Parallelism and the ARM Instruction Set Architecture

Leveraging parallelism on several levels, ARM's new chip designs could change how people access technology. With sales growing rapidly and more than 1.5 billion ARM processors already sold each year, software writers now have a huge range of markets in which their ARM code can be used.

John Goodacre
Andrew N. Sloss
ARM

Over the past 15 years, the ARM reduced-instruction-set computing (RISC) processor has evolved to offer a family of chips that range up to a full-blown multiprocessor. Embedded applications' demand for increasing levels of performance and the added efficiency of key new technologies have driven the ARM architecture's evolution.

Throughout this evolutionary path, the ARM team has used a full range of techniques known to computer architecture for exploiting parallelism. The performance and efficiency methods that ARM uses include variable execution time, subword parallelism, digital signal processor-like operations, thread-level parallelism and exception handling, and multiprocessing.

The developmental history of the ARM architecture shows how processors have used different types of parallelism over time. This development has culminated in the new ARM11 MPCore multiprocessor.

RISC FOR EMBEDDED APPLICATIONS

Early RISC designs such as MIPS focused purely on high performance. Architects achieved this with a relatively large register set, a reduced number of instruction classes, a load-store architecture, and a simple pipeline. All these now fairly common concepts can be found in many of today's modern processors.

The ARM version of RISC differed in many ways, partly because the ARM processor became an embedded processor designed to be located within a system-on-chip device.[1] Although this kept the main design goal focused on performance, developers still gave priority to high code density, low power, and small die size.

To achieve this design, the ARM team changed the RISC rules to include variable-cycle execution for certain instructions, an inline barrel shifter to preprocess one of the input registers, conditional execution, a compressed 16-bit Thumb instruction set, and some enhanced DSP instructions.

- *Variable cycle execution*. Because it is a load-store architecture, the ARM processor must first load data into one of the general-purpose registers before processing it. Given the single-cycle constraint the original RISC design imposed, loading and storing each register individually would be inefficient. Thus, the ARM ISA instructions specifically load and store multiple registers. These instructions take variable cycles to execute, depending on the number of registers the processor is transferring. This is particularly useful for saving and restoring context for a procedure's prologue and epilogue. This directly improves code density, reduces instruction fetches, and reduces overall power consumption.
- *Inline barrel shifter*. To make each data processing instruction more flexible, either a shift or rotation can preprocess one of the source registers. This gives each data processing instruction more flexibility.

0018-9162/05/$20.00 © 2005 IEEE

- *Conditional execution.* An ARM instruction executes only when it satisfies a particular condition. The condition is placed at the end of the instruction mnemonic and, by default, is set to always execute. This, for example, generates a savings of 12 bytes—42 percent—for the greatest common divisor algorithm implemented with and without conditional execution.
- *16-bit Thumb instruction set.* The condensed 16-bit version of the ARM instruction set allows higher code density at a slight performance cost. Because the Thumb 16-bit ISA is designed as a compiler target, it does not include the orthogonal register access of the ARM 32-bit ISA. Using the Thumb ISA can achieve a significant reduction in program size.

  In 2003, ARM announced its Thumb-2 technology, which offers a further extension to code density. This technology increases the code density by mixing both 32- and 16-bit instructions in the same instruction stream. To achieve this, the developers incorporated unaligned address accesses into the processor design.

Nonsaturated (ISA v4T)	Saturated (ISA v5TE)
PRECONDITION	**PRECONDITION**
r0=0x00000000	r0=0x00000000
r1=0x70000000	r1=0x70000000
r2=0x7fffffff	r2=0x7fffffff
ADDS r0,r1,r2	QADD r0,r1,r2
POSTCONDITION	**POSTCONDITION**
result is **negative**	result is **positive**
r0=**0xefffffff**	r0=**0x7fffffff**

***Figure 1.* Nonsaturated and saturated addition. Saturation is particularly useful for digital signal processing because nonsaturations would wrap around when the integer value overflowed, giving a negative result.**

- *Enhanced DSP instructions.* Adding these instructions to the standard ISA supports flexible and fast 16 × 16 multiply and arithmetic saturation, which lets DSP-specific routines migrate to ARM. A single ARM processor could execute applications such as voice-over-IP without the requirement of having a separate DSP. The processor can use one example of these instructions, SMLAxy, to multiply the top or bottom 16 bits of a 32-bit register. The processor could multiply the top 16 bits of register r1 by the bottom 16 bits of register r2 and add the result to register r3.

