electrical and computer engineering at the University of California, Davis. His research interests are self-timed logic, computer architecture, and software engineering. He received an MS in electrical and computer engineering from the Indian Institute of Science at Bangalore and a PhD in computer science from the University of Utah. He received the National Science Foundation Research Initiation Award in 1993 and the Faculty Early Career Development (CAREER) award in 1997.

Contact Werner at Semiconductor Research Laboratory, Hitachi America Ltd., Research and Development Division, 201 E. Tasman Dr., San Jose, CA 95134; twerner@halsrl.com. Akella can be reached at Asynchronous Systems Research Group, Dept. of Electrical and Computer Eng., University of California, Davis, CA, 95616; akella@eecs. ucdavis.edu