IntellaSys Corporation
10080 N. Wolfe Road, Suite SW3-190
Cupertino, CA 95014
voice 408-446-4222
fax 408-446-5444

A 21st Century Sea Change Taking Place in Embedded Microprocessors

David Guzeman, Chief Marketing Officer,
IntellaSys Corporation, Cupertino, CA

It has been 30 years since the 8048/8051 microprocessors appeared on the market and changed the world's view of what an embedded microcontroller should look like. Over the years, each new microcontroller has tended to follow that basic architecture, adding improvements at each step in order to stay in step with the increasingly demanding applications. As long-lived and important as that original architecture has been, it is now time to embrace a new multicore architecture, one designed from the ground up to handle the applications of the 21st century. In this white paper, we will discuss the sea changes in architecture design that are being driven by demands for higher operating speeds and lower power dissipation.

20th Century Applications

The typical application from the 20th century used an 8-bit microcontroller – a bit banger – that could read sensors, do a small amount of data processing, and then drive some I/O lines, probably parallel, in order to send characters to a display or record a data byte onto tape or some other data logging device. Additional I/O lines could scan a simple keyboard or set of switches, and the whole thing could be driven within time constraints by an on-chip real time clock that could provide precise timing references to sync data transfers, and perform other time-driven tasks.

These applications used only a small amount of memory, perhaps 64 to 256 bytes of RAM, and most of that was integrated on the chip. Although provisions were made to access external memory as well, this was initially a primitive interface consisting of just an address and data bus and relied on the processor to read and move data in and out of external memory under software control.

Thus the emphasis was on controlling I/O within tight time constraints with very little actual data manipulation done by the processor chip. That's fortunate because the processor was extremely limited in its data processing capability anyway and was very slow running at clock rates of a few Megahertz. As limited as these chips were, they were sufficient to control countless simple applications ranging from wall thermostats to simple home automation systems. In fact, at this moment I'm typing this paper on a recently introduced laptop that uses a derivative of that original 8048 chip for the sole purpose of reading keyboard clicks. Over time, processors were introduced that were even smaller with less capabilities that sold for, presumably lower prices. At the same time, others came out that were more advanced, both 16 and 32 bit versions, and with much faster and more sophisticated external memory interfaces using DMA controller circuitry. Still, the basic idea has remained the same. One or two processors on a chip, reading data from input lines and sending data to output lines, and wiggling I/O control pins as appropriate… all to the metronome of an external reference real time clock.

Consumer Electronics is Driving 21st Century Applications

But now the nature of the applications has changed dramatically. In addition to the traditional real time bit banging, a new dimension of processing capability has been added – the processing of algorithms. Today the high-volume applications are multimedia consumer aps that range from tiny MP3 music players to cell phones with video capability. Moreover, the long awaited avalanche of high-definition televisions has begun, and along with those televisions, consumers are suddenly perceiving the need for home networks that move video and music from room to room.

Multimedia Capability

All new consumer applications have digital data at their heart, and that implies extensive digital signal processing in any device that displays or plays that data. The various file formats for multimedia have been carefully designed with an eye toward digital processing by using mathematical algorithms – Fast Fourier Transforms (FFTs), discrete cosine transforms (DCTs), and so forth. The high bandwidth required to serve multimedia applications requires that 21st century processors have dedicated circuitry for processing those algorithms. But at the same time, none of the earlier requirements for general purpose I/O and real time clocks has gone away. New chips must handle both!

Bitstream Orientation

Whereas earlier processors viewed external memory as the source and destination of applications data, modern processors must be able to operate with high-speed bitstreams of data arriving from the internet, USB and 1394 cables, as well as cable and satellite television services. The USB 2.0 interface, now nearly ubiquitous on consumer products such as cameras, MP3 players, and even cell phones, requires up to 480 mbit/sec. The 1394 interface is commonly used in video applications and comes in 200/400/800 mbit/sec rates. Even gigabit Ethernet is beginning to appear in homes with even higher data rates yet. Today's processors have to deal with these data rates, all of which are staggeringly fast by 20th century standards.

