WHITE PAPER

BROADCOM®

Practical System Design and Debug Considerations for Multiprocessing in the Embedded Environment

16215 Alton Parkway • P.O. Box 57013 • Irvine, CA 92619-7013 • Phone: 949-450-8700 • Fax: 949-450-8710

MULTI-PRO-WP100-R
09/12/02

REVISION HISTORY

Revision	*Date*	*Change Description*
MULTI-PRO-WP100-R	09/12/02	Initial release.

Broadcom Corporation
P.O. Box 57013
16215 Alton Parkway
Irvine, CA 92619-7013

TABLE OF CONTENTS

LIST OF FIGURES

The practice of using multiple processors in embedded designs has become almost universal over the last ten years. The practical reason for this is to provide system-wide processing power that greatly exceeds the capabilities of a commercially-available microprocessor. Until recently, virtually all of the multiprocessing configurations have been constructed at the board or system level by the user. Now, however, there are new devices coming onto the market that incorporate multiple microprocessors on the same die. Using these devices effectively requires an understanding of why these devices exist and the options they provide to a system designer.

PROCESSING TOPOLOGIES

The most common multiprocessor architecture is a collection of processors in a loosely-coupled configuration. The term *loosely-coupled* indicates that these processors interact over a communication channel. This communication channel does not have to be a conventional serial or parallel channel. It is quite often a shared area of memory used by processors on the same board, or over a backplane. The key characteristic of these processors is that they have their own independent memory subsystems. Each processor acts more or less as an independent node.

A more recent architecture in the embedded space is a *tightly-coupled* processing pair, or array. These processors share a common bus and addressable memory space. In the overall system configuration, they operate as a single node, and appear as a single processing element to the rest of the system. Whereas adding more loosely-coupled processing nodes increases the overall processing power of a system, adding more tightly-coupled processors increases the power of a given node.

METHODS OF INCREASING PROCESSING POWER

When it comes to increasing the processing power of a single node, why add multiple processors instead of using a single processor that is simply more powerful? The reason is that there are limitations to what can be done to increase the speed of a single processor.

For example, increasing the clock speed of the processor necessitates having to design board-level signal paths and wide busses that can keep up with the higher clock speed. Also, even though clock speeds are always increasing, at any given time the fastest might not be fast enough.

Besides increasing clock speed, another approach to increasing processor performance is to increase the parallelism of instruction execution within a CPU. This is already being done in modern RISC CPUs, but there is a practical limitation to this approach. When instructions are executed serially, they can always execute when they are fetched. When two instructions are issued in parallel, however, there is the chance that one of the instructions will affect or be affected by the other instruction. If this happens, the hardware must be prepared to interlock and force serialization of the instructions. With dual instructions, there is the opportunity for a single interlock. For the case where three parallel instructions are issued, there is the opportunity for three interlocks. Four parallel instructions could potentially cause six interlocks. Thus, increasing the parallelism of instruction execution can have diminishing returns. Although code can always be hand-written (usually by the CPU vendor) to display the potential processing power of a CPU, the practical output of compilers used by a commercial software development team rarely is able to make use of processors that can issue more than four instructions per clock.

Another drawback associated with increased parallelism (and to some extent, increasing clock speed) is that the disadvantages of additional design complexity and power consumption offset much of the benefit of performance gain. The relatively low power consumption of the MIPS®/SB-1 processor compared to a Pentium® processor illustrates the difference in power consumption between a multiple processor and a single processor.

Because of the limitations and drawbacks associated with higher clock speed and internal parallelism, the best option for effectively and reliably execute the code image is to add additional processors. This method allows the execution of parallel threads, something not possible in a single, highly-parallel CPU. On the other hand, with more processing power comes more complexity. Therefore, the key to getting the most out of using these systems is the proper understanding of code execution, memory allocation, and how system resources will be distributed.

TYING TWO PROCESSORS TOGETHER IN HARDWARE

Multiple processors must be connected in hardware so that they can operate as a single node. A generic configuration is shown on Figure 1. This oversimplified diagram shows the key point that the processors have access to a common bus, which is the most basic form of a coherent interconnect. Each processor can perform identical functions with respect to any other device on the common bus. Also, any device has an equal opportunity to generate interrupts to either processor.

