Multi-core and Multi-threaded SoCs Present New Debugging Challenges

Earl Mitchell
Senior Software Engineer
MIPS Technologies Inc
August 2003

Introduction

Experienced developers know that good tools support is critical for successful implementation, debugging, and maintenance of embedded products. SoC designs have introduced a new set of problems that make good tools support even more critical. Some problems can be attributed to the inherent decrease in accessibility and visibility for components used in SoC designs; others can be attributed to the increasing complexity of hardware and software. In particular, the increasing use of concurrency in SoC designs will introduce problems that the current generation of tools are not well suited to help debug. None of these concurrency techniques are new but in most cases their use in embedded designs has traditionally been limited compared to usage in desktop and enterprise (supercomputing) products.

Developers must be aware of these issues as early in the design process as possible in order to ensure they will have the tools necessary to deal with these challenges later in the product development cycle. SoC-based designs are heavily dependent on on-chip debug support. Most processor core vendors do not develop tools themselves. Nevertheless, they must still provide on-chip debug support required to facilitate the development of such debugging tools.

Overview

Concurrency is used in SoC designs to increase the performance and scalability of various multimedia, networking, and communications products. Multitasking and multithreading are techniques used to simulate concurrency thru cpu time-sharing. Time-sharing is controlled by a scheduler which manages context-switching between multiple tasks or threads. The terms process, task, and thread are sometimes used interchangeably but there are subtle differences between the three. A task is a sequence of instructions, which can be suspended or resumed. Multiple tasks may or may not share the same address space (e.g. text, data, and global regions are accessible to other tasks). The state of a task is represented by its context which is a snapshot of the processor's state. This snapshot records values for the program counter, stack related registers, various general purpose registers, and some core specific registers related to privilege modes and exception handling. Schedulers use context records to suspend and resume tasks. A thread is an instance of a task. Thread groups are multiple instances of the same task. Threads in the same group all share the same program space and some context state. The advantage of using threads is that schedulers can exploit the shared context within a thread group to perform context switching faster for that group. That is, the scheduler does not have to save as many registers when switching between context for a thread group. Some RTOS vendors only support the task model, and some support the thread model.

A process is a task running on an operating system that uses an MMU to prevent other tasks from accessing its memory regions. Threads running on such operating systems are called lightweight processes and they share virtual address spaces. Lightweight processes relax memory protection constraints and behave more like tasks. For example, schedulers can take advantage of the shared program space to speed up inter-process-communication by passing pointers to messages instead of copying the entire message. Also, the shared program space uses memory more efficiently and improves the cache hit rate. The process model is used by “full featured” OSes like Linux, which has become a popular choice for embedded use. Some platforms like Java provide their own application level thread schedulers but these simply map their application threads into an OS level task or thread.

Concurrency techniques used in hardware design increase performance by using multiple instruction pipelines, reducing idle cycles for shared resources (e.g. pipelines, buses), and by distributing the processing load across multiple cores. In general, it is easier to gain processing power by adding more cores than to increase the clock speed. Multi-core SoCs consist of some combination of general purpose cpus, DSPs, and application specific cores. Some cores provide multithreading support to increase cpu utilization and to boost context-switch performance.

Core vendors are also experimenting with two types of multithreading microprocessor architectures. The first provides extra logic to present OS with an appearance of multiple virtual processor units instead of one physical processor. The idea is to allow idle units to be used by a second thread while the current on is running. The second type uses internal scheduling algorithms to reduce idle pipeline cycles by issuing instructions from a different thread when the current one stalls. The problem here is what criteria should be use to choose the new thread to switch to? To avoid QoS problems a priority based scheme would be required, and it would have to be controllable by the OS scheduler.

