New Degrees of Parallelism in Complex SOCs

MIPS Technologies, Inc.
July 2002

New more feature-rich electronic devices appear everyday, while existing devices continue to evolve and converge into more elaborate ones. There remains, however, a strong impetus to integrate these functions onto as few silicon chips as possible, which is driving SOC complexity up at a staggering rate. Moore's Law provides the ammunition to meet this challenge, but bringing the power of semiconductor technology to bear on SOC performance in the future will require different architectural approaches than those used in the past. The traditional techniques of pipelining and superscalar instruction issue are approaching their limits and other techniques like vector processing, multithreading, and chip multiprocessing will be called on to carry the ball.

SOC Complexity Skyrockets

As the marketplace continues to become more adept with technology, companies are making each new electronic device more sophisticated than the last. Every new generation has more features and higher throughput. Convergence is compounding this growth, resulting in some truly elaborate devices.

Still, the benefit of integrating functions onto fewer silicon chips remains as compelling as ever. The result is a staggering growth rate of SOC complexity. While Moore's Law continues to provide the transistors necessary to construct these incredibly complex devices with reasonable manufacturing costs, some major changes are afoot.

One force driving change is the increasing need to be earlier to market. New device markets have steeper growth rates than those in the past and products are becoming obsolete much more quickly. Just a couple of months one way or the other in today's markets can make the difference between success and failure, profit and loss.

A second force at play is the rapid growth in development costs. Design productivity is not keeping pace with device complexity or the semiconductor technology enabling it, so design costs are rising. Worse, the fixed-costs of putting an SOC into production are rising even faster. By some estimates, 90nm mask costs will exceed one million dollars per set. In 1995 on 0.35µm technology, design and mask costs usually amounted to less than 15% of the costs of a 250K-unit run of product; in 2003 in 90nm technology, an equivalent design might see design and mask costs approach 65%.

Driving Performance

Rising development costs motivate companies to design fewer SOCs, but to make each one more flexible and programmable. Doing so allows designs to be reused to take advantage of economies of scale and shorten the average time it takes to get a device to market. Moreover, programmability will allow companies to keep products in the market longer, boosting integrated profits.

For programmability to be an option, however, embedded-processor cores must deliver the type and level of performance needed to implement functions that today require hardwired logic blocks or specialized (difficult to program) processors. Delivering this level of performance in a cost-effective, power-efficient, easy-to-program processor will require different architectures and techniques than those commonly used today.

While the functions implemented in SOCs are diverse and dependent on the particular application domain the SOC will serve, there are some general trends cutting broadly across many embedded markets. It is generally true, for example, that in most electronic devices digital signal processing and multimedia processing are becoming larger components of the overall workload. In fact, these tasks are to a large extent driving the architecture of embedded processors[1] and SOCs.

As these forces converge it will drive the demand for performance well beyond the 55% compound annual performance growth rate the industry has sustained over the past 30 years.

Current Techniques Fall Short

High-performance embedded processors have traditionally relied mainly on clock frequency and superscalar instruction issue to boost performance. Caches have played an important part in enabling the potential of these techniques in the face of increasing memory latencies. While frequency and superscalarity have served us well and will continue to be used, they have limitations that will limit the gains we can expect from them in the future.

The gains in operating frequencies, which have historically come at a rate of about 35% per year, are attributable to two factors, each of which has contributed roughly half the gains: semiconductor feature scaling and deeper pipelining.

The intrinsic switching delay of semiconductor devices ($\tau = CV/I$) has improved steadily at a rate of around 20% per year, due mostly to device scaling. Global interconnect delays have scaled less favorably, limiting overall chip gains to somewhat less. Over the next several years, the ITRS 2001[2] calls for intrinsic switching delays to improve at about 17% per year. Local interconnect performance will scale similarly, but global interconnects will lag, requiring circuit design tricks to keep pace.