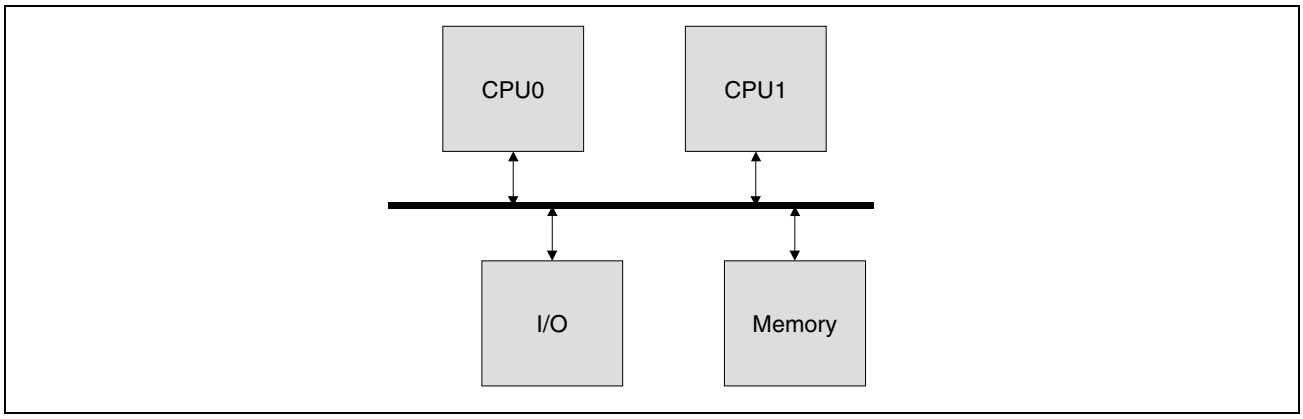

Figure 1: Simple Dual Core Configuration

A more complex, and therefore more accurate diagram is shown on Figure 2 on page 3. This is actually similar to the internal configuration of the Broadcom BCM1250 processor. As can be seen by these diagrams, a single, high-speed coherent interconnect in the form of a common bus provides the tightly-coupled aspect of the configuration. This bus is the both the advantage and the vulnerability of the system.

