Simultaneous Multi-processor Cores for Efficient Embedded Applications

Earle Jennings
CTO, QSigma, Inc., Sunnyvale, CA 94089, USA, US Citizen.

Corresponding author. Tel.: +1 510 292 8328; email: earle.jennings@qsigmainc.com
Manuscript submitted April 15, 2017; accepted July 20, 2017.
doi: 10.17706/jcp.13.4.371-382

Abstract: This paper introduces Simultaneous Multi-Processor (SMP) cores. These SMP cores offer a high performance, efficient application target for the embedded system developer. SMP cores can be reprogrammed like a microprocessor in response to application requirement changes. They do not require caching, or superscalar instruction processing, greatly reducing silicon size and energy consumption. Also the power to any unused resources is gated off each clock cycle. This new class of instruction processors is discussed and shown through a core architecture implementing multiple simultaneous processes. This approach solves an inherent problem in VLIW instruction processing, giving the advantages of VLIW, while dramatically reducing instruction memories, and eliminating the need for instruction caching. Examples are given of the simultaneous processes of multiple threads. Merging these processes is shown. The SMP cores achieve the effect of superscalar instruction processing and multi-thread control, through a compile time procedure, without any additional hardware.

Key words: Caches, embedded controllers, SOC, superscalar microprocessors.

1. Background

Today, an embedded application developer chooses from three implementation targets, scalar microprocessors, superscalar microprocessors, or customized IP cores tailored to a specific purpose. Customized IP cores are efficient because they leave out components unnecessary for their purpose. However, significant redesign is needed for them to meet changing application requirements, delaying product entry into the market. Scalar microprocessors have limited performance, but can be reprogrammed as application requirements change. Superscalar microprocessors improve the performance of microprocessor applications through a combination of caching and superscalar instruction interpretation, with multi-thread control. However, both caches and super-scalar mechanisms require a large silicon and energy overhead.

Today's computer architectures are derived from the von Neumann architecture first deployed in the 1950's [1]. The von Neumann architecture implements a central processing unit (CPU), which operates a program counter. An instruction is fetched, based upon the program counter, by accessing a location in a memory. The CPU responds to the fetched instruction by translating it into some internal sequence of states, generally referred to as executing the instruction. The program counter may be altered, and the CPU repeats the process of fetching and executing instructions. Three early computers, built on this architecture [2], pioneered features commonly incorporated into today's microprocessors.

These three computers are the IBM 360, with its use of caching, the VAX-11, with its multi-tasking and virtual memory environment, and the Pentium, as representative of superscalar microprocessors. The IBM 360 introduced caches as a way to interface slow, but large, memories to the CPU. The VAX-11 successfully ran many different programs on the same CPU during a small time interval, where each program could pretend that it ran in a huge memory space. A superscalar microprocessor interprets an intermediate language of a simpler architecture, such as the 80486, or the Arm 7, into smaller (pico) instructions [3]. The pico-instructions are scheduled into streams that simultaneously operate data processing resources, such as floating point arithmetic units, at far higher performance than the intermediate language supports. All of these innovations make for better general-purpose computers. However, they are inefficient in addressing the situation of high performance computers (HPC) [4], the power requirements for Digital Signal Processing (DSP) circuits, and the requirements for System On a Chip (SOC) components [5]. The Simultaneous Multi-Processor (SMP) core research results are applicable to DSP and HPC, however this paper focuses on embedded cores for SOC.

2. A New Class of Instruction Processors

Fig. 1 shows, on the left, executing an Amdahl compliant algorithm [6] with a parallel part (PP) and a sequential part (SP). Shown on the right is a Simultaneous Multi-Processor (SMP) core, implementing a process state calculator simultaneously issuing at least two process state indexes for executing multiple simultaneous processes. Each simultaneous process separately owns the instructed resources of the core. Each owned instructed resource includes its own local instruction processor, which simultaneously responds to the process state of its owning process, to generate a local instruction that directs the instructed resource's operation as part of the owning process. Data processing resources, such as a data memory port, an adder, and so on, are called instructed resources. Each process owns separate instructed resources so that the Parallel Part (PP) and the Sequential Part (SP) need not stall each other. Owning a resource means that one, and only one, process within a task stimulates its instruction processing with its process state.