Figure 1 shows how saturation can affect the result of an ADD instruction.[2] Saturation is par-

A Short History of ARM

Formed in 1990 as a joint venture with Acorn Computers, Apple Computers, and VLSI Technology (which later became Philips Semiconductor), ARM started with only 12 employees and adopted a unique licensing business model for its processor designs. By licensing rather than manufacturing and selling its chip technology, ARM established a new business model that has redefined the way industry designs, produces, and sells microprocessors. Figure A shows how the ARM product family has evolved.

The first ARM-powered products were the Acorn Archimedes desktop computer and the Apple Newton PDA. The ARM processor was designed originally as a 32-bit replacement for the MOS Technologies 6502 processor that Acorn Computers used in a range of desktops designed for the British Broadcasting Corporation. When Acorn set out to develop this new replacement processor, the academic community had already begun considering the RISC architecture. Acorn decided to adopt RISC for the ARM processor. ARM's developers originally tailored the ARM instruction set architecture to efficiently execute Acorn's BASIC interpreter, which was at the time very popular in the European education market.

The ARM1, ARM2, and ARM3 processors were developed by Acorn Computers. In 1985, Acorn received production quantities of the first ARM1 processor, which it incorporated into a product called the ARM Second Processor, which attached to the BBC Microcomputer through a parallel communication port called "the tube." These devices were sold mainly as a research tool. ARM1 lacked the common multiply and divide instructions, which users had to synthesize using a combination of data processing instructions. The condition flags and the program counter were combined, limiting the effective addressing range to 26 bits. ARM ISA version 4 removed this limitation.

