Comprehensive tool chains needed to debug multiple processor cores


Comprehensive tool chains needed to debug multiple processor cores
The increasing reliance on complex multicore designs is driving the need for comprehensive debugging tools that can answer a variety of challenges. With multiple cores and support structures often embedded within a shared SoC architecture, conventional piecemeal debugging methods are no longer viable, because most of the interprocessor traces are unavailable for direct external monitoring.

So, instead of having a plethora of convenient debug access points at the external pinouts for each processor, today's complex multicore SoC designs effectively bury all these interprocessor links within the chip-level implementation. And, by bringing large amounts of support logic such as memories, mixed-signal, FPGAs, flash and I/O structures into the chip itself, highly integrated SoCs further complicate the challenges of debugging the final product. When the design involves a heterogeneous mix of different vendors' processors, such as ARM cores and DSP cores, the debug environment also must cope with inherent differences, like bus structures and data flow characteristics.

In such environments, more complex interactions are common between the microcontroller and processor and the DSP. That includes mixed feeds to displays, interaction of pointing devices with signal and data flow, more use of the DSPs as coprocessors for the controller for advanced application-specific computations and transformations, etc. This has led to the need for more comprehensive interprocessor debugging tools.

As the higher level debug process progresses and the overall design becomes more stable, it often becomes necessary to focus on detailed analysis and debug of specific subsegments to optimize key functions or tune for maximum performance, or both. In these instances, the debug tools must be able to manage all aspects of the design while homing in on a specific portion. For example, in a cell phone design the DSP section might already be considered solid, with the majority of the debug process shifting to focus on the microcontroller section. However, because the DSP is a critical aspect of the overall data flow, the debugger still needs to be able to interactively reset, run and control the DSP section in conjunction with debug of the microcontroller.

The first requirement for optimizing comprehensive debug is to enable the user to communicate with and control all the cores at the same time via a common interface. The access to each core also must
be completely independent of the surrounding environment, meaning that the state of another core or section of the design should not limit or hamper the designer's ability to control the runtime actions of any targeted core.

The need for independent access can be especially important in design situations that involve a master/slave relationship between processors. For instance, in the cell phone, the ARM CPU cores and DSP core communicate via cross-interrupts and shared memory (or mailboxes). However, the DSP has its own private memory and peripherals for high-speed data flow, such as analog-to-digital or digital-to-analog converters, or direct access to network interfaces.

The CPU cores connect to an Advanced Microcontroller Bus Architecture (or Amba, an ARM standard) High-Performance Bus in one or more layers to access peripherals. Typically, the DSP is loaded by the ARM processor and held in reset, thereby also requiring it to use the processor to access peripherals not directly in the DSP's data flow.

Debugging such a design can be especially tricky when you consider the complexity of the data flow and interprocessor control functions. For instance, during debug it can be quite advantageous to be able to run the DSP independently and analyze its operation, even if the main processor core is not yet up and running to handle the control functions or to provide access to the other peripherals.

Using the cell phone example, a designer would typically have to go through the following specific steps:

- Connect the emulator/debugger(s) to the JTAG port(s).
- Configure the memory interface (pointers to RAM and Flash, setting wait states, etc.).
- Load application software.
- Configure DSPs via the ARM7TDMI core (setting shared memory with correct DSP information).
- Take the DSP out of reset via the ARMTDMI core.
- Stepping through the specific debug operations.
- Capturing the state of all processors upon failure states or specified conditions.

These steps are further complicated by the fact that at any point in time various portions of the design must be simulated in order to debug other parts. For instance, instead of running with a real antenna, the above design would likely be debugged using an antenna simulator that would also have to be set up and controlled.

In a traditional debug environment, where separate tools are used for controlling each processor, the above series of steps can become complex and time consuming. A designer focused solely on the DSP side would also have to learn the debug environment for the microprocessor cores to set up and start the DSP. In practice this would call for bouncing back and forth between two distinct debuggers at various steps in the process, which slows down the overall process as well as making it difficult to automate. Worse yet, with each crash, the developer would have to manually walk through all the setup steps to restart the process and get back to the point of failure.
In contrast, with a single comprehensive debug environment, developers from both the microprocessor core and DSP sides can share common scripts for setting up or emulating specific parts of the design or both, and can use similar processes to automate major chunks of the debug process.

In addition to reducing the learning curve for developers focused on different parts of the design, the use of common debug tools, scripts and processes also makes for a much smoother collaboration effort when they have to call on each other for help and support. Instead of wasting time trying to figure out if the developer "from the other side" has correctly set up "my" processor, if all developers have used the same tools and scripts they can jump directly to analyzing the results within a debug environment that is familiar to all of them.

In addition to independent control over processors, it is vital that the comprehensive debug environment also support synchronous operation of some or all of the processing cores. It is critical...
to be able to precisely control synchronized actions such as Start, Step or Stop across multiple processors to capture the state of each core in relation to other cores. In order to analyze real-time processing it is imperative to be able to start the multiple processors at same time or halt them simultaneously or both.

The debug environments' Halt functions also have to provide the flexibility for either implementing a user-initiated Halt (to inspect the process at a predetermined point) or a break-triggered Halt (to capture the system state upon occurrence of a failure). Furthermore, the debug system also has to provide the flexibility for independently continuing operations throughout the rest of the system even if one of the cores fails.

Because the interfaces between the microprocessor cores and DSPs are becoming more intermingled with the real-time operating system and platform-level software, it is also critical for the product design and debug environment to provide an awareness of the operating system and to allow for control of the hardware through the higher level operating system. Basically, from a hardware design and debug standpoint, the tradeoffs involve gaining functionality by giving up a significant amount of flow control to the software. This results in a state machine where the debugger is in control of the nodes but not the transitions. Therefore, when considering the use of higher level real-time operating system functions or platform operating system control or both, designers must be aware of the increased difficulties with debug and ensure that their debug tools can effectively retain control over both the software and hardware.

The overriding goal of a comprehensive debug environment is to put the user in complete control of all parts of the SoC design, with full flexibility to start, stop and analyze any process or subprocess either synchronously or independently. A critical factor in achieving this is to bring together all the debug processes under control of a single user interface, but without compromising on the specific capabilities for debugging each processor core.

To be effective, the user interface must allow the designer to flexibly model the target architecture within the system, such as defining all relevant chip-level and board-level attributes, specifying register formats, memory layouts, peripherals, ASICs or other devices. Data flow parameters also must be defined, including communication buses, shared memory structures, memory-mapped peripherals, mailboxes, etc. Entering the nominal values or limits, or both, for complex bit fields and data buffers also provides the underlying context for triggering failure states or out-of-parameter conditions that may occur during debug.

The use of comprehensive and uniform system modeling also allows for detailed examination and analysis of all multiprocessor interactions, communication and data flow parameters throughout the design. These system-level debug environments lay an effective foundation for implementing trace capture circuitry in future SoC designs, which can potentially allow precise correlation of detailed cycle-by-cycle information and register visibility if required.
See related chart