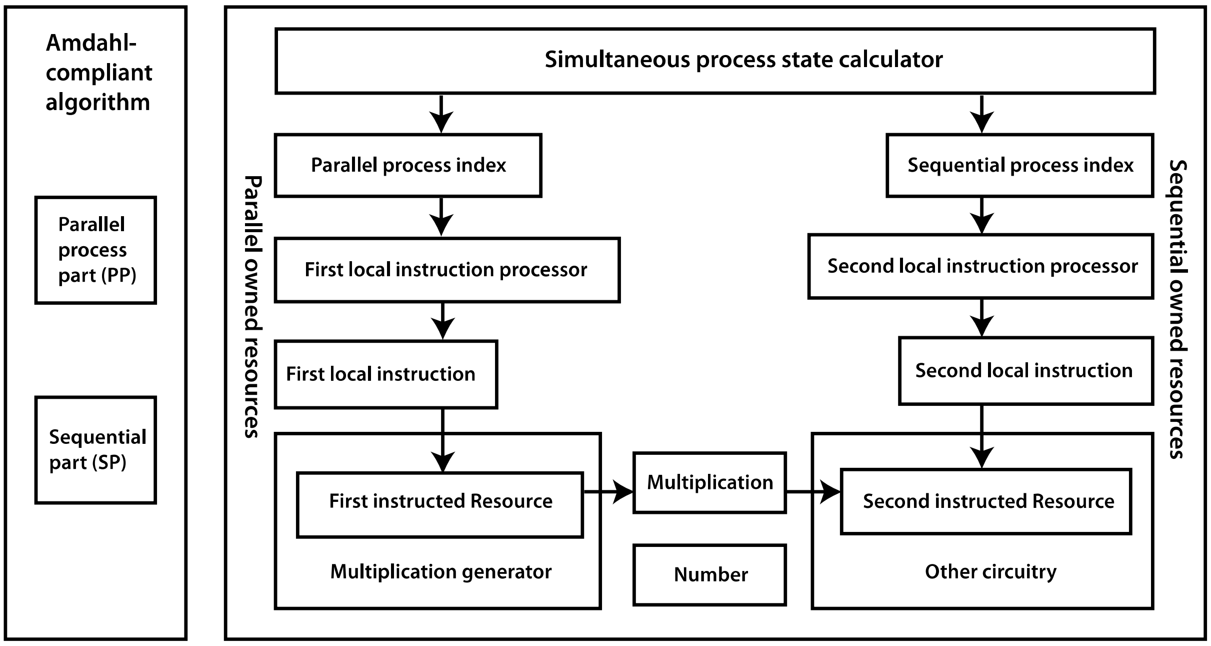

Amdahl-compliant algorithm
Parallel process part (PP)
Sequential part (SP)
Simultaneous process state calculator
Parallel process index
Sequential process index
Parallel owned resources
Sequential owned resources
First local instruction processor
Second local instruction processor
First local instruction
Second local instruction
First instructed Resource
Multiplication
Second instructed Resource
Multiplication generator
Number
Other circuitry

Fig. 1. An amdahl-compliant algorithm and its components implemented by the SMP core.

A SMP core program designates the resources owned by the specific simultaneous processes of a task. A process state calculator issues a process index for each of the simultaneous processes. Local resources performing data processing, memory access, I/O and feedback are each owned and operated by specific processes, or are not used at all by that task. Ownership may vary for different tasks, but within one task is fixed.

2.1. Benefits of the SMP Core

The SMP core simultaneously performs both processes PP and SP as shown in Fig. 2. This is compared to the conventional computer (scalar microprocessor) that may execute, at most, one of the processes at a time.

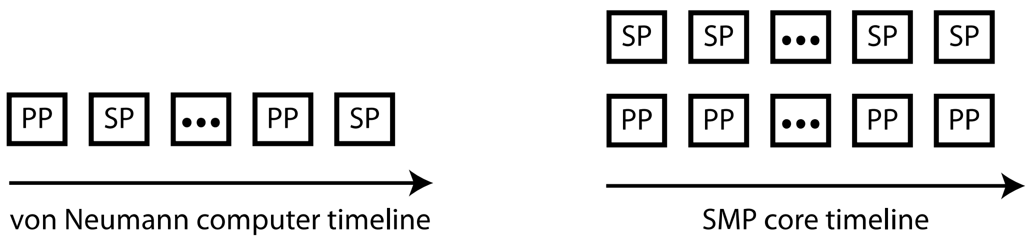


Fig. 2. Timeline of von Neumann computer compared to SMP core.

Before superscalar microprocessors, there was an inherent problem with conventional microprocessors. They did not give fine enough control of each data processing resource. This fundamentally limited their performance. In the 1990's two approaches were considered, Very Long Instruction Word (VLIW) machines and superscalar microprocessors. The VLIW machines had very detailed instructions for each resource, but required very long, and deep, instruction memories to be effective, which increased the instruction memory demands. Consider a third alternative, the SMP core. Assume that the PP and SP processes each have a range of 8 process states (instructions). This new core is driven by separately accessible, process-owned local instructions, shown in Fig. 3.