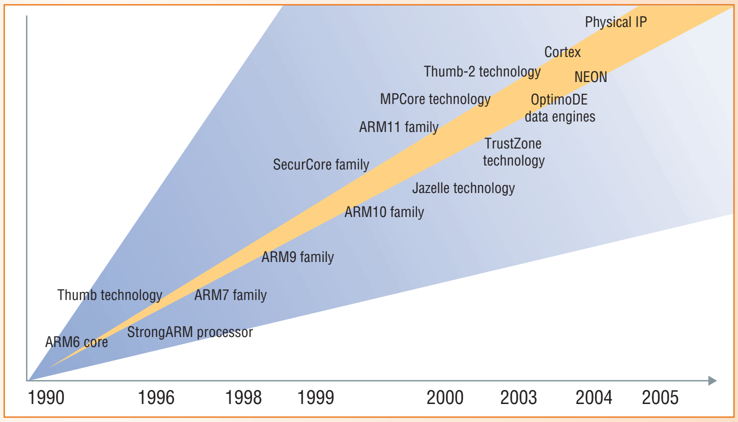

***Figure A.* Technology evolution of the ARM product family.**

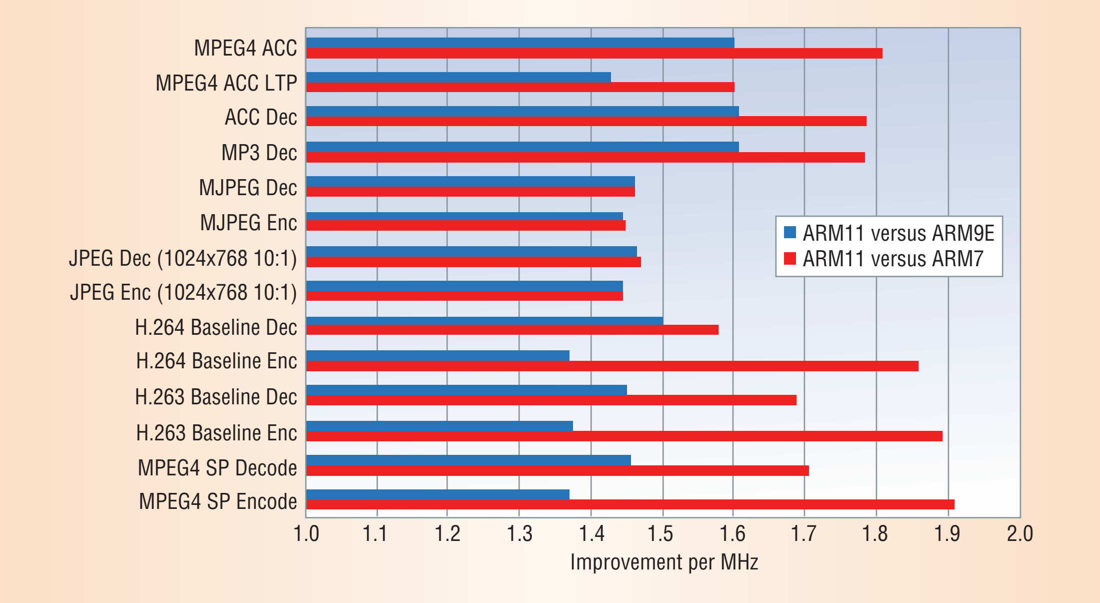

Figure 2. SIMD versus non-SIMD power consumption. The lightweight ARM implementation of SIMD reduces gate count, hence significantly reducing die size, power, and complexity.

ticularly useful for digital signal processing because nonsaturations would wrap around when the integer value overflowed, giving a negative result. A saturated QADD instruction returns a maximum value without wrapping around.

DATA-LEVEL PARALLELISM

Following the success of the enhanced DSP instructions introduced in the v5TE ISA, ARM introduced the ARMv6 ISA in 2001. In addition to improving both data- and thread-level parallelism, other goals for this design included enhanced mathematical operations, exception handling, and endian-ness handling.

An important factor influencing the ARMv6 ISA design involved increasing DSP-like functionality for overall video handling and 2D and 3D graphics. The design had to achieve this improved functionality while still maintaining very low power consumption. ARM identified the single-instruction, multiple-data architecture as the means for accomplishing this.

SIMD is a popular technique for providing data-level parallelism without compromising code density and power. A SIMD implementation requires relatively few instructions to perform complex calculations with minimum memory accesses.

Due to a careful balancing of computational efficiency and low power, ARM's SIMD implementation involved splitting the standard 32-bit data path into four 8-bit or two 16-bit slices. This differs from many other implementations, which require additional specialized data paths for SIMD operations.

Figure 2 shows the improvements in MHz that various codecs require when using the ARMv6 SIMD instructions introduced in the ARM11 processor.

The lightweight ARM implementation of SIMD reduces gate count, hence significantly reducing die size, power, and complexity. Further, all the SIMD instructions execute conditionally.

To improve handling of video compression systems such as MPEG and H.263, ARM also introduced the sum of absolute differences (SAD) concept, another form of DLP instructions. Motion estimation compares two blocks of pixels, R(I,j) and C(I,j), by computing

$$SAD = \sum | R(I,j) - C(I,j) |$$

Smaller SAD values imply more similar blocks. Because motion estimation performs many SAD tests with different relative positions of the R and C blocks, video compression systems require very fast and energy-efficient implementations of the sum-of-absolute-differences operation.

The instructions USAD8 and USADA8 can compute the absolute difference between 8-bit values. This is particularly useful for motion-video-compression and motion-estimation algorithms.

THREAD-LEVEL PARALLELISM

We can view threads as processes, each with its own program counter and register set, while having the advantage of sharing a common memory space. For thread-level parallelism, ARM needed to improve exception handling to prepare for the increased complexity in handling multithreading on multiple processors. These requirements added inherent complexity in the interrupt handler, scheduler, and context switch.

One optimization extended the exception handling instructions to save precious cycles during the time-critical context switch. ARM achieved this by adding three new instructions to the instruction set, as Table 1 shows.

Programmers can use the change processor state (CPS) instruction to alter processor state by setting

Table 1. Exception handling instructions in the ARMv6 architecture.

Instruction	Description	Action
CPS	Change processor state	CPS<effect> <iflags>,{,#mode} CPS #<mode> CPSID <flags> CPSIE <flags>
RFE	Return from exception	RFE<addressing_mode> Rn!
SRS	Save return state	SRS<addressing_mode>,#<mode>{!}

```
ARMv4T ISA                          ARMv6 ISA
; Copy CPSR                                ; Change processor state and modify
      MRS     r3, CPSR                     ; select bits
; Mask mode and FIQ interrupt                   CPSIE      f, #SVC
      IC      r3, r3, #MASK|FIQ
; Set Abort mode and enable FIQ
      ORR     r3, r3, #SVC|nFIQ
; Update the CPSR
      MSR     CPSR_c, r3
```

Figure 3. ARMv6 ISA change processor state instruction compared with the ARMv4T architecture.

the current program status register to supervisor mode and disabling fast interrupt requests, as the code in Figure 3 shows. Whereas the ARMv4T ISA required four instructions to accomplish this task, the ARMv6 ISA requires only two.