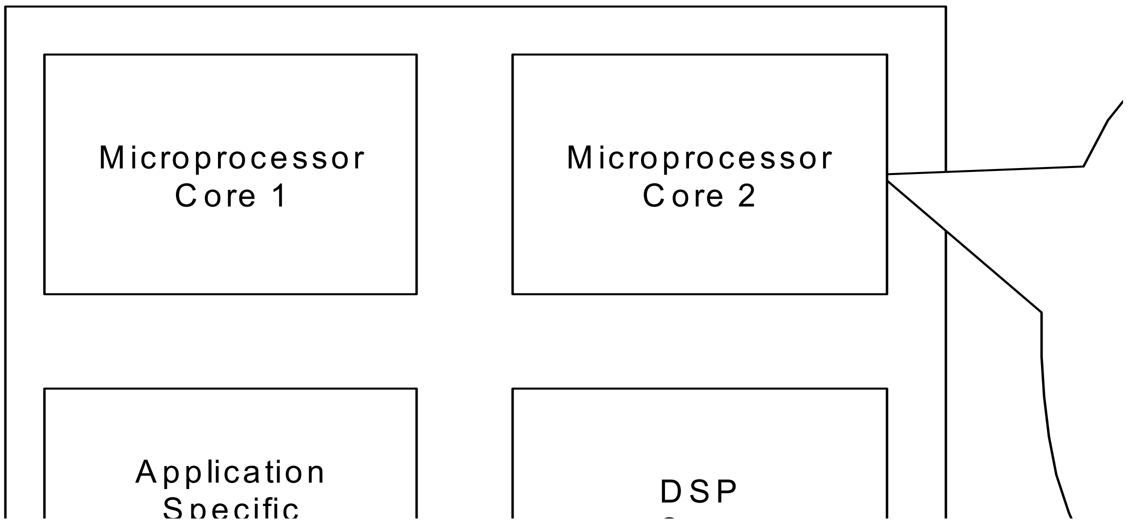


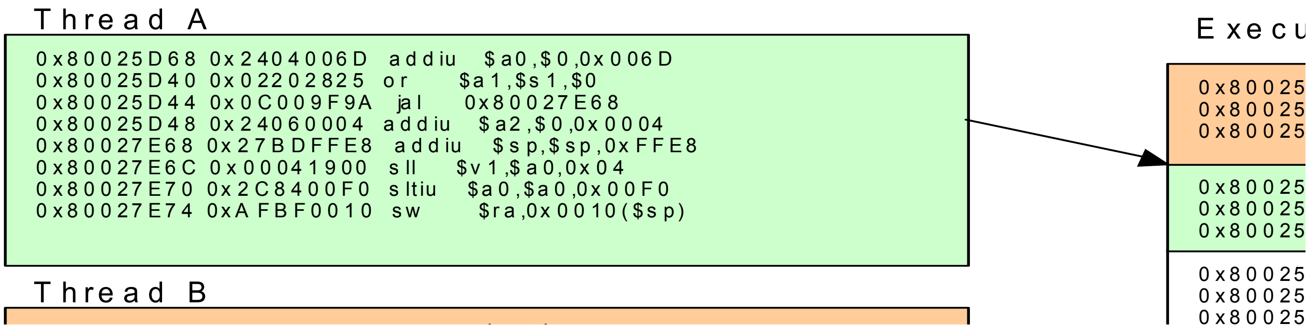
Figure 1:A typical SoC exploiting concurrency at multiple levels in the design.

The objective in any debugging scenario is to figure out what is going on. To accomplish this in a multitasking, multithreading, multi-core system the debuggers must evolve to be task-aware and thread-aware in order to help the developers isolate which threads of execution are causing problems. They should also provide concurrent execution control capabilities and more informative trace information. All of these features would require on-chip support to implement.