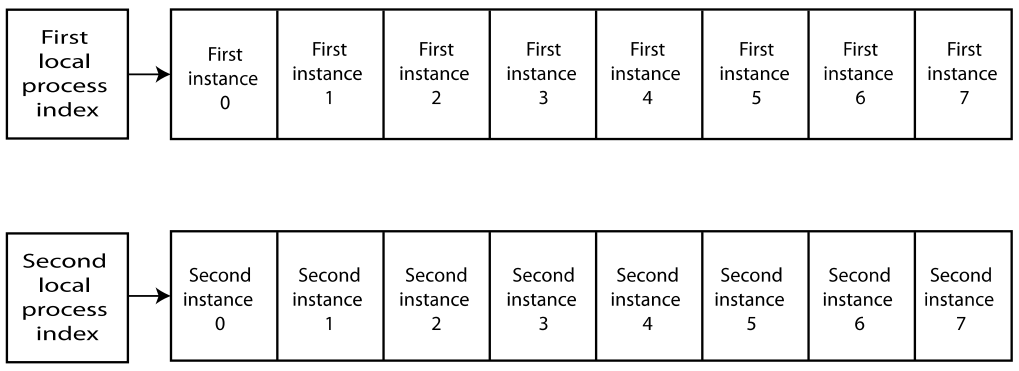
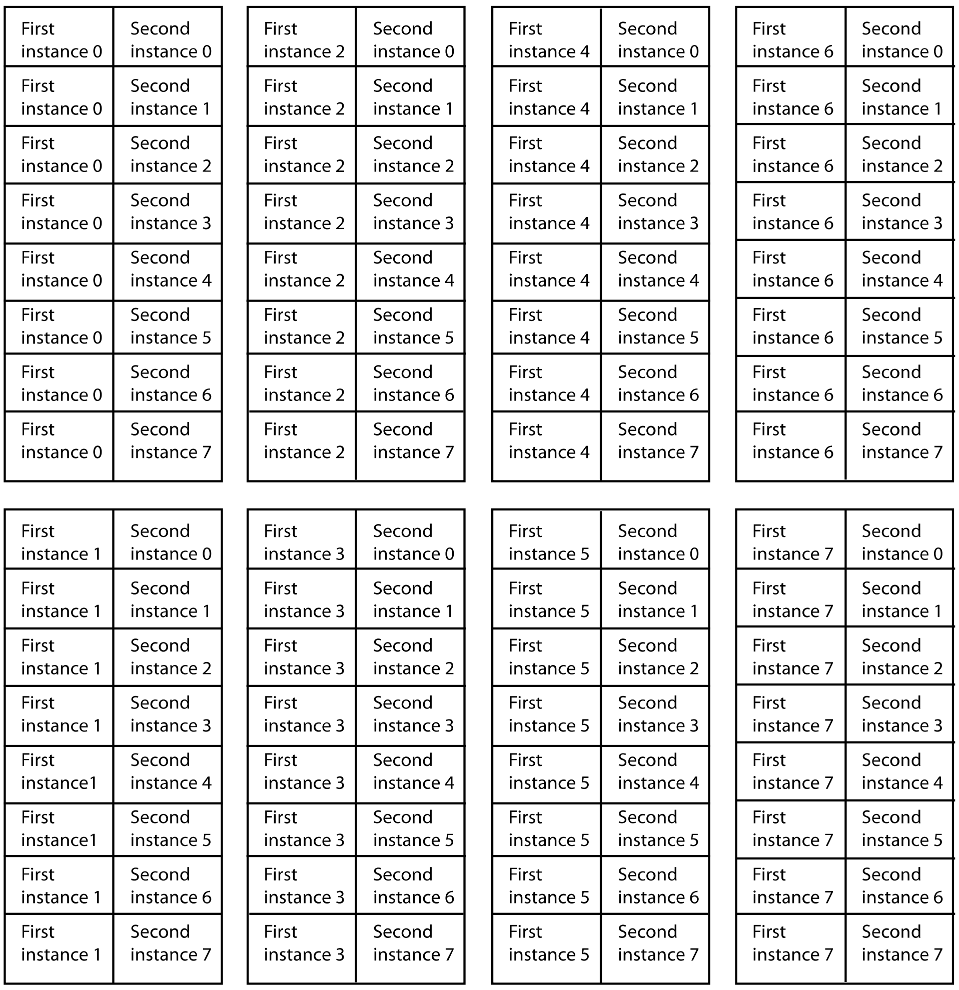


Fig. 3. Local, process-owned instruction memories range over eight instructions.

A VLIW instruction memory supporting these same independent operations requires a much larger VLIW memory of 64 instructions, as shown in Fig. 4. The simultaneous processes, and the local instructions for their owned instructed resources, remove this otherwise required large VLIW memory. They also remove the need for instruction caching. Starting from the core, the sequential part and parallel parts of the conventional computer become the simultaneous processes, and incorporate the advantages of three new features. First, all feedback is external to arithmetic components, such as the FP adders. The operation of accumulating feedback is triggered by the state of the feedback queues. This feedback scheme supports an alternative to FP multiply-accumulate operations, which runs at the speed of the multiplier, without concern for how the adders are implemented, or the latency of the adders. Second, the adders are extended to support comparisons with the winning input operand, and its index, sent as the adder output. Winners may be the maximum, or the minimum, as specified by the program. Third, communication between the parallel part and the sequential part is through feedback, with the feedback queue status triggering actions in the receiving process.



First instance 0	Second instance 0	First instance 2	Second instance 0	First instance 4	Second instance 0	First instance 6	Second instance 0
First instance 0	Second instance 1	First instance 2	Second instance 1	First instance 4	Second instance 1	First instance 6	Second instance 1
First instance 0	Second instance 2	First instance 2	Second instance 2	First instance 4	Second instance 2	First instance 6	Second instance 2
First instance 0	Second instance 3	First instance 2	Second instance 3	First instance 4	Second instance 3	First instance 6	Second instance 3
First instance 0	Second instance 4	First instance 2	Second instance 4	First instance 4	Second instance 4	First instance 6	Second instance 4
First instance 0	Second instance 5	First instance 2	Second instance 5	First instance 4	Second instance 5	First instance 6	Second instance 5
First instance 0	Second instance 6	First instance 2	Second instance 6	First instance 4	Second instance 6	First instance 6	Second instance 6
First instance 0	Second instance 7	First instance 2	Second instance 7	First instance 4	Second instance 7	First instance 6	Second instance 7