The other half of the frequency gains have come mainly from deeper instruction pipelines, although improved design tools and circuit-design innovations have also contributed. Deeper pipelining distributes the work of executing an instruction over more discrete clock cycles, reducing the amount of work done each cycle and allowing the operating frequency to rise proportionally.

Pipelining comes at a cost. Deeper pipelines are less efficient. Each additional pipeline stage introduces overhead, to the tune of a couple of fan-out-four-equivalent (FO4) inverter delays per stage. Architectural efficiency also suffers. Deeper pipelines have long restart penalties on changes in control flow, requiring complex mechanisms such as branch predictors and speculative execution.

There are practical limits to how far pipelining can be pushed and still deliver a performance gain. Well-formed clock pulses are difficult to deliver below a period equivalent to about seven FO4 inverter delays. Dividing logic components, such as ALUs, into pieces smaller than about 15 FOV inverters ends up creating more work than can be recovered from the frequency gain. While embedded-processor pipelines are not yet being pushed as far as those in their PC-processor cousins, efficiency is still an issue, and it will limit the frequency gains we can expect from deeper pipelines in the long run.

It is an unfortunate misperception in the market that processor frequency is a reliable measure of performance and more is always better. But high frequency has downsides, whether from more aggressive circuit design or deeper pipelines. The main disadvantage is power dissipation: longer pipelines are less efficient, faster devices leak more current, and switching devices at a higher rate requires considerably more energy.

With high frequency designs, static power dissipation is high because fast (low-V_t) transistors have high subthreshold leakage characteristics. Dynamic power dissipation (CV^2F) is higher because switching capacitance (C) goes up with pipeline stages, frequency (F) goes up (obviously), and the supply voltage (V_{dd}) must be raised in proportion to frequency for the circuit to operate. Semiconductor scaling has in the past saved the day on dynamic power because each generation operates at lower V_{dd}. Voltage scaling, however, will slow as we approach the limits set by silicon physics. Moreover, lower supply voltages (V_{dd}) force switching thresholds (V_t) down, boosting static power.

Superscalar instruction issue will also approach limits. This technique introduces many of the same efficiency losses as pipelining, plus others relating to parallel dependency analysis. Instruction dispatch logic complexity goes up exponentially with issue width. Dynamic instruction reordering is often required to sidestep hazards and find enough instruction parallelism to fill the issue slots; complexity goes up with the depth of this instruction window. Extra pipeline stages may have to be added to prevent this complexity from reducing frequency, creating a snowball of complexity.

Regardless of the cleverness of a superscalar design, the technique is fundamentally constrained by the amount of instruction-level parallelism (ILP) in programs. If there is not much ILP available, superscalar efficiency will be low, leaving execution units under utilized. While theoretical studies show high ILP in some programs, others show precious little. Furthermore, much of the ILP discovered in these studies turns out to be data-level parallelism (DLP). If DLP is extracted by other less expensive means, which we will discuss later, the superscalar hardware can be left with little to chew on.

The biggest problem with superscalar techniques is the rapidly diminishing returns on transistor usage. While significant speedups are achieved in moving from a single-issue scalar design to a dual-issue design, very much less is gained by adding a third issue slot. Beyond four-issue, the gains are essentially nil. In combination with deep pipelining, which is the common usage, superscalar design, when pushed too far, results in very complex logic with very little to show for it.

Very-long-instruction-word (VLIW) machines were devised to reduce the complexity of issuing multiple instructions in parallel. VLIWs foist the task of instruction scheduling from hardware onto the compiler, thus—theoretically—mitigating the diminishing returns of superscalars. Results, however, have been disappointing. In practice, VLIWs end up requiring many of the same complex mechanisms as superscalars: branch predictors, forwarding paths, pipeline interlocks (e.g. scoreboards), etc. Plus, new mechanisms, such as predication, code compression, and rotating registers, have to be tacked on to compensate for the absence of dynamic instruction-scheduling facilities. VLIWs suffer badly on some types of code because compilers don't have the benefit of runtime information to efficiently schedule the machines resources. As a result, the complexity of VLIWs has not been convincingly lower than that of superscalars with similar performance.