Task and Thread Awareness

Some debuggers are intelligent enough to access OS kernel data structures in memory, or utilize APIs provided by the OS to retrieve task and thread state information. This approach is limited in that it is OS specific and it can only access state information when execution is suspended. Task and thread awareness is required in order to support thread-specific and task-specific breakpoints. Most debuggers with thread support (e.g.GDB) use an all-or-none approach. That is, when a thread-specific breakpoint triggers it suspends execution for the entire thread group. This avoids the problem of shared data changing while the selected thread is suspended. When execution resumes it restarts all the threads in the group. Single-stepping the selected thread in lockstep with other threads in the group requires OS support. Halting the core for a thread-specific breakpoint is difficult because a thread group shares the same set of instructions. An instruction breakpoint will cause the core to halt execution for any thread in that group which executes the same instruction. To only halt for a particular thread the debugger would have to restart the threads for which the breakpoint is disabled. The core could provide more sophisticated breakpoints which only trigger on that instruction address for a specified task or thread. For example, when the OS scheduler performed a context-switch it could modify a register identifying which task or thread is now active. Some debuggers implement process-specific breakpoints by looking at the active virtual addresses. But this does not work for tasks and threads which can share address spaces.

It would also be beneficial to incorporate more task- and thread-awareness into instruction trace logs (see Figure 2). Trace logs only show which instructions were executed at some point in time. There is no task id or thread id associated with the trace frames. Again if the OS scheduler notified the core which task or thread was now active for a context-switch then the trace block could potentially access that information and record it in the trace log. Currently the only alternative is to use instrumentation techniques which are intrusive and do not provide assembly level trace information.



```
0x80025D68 0x2404006D  addiu   $a0,$0,0x006D
0x80025D40 0x02202825  or      $a1,$s1,$0
0x80025D44 0x0C009F9A  jal     0x80027E68
0x80025D48 0x24060004  addiu   $a2,$0,0x0004
0x80027E68 0x27BDFFE8  addiu   $sp,$sp,0xFFE8
0x80027E6C 0x00041900  sll     $v1,$a0,0x04
0x80027E70 0x2C8400F0  sltiu   $a0,$a0,0x00F0
0x80027E74 0xAFBF0010  sw      $ra,0x0010($sp)
```

```
0x80025
0x80025
0x80025
0x80025
0x80025
0x80025
0x80025
0x80025
0x80025
```

Figure 2: Three threads executing the same snippet of code. Without enhanced trace support incorporating the context switch information it would be very difficult to figure out which thread was executing at each frame in the log.

Debugging Concurrent Programming Problems

Most concurrent programming problems can be attributed to a lack of proper synchronization in the access of shared resources (e.g. cpu and bus cycles, memory, and various devices). The problems are manifested in the form of data corruptions, race conditions, deadlocks, stalls, and starvation. The occurrence of these problems is often unpredictable and hard to reproduce. For example, executing the same code with the same input can produce different results or produce the same result but with different completion times. They defy traditional source debugging techniques like “stepping thru code” which is too intrusive and only focuses on a single thread of execution.

As a result, the most popular solutions for attacking these types of problems are “using printfs” or event logging based techniques which instrument the software. Tool vendors claim their instrumentation techniques typically add about 2-5% overhead. In most cases these techniques are sufficient and the overhead is acceptable. But in highly optimized architectures code instrumentation techniques introduce unacceptable changes in pipeline and cache behavior as well as adding more non-deterministic side effects. Unfortunately, this case will become more common as SoC designs continue to scale up in processing speed and degree of concurrency. The alternative is to use non-intrusive real-time trace debugging techniques. Again it would be very useful to add context-switch events into the trace to determine which task or thread was active at the time.

Since the goal of using concurrency techniques is to increase resource utilization (i.e. reduce idle cycles), it would be desireable to have various counters and statistics (e.g. cache hits/misses, cpu/pipeline stalls) provided by the core to measure the effectiveness of these techniques.

Debugging Multiple Cores
Most developers are already aware of multi-core accessibility issues and the use of JTAG scan chains to provide probes access to all the cores. Likewise, tools vendors are modifying their IDEs to support concurrent debugging of multiple heterogeneous cores so engineers are not forced to use multiple debuggers. In order to debug synchronization problems involving multiple cores we need concurrent execution control. For instance, synchronized breakpoints that halt one more selected cores can be implemented if the cores provide external pins which can be used to signal other cores to halt. When a breakpoint occurs on one core it asserts the pin to halt the other cores. Currently debuggers try to emulate synchronized breakpoints by issuing halt commands thru the JTAG interface. But this has significant latency problems.