First instance 1	Second instance 0	First instance 3	Second instance 0	First instance 5	Second instance 0	First instance 7	Second instance 0
First instance 1	Second instance 1	First instance 3	Second instance 1	First instance 5	Second instance 1	First instance 7	Second instance 1
First instance 1	Second instance 2	First instance 3	Second instance 2	First instance 5	Second instance 2	First instance 7	Second instance 2
First instance 1	Second instance 3	First instance 3	Second instance 3	First instance 5	Second instance 3	First instance 7	Second instance 3
First instance1	Second instance 4	First instance 3	Second instance 4	First instance 5	Second instance 4	First instance 7	Second instance 4
First instance1	Second instance 5	First instance 3	Second instance 5	First instance 5	Second instance 5	First instance 7	Second instance 5
First instance 1	Second instance 6	First instance 3	Second instance 6	First instance 5	Second instance 6	First instance 7	Second instance 6
First instance 1	Second instance 7	First instance 3	Second instance 7	First instance 5	Second instance 7	First instance 7	Second instance 7

Fig. 4. VLIW instruction memory accessed by a single instruction pointer.

2.2. Block Diagram of the Core

Fig. 5 shows a block diagram of a core that includes a multiplier and an instruction pipeline organized into five instruction pipe stages. The execution wave front passes through the instruction pipe stages in a fixed pattern. Each instruction pipe includes one or more pipe stages. The latency for the execution wave front to traverse instruction pipes may differ. The process state calculator is in pipe 0. Each process operates based upon a process index, and possibly loop output(s), by generating an instruction at each instructed resource to perform the execution wave front as it passes through that resource. Assume that up to four simultaneous processes can execute in each SMP core as shown.

Feedback paths do not go through the arithmetic. Instead, feedback is in separate hardware with a consistent status structure used to trigger process state changes based upon data availability. This allows for a simple, consistent software notation that controls all computing actions based upon when the data is available, whether it is from a local arithmetic resource, or across a computer floor of several hundred cabinets. All the power for the next execution wave front is gated off, if no operations are to be performed.

The core is shown executing four simultaneous processes by generating four process indexes that each drive instruction processing for the instructed resources owned by one of these processes. Each instructed resource is instructed by a local instruction generated in response to the process index of the owning simultaneous process. Both the parallelizable and sequential parts may be implemented as simultaneous processes that do not stall each other as they execute.

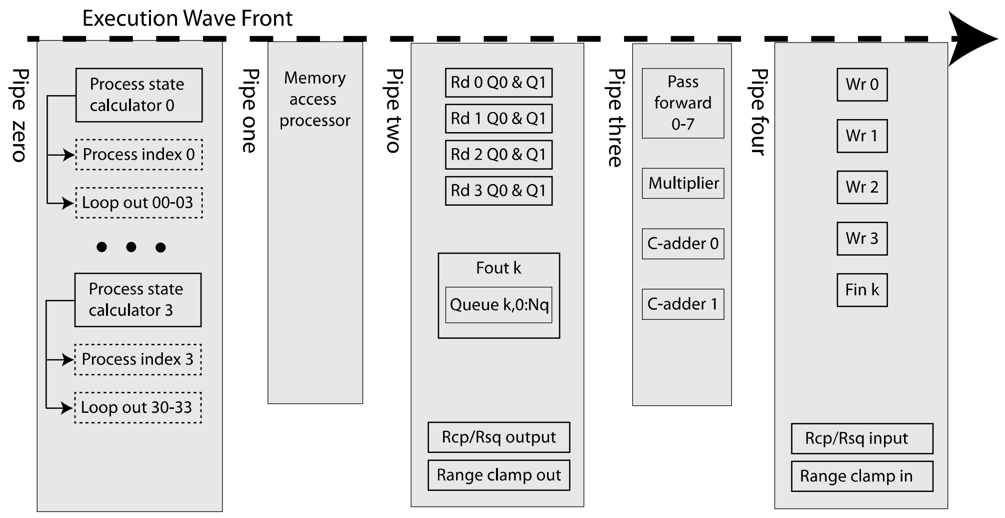


Fig. 5. SMP core block diagram.

Locally generated instructions selected from multiple process indexes insure operational diversity in controlling the resources, while minimizing instruction redundancy. Matrix inversion by Gaussian elimination requires less than 24 local instructions. Large external VLIW memories and instruction caches are not required, greatly improving energy efficiency.

The execution wave front replaces a traditional buss and provides substantial benefits. Multiple queues in a single feedback output port enable a hierarchical response to data availability, allowing a single adder to act like a cascading adder network for accumulation in FIRs and dot products, as well as pivot entry calculation in matrix inversion and LU decomposition. All of these algorithms are implemented so that the multiplications do not stall, independent of core clock frequency, or the number of pipe stages in the arithmetic circuits. The other circuitry keeps up with the multiplications providing maximum performance, at the least energy cost for the required operations.