New Techniques to the Rescue

Pipelining and superscalar techniques have proven effective as far as they go, and most embedded processors aren't yet pushing them to their limits. But high-end and mid-range embedded processors soon will if the demand for performance materializes as we expect.

Fortuitously, 90nm semiconductor technology will arrive in time to enable some new techniques to pick up where pipelining and superscalar techniques leave off. In 90nm technology, logic densities will approach 10^6 transistors/mm^2, and embedded SRAM densities will exceed 10^6 bits/mm^2. At these densities, the marginal manufacturing costs for a million logic transistors or a million bits of memory will be on the order of six cents ($0.06) on a nominally sized SOC in the range of 80mm^2.

Three techniques that will come to the fore are vector processing, multithreading, and chip multiprocessing (CMP). These techniques have two characteristics in common: they exploit different levels of parallelism than pipelining and superscalar issue, and they are transistor intensive. Unlike pipelining and superscalar techniques, which are extraordinarily complex, vectors, threading, and CMP are simpler, relying more on arrayed datapath elements than complex control structures.

Pipelining and superscalar techniques both exploit fine-grain instruction-level parallelism (ILP); pipelining exploits ILP by temporal means, superscalar by spatial means. Vector processing, in contrast, exploits fine-grain data-level parallelism (DLP); multithreading exploits medium-grain thread-level parallelism (TLP); and chip multiprocessing exploits coarse-grain process-level parallelism (PLP).

Multithreading

As previously pointed out, one limitation on ILP hardware is the lack of parallelism in a single instruction stream. Another impediment is the presence of long-latency operations, such as memory accesses. Both problems stall the instruction pipeline, reducing the average instruction issue rate and leading to under utilization of the execution units.

The idea behind multithreading is to exploit a higher-level of parallelism, thread-level parallelism, to keep execution units busy during times the pipeline would otherwise be stalled. A thread is an instruction stream that is data, memory, and control independent of other streams. The independence of threads can be exploited by switching to a new thread when one thread stalls.

Many forms of multithreading have been devised over the years. The coarsest (and oldest) form is simple interrupt-driven multitasking. This age-old technique has been enhanced over the years with various hardware structures to reduce granularity by lowering the overhead of switching threads. Indeed, the key to effective multithreading is lower thread-switch overhead, which allows more pipeline stalls to be covered.

The ultimate in fine-grain multithreading, called simultaneous multithreading[3] (SMT), was first described by Susan Eggers and Dean Tullsen at the University of Washington. In this approach, instructions from several threads are interleaved onto the machine's execution units. Such fine-grained threading makes it possible to cover latencies as short as those arising from even single-cycle pipeline hazards, eliminating many of the IPC-robbing bubbles typically found in superscalar pipelines.

All forms of hardware-assisted multithreading require the duplication of processor resources to hold the architectural state of each running thread within the core. While this might sound expensive, state- and context-holding resources typically amount to a small portion of the core's transistors.

Part of the beauty of SMT is that it uses facilities similar to the out-of-order execution facilities that many superscalar designs use to improve efficiency. SMT piggybacks easily on top of existing register-renaming and reorder-buffer facilities, adding only slightly to the complexity of a superscalar design.

Throughput improvements over single-thread superscalar implementations can be impressive. Results are dependent on the specifics of the underlying hardware and the workload, but investments of an additional 10% in hardware for a second thread can boost throughput on the order of 30%.

Moreover, because of its ability to fill issue slots and boost execution-unit utilization, SMT enables the construction of wider superscalar machines than would otherwise make sense. Wide superscalars do have the distinct advantage of high single-thread performance on threads rich in ILP. SMT machines can easily shift hardware resources among threads, giving high peak performance on a single thread or high aggregate throughput on several threads.

There is, however, the matter of software. Unfortunately, the thread-level parallelism on which SMTs thrive must be explicitly identified to the hardware. While obvious parallelism exists in separate tasks, which a multitasking OS could schedule onto an SMT, that level of parallelism may actually be more suited to chip multiprocessors, which we discuss later. The ideal role for SMT would be executing smaller threads from a single program, where the sharing of instructions and data in the caches is essentially free and very fast.