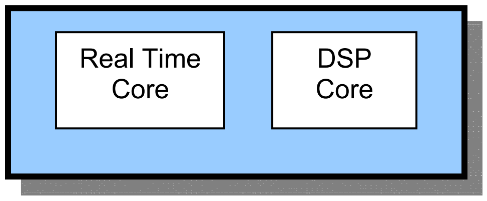
To make matters worse, the new High Definition Audio-Video Network Alliance (HANA) standard for home networking assumes up to FOUR 1394 bitstreams that may reach 800 Mbit/sec. And MP4 formatted data assumes multiple bitstreams for audio and video plus optional additional streams for things like subtitles and still images. In many cases, the same processor that is decoding the MP4 bitstream from a buffer memory must also handle the incoming bitstream as well, so that as many as four or five of these high-speed bitstreams must be handled at once.

Fast External Memory Interface

With requirements for fast data are mapped over to the external memory as well, the days when the processor only needed to address a few hundred bytes of data are long over. Today, some specialized processors aimed at video applications, for instance, must be able to handle 128 Mbytes of DDR SRAM memory. While it is relatively easy to implement larger address ranges on a processor chip, the speed of these memory interfaces is now critically important. The large address space translates into many pins on the processor dedicated to the external memory interface. The fact that there are multiple bitstreams required for many of these applications means that there must be an easy way to quickly switch the address bits on the memory interface. Most modern processors use full-blown DMA (direct memory access) controllers for this interface – typically three of them. Some even go the extra step of allowing indexed addressing in the controller. That's convenient, for instance, when the device is fetching multi-byte vectors from memory.

Low Power Dissipation

Many modern consumer devices are battery operated. The high processing load, combined with a display, and sometimes even a disk drive, place a heavy load on the batteries in these devices. As a result, power is at a premium and the processor itself must be capable of low-power operation to maximize battery life. Of course, low-power does not normally go hand-in-hand with high processing speed, so this represents a serious design tradeoff.



A chip with just two cores, one for real time tasks and the other for algorithm processing.

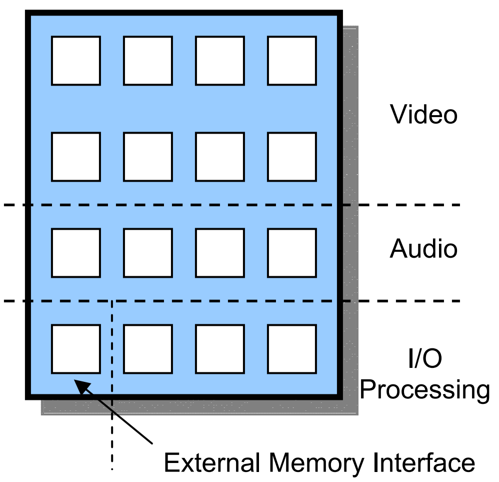
21st Century Multicore Processor Architecture

Changes in the nature of applications clearly require corresponding changes in the processor chip's architecture. For instance, the need for multimedia capability requires special high-speed arithmetic circuits. And the need for that high-speed processing has led chip designers to add core processors to the chip so that those tasks that require real time processing can be run on one core while the other tasks can be run on a second core.

Multiple Processor Cores

The approach of trying to segregate the tasks into two groups – real time and non real time – fails for the simple fact that in modern applications MOST of the tasks have a real time component to them. Simply put, multimedia applications are driven by high-speed computing elements that are racing to complete their algorithms within a tiny slice of time before the next batch of multimedia data arrives. Failure to do so means there is a gap in the music or a glitch in the video.

Recent trends have been to incorporate one or even two DSP cores with high-speed multiply / accumulator circuitry

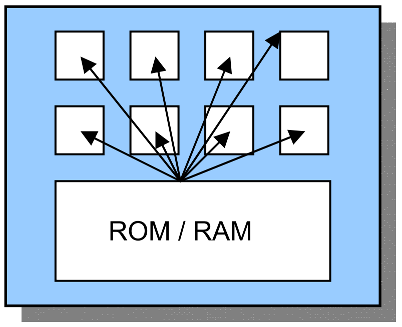


A chip with multiple cores showing how an application can be spread across them to maximize processing power.

that keep pace with those multimedia bitstreams. But this approach appears to be reaching its limit whereas the demand for additional and higher speed bitstreams seems to know no bounds. A much better approach is to integrate several more core processors onto the chip, each simpler than the complex DSP core, but each containing a high speed multiplier / accumulator. Properly designed, these core processors can take on complex algorithms by spreading the computing load across them and sharing the task. Of course this requires rewriting the algorithm in a way that facilitates this breaking up and sharing the task but the result can be an incredible increase in processing capability.