2.3. The Flat Time Execution Model

The flat time execution model is a semantically accurate portrayal of the performance of the execution wave front through the instruction pipes. The flat time execution model is designed to reveal all the relevant states of each execution wave front as it traverses the core. This helps the application developer quickly diagnose and fix programs. The flat time model shows the process ownership within each instruction pipe, whether the instructed resource is used, and if used, presents the following: The local performed instruction, the selected inputs, any operations performed upon the inputs to create the operands used in the resource's data processing, and if applicable, the output from the resource. The data memory read ports do not have data inputs, but have read addresses, which are shown. Similarly, data memory write ports have address inputs and data inputs, but have no outputs. This not only simplifies the programming, but also optimizes concurrency and task switching characteristics. The execution wave front insures all data results coming out of the instruction pipe are based on data that went into the instruction pipe with the execution wave front. Further simplicity results from requiring the inputs of each instruction pipe to come from the outputs of the previous instruction pipe.

2.4. Power Management and Monitoring

Fig. 6 shows the process state calculator generating a usage vector for each of the processes that indicates which instructed resources are owned, and used, by a process on this execution wave front.

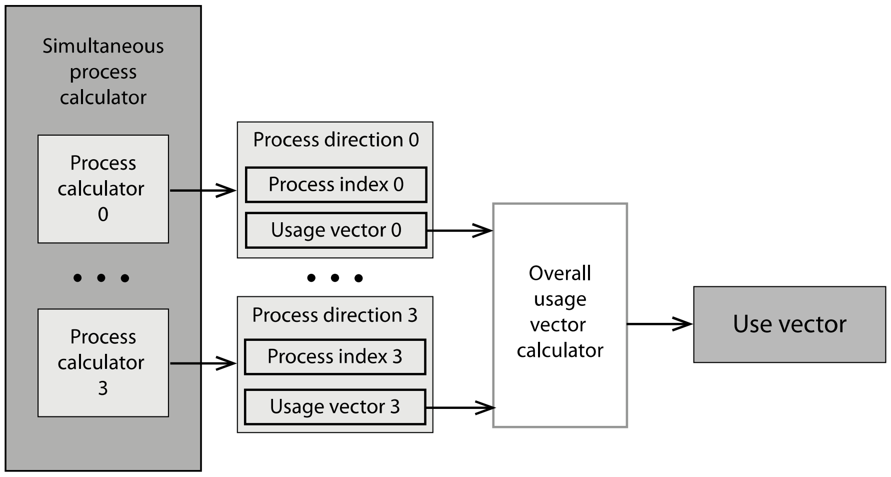


Fig. 6. A simultaneous process state calculator generates the usage vector.

Fig. 7 assumes the power domain of an instructed resource is CMOS-like logic and power technology. One component of the usage vector, Use bit is shown driving a power gate. A gated resource power is generated by the power gate in response to the Use bit of the usage vector. The instructed resource uses the gated resource power as the execution wave front traverses the instructed resource. When the use bit is off, the instructed resource does not consume power, When on, it does. Some implementations may gate the clock to effect control of the power.

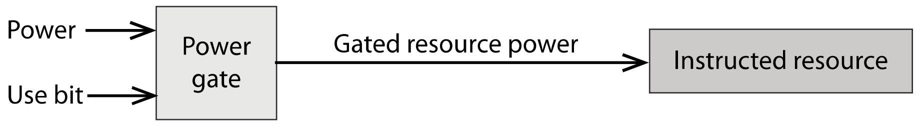


Fig. 7. Gating off power to an instructed resource based upon the usage vector.

3. Programming the SMP Core

Programming is all about defining the simultaneous processes required to operate the SMP core. A thread is one, or more, simultaneous processes benefiting from being in one core. The first step in defining the processes of each thread is to define what instructed resources are owned by each of the processes. These owned resources may include input queues, feedback queues, memory read port queues, arithmetic units, feedback portals, output portals, and memory write ports. The SMP core contains two adders. To simplify programming, both adders can perform the same operations. These include an inline comparison that may be chained, without branching, to calculate the pivot for matrix inversion, or the maximum, or minimum, of a vector or matrix.

Four threads are commonly found in application programs in real-time systems such as Systems On a Chip

(SoC). The first is a vector dot product, the second is a Finite Impulse Response (FIR) filter, the third is a Fast Fourier Transform (FFT) and the fourth calculates the maximum, or minimum, of a vector. In these examples, assume that the arithmetic is in single precision, floating point.

3.1. Vector Dot Products and Overcoming the Myth of Multiply-Accumulate

A SMP core performs the vector dot product on two vectors acting as inputs. They may either reside in the core, or be presented as streams to the core. In most standard implementations of dot products, some form of multiply-accumulate is used [7]. There are two problems with this approach. First, the feedback of the adder is needed for accumulating the next multiplication. In high speed (at least 100 MHz clock cycles), the floating point adders require at least two pipe stages, and accumulation cannot occur more often than once every two clock cycles. The multiplier has to wait for the accumulated result and therefore stalls at least 50% of the time. Second, each accumulation incurs a rounding error. The rounding error bounds grow linearly with the number of products being summed. Both of these problems are addressed by proper configuration of the SMP core. In the SMP core, the multiplier receives the two corresponding vector components and generates the vector product component, which is fed back from the multiplier to a product queue feeding a C-adder (see Fig. 5, Pipe 3). The C-adder can operate on three operands in each execution wave front. Rather than accumulate one product every clock cycle as in multiply-accumulate, it more efficiently adds three products together to make a first level sum. The first level sums are fed back to a first level sum queue, which has a higher priority trigger. When there are enough elements in this queue to initiate an execution wave front using three first level sums, the C-adder operates on these three sums to create a second level sum of products. The C-adders output incorporates up to nine products into one entry. This entry is fed back into a second level sum queue, and so on. This thread resolves both dot product problems. First, the multiplier is not stalling and the adder is keeping up, rather than stalling the multiplier until the adder can finish. Second, one rounding error is accumulated for each level of sums, so that the rounding error bound is on the order of ceiling(log3(N)) * ½ LSB, as opposed to the conventional bound for multiply-accumulates of O(N) * ½ LSB.