Programmers can use the save return state (SRS) instruction to modify the *saved program status register* in a specific mode. The updating of SPSR in a particular ARMv4T ISA mode involved many more instructions than in the ARMv6 ISA. This new instruction is useful for handling context switches or preparing to return from an exception handler.

MULTIPROCESSOR ATOMIC INSTRUCTIONS

Earlier ARM architectures implemented semaphores with the swap instruction, which held the external bus until completion. Obviously, this was unacceptable for thread-level parallelism because one processor could hold the entire bus until completion, disallowing all other processors. ARMv6 introduced two new instructions—load-exclusive LDREX and store-exclusive STREX—which take advantage of an exclusive monitor in memory:

- LDREX loads a value from memory and sets the exclusive monitor to watch that location, and
- STREX checks the exclusive monitor and, if no other write has taken place to that location, performs the store to memory and returns a value to indicate if the data was written.

Thus, the architecture can implement semaphores that do not lock the system bus that grants other processors or threads access to the memory system.

The ARM11 microarchitecture was the first hardware implementation of the ARMv6 ISA. It has an eight-stage pipeline with separate parallel pipelines for the load/store and multiply/accumulate operations. With the parallel load/store unit the ARM1136J-S processor can continue executing without waiting for slower memory—a main gating factor for processor performance.

In addition, the ARM1136J-S processor has physically tagged caches to help with thread-level parallelism—as opposed to the virtually tagged caches of previous ARM processors—which considerably benefits context switches, especially when running large operating systems.

A virtually tagged cache must be flushed every time a context switch takes place because the cache contains old virtual-to-physical translations. In the ARM11, the *memory management unit* logic resides between the level 1 cache and the processor core. The reduction in cache flushing has the additional benefit of decreasing overall power consumption by reducing the external memory accesses that occur in a virtually tagged cache. The physically tagged cache increases overall performance by about 20 percent.

INSTRUCTION-LEVEL PARALLELISM

In ILP, the processor can execute multiple instructions from a single sequence of instructions concurrently. This form of parallelism has significant value in that it provides additional overall performance without affecting the software programming model.

Obviously, ILP puts more emphasis on the compiler that extracts it from the source code and schedules the instructions across the superscalar core. Although potentially simplifying otherwise overly complex hardware, an excessive drive to extract ILP and achieve high performance through increased MHz has increased hardware complexity and cost.

ARM has remained the processor at the edge of the network for several years. This area has always seen the most rapid advancements in technology, with continuous migration away from larger computer systems and toward smaller ones. For example, technology developed for mainframes a decade or two ago is found in desktop computers today. Likewise, technologies developed for the desktop five years ago have begun appearing in consumer and network products. For example, symmetric multiprocessing (SMP) is appearing today in both desktop and embedded computers.

Continued demand for performance at low power has led to minimizing the overall power budget by adding multiple processors and accelerators.

PERFORMANCE VERSUS POWER REQUIREMENTS

For several years, the embedded processor market inherited technology matured in desktop computing as consumers demanded similar functionality in their embedded devices. The continued demand for performance at low power has, however, driven slightly different requirements and led to the overall power budget being minimized by adding multiple processors and accelerators within an embedded design. Today, the demand for high levels of general-purpose computing drives using SMP as the application processor in both embedded and desktop systems.

In 2004, both the embedded and desktop markets hit the cost-performance-through-MHz wall. In response, developers began embracing potential solutions that require SMP processing to avoid the following pitfalls:

- *High MHz costs energy.* Increasing a processor's clock rate has a quadratic effect on power consumption. Not only does doubling the MHz double the dynamic power required to switch the logic, it also requires a higher operating voltage, which increases at the square of the frequency. Higher frequencies also add to design complexity, greatly increasing the amount of logic the processor requires.
- *Extracting ILP is complex and costly.* Using hardware to extract ILP significantly raises the cost in silicon area and design complexity, further increasing power consumption.
- *Programming multiple independent processors is nonportable and inefficient.* As developers use more processors, often with different architectures, the software complexity escalates, eliminating any portability between designs.

In mid-2004, PC manufacturers and chip makers made several announcements heralding the end of the MHz race in desktop processors and championing the use of multicore SMP processors in the server realm, primarily through the introduction of hyperthreading in the Intel Pentium processor. At that time, ARM announced its ARM11 MPCore multiprocessor core as a key solution to help address the demand for performance scalability.