Spreading the computing task in this way has a second advantage. Whereas chips based on DSP cores have little flexibility, chips based on an array of core processors can be programmed to bring the optimum number of cores to bear on the problem. Need more speed? Simply assign more cores to the task. This approach has the benefit of strong computing power within the cores, so that unlike the DSP which consists mainly of high-speed arithmetic circuits, the core processors add high-speed conditional branching plus all the other powers of traditional computing elements. As a result, the multiple core approach is extremely flexible and its ability to solve problems is not limited to high-speed arithmetic.

The flexibility of multicore chips means they can be brought to bear on a wide variety of problems by simply assigning cores to the different tasks required. One can be assigned to managing external memory, perhaps eight more could be directed to doing the FFTs to process the multimedia algorithm, and several more can drive the various I/O subsystems in the application. This sharply contrasts to the traditional single-processor approach for handling multiple tasks. As everyone knows, that approach directs the single processor to work on one task for some period of time and then switch to another, and so on and so on, providing the illusion of a multi-tasking processor. In cases where some of the tasks are I/O bound and



A chip with 8 cores showing the bottle-neck that occurs when accessing a common shared memory.

spend significant time waiting for data to be received, that illusion holds up pretty well. But for tasks that are not waiting for data, the illusion breaks down and no one is fooled – the processor is simply sharing its resources among the tasks and the burden is painfully evident. The problem is exacerbated by the context switching time needed by the processor to save registers and application data as it moves from task. The larger and more complex the processor, the greater the context switching time and the more the illusion of multitasking breaks down. The multicore approach turns this on its head by assigning one or more processors to each task. The context switching time is zero for the simple reason that the individual processors never switch tasks, and the illusion of a multitasking chip becomes reality.

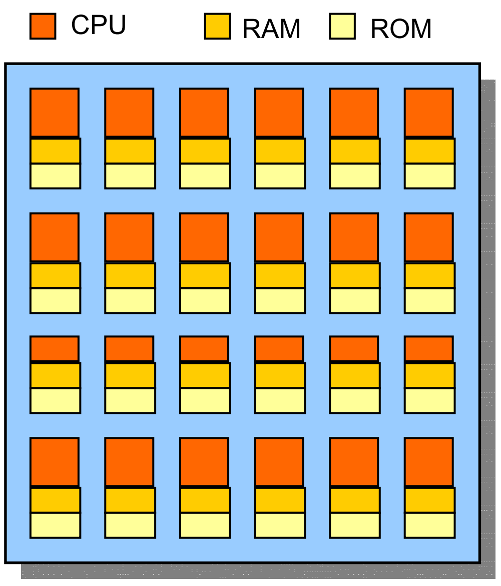
Local RAM/ROM Memory

Whenever multiple processors are incorporated into designs, the issue of memory access rears its ugly head. Most multicore chip designs combine several cores with a common memory structure. While this simplifies the design since each core consists of only the processor itself, the savings is replaced with the extremely difficult problem of sharing the common memory among multiple cores and arbitrating their accesses to it. This normally involves either some sort of arbitration network or crosspoint switch. This approach is workable when only 3 to 4 cores are contemplated, but when the chip design calls for dozens, as it does here, the complexity of sharing memory becomes daunting. In addition, as more and more core processors require memory access, the sharing becomes less and less efficient and quickly becomes a killer bottleneck that negates all of the processing gains that came with multiple cores.

The solution is to replace the common, shared memory with local memory that is local to each core processor. In this arrangement there is no need for memory arbitration or crosspoint switches because the cores are simply accessing their own, private RAM / ROM memory stores.

The concept of a common memory store offers one big advantage, namely the optimization of chip memory size by simply allocating to each core processor the amount of memory that core needed. When each core has its own local memory store, the size of that memory will always be a compromise. If it's too small, the cores will be handicapped – too large and it will be wasted and the chip will grow larger at the cost of efficiency.