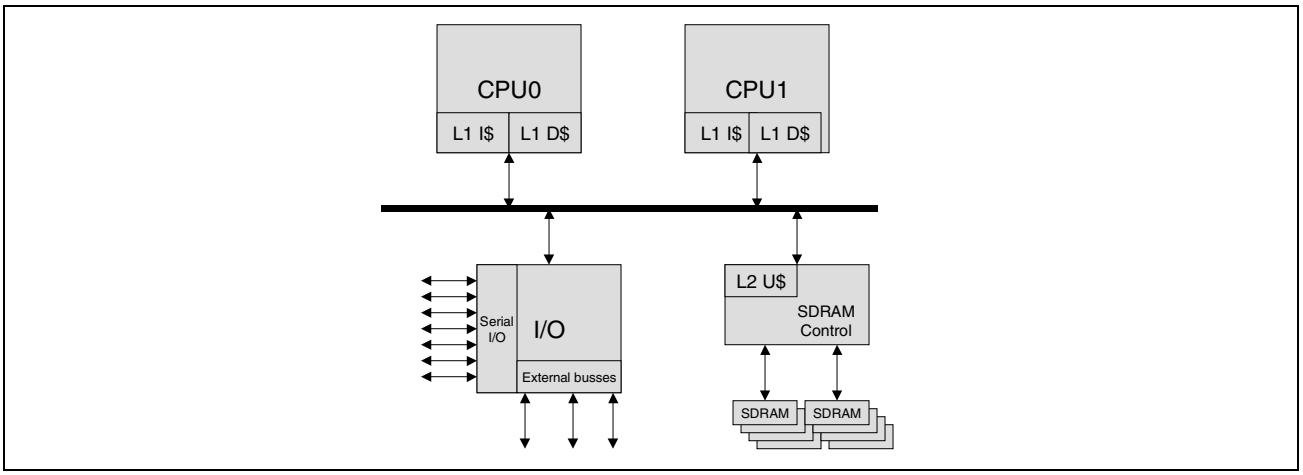

Figure 2: Practical Dual Core Configuration

Because both processors are on the main bus, resources are allocated to each processor by the software. This can be done as part of the system definition, or dynamically at execution time. This is obviously an advantage in that the memory allocation and I/O servicing responsibility remains fluid. As an example: the system designer can partition the tasks associated with a communication stack to each processor according to protocol level, transmit versus receive, communication channel, and so forth. Also, these assignments can evolve and be changed even after the product has been shipped to customers.

The problem with this configuration is that the main bus is also the opportunity for bottlenecks during operation. The solution to this problem is a wide, high-speed bus. This bus should also have a protocol that can support cache coherency and peer-to-peer communication between processor L1 caches. Of course, placing and routing a wide, high-speed, multidrop bus on a board is not a small challenge. Extending the bus past 133 MHz in speed and 64 data bits in width stresses practical design considerations. The escape pattern for modern ball grid array devices is complex in itself. All of these issues tend to push the board layer count to the edge of economic viability.

This interconnection issue is the major obstacle to tightly-coupled processor systems. It is also the primary reason that silicon vendors are moving to place multiple processors on a single die. Semiconductor manufacturing techniques allow for much higher clock speeds and bus widths than are practical on a circuit board. The BCM1250 internal bus width is 256 data bits and 128 address and control signals. It operates at 50 percent of the CPU clock frequency, or 500 MHz for 1 GHz devices.

Software Design for Multiple Processors

Despite the hardware issues, the most complex challenges to a multiple processor design are in the software. At the highest level, the question is how to partition the code execution among the processors. This is obviously a critical decision that must be made early in the system design cycle. An improper method of allocating execution can eliminate any potential advantage of using more than one processor.

The two most common approaches to allocation functions across processors in a tightly coupled system are through static assignment at system design time, or through symmetric multiprocessing (SMP). Static allocation can be used at any level in the software architecture, and is always present at the lowest levels, especially at boot time. SMP allows the operating system to dynamically schedule tasks to whichever CPU has the most available idle cycles. As a result, this level of allocation

sits above the operating system scheduler. It should be noted that more advanced SMP operating systems will also allow the dynamic scheduling of operating system services such as communication stacks.

Making the allocation of execution cycles across processors more dynamic causes the system to be less deterministic. This has a huge impact on the debugging environment. Because the OS can assign the execution of a task to a processor dynamically, only the operating system can track a task to allow visibility with debug tools. Typical in-circuit emulators and basic debuggers are unable to deal with this type of environment. It is important that the operating systems provide adequate debugging support, and not just support only the execution environment for SMP.

SMP also requires that the operating system manage both cores. While it is possible that an operating system could have most of its code confined to one CPU (with only a minimal scheduler on the second core), some operating systems duplicate most of the operating system on each processor. Each CPU ends up allocating pretty much the same number of cycles to executing overhead instructions as the other CPU in the pair. There is a significant difference between the advertised versus the technical ability to support multiple processors. To ensure that SMP provides enhanced performance over a static asymmetric design, it is necessary to make a very close examination of the implementation details of how the operating system consumes CPU cycles.

An embedded system is typically a closed environment; therefore it is rare to have unexpected new code introduced into the system that is not part of a planned upgrade. Because of this, the allocation of tasks across the CPU can be done during software design. This allows the operating system to be restricted to one CPU, and to have operating-system-agnostic code run on the second CPU. The advantage to this is that the CPU without the operating system is able to devote a larger percentage of its cycles to the application. It also means that critical code that has response time restrictions can be executed without the overhead of operating system context switching.

MEMORY ALLOCATION

When more than one processor has access to a common memory system, do the processors allocate from a common heap? Do they have separate executable images, or do they use a single reentrant image? There are a number of issues to consider when allocating memory space between processors. These issues include code and data space, which are treated differently.

The data space of a processor contains static initialized and uninitialized variables, dynamically allocated memory (heap), and the stacks. The individual characteristics of these different data types determine if they are to be shared between processors.