Introduced alongside the ARM11 MPCore, a set of enhancements to the ARMv6 architecture provides further support for advanced SMP operating systems. In its move to support richer SMP-capable operating systems, ARM applied these enhancements, known as ARMv6K or AOS (for Advanced OS Support), across all ARMv6-architecture-based application processors to provide a firm foundation for embedded software.

The ARM11 multiprocessor also addressed the SMP system design's two main bottlenecks:

- interprocessor communication with the integration of the new ARM Generic Interrupt Controller (GIC), and
- cache coherence with the integration of the Snoop Control Unit (SCU), an intelligent memory-communication system.

These logic blocks deliver an efficient, hardware-coherent single-core SMP processor that manufacturers can build cost-effectively.

PREPARATIONS FOR ARM MULTIPROCESSING

To fully realize the advantages of a multiprocessor hardware platform in general-purpose computing, ARM needed to provide a cache-coherent, symmetric software platform with a rich instruction set. ARM found that a few key enhancements to the current ARMv6 architecture could offer the significant performance boost it sought.

Enhanced atomic instructions

Researchers can use the ARMv6 load-and-store exclusives to implement both swap-based and compare-and-exchange-based semaphores to control access to critical data. In the traditional server computing world of SMP there has, however, been significant software investment in optimizing SMP code using *lock-free synchronization*. This work has been dominated by the x86 architecture and its atomic instructions that developers can use to compare and exchange data.

Many favored using the Intel cmpxchg8b instruction in these lock-free routines because it can exchange and compare 8 bytes of data atomically. Typically, this involved 4 bytes for payload and 4 bytes to distinguish between payload versions that could otherwise have the same value—the so-called A-B-A problem.

The ARM exclusives provide atomicity using the data address rather than the data value, so that the routines can atomically exchange data without experiencing the A-B-A problem. Exploiting this would, however, require rewriting much of the existing two-word exclusive code. Consequently, ARM added instructions for performing load-and-store exclusives using various payload sizes—including 8 bytes—thus ensuring the direct portability of existing multithreaded code.

Improved access to localized data

When an OS encounters the increasing number of threaded applications typical in SMP platforms, it must consider the performance overheads of associating thread-specific state with the currently executing thread. This can involve, for example, knowing which CPU a thread is executing on, accessing kernel structures specific to a thread, and enabling thread access to local storage. The AOS enhancements add registers that help with these SMP performance aspects.

CPU number. Using the standard ARM system coprocessor interface, software on a processor can execute a simple, nonmemory-accessing instruction to identify the processor on which it executes. Developers use this as an index into kernel structures.

Context registers. SMP operating systems handle two key demands from the kernel when providing access to thread-specific data. The ARMv6K architecture extensions define three additional system coprocessor registers that the OS can manage for whatever purpose it sees fit. Each register has a different access level:

- user and privileged read/write accessible;
- read-only in user, read/write privileged accessible; and
- privileged only read/write accessible.

The exact use of these registers is OS-specific. In the Linux kernel and GNU toolchain, the ARM application binary interface has assigned these registers to enable *thread local storage*. A thread can use TLS to rapidly access thread-specific memory without losing any of the general-purpose registers.

To support TLS in C and C++, the new keyword *thread* has been defined for use in defining and declaring a variable. Although not an official extension of the language, using the keyword has gained support from many compiler writers. Variables defined and declared this way would automatically be allocated locally to each thread:

```
__thread int i;
__thread struct state s;
extern __thread char *p;
```

Supporting TLS is a key requirement for the new Native Posix Thread Library (NPTL) released as part of the Linux 2.6 kernel. This Posix thread library provides significant performance improvements over the old Linux pthread library.

Power-conscious spin-locks

Another SMP system cost involves the synchronization overhead required when processors must access shared data. At the lowest abstraction level in most SMP synchronization mechanisms, a *spin-lock* software technique uses a value in memory as a lock. If the memory location contains some predefined value, the OS considers the shared resource locked, otherwise it considers the resource unlocked. Before any software can access the shared resource, it must acquire the lock—and an atomic operation must acquire it. When the software finishes accessing the resource, it must release the lock.

Spin-lock causes the processor to spin around a tight loop while continually attempting to acquire a lock.

In an SMP OS, processors often must wait while another processor holds a lock. The spin-lock received its name because it accomplishes this waiting by causing the processor to spin around a tight loop while continually attempting to acquire the lock. A later refinement to reduce bus contention added a back-off loop during which the processor does not attempt to access the lock. In either case, in a power-conscious embedded system, these unproductive cycles obviously waste energy.