3.2. FIR Filters

Assume for the moment that the FIR has n tap coefficients $a_i, i = 0, \ldots, n-1$ acting upon a sample stream B_j. The output of the filter is $out_i = \sum_{j=0}^{n-1} a_j B_{j+i}$. Because the taps and a buffer providing a window onto the data are required, pointer roll over must be supported to associate the correct tap with the correct data. Conventionally microprocessors use application coding to do this. However, the memory processors of the SMP cores are required to perform these pointer operations so that the multiplications never stall.

3.3. FFTs

FFT's are very common vector functions in signal processing. There are two implementation issues targeting the SMP cores. First, FFT's require addressing a data store in two distinct modes. First, a bit-flipped address needs to be generated. This flips the top bit and the bottom bit, the first from the top and the first from the bottom, and so on. Second, more standard addressing occurs in all but the early pass of an FFT. Bit-flipping addressing and standard addressing are commonly handled in applications coding. However, the memory processor of the SMP cores fulfills this address generation requirement internally, removing this burden from the application coding. Second, FFT's perform complex number multiplications, which are accumulated as complex number temporary values, to generate each step operating on the input (or previous pass) data. This can naturally and efficiently be performed with the three operand C-adders.

3.4. Finding a Maximum (or Minimum) of a Vector

Traditionally, the maximum (or minimum) of two numbers is iteratively formed, by subtracting one entry from a running maximum/minimum tally. The tally is initialized to the first entry. Using the arithmetic operation's sign result (less than 0 or greater than 0), either the old running tally or the current entry is selected as the new tally. This approach is inherently inefficient, because each comparison step must flush the adder pipes and then assemble the result. In the SMP core, two or three operands can be compared and their result generated in one execution wave front, sending these results back through a first level feedback queue. When enough (two or more) entries are in this queue, it's status triggers an execution wave front that takes the top two, or three, partial comparisons and generates a second level comparison, which is then fed back to a second level comparison queue, and so on. There is no dependence upon flushing the adder's pipes and then subsequently building the comparison result. Rather than performing this thread in about O(n)* (adder_pipe_latency + comparison result construction), the result is generated in at most O(n)+ ceiling(O(log3(n))) * (adder_pipe_latency+feedback_overhead).

4. Multi-threading as a Compile Time Operation

Table 1. Input Processes of the Threads

Instructed resource	Dot product input process	FIR filter input process	FFT input process	Calculate maximum
In queues	Dot-A, Dot-B	FIR-In	FFT-In	
Feedback queues				
Memory read queue		Tap-Read, FIR-in	FFT-coef read FFT-pass data	
Multiplier	Yes	Yes	Yes	
C-adder 0				
C-adder 1				
Memory write ports		FIR-in	FFT-in	
Feedback in	Product Fdbk In	FIR product 1 term	FFT product 1 term	
Output portal				

Table 2. Accumulation Processes of the Threads

Instructed resource	Dot product accumulate process	FIR filter accumulate process	FFT accumulate process	Calculate maximum process
In queues				Max-in queue
Feedback queues	Product fdbk 1, Dot accum 1 to n			Max-fdbk 2 to max-n
Memory read queue		Tap-Read, FIR-in	FFT-coef read FFT-pass data	
Multiplier				
C-adder 0	Yes	Yes	Yes	Yes
C-adder 1				
Memory write ports		FIR-write accumulate	FFT-in	
Feedback in	Dot accumulate fdbk in	FIR accumulate	FFT accumulate	CMax fdbk in
Output portal	Yes	Yes	Yes	Yes

The four threads discussed above, each require no more than two SMP processes. One process performs the multiplications and the other accumulates the additive results. For example, assume the following:

- The dot product vectors are each of length = 729 = 36. The dot product initial process generates all the dot product terms, A[i]*B[i] with one multiplication for each product, thereby requiring one execution wave front per input pair A[i] and B[i]. In this example, the vectors are not stored internally. These product terms are accumulated by feedback to a first queue. When three dot product terms are in the queue, the process state triggers output of these three terms, which are added in one C-adder, and then fed back through a second feedback path to a second queue for accumulation. Six queues, including the first queue for the dot product terms, are operated to accumulate the dot product.

- The FIR has n = 27 taps. FIR's first process performs n products for each output. With each new input, all n taps can be processed against that input to generate the incremental values needed for n successive outputs.
- The FFT performs a 1K complex FFT [8], which can be implemented as 5 radix 4 steps. Each radix 4 step inputs the partially processed data from the previous step, or the initial input as real numbers, each representing one of two components of a complex number. Each of these real numbers is multiplied by two coefficients to form the components needed for a complex number multiplication. Bit flipping occurs in the first radix step, otherwise the other steps do not use bit flipping.
- The vector being calculated for its maximum also has a length = 729 = 36. Calculating the maximum of a vector does not store the vector to simplify this discussion.