The stacks must be separate for each processor. In an operating system managed environment, each task might have its own stack, and if SMP is used, the task and its stack can be passed back and forth between CPUs. However, the system stacks must be separate. It is the system stack that is usually referred to as "the stack" and it is this data structure that is allocated in the linker command file.

Static variables, whether initialized or not, present more of a sticky problem. If a task is going to be passed back and forth between CPUs, or may be scheduled to run dynamically between CPUs, then it must be able to access these static variables. Variables that are local to a task can be confined to an area of memory, and a pointer can be passed with the task. For variables that are system-wide in scope, a common area of memory must be specified and available to both CPUs. The allocation of static variables is far simpler if tasks are not dynamically shared among processors. The only variables that need to be accessed by both processors are those that are specifically designated for communication.

The heap can be shared or separate. A shared heap is theoretically more efficient because each processor can use as much remaining memory as needed. Allocating separate heaps for each processor can result in one running out of memory while

the other still has unused space available. Again, practical considerations in the embedded system tend to mitigate the advantage of a common heap. It is usually easier to predict the heap usage by each processor. Also, a common gateway routine, which would be used by both processors to allocate and free the heap, can be a bottleneck between CPUs.

When using multiple CPUs, each typically has its own MMU. Therefore it is possible to have address aliasing between processors. The advantage of having separate virtual address maps is that code for each processor can be completely protected from the other processors. This can be a significant factor in designing a stable system. On the other hand, if identical maps are used, then simply passing pointers allows the exchange of information between processors. This method of passing by reference can have an enormous performance advantage over copying data between CPUs. Fully protected memory support is usually supplied by an operating system, which usually support shared physical addresses for memory areas that are accessed often. This allows a compromise between full isolation and maximum performance.

INTERPROCESSOR COMMUNICATION (IPC)

During code execution, it is common to allow processors to communicate with each other. Because task scheduling is quite often asynchronous, most of the issues associated with having processors communicate are already handled by single processor designs, with some important exceptions.

In single processor systems, task entry and exit usually pass through operating system code. Therefore, raising a flag or semaphore that alters execution priority will cause the appropriate code to execute on the next system call or clock tick. In multiple processor systems, it may be necessary to cause an exception of the other CPU to guarantee a timely response. The BCM1250 has mailboxes that can trigger an exception when written. A CPU can set a flag in these registers, or write an entire pointer to cause the other CPU to respond immediately.

A more subtle issue is that of testing and setting of flags or semaphores. RISC machines do not allow indivisible read-modify-write cycles. They provide atomic access through bus snooping and conditional stores. Basically, a read of a semaphore is performed, followed by a conditional write instruction to seize the flag if it is available. If there are no intervening exceptions or writes to the semaphore, then the write executes with a condition that can be detected by the CPU. This mechanism will work at its basic level with single or multiple processors. However, slight differences in the execution environment for multiple processors can result in live lock.

In single processor systems, tasks can be nested within each other because of preemptive scheduling. Because preemptive task scheduling passes through the processor's exception mechanism, this can allow a task to attempt to take a semaphore within the test and conditional store of another task. This is not a problem because the second task will get the semaphore and the first task will see its store fail. This can, in theory, nest as deeply as the entire task schedule, and will function and maintain order in all single processor systems. The only caveat is that the test and set loops must be nested by preemption through an exception.

In multiple processor systems, things can get more complicated. It is possible for two CPUs to enter a test and set loop on the same flag at the same time. Processor "A" would perform a read of the flag, followed by processor "B." Although the read of the flag by processor "B" is not guaranteed to cause the failure of the subsequent store of processor "A," this behavior is implementation dependent. It is possible for the two processors to spin forever, with the read of each causing the store of the other to fail. To solve this problem, the BCM1250 processor supports only the MIPS LL/SC instructions to cacheable memory. This forces the flag location to reside in L1 data cache of one of the processors. This automatically introduces a different latency for each processor. This prevents the occurrence of a live lock situation on a flag. In fact, the device is specifically designed to prevent such an event. However, when dealing with multiple processors that have not had this issue addressed by the silicon designers, the software must be prepared to detect and survive extensive time in a spin lock.