The AOS extensions include a new instruction pair that lets a processor sleep while waiting for a lock to be freed and that, as a result, consumes less energy. The ARM11 multiprocessor implements these instructions in a way that provides next-cycle notification to the waiting processor when the lock is freed, without requiring a back-off loop. This results in both energy savings and a more efficient spin-lock implementation mechanism.

Figure 4 shows a sample implementation of the spin lock and unlock code used in the ARM Linux 2.6 kernel.

Weakly ordered memory consistency

The ARMv6 architecture defined various memory consistency models for the different definable memory regions. In the ARM11 multiprocessor, spin-lock code uses coherently cached memory to store the lock value. As a multiprocessor, the ARM11 MPCore is the first ARM processor to fully expose weakly ordered memory to the programmer. The multiprocessor uses three instructions to control weakly ordered memory's side effects:

```
static inline void _raw_spin_lock(spinlock_t *lock)
{
    unsigned long tmp;

    __asm__ __volatile__(
     1:  ldrex    %0, [%1]           ; exclusive read lock
         teq      %0, #0             ; check if free
         wfene                       ; if not, wait (saves power)
         strexeq  %0, %2, [%1]       ; attempt to store to the lock
         teqeq    %0, #0             ; Were we successful ?
         bne      1b                 ; no, try again
      :   “=&r” (tmp)
      :   “r” (&lock->lock), “r” (1), “r” (0)
      :   “cc”, “memory”
    );

    rmb();    // Read memory barrier stops speculative reading of payload
}             // This is NOP on MPCore since dependent reads are sync’ed

static inline void _raw_spin_unlock(spinlock_t *lock)
{
    wmb();    // data write memory barrier, ensure payload write visible
              // Ensures data ordering, but does not necessarily wait
    __asm__ __volatile__(
        str %1, [%0]                 ; Release spinlock
        mcr p15, 0, %1, c7, c10, 4   ; DrainStoreBuffer (DSB)
        sev                          ; Signal to any CPU waiting
     :  “r” (&lock->lock), “r” (0)
     :  “cc”, “memory”);
}
```

Figure 4. Power-conscious spin-lock. This sample implementation shows the spin-lock and unlock code used in the ARM Linux 2.6 kernel.

- *wmb()*. This Linux macro creates a write-memory barrier that the multiprocessor can use to place a marker in the sequencing of any writes around this barrier instruction. The spin-lock, for example, executes this instruction prior to unlocking to ensure that any writes to the payload data complete before the write to release the spin-lock, and hence before any other processor can acquire the lock. To ensure higher performance, the barrier does not necessarily stall the processor by flushing data. Rather it informs the load-store unit and lets execution continue in most situations.
- *rmb()*. Again from the Linux kernel, this macro places a read-memory barrier that prevents speculative reads of the payload from occurring before the read has acquired the lock. Although legal in the ARMv6 architecture, this level of weakly ordered memory can make it difficult to ensure software correctness. Thus, the ARM11 multiprocessor implements only nonspeculative read-ahead. When the possibility exists that a read will not be required, as in the spin-lock case—where there is a branch instruction between the teqeq instruction and any payload read—the read-ahead does not take place. So, for the ARM11 MPCore multiprocessor, this macro can be defined as empty.
- *DSB (Drain Store Buffer)*. The ARM architecture includes a buffer visible only to the processor. When running uniprocessor software, the processor allows subsequent reads to scan the data from this buffer. However, in a multiprocessor, this buffer becomes invisible to reads from other processors. The DSB drains this buffer into the L1 cache. In the ARM11 multiprocessor, which has a coherent L1 cache between the processors, the flush only needs to proceed as far as the L1 memory system before another processor can read the data. In the spin-lock unlock code, the processors issue the DSB immediately prior to the SEV *(Set Event)* instruction so that any processor can read the correct value for the lock upon awakening.