These four processes are commonly found in a variety of applications including statistical analysis [9].

The first three of these four threads has an input process as shown in Table 1.

Each of these four threads has an accumulation process as shown in Table 2.

4.1. Multi-threading

The four threads discussed above include three input SMP processes and four accumulation SMP processes. These threads illustrate corresponding processes that can be merged. Their arithmetic resources are the same, and their use of queues is non-overlapping. For example, a merged thread composed of the four threads discussed above operates in one SMP core as two merged processes. At this point, the real-time system requirements become relevant. Two examples involving the same merged thread in the SMP core illustrate this. In a first real-time system, all input processes are continuously receiving data on every clock cycle. The merged input process cannot keep up with the inputs. In a second real-time system, the four threads need to generate their outputs once every millisecond, and the SOC operates with a 400 MHz clock. The merged threads can efficiently operate within one SMP core, requiring only the energy needed for the calculations. In SMP cores, management of multi-thread mergers is the responsibility of the programmer using compile-time tools.

4.1.1. Merging the input processes

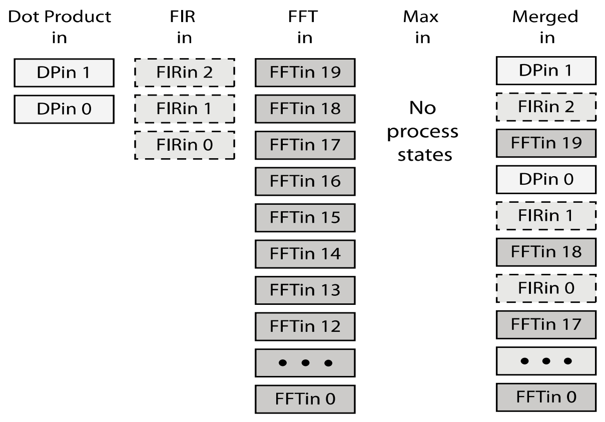


Fig. 8. Input processes of three threads merged into one process.

Fig. 8 shows the individual process lists on the left and the merged process list on the right. Assume that the input processes have the following process states: the vector dot product has two, the FIR has three, and the FFT has twenty. These process states are ordered so the most probable process states have lowest priority. This insures that the process states with the least probability have the highest priority, which is shown as the top process state of their process. To merge these threads requires merging these separate process state lists into a single process state list. The merged process resource ownership is the union of the owned resources of the merged processes.

4.1.2. Merging the accumulation processes

Fig. 9 shows the individual accumulation process lists on the left and the merged accumulation process list on the right. These processes have the following process states: the vector dot product has seven, the FIR has three, the FFT has four, and calculating the maximum has eight.

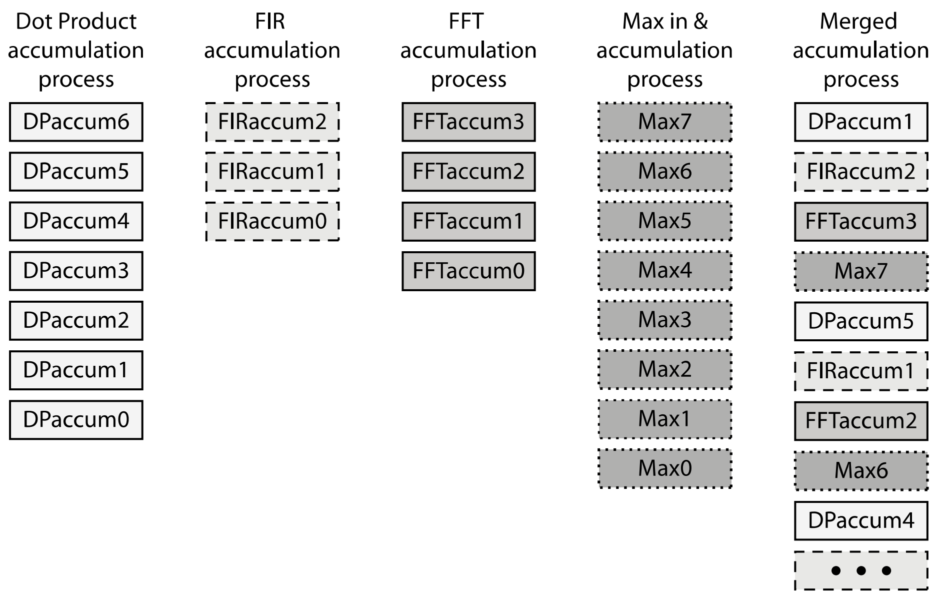


Fig. 9. The accumulation processes of four threads merged into one process.

5. Conclusion

Simultaneous Multi-Processor (SMP) cores offer a new, high performance, efficient target for the embedded application developer. SMP cores can be reprogrammed like a microprocessor in response to application requirement changes [10], [11]. These cores have the advantages of VLIW architecture, but require only a small fraction of the instruction space. Simultaneous processes of the programmed core provide the performance of the superscalar microprocessor, without the runtime hardware overhead of superscalar instruction interpretation. Multi-threading is under the application developers' control and also has no runtime hardware overhead. The execution wave front replaces a traditional buss. The flat-time model of the execution wave front revolutionizes, by simplifying, program code development. This model reveals all the relevant information the code developer needs to verify their program through the execution wave front's traversal of the instruction pipes.