Today, however, especially in the embedded world, programmers rarely construct programs in multithreaded fashion. While this technique is growing more popular for other reasons, it is not likely to happen on a large scale anytime soon. The hoped-for solution is that compilers can be taught to automatically divide programs into independent threads. Scheduling loop iterations as different threads is one obvious possibility, but finding more general independent control flows is the Holy Grail.

With the adoption of SMT in the Pentium 4, which Intel labels "hyper-threading," research into multithreading is gearing up. Led by John Shen, director of Intel's microarchitecture lab, research is underway on clever ways to bring thread hardware to bear on single programs, such as speculative precomputation and adaptive dynamic prefetch threads[4].

Embedded systems offer another opportunity for SMT. In many systems, real-time response is critical. Traditional single-threaded processors take long and variable amounts of time responding to priority interrupts. Multithreaded processors keep multiple tasks ready to run at a nanosecond's notice, eliminating the overhead of switching to a high priority task, greatly accelerating real-time response.

Vector Processing

A serious obstacle to scaling multi-issue machines to higher issue rates is the Flynn bottleneck. This bottleneck describes the difficulty of fetching and issuing many instructions in parallel, which explains the exponential growth in the complexity of superscalar machines. Deeply pipelined machines aren't much better. From the perspective of instructions per cycle (IPC), scheduling a deep pipeline is as difficult as scheduling a wide-issue pipeline.

Vector processors address Flynn's bottleneck. In a vector machine, a single instruction performs many operations, perhaps an entire loop's worth, greatly reducing the instruction fetch and issue bandwidth requirements for a given amount of work.

The limitation on vector architectures is that they apply only where high degrees of data-level parallelism exist. That is, where operations on the data set are such that the result of any operation is independent of other operations. This characteristic is why early vector machines like the Illiac-IV, CDC Star, TI ASC, and CRAY-1 were developed for scientific applications where computations are based mostly on matrix algebra.

Early vector machines, however, suffered badly from Amdahl's Law, which shows that the overall speedup is dramatically lowered by even small amounts of scalar code that cannot be vectorized. Since early vector machines had very poor performance on the scalar portions of the code, they were relegated to niche markets like weather forecasting and weapons simulations. Vector processors fell even further out of favour as computing turned more toward PCs. For the first decade of the PC era, workloads were almost entirely scalar in nature, requiring architectures more adept at general-purpose and control-oriented processing.

Several years ago, however, digital multimedia technologies began to take hold on PCs. Today, multimedia content is growing rapidly relative to other tasks, and it is demanding more and more of the CPU's attention. Given its phenomenal market appeal, multimedia will dominate the PC workload, if it doesn't already, and is now rapidly migrating down into consumer devices of every sort.

The switch in emphasis from general-purpose to multimedia workloads changes the architecture called for in embedded processors. Multimedia processing is more akin to scientific processing than to general-purpose processing. Multimedia workloads are largely concerned with processing (filtering, decompressing, etc.) streams of digital signals using block-oriented algorithms. This processing tends to be computationally intense and highly data parallel, much like scientific applications.

Multimedia processing, however, has an additional requirement not usually imposed on scientific applications: it must often be performed in real time. This requirement puts extraordinary pressure on the performance of the processor. Traditional superscalar or pipelined embedded processors are poorly equipped for the job. Indeed, the out-of-order and speculative techniques often used to boost their efficiency can be detrimental to real-time processing. When pushed very far, the execution schedule can become highly nondeterministic. Uncertainty in the execution time degrades real-time performance because it requires guard-banding of deadlines, which throws away potentially useful CPU cycles.

Re-enter vector processing. Massive transistor budgets will enable a new, and in many ways superior, form of vector processing than that found in early vector machines. The new vector architecture has been called single-instruction-multiple-data (SIMD), although this is a somewhat more restrictive use of the term than Flynn originally intended in his taxonomy of computer architectures.