ARM11 MPCORE MULTIPROCESSOR

The ISA’s suitability is not the only factor affecting the multiprocessor’s ability to actually deliver the scalability promises of SMP. If they are poorly implemented, two aspects of an SMP design can significantly limit peak performance and increase the energy costs associated with providing SMP services:

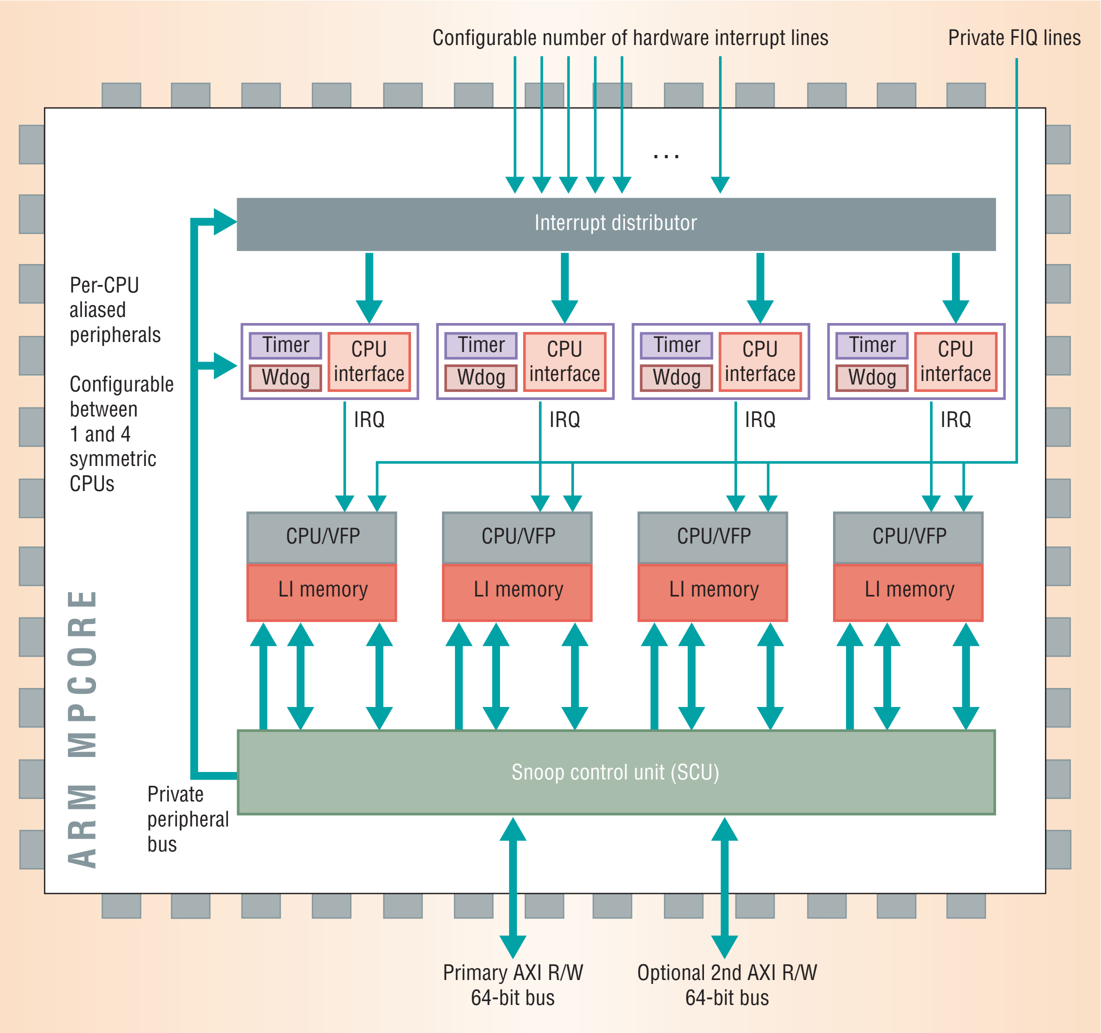

Figure 5. ARM11 MPCore. This multiprocessor integrates the new ARM GIC inside the core to make the interrupt system's access and effects much closer and more efficient.

- *Cache coherence*. Developers typically provide the single-image SMP OS with coherent caches so it can maintain performance by placing its data in cached memory. In the ARM11 multiprocessor, each CPU has its own instruction and L1 data cache. Existing coherency schemes often extend the system bus with additional signals to control and inspect other CPUs' caches. In an embedded system, the system bus often clocks slower than the CPU. Thus, besides placing a bottleneck between the processor and its cache, this scheme significantly increases the traffic and hence the energy the bus consumes. The ARM11 MPCore addresses these problems by implementing an intelligent SCU between each processor. Operating at CPU frequency, this configuration also provides a very rapid path for data to move directly between each CPU's cache.
- *Interprocessor communication*. An SMP OS requires communication between CPUs, which sometimes is best accomplished without accessing memory. Also, the system must often regulate interprocessor communication using a spin-lock that synchronizes access to a protected resource. Other SMP OS communication between CPUs is best accomplished without accessing memory. Systems frequently must also synchronize asynchronously. One such mechanism uses the device's interrupt system to cause activity on a remote processor. These software-initiated interprocessor interrupts (IPI) typically use an interrupt system designed to interface interrupts from I/O peripherals rather than another CPU.