The next research goal is to insure that any program for an existing superscalar microprocessor can be correctly implemented in these SMP cores. This is achieved by considering the known good superscalar microprocessor, and it's Intellectual Property (IP) [12]-[14], as a prototype for the compile-time tools, and the target SMP cores. This IP includes models of the data processing unit and the superscalar instruction interpreter. It also includes the verification and test sets confirming the microprocessor for industrial release. Compile-time tools can be generated from the hardware scheduler of the superscalar instruction interpreter. This utility will collect the threads of an application program for later merging and placement. These SMP cores can be generated in a series of steps that prove, by construction, that the operational semantics of the microprocessor's data processor are preserved. The data processor will be organized into the instruction pipes. The data processor is then augmented through a correct-by-construction insertion of queues. This creates the initial SMP cores. These cores can be proven to perform correctly as the target of programs, by using the existing microprocessor verification and test set. Next, these initial SMP cores can be semantically extended to account for more than two operand additions, comparisons, non-linear operations, etc.. These can be inserted into a subsequent SMP core to extend its performance and efficiency.

Acknowledgment

The author thanks Heather Murphree for her inspired support and feedback since 1984. The author also thanks George Landers and Lavelle Gibson for the pleasure and honor of working with them since 1991.

References

[1] Goldstine, B., & Neumann, V. (1946). Preliminary discussion of the logical design of an electronic computing instrument. *John von Neumann Collected Works, MacMillan Co., 5,* 34-79.

[2] Patterson, D., & Hennessey, J. (1990). *Computer Architecture: A Quantitative Approach.* Morgan Kauffmann Publisher.

[3] Johnson, W. (1991). *Superscalar Microprocessor Design, P. T. R.* Prentice Hall.

[4] Lucas, R. (2014). *Top Ten Exascale Research Challenges DOE ASCAC Subcommittee Report*. Sponsored by the US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research.

[5] Shalf, J. (2013). Computer architecture for the next decade. *Proceedings of EEHPC Workshop*.

[6] Amdahl, G. (1967). The validity of the single processor approach to achieving large-scale computing capabilities. *Proceedings of AFIPS Spring Joint Computer Conference* (pp. 483–85).

[7] Muller, J. (2006). *Elementary Functions: Algorithms and Implementation* (2nd ed.). Boston: Birkhauser.

[8] Van Loan, C. (1992). *Computational Frameworks for the Fast Fourier Transform.* Philadephia: SIAM.

[9] Rizzo, M. L. (2008). *Statistical Computing with R.* Chapman and Hall.

[10] Jennings, E. (2010). Breaking Amdahl's law by changing the execution architecture. Retrieved from http://www.qsigmainc.com/?page_id=99

[11] Jennings, E. (2016). Introduction to an integrated exascale computing architecture: Part one. Retrieved from http://www.qsigmainc.com/?page_id=99

[12] Perry, D. (1994). *VHDL* (2nd ed.). New York: McGraw-Hill, Inc.

[13] Mishra, K. (2013). Advanced chip design: Practical examples in verilog. Retrieved from www.amazon.com

[14] Bhasker, J. (2004). *A SystemC Primer* (2nd ed.). Allentown: Star Galaxy Publishing.

Earle Jennings is a member of IAENG. He was born in Bozeman, Montana and began designing computers at the age of 14. His first design was a relay based computer made of salvaged parts from a local telephone switching station. He received the bachelor's degree in mathematics from Thomas Edison University in 1989 and the master's degree in applied mathematics from the University of Texas at Dallas in 1996. He passed the US patent and trademark office patent registration examination in 1999 to become a patent agent.

He designed the first solid state disk drive to interface to a micoprocessor in 1980 when he was the lead designer at Pullman Software Systems in Pullman, Washington. Currently, He is the chief technical officer of QSigma, Inc., a start-up committed to researching fundamental solutions to fundamental problems, in Sunnyvale, CA. He is a patent agent who operates a patent legal practice and also an associate of bell and associates, LLP, of San Francisco, CA. Further employment details may be found on his linkedin.com page. His research interests include compiler related topics, parallel processors, digital signal processing, real-time systems, communications related topics, VLSI, FPGA, programmable logic, computer arithmetic, and instruction processing. His compiler related topics include circuitry compilation, merged syntactic and semantic processing, code generation and optimization, intermediate languages, and interpreters. His communications related topics include wireline and wireless protocols, routers, access points, error correcting codes, fault detection, recovery and resilience, networks in a chip and messaging protocols for simultaneous multi-processor cores and modules of these cores.

Jennings is a member of the institute for electric and electronic engineers (IEEE), the association of computing machinery (ACM), the society for industrial and applied mathematics (SIAM), the American society of mechanical engineers (ASME), and the national association of patent practitioners (NAPP). Patents listing him as an inventor can be provided upon request. Many of his relevant writings can be found either on the linkedin page or the QSigma website. Jennings has written hundreds of US patent applications and is an expert in international patent law.