SIMD architecture has a number of advantages in modern processors. For starters, SIMD execution units are easy to add to existing processing engines. A SIMD instruction is identical in every respect to a scalar instruction except that the scalar operands are replaced by short fixed-length vector operands. SIMD instructions can be added to work on operands in the existing general register file (which only makes sense in 64-bit architectures), the floating-point register file, or even a new wider register file especially for SIMD vectors.

The result is that SIMD integrates seamlessly with scalar processing. Microarchitecture techniques that accelerate scalar engines will accelerate the vector engine in the same way, acting as a multiplier on the vector speedup. Moreover, SIMD machines suffer less from Amdahl's law because of the fast scalar engine underneath and the fine-grain integration of vector and scalar instructions, greatly reducing the problem that plagued early vector machines.

SIMD architecture relies on spatially parallel execution units, unlike early vector machines that relied more on deeply pipelined units. The use of parallel units is enabled by inexpensive transistors. Most modern processor architectures have now added SIMD instruction-set extensions of some variety, and at 90nm we expect these extensions to become heavily used in embedded processors.

SIMD vectors are especially powerful in embedded systems that perform digital-signal and multimedia processing. Let's say, for example, a multimedia stream requires processing of 16-bit fractional data elements. A 128-bit wide SIMD vector unit would, with a single instruction, perform eight parallel multiply-add operations, which are the heart and sole of many signal-processing algorithms. On these algorithms, speedups approaching the SIMD parallelism are possible in many critical inner loops. Overall application speedup will be less due to Amdahl, but it can still be large, since these loops often occupy the majority of CPU cycles.

SIMD speedup does not have to be taken in the form of performance. It could be taken as a power savings. Let's say we were to add SIMD to a processor and get some degree of application-level speedup. This gain could instead be used to reduce operating frequency while holding performance constant. The SIMD units add transistors, but dynamic power consumption goes up only linearly with transistors. Lowering frequency, on the other hand, not only lowers the power directly, but it also allows the supply voltage to be lowered, which is a quadratic term in the power equation ($P=CV^2F$).

Furthermore, while the SIMD units may add transistors, they may not add silicon area. A lower operating frequency would allow the core to be optimized for area rather than speed. The difference can be significant; indeed, die size might actually be lower with the SIMD units than without.

The downside of SIMD architecture is that data-level parallelism must be explicitly exposed to the hardware by the programmer or by the compiler. Finding and exposing data-level parallelism is not always easy. Most programmers are not experienced in doing it, nor do they have much interest in worrying about such details—they'd much rather trust the compiler to take care of it for them. The majority of current compilers, however, don't schedule for SIMD. While vectorizing-compiler technology exists for FORTRAN on supercomputers, the now ubiquitous C language is more difficult to vectorize. Successful C vector pre-processors have, however, been demonstrated by companies like Veridian Systems (www.psrv.com). With SIMD now becoming a regular feature of major PC and server processor architectures (e.g., x86-SSE, PowerPC-AltiVec, SPARC-VIS, etc.), SIMD vectorizing C compilers will undoubtedly materialize.

Even under the best of circumstances, however, there is a level of SIMD potential that compilers will never uncover. The simple fact is that only so much speedup can be gained by vectorizing a basically scalar algorithm. Much bigger gains can be achieved by fundamentally redesigning the algorithm for execution on a data-parallel machine. This task requires humans, although high-level tools like MATLAB (www.mathworks.com) and Mathematica (www.wolfram.com) can help. Fortunately, again, the presence of SIMD in the major processor architectures will eventually produce a vector-savvy base of algorithm programmers.

Chip Multiprocessing

At the other end of the spectrum from DLP lies an entirely different level of parallelism: process-level parallelism. PLP is high-level, coarse-grain parallelism that exists due to the natural independence of separate programs or processes. Current SOCs have considerable PLP and, as SOCs integrate more functions, PLP will become even more abundant.