Figure 5 shows how the ARM11 MPCore integrates the new ARM GIC inside the core to make the interrupt system's access and effects closer and more efficient. ARM designed the GIC to optimize the cost for the key forms of IPI used in an SMP OS.

Interrupt subsystem

A key example of IPI's use in SMP involves a multithreaded application that affects some state within the processor that is not hardware-coherent with the other processors on which the application process has threads running. This can occur when, for example, the application allocates some virtual memory. To maintain consistency, the OS also must apply these memory translations to all other processors. In this example, the OS would typically apply the translation to its processor and then use the low-contention private peripheral bus to write to an interrupt control register in the GIC that causes an interrupt to all other processors. The other processors could then use this interrupt's ID to determine that they need to update their memory translation tables.

The GIC also uses various software-defined patterns to route interrupts to specific processors through the interrupt distributor. In addition to their dynamic load balancing of applications, SMP OSs often also dynamically balance the interrupt handler load. The OS can use the per-processor aliased control registers in the local private peripheral bus to rapidly change the destination CPU for any particular interrupt.

Another popular approach to interrupt distribution sends an interrupt to a defined group of processors. The MPCore views the first processor to accept the interrupt, typically the least loaded, as being best positioned to handle the interrupt. This flexible approach makes the GIC technology suitable across the range of ARM processors. This standardization, in turn, further simplifies how software interacts with an interrupt controller.

Snoop control unit

The MPCore's SCU is an intelligent control block used primarily to control data cache coherence between each attached processor. To limit the power consumption and performance impact from snooping into and manipulating each processor's cache on each memory update, the SCU keeps a duplicate copy of the physical address tag (pTag) for each cache line. Having this data available locally lets the SCU limit cache manipulations to processors that have cache lines in common.

The processor maintains cache coherence with an optimized version of the MESI (modified, exclusive, shared, invalid) protocol. With MESI, some common operations, such as A = A + 1, cause many state transitions when performed on shared data.

To help improve performance and further reduce the power overhead associated with maintaining coherence, the intelligence in the SCU monitors the system for a *migratory line*. If one processor has a modified line, and another processor reads then writes to it, the SCU assumes such a location will experience this same operation in the future. As this operation starts again, the SCU will automatically move the cache line directly to an invalid state rather than expending energy moving it first into the shared state. This optimization also causes the processor to transfer the cache line directly to the other processor without intervening external memory operations.

This ability to move shared data directly between processors provides a key feature that programmers can use to optimize their software. When defining data structures that processors will share, programmers should ensure appropriate alignment and packing of the structure so that line migration can occur. Also, if the programmers use a queue to distribute work items across processors, they should ensure that the queue is an appropriate length and width so that when the worker processor picks up the work item, it will transfer it again through this cache-to-cache transfer mechanism. To aid with this level of optimization, the MPCore includes hardware instrumentation for many operations within both the traditional L1 cache and the SCU.

The ARMv6K ISA can be considered a key multiprocessor-aware instruction set. With its foundation in low-power design, the architecture and its implementation in the ARM11 MPCore can bring low power to high-performance designs. These new designs show the potential to truly change how people access technology. With more than 1.5 billion ARM processors being sold each year, there is a huge range of markets in which ARM developers can use their software code. ■

References

1. D. Seal, *ARM Architecture Reference Manual,* 2nd ed., Addison-Wesley Professional, 2000.
2. A. Sloss et al., *ARM System Developer's Guide*, Morgan Kauffman, 2004.

John Goodacre *is a program manager at ARM with responsibility for multiprocessing. His interests include all aspects of both hardware and software in embedded multiprocessor designs. Goodacre received a BSc in computer science from the University of York. Contact him at john.goodacre@arm.com.*

Andrew N. Sloss *is a principle engineer at ARM. His research interests include exception handling methods, embedded systems, and operating system architecture. Sloss received a BSc in computer science from the University of Hertfordshire. He is also is a Chartered Engineer and a Fellow of the British Computer Society. Contact him at andrew.sloss@arm.com.*