Fortunately the size of that local memory is easy to set. By writing code and experimenting with typical algorithms



A chip with 24 cores, each with its own local RAM and ROM. With local memory distributed this way, there is no memory bottleneck.

that must be handled by the chip, it quickly becomes clear that the requirements fall into two sizes… 1,000 bytes and less and a much larger size… megabytes or even hundreds of megabytes. This second memory size occurs when large buffers are used for handling multimedia data, but that only applies to a few of the cores on the chip. Clearly, adding megabytes of local memory to each core would be extremely wasteful, even if it were practical. The first memory size, 1,000 bytes, is quite practical with today's mainstream semiconductor processes and is proving more than adequate as a working size for local core memory.

The final solution obviously is to have a relatively small local memory store for each core, on the order of 1,000 bytes, for code and data storage *plus* access to a much larger external memory for multimedia buffer requirements that is used by only a handful of cores.

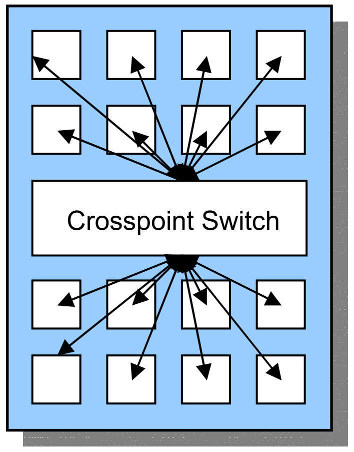
Communications between Cores

It is readily apparent that the idea of a multicore chip is not that of a set of core processor *islands,* each with its own set of I/O pins standing independently from the others. We have already described, for instance, how compute-intensive algorithms can be spread and shared among multiple core processors. Obviously that implies a level of communication and cooperation among the cores.

Communications between core processors takes two forms: passing status signals and passing blocks of data. Conceptually there is no difference between the two although there is a significant difference in the communication speed. For instance, a status signal might be sent to a neighboring core indicating that data is ready for transfer, and then the cores communicate by passing that block of data between them. While both of these communications approaches must be efficient, the way that efficiency is achieved may be completely different. We will return to this in a moment.

As in the case of shared memory, communications between processors can be handled in several ways. If there are only a couple of core processors involved, it's practical to provide circuitry for each to communicate with the others. But as the number of cores increases into dozens, the chip area and complexity of the communications circuitry becomes prohibitive. Another way to implement inter-core communications is to limit the communications to a smaller set of cores, typically to just a core processor's nearest neighbors. This is far simpler and very practical.

The implementation of inter-core communications structures goes right to



A 16-core chip using a crosspoint switch for core-to-core communications. This quickly becomes a bottleneck with more than four or five cores.

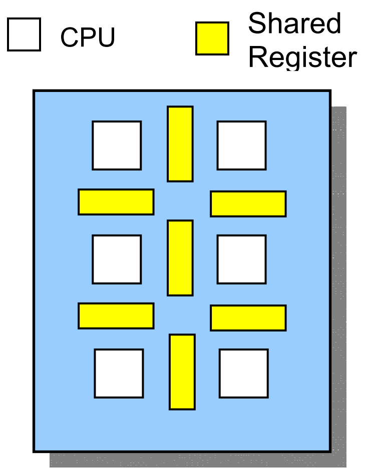
the heart of the philosophy of bringing a sea of processors to bear on a problem. How are communications channels and processes created? As computer users, we are accustomed to letting the computer make many of the decisions regarding the applications we run. For instance, when our word processor application needs more memory as our document grows, we rely on the computer to find a block of memory and assign that block to our word processor program, a process that might entail reassigning blocks and moving some to disk. That process is completely invisible to us and is done, as needed, by the computer.

Less obvious is the fact that the memory allocation system and even the disk operating system were designed to make this process efficient to drive for a software entity, in this case, the word processor program. The system was designed from the very beginning with the idea that it would be the computer operating autonomously that would allocate the block of memory and move other blocks to the disk drive, as opposed to a human being.