BUILD METHODS

There are several approaches that can be used to construct an executable image. If the operating system is not designed for multiple processors, then the system designer has to perform a build that will allow code to execute on each processor. One method is to build two independent applications, creating a separate image for each processor (see Figure 3). On the Broadcom BCM1250 processor, both cores use the same reset vector; however, only one core is released from reset at a time. This allows the first core to perform system-wide one-time initialization before releasing the second processor. If the second processor is going to start fetching code from the same boot flash device as the first CPU, then the startup code must first check the processor number field in the processor ID register. The code will branch based on this value and from that point on execute independently on each CPU. Using this method, one build is performed and, while each processor executes out of an isolated area of the final image, the code is the result of a single link. Therefore, global memory locations shared by the processors can be referenced to symbols, which are located at link time.

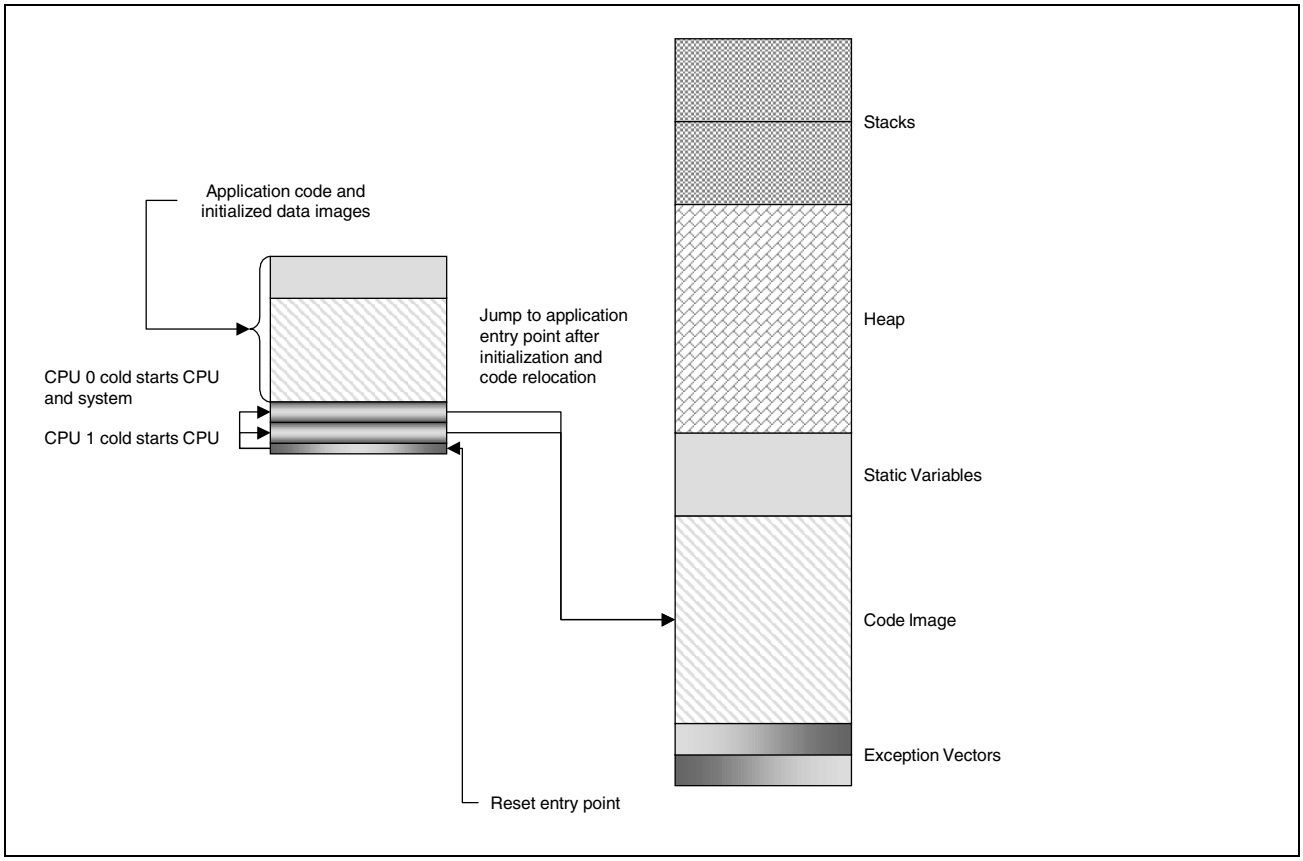

Figure 3: Single Code and Data Sections

A variation of this method is to build separate images for each application and a third routine to sit at the reset vector (see Figure 4 on page 7). Each of the main executable images would have a predefined static entry address for startup. The small reset vector code could read the processor number and then jump to one of the two specified entry points.

The other option would be to build a common image that is simultaneously executed by both processors. Because each of the CPUs on the Broadcom 1250 processor has its own 32 K 4-way set associative instruction cache, there is little concern of a bottleneck when both processors are trying to fetch instructions from the same part of the code image. In such systems,

the initialization assembly code can load the CPU registers for the stack, heap, and static data to point to different memory areas for each processor. Thus, while both processors are running out of the same image, they are effectively running independently.