The PLP ground is fertile because it cannot be harvested with pipelining, superscalar, or SIMD techniques. PLP is similar to TLP, the distinction being one primarily of granularity. Due to their similarity, multithreading hardware can also get at PLP, but there are advantages to using CMP where possible, leaving multithreading hardware to the fine-grain threads to which CMP is less well suited.

Where PLP exists, CMP is an ideal means of exploiting it. Compared with building a single monolithic processor fast enough to execute two parallel tasks in a given time, two lesser processors running in parallel offer a superior solution.

One reason is real-time response. While a single high-speed CPU gives the best response time in a single-tasking environment, as the task load increases queuing theory predicts that the response time goes badly nonlinear. What's really needed in most multimedia systems and SOCs, however, is adequate response with multiple tasks running rather than blinding response with a single task.

Multiple processors lower the multiprogramming level, providing a more desirable response-time profile and a higher task capacity. A processor with the horsepower to produce the same average response time on two tasks as two lesser processors, for example, would have 20% worse response time to four tasks and twice as bad a response to eight.

In this example, the single processor would have to be more powerful by about 20% than each of the lesser processors. Obviously, the smaller processors would be less complex and easier to build. They could also run at higher frequency because longer wires switch more slowly than short ones. Furthermore, the decentralized control structure of the multiprocessor would scale better with process technology than the more centralized control structures of a single, larger, more powerful processor.

CMP construction also offers scalability in another dimension. Today, SOCs are commonly designed as an assortment of specialized processing elements and hardwired logic blocks connected in a bizarre organization reminiscent of a Picasso masterpiece. This approach is usually taken on the basis of silicon efficiency. But such designs are inherently unscalable. Conceptually, a regular CMP array would provide a more scalable approach, providing more or less aggregate performance through the simple addition or removal of processor cores.

While the CMP approach might be less silicon efficient than the specialized design, the difference can be mitigated to some extent by the redundancy of the processor array, which creates an effectively smaller die. Even ignoring these effects, however, the lower efficiency of the CMP architecture will become irrelevant over time. As the trends we discussed previously take hold, the more scalable, more programmable, more reusable, CMP approach will become the most cost-effective solution, even if not the one with the smallest die size.

Summary

A number of trends are converging that promise to radically change the architecture of embedded processors and SOCs. The limits of instruction-level parallelism will be reached and the traditional techniques of deeper pipelines and superscalar instruction issue will run out of steam. Frequency gains through process scaling alone will not keep processors on their historical performance pace.

At the same time, the clear economic benefits of more programmable SOCs and the insatiable appetite for multimedia processing power will drive the performance demands on embedded processors up dramatically—faster even than that predicted by Moore’s Law. Fortunately, the massive transistor budgets and falling transistor costs afforded by 90nm process technology will enable new architectural techniques to tap into the levels of parallelism that have heretofore gone largely ignored by embedded processors.

Multithreading will exploit thread-level parallelism to boost throughput and execution unit efficiency, despite increasing memory latency. Vector processing in the modern form of SIMD execution units will be added to conventional embedded RISC cores to exploit the data-level parallelism abundant in multimedia streams. Chip multiprocessing will exploit process-level parallelism to raise task capacity and improve real-time response while at the same time offer a superior architecture for more scalable, more programmable SOCs.

While each of these techniques is powerful in its own right, each exploits a different level of parallelism and can be readily combined together, along with conventional pipelining and superscalar techniques, to create embedded processor cores with performance adequate to radically change the way SOCs are designed.

---

[1] Diefendorff, K. and Dubey, P., "How Multimedia Workloads Will Change Processor Design," *IEEE Computer*, September 1997, 43-45.

[2] "International Technology Roadmap for Semiconductors," 2001 Edition, http://public.itrs net/.

[3] Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," ISCA95.

[4] Shen, "Multi-threading for Latency," Fifth Workshop on Multithreaded Execution, Architecture and Compilation, 34th International Symposium on Microarchitecture, 2001.