In the case of the multicore chip, just how will the cores be assigned to perform the various tasks that make up the application? It is not going to be the application program itself, or even some operating system "in the sky." The process of assigning cores to tasks is done by the designer / programmer who maps the application onto the chip, not by some development system program. The mapping process is one of the most basic, fundamental parts of the design problem. To do it, the designer must ask which tasks communicate the most data, and then assigns adjacent cores to those tasks to optimize the core communications. If this core assignment process was going to be done in some automated fashion by the development system, then it would be appropriate to design an inter-core communications system optimized for that automated assignment process. But since it is done by the human designer, it is much better to use the simplest, most efficient communications structure that simply restricts the core communications to nearest neighbors. Of course, it is always possible to have cores relay data and status signals to more remote cores, but by restricting direct communications to nearest neighbors, the chip design is made much simpler and there is no real cost to the applications designer who was going to do the assign core tasks anyway.

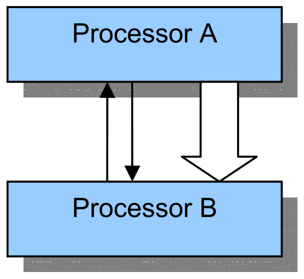
This conflict between automatic design and design by humans targeting specific applications will arise over and over again. Whereas our computer functions one moment as a word processor and the next as a movie player or a financial spreadsheet calculator is completely different from how embedded processors function. An embedded processor chip does not switch back and forth between being a camera and a wall thermostat, and for that reason we should NOT compromise chip design by burdening it with generic do-anything, anywhere, anytime structures like large crosspoint switches that allow communication between any two on-chip core processors.

Once the decision has been made to limit communications to nearest neighbor cores, the communications structures become much simpler and it is possible to make them even more efficient. Communications between cores now takes place through shared registers and there is no need for conflict resolution or priority networks. But what is possible



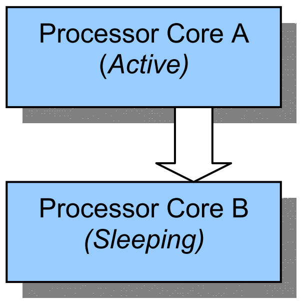
A 6-core array (2x3) using shared registers for core-to-core communication removes the bottleneck issue.

is to combine some aspects of status signals with the communication of data. Traditionally two processors passing data through a shared register will poll a status bit somewhere to determine the state of the transfer. Processor A sends data to the register and sets the status bit HIGH signaling that data is present and needs to be read. Processor B is polling that status bit in a software loop waiting to see it go HIGH indicating that fresh data is present in the register. After reading the data, processor B resets the status bit LOW indicating the data has been read and the register is ready for another transfer. There are many variations on this theme, but the sad fact is that more time is spent in having the two processors read the status bit, test it, and write it, than is spent actually transferring the data.

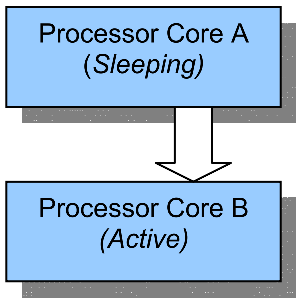


Two processor chips communicating a word from A to B using two hand-shake lines. In these arrangements more time is spent reading and writing status bits to the handshake lines than in transferring the data.

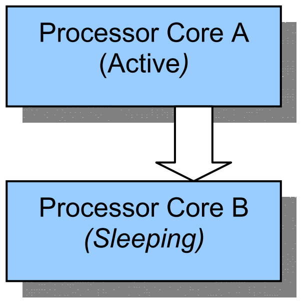
The multicore chip offers a much simpler solution. Write the code for core-processor A so that it always assumes the register is empty and waiting for data. Its loop no longer contains code for testing and writing the status bit, but becomes simply SendData – SendData – SendData, and so on. Likewise the code for core-processor B assumes there is always data waiting so that its loop is now simply ReadData – ReadData – ReadData, etc. How is this done in practice? Core-processor A, the sending core, attempts to send data to the shared register and if there is still unread data in the register, core-processor A simply stops running. It stops until the data in the register has been read by B, and at that point A starts back up again on the very instruction it had started before, i.e. SendData. Thus, from a code standpoint, core-processor A always assumes the register is empty and waiting for more data… there is no reason to read and test a status bit. Core-processor B does something similar. Its code always assumes the register is full of unread data. As it begins to execute the ReadData instruction to get that data from the register, if it turns out there is no unread data in the register, it too simply stops running. When new data does appear, B finishes executing its ReadData instruction which then successfully gets the data from the register. Again, there is no need for reading, testing, and setting a status bit.