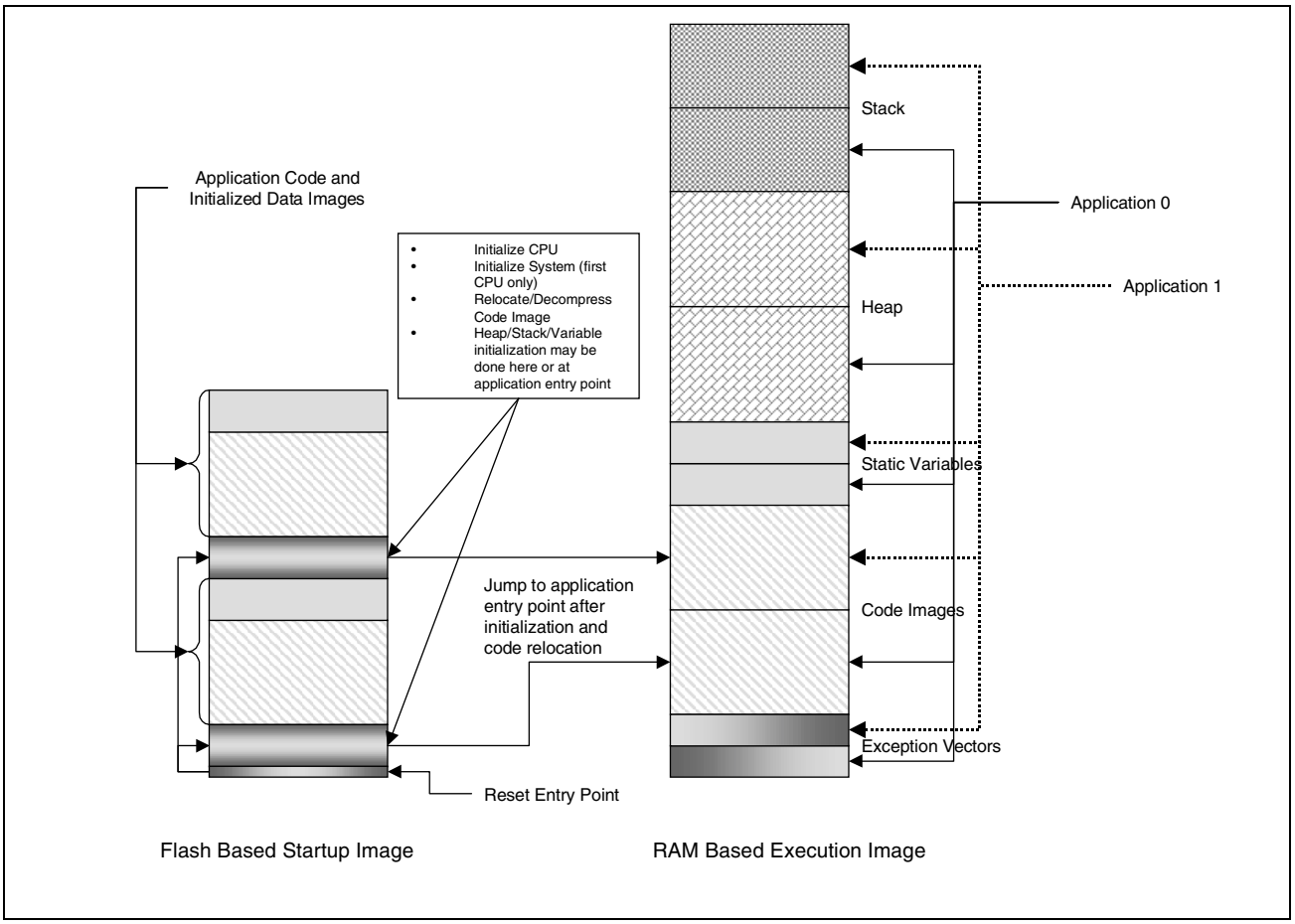

Figure 4: Separate Code and Data Sections

PRACTICAL CONSIDERATIONS

Although there is an almost infinite variety of ways to approach a multiprocessor design, there is usually a limited number of methods that are feasible for a particular system. The most effective way to choose system architectures is to focus on the methods that will be used to get them working.

Start by looking at the tool suite that will be used for development and debug. Although running one processor without an operating system can provide better throughput on that CPU, are the tools available to debug in a real-time operating system (RTOS)-free environment? It is important that the development team can work with emulators and/or debuggers that do not require RTOS services. Most RTOS debug tools are very powerful, but their features are supported by extensive facilities running in the operating system on the target. RTOS-agnostic tools are usually more basic in their operation. An RTOS will allow an engineer to stop or pause a task while the rest of the system continues to run. Typical emulators and debuggers

that are not running with an RTOS will halt the CPU when a breakpoint is hit. Running one CPU with an RTOS and one without will result in separate development environments for each processor.

If SMP, or any other method that dynamically allocates tasks between processors is used, the ability to corral such tasks is important. If a breakpoint is set, will it stop the task on either CPU, or only on one? Software breakpoints will stop an execution thread regardless of where it is being executed; hardware execution breakpoints are usually local to a single CPU. Unless the operating system provides extensive and easy-to-use tools for handling such events, assigning a task to a particular CPU is the most practical solution. It is also important to remember that when one CPU halts for a debug event, the other CPU is likely to continue to run. While asynchronous operation is to be expected during normal operation of more than one CPU, be prepared for unexpected behavior if one CPU suddenly begins executing alone.