Multiple cores also present new challenges for trace support. Ideally you would like to have an on-chip trace buffer for each core. Unfortunately, this may be too expensive and if the cores are heterogeneous and/or running at different clock frequencies then correlating the trace dumps would be difficult. Having correlated trace dumps would be very useful in debugging an SMP-like architecture where one core can be stalled by another waiting for a shared resource. But correlation requires a common timebase or point-of-reference. A common timebase could be implemented by having one core periodically send signals to other cores to reset (sync) an internal counter. Alternatively, a single on-chip trace buffer could be shared by the cores or a single off-chip trace buffer within the probe could be used (see Figure 3). In the latter case the probe could provide the timebase for time-stamping.

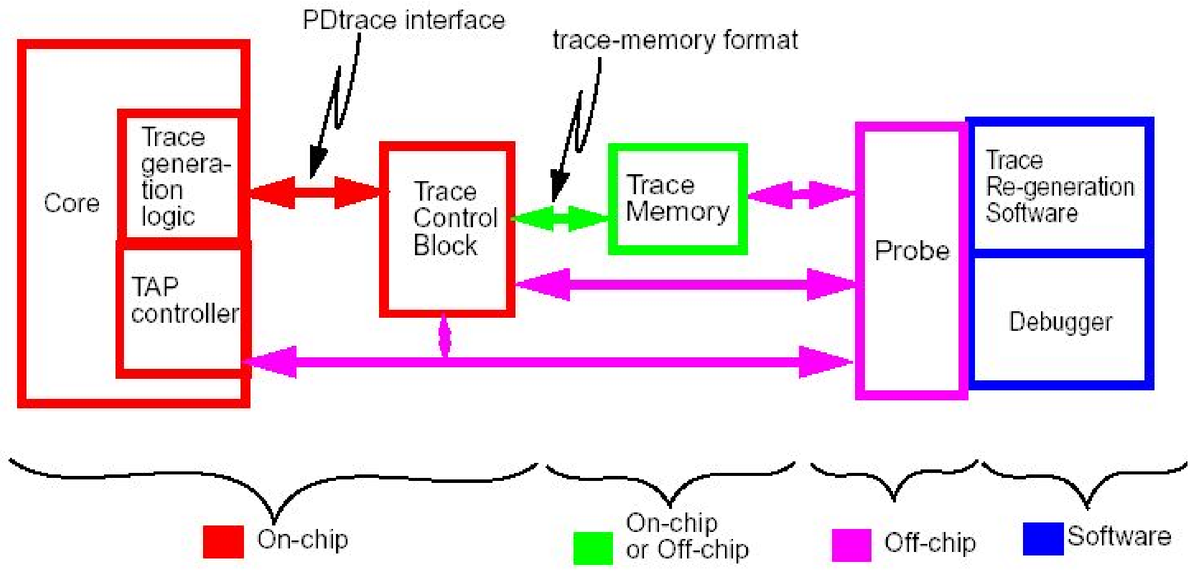


Figure 3: Possible configurations for trace capture.

Conclusion

We have seen how adding on-chip support can facilitate the implementation of more advanced concurrent debugging features. So why are so many SoCs being produced today with little or no on-chip support? The answer is cost and time-to-market pressure. On-chip debug support does not come for free. It increases the cost and complexity of the chip. To address these issues some vendors are now offering on-chip debug support in the form of IP blocks. Eventually manufacturing cost will drop and it will become more practical to produce chips with internal debug support. Currently, developers must get by using other debugging techniques like instrumentation. But as SoC designs scale up in complexity and performance, these techniques will eventually become too intrusive. The result will be an increase in the migration of debug support from software into hardware, and the hardware will become more task- and thread-aware as context switching support is provided in hardware for the OS scheduler.