Core A begins the transfer process of sending a word to the sleeping core B.



Core B completes the transfer process by waking up and accepting the data word. Meanwhile core A goes to sleep while B accepts that word.



The process of B accepting the data word wakes up core A which begins the process again, sending the next word. B has gone back to sleep. From each processors point-of-view, the other processor is always ready to either send or receive. There is no need for handshake lines.

This technique will be unfamiliar to most readers because it is not an option in systems where the processors are on different chips. The reason it works is that, when both cores are on the same silicon chip, there are circuit techniques for starting and stopping core processors that can be utilized. The key is that the start / stop process has to be very fast – on the order of one instruction execution time to be really effective. But when that can be achieved, the speedup in data transfer between core processors is dramatic and improves the throughput by a factor of several times. In effect, it completely eliminates the software signaling between cores for many types of data transactions.

If the core processors are designed to use memory-mapped I/O, even more interesting types of communication can occur between cores. In this system, I/O registers are treated as memory addresses which means that the same instructions that read and write memory also perform I/O operations. But in the case of multicore chips, there is a powerful ramification of this choice for I/O structure. Not only can the core processor read and execute instructions from its local ROM and RAM, it can also read and execute instructions presented to it on I/O ports or registers.

Now the concept of tight loops transferring data without the need for reading, testing, and writing status bits becomes incredibly powerful. It allows instruction streams to be presented to the cores at I/O ports and executed directly from them. And since the shared registers between cores are essentially the same as I/O ports, that means that one core can send a code object to an adjoining core processor which can execute it directly from the shared register with no need to actually transfer the code to the other processors local memory. Code objects can now be passed among the cores, which execute them at the registers. The code objects arrive at a very high-speed since each core is essentially working entirely within its own local address space with no apparent time spent transferring code instructions.

Real Time Clocks

As traditional processors have grown in processing speed and complexity, they have moved further and further away from their ability to handle tasks in real time, meaning the time to process code is indeterminate and will vary from cycle to cycle. This is largely due to the introduction of increasingly larger caches used by the processor to reduce external memory accesses. Thus, on one loop through the code the instructions are all fetched externally, but on the next they are contained within the cache. At the same time, as processor complexity has grown, the number of CPU registers has increased as well. Accordingly, the amount of time required to save the contents of those registers during interrupt handling has increased. All of this makes modern processors ill-suited for embedded applications, to say nothing of the large memory requirements and sheer chip cost.

Embedded processors have always stressed the ability to handle real time applications, to process code in a guaranteed time slot, to handle events and displays within a tightly controlled (and shrinking) time allotment. Single processor chips use a real time clock, supplied by an external reference, to setup and control those tasks. But what is the ideal arrangement in a multicore chip?

Thinking about the application as a set of related tasks and subtasks, with cores assigned to each, provides an answer. Modern applications, especially those that are multimedia intensive, are not characterized by one or two tasks that must be accomplished within a time slot. Today, many if not most of the tasks have a real time component to them. Consequentially, one core will need to have access to the real time clock reference which it uses to inform the other cores by sending status signals to them in the form of messages, or each core must have the capability of accessing that reference clock directly. Of the two, the latter is a much better solution.

If status signals can be eliminated by each core having its own access to the real time clock, that combined with the lack of need for status signals to transfer data between cores, goes a long way to eliminating the status signal form of communication between cores altogether. Notice we are not suggesting that a system clock signal be distributed across the cores requiring millions of nodes to be switched synchronously to the beat of that clock. For the real time clock to be effective, only a handful of nodes in each core must be switched, and the effect on power dissipation is negligible. A simple counter on each node is more than sufficient to make each node self-sufficient in terms of real time processing.

Low Power by Design

As more and more embedded processor chips find themselves in mobile applications, the requirement for low power dissipation has become critically important. In traditional designs this is achieved through excruciating attention to detail, carefully determining the speed at which each signal path must operate and then choosing transistor sizes appropriate to that speed. Only the highest speed paths are implemented with large power-hungry transistors.

But the multicore chip, with the ability to start and stop core processors as data is presented or denied, has a much simpler power-saving mechanism. Cores that are not processing data are not running and therefore are not dissipating any power. Cores only run as they are needed and the turning on and shutting off is completely automatic and need not be invoked by the program.