Similar considerations are also important for the data areas of memory. The more fluid and open memory usage is between processors, the more difficult it is to identify a particular access. Having separate heaps and initialized data areas with tightly controlled communication structures at fixed memory addresses greatly helps in being able to isolate a problem. Also be aware of potential stalls from one CPU waiting at a spinlock for access to a common structure if the other CPU is stopped at a breakpoint.

CONCLUSION

A realistic method of determining the layout of execution and memory allocation is to see everything from the debug point of view. At each system design decision, stop and ask how each component will be handled if it doesn't work. Some specific issues that should be determined early in the design process include the following:

- Determine if it is reasonable to restart from a breakpoint on one CPU without restarting the entire system. This can kill the debug schedule if the system must be restarted and it takes a while to reach a particular state.
- Does one CPU require the other to be alive for it to remain stable? The ability to isolate and determine the basic functionality of each processor is critical.
- In statically allocated execution environments, be prepared to reassign tasks between CPUs after a first pass at execution determines real-world conditions. This is especially sensitive if one CPU is running an operating system and the other is not.
- For common data structures, do the tools allow trapping accesses for either processor as well as isolating accesses from a single processor? The Broadcom 1250 processor has internal bus trapping that supports both options.

It is often true that the more theoretical or novel approaches to multiprocessor designs do not work as expected in actual practice. A system that cannot be properly debugged cannot be shipped, and there is no point in designing a system that won't leave the laboratory.

Broadcom Corporation

16215 Alton Parkway
P.O. Box 57013
Irvine, CA 92619-7013
Phone: 949-450-8700
Fax: 949-450-8710