The effect on power dissipation is much larger when complete cores are shut down than by trying to gauge and size signal paths. In fact, this approach has a second benefit. Because of the automatic synchronization of data passing between cores, there is absolutely no reason to make the cores themselves synchronous. That means, there is no reason to have a central clock to which each core must beat. Data transfers always take place at the highest possible speed – an external clock adds nothing but complexity. Now the central clock is replaced by an individual clock for each core – a simple ring oscillator – that runs as fast as the native speed of the silicon allows. No central clock means there is no giant clock tree with millions of transistor nodes dissipating power at each tick. Instead, the tiny individual clock oscillators run on each core, but *only if that core is running.* If a core has been stopped because data is either unavailable at its shared register or has not yet been read by a neighbor, the ring oscillator is also stopped. Clock dissipation only occurs in running cores, and even then these are fully asynchronous with regard to each other so that the power dissipation is spread over time.

In a chip such as this, with dozens of core processors, only a fraction of those cores are running at any given time. Some of these cores will be off for significant amounts of time because the chip is in a mode that does not run tasks involving those cores. But even the cores that are running are doing so in short spurts, first turning on and executing code as fast as silicon will allow. Then immediately shutting back off as they exhaust the data presented to them or waiting for a neighbor to pick it up and continue. In this type of environment, we estimate only a third of the cores would be running at any given instant, though a few nanoseconds later, a different group of cores would be active, but still only about a third. This effectively reduces the power dissipation of the entire chip by a factor of 2/3 while at the same time ensuring that each core runs at the maximum possible speed of the silicon with no compromises.

Instruction Sets

Instruction sets are mostly determined by the register set associated with the processor. In the case of the multicore chip, however, the core processors are carefully designed to provide maximum speed with minimum size and complexity. In other words, they are RISC processors, that are carefully optimized to run code using a very simple reduced instruction set. By far the best match of processor architecture and processor language is to have the processor execute instructions in some high-level RISC language as native machine code. This accomplishes two things: first it packs the maximum amount of functionality into the smallest programs and second it maximizes the speed of execution by eliminating the need for intermediate translation between high-level source code and machine code. The first is critical in chips with limited memory sizes and the second is equally critical when processing demanding multimedia application algorithms.

That leaves the question of which high-level language to implement as the machine code instruction set on these core processors, and here, the choices are few. Most modern high-level languages are designed to pass large amounts of data to a set of functions and subroutines as frames on the return stack. This process is largely invisible to the programmer as it is hidden behind the machinations of the language compiler. But that approach is wildly inefficient for core processors of the type we're envisioning as the embedded chip of the future. In this case, the processor may be RISC but languages like C and C++ are definitely not RISC. Fortunately there is a language that is optimum for these types of cores – so optimum in fact it appears that it was designed with multicore chips in mind. That language is Forth.

Forth is ideal for small processor cores for several reasons, but the first is simply that it does not use a large number of processor registers. The hardware needed to implement a Forth-based processor is minimal. And because Forth programs are written by defining new words and then using those to define higher-level words yet, it is easy to identify a small set of core words – the kernel – that everything else is built on, and then building those core words into the processor as dedicated circuitry. The result is blinding speed in a very small core processor.

By implementing as few as 32 instructions in that core set, it is possible to achieve the ideal RISC compromise where the minimum instruction set handles the majority of applications code directly within that set and at the same time does not pad out the set with seldom used instructions that complicate the circuitry and ultimately slows execution. Clearly, an instruction set with only 32 instructions can be implemented in as little as five bits, but by recognizing that some instructions only apply in certain contexts, it's possible to pack multiple instructions into a small instruction word… as many as four instructions in an 18-bit word.

Instruction packing like this achieves an automatic caching effect with no need for setting up L1 and L2 caches. Instead, each instruction fetch brings four instructions into the core processor. Although this built-in cache is certainly small, it is extremely effective when the instructions themselves take advantage of it. For instance, micro for – next loops can be constructed that are contained entirely within the bounds of a single 18-bit instruction word. These types of constructs are ideal when combined with the automatic status signaling built into the I/O registers because that means large blocks of data can be transferred with only a single instruction fetch. And with this sort of instruction packing, the concept of executing instructions being presented on a shared I/O register from a neighboring processor core takes on new power because now each word appearing in that register represents not one, but four instructions. These types of software / hardware structures and their staggering impact on performance in multicore chips are simply not available to traditional languages – they are only possible in an instruction set where multiple instructions are packed within a single word and complete loops can be executed from within that word.

No Central Operating System

The idea of multiple cores on a single chip is certainly not new, and in fact there are at least a dozen already on the market or about to be introduced. But virtually all of these are made up of two or four cores where those cores are large, complex processors designed to run desktop applications such as Windows. There is certainly a place for these — not in highly compact embedded applications — but in large servers. Such multicore processors all rely on a central operating system to load and direct the core processors.

This arrangement is usually typified as SMP – Symmetric MultiProcessing – where each of the cores is identical. To be successful it assumes that the software being run has been written in a multi-threaded form. The operating system, probably running on one of the cores, takes that code and loads it onto the remaining cores by separating the code into blocks which set off the individual threads. It loads the cores in a way to equalize the processing load across the cores using the threaded code blocks as the basic code increment. Where applications have been written in this multithreaded format, the multicore SMP approach works fairly well.

Of course not all software is written that way, but even when it is not, the central operating system can load entire programs onto individual cores, so that some benefit of the multiple cores can be seen. But none of this applies in the case of embedded processors. There are no disk drives, no loading of cores with tasks on-the-fly, dynamically controlled by a central operating system. Simply put, there is no central operating system in an embedded processor. In the case of multicore chips, the role of the central operating system has been replaced with the concept of the thoughtful programmer.

For these kinds of chips, code is written for specific cores on the chip. It is not designed to run independently on any given core, since each core is connected to the outside world with a different set of I/O functions. The code only makes sense in the context of the core for which it was written. This is not a drawback of the approach, since the system has already been determined to be a camera, for instance, and not a camera one minute and a breadmaker the next. If the cores were to have totally different tasks minute to minute, you could argue for the presence of a controlling program like a central operating system. But since that flies in the face of the entire concept of the embedded processor, there is no central operating system.

This presents a slight problem. PCs, for instance, do not simply have an operating system, they also have a BIOS (Basic Input Output System) the operating system is built on. That BIOS implements the most basic level of I/O drivers in the system. And while the multicore embedded processor needs no central operating system, it still has the need for basic input / output drivers. And if we are going to avoid the idea of central, shared memory we are going to have to accept the idea of each core processor having its own BIOS.

Since each core has its own ROM memory, it also has the ability to have its own BIOS. In addition to simple input / output functions, the core processor BIOS can have all sorts of helper routines as well. These BIOS routines are not simply copies, replicated in each core's ROM across the chip. They must be individualized to handle the individual personalities of the cores. Although the cores themselves are the same, their location within the chip array makes them unique. Some connect to certain types of I/O, while others connect to other cores. Cores in the middle of the array probably have no external I/O at all beyond the shared registers that are used for inter-core communications.

Multiple I/O Interfaces

As embedded processors have moved from the original form of the 8048/8051 to modern processors, the nature of the I/O has changed as well. This is true regardless of whether we're discussing single processors or multiple core processors. Whereas originally simple parallel I/O lines plus a serial interface was sufficient, chips today must interface with other predetermined interfaces like USB, 1394, and SPI (Serial Protocol Interface).

Today there are hundreds of peripheral chips utilizing the SPI interface, and a processor chip that provides a SPI interface (or multiple SPI interfaces). This opens up a world of inexpensive, powerful peripheral functions that can be easily incorporated into the system.

Scaleable Embedded Arrays

All of this time we've been discussing multicore chips without regard for the layout, the arrangement of the cores. But if the cores are identical, outside of their ROM contents that is, then the number of cores in the array is largely arbitrary and is set by the simple economics of the chip size as related to the demands of specific applications for processing power.

Chips laid out by simply replicating cores makes them scaleable – if there are not enough cores to do the job, pick one with more. Additionally, many (but not all) of the same structures available for inter-core communications are also available as cores communicate from chip to chip. Accordingly, applications can be scaled by adding multiple chips to increase the number of cores, memory, and I/O.

IntellaSys specializes in innovating multicore processor solutions that target embedded applications requiring low-power operation, fast operating speed and a small footprint. For more information visit: www.intellasys.net.