ARM intrusive debugging for post-silicon SoC validation


Debugging large RTL projects has become increasingly complex. With different types of applications evolving, there is a need for different ways to enable debug hooks in every application. Although the debug architecture of the SOC is at par with its complex design to make non-intrusive debugs easy, but there are multiple scenarios which require intrusive debugging. This article compiles a few scenarios which depict the use of intrusive debugging for validation of an SOC.

Low-power handshake

As the demand of power saving has increased in applications, there are multiple power domains in an SOC which are power-gated or ungated based on requirements of the use case. During low power modes, one or multiple power domains of the SOC are powered down while others are still running. In such cases, occurrence of any issue and its debug becomes more complex. Debugging during power saving modes is not always possible since sometimes SOC debug logic itself is powered down. To capture the debug information from a running application during such power-gated modes, the debugger should be continuously communicating with the SOC and should capture all the relevant information before entry and after exit from such modes.

Whenever there are application crashes during power saving modes, data capture becomes more complex if there is no mechanism available to keep the debugger connection intact. Therefore, handshaking is important before entering the power gating mode to allow the debugger to restore data and enable debug as soon as the DUT leaves power-gating mode.
The SOC should initiate the handshake before entering low power mode. After the SOC enters low power mode, the debugger tries to establish a connection with it and waits for the low power mode exit. When the SOC wakes up from low power mode, it waits until the handshake takes place with the debugger and debug continues. Therefore, all the data remains intact with the debugger making debug easier. Figure 2 shows the sequence of events describing SOC-Debugger handshake.

![Flow diagram illustrating debugger handshake](image)

**Basic example**

There is a missing configuration in the boot code which causes a reset after low-power exit in a specific scenario. When in non-debug mode, it will be very difficult to find where exactly the problem exists (i.e., in low-power exit, boot code, or application code). Thus, if low-power handshake is enabled, some delay will be added in low-power exit, but the missing configuration can be found as debug of the core can be enabled immediately after low-power exit, and boot code can be stepped through to locate the issue.
### Semi hosting

In validation there is the requirement of a constant exchange of information between a host machine and DUT. This information can be test case number to be run, test case variables required by the test case at runtime, results of test cases, print logs, etc. Semi hosting can be used by the validation engineer to develop an environment in the host machine which can enable them to run the entire test suite with the help of the debugger. Many debuggers, like Lauterbach, ARM RVDS, and J-Link, provide their library functions which can be used from the host machine seamlessly. Libraries can also be invoked from the firmware running on the DUT based on requirements.

![Figure 3: Semi-hosting block diagram](image)

### Basic example

A basic example of semi hosting used in validation is the reset-test-resume scenario, where the DUT wants to store its parameters in host machine before a reset. In that case, whenever a specific memory location is written by DUT with a specific value, the host machine uses debugger APIs to store the context and waits for a reset while DUT code provides a reset. After the reset, the host writes the parameters back to the memory and the DUT code execution is resumed. Now the DUT can access the previously saved parameters and compare them with the new ones.

### Cross Triggering

Cross triggering is yet another example of intrusive debugging. In a multiprocessor environment, cross triggering provides a common synchronized platform to the user for debug. Various input and output signals of different processors are mapped to a single matrix with multiple channels. Each input is broadcast on the channel and the enabled output on that channel is routed to the processor. Apart from providing support to semi hosting and other debug mode features, various non-intrusive debugging techniques as traces, watchpoints, and interrupts are synchronized by enabling cross triggering. Cross triggering interface provides a synchronized and binding platform for the debugging of complex multiprocessor applications. An example of a cross-triggering matrix is as shown in the below block diagram.
The CTI Input 3 of processor A is enabled and broadcast at channel 0 of the CTI Matrix.

Since the CTI Output5 of processor B is enabled, the trigger from channel 0 of the matrix is routed to Processor B, CTI Output5.

**Basic example**

If we want to observe a core-traffic profile through HTM traces, it is difficult to enable core execution at the exact time when HTM traces are enabled/disabled. In this case, we can use CTI input of HTMTrigger to trigger the CTI output which is an interrupt to the desired core. On receiving the interrupt, the core can begin execution of the profile to be taken.
DAP access in case of caches and coherency testing

Cores can have individual or unified/shared caches. In the case of symmetrical multiprocessoring, snoop channels also may be enabled. Coherent systems can also have non-core masters on the snoop bus of the SOC. In the post-silicon phase, such scenarios are challenging to validate as there is no visibility of caches or dirty RAMs. DAP access of a debugger can be used in this case.

Basic example

Let us consider the following example to understand the usage in a shared and coherent system with two cores (A & B), having independent L1 caches with read/write allocate and write back policy. Core A writes to a virtual address after which coherent Core B accesses the same virtual address, assuming a similar virtualization scheme used in both cores. Core B correctly reads the new value as updated by Core A. But how do we know that Core B has read the data via the snoop channel only, and that the data was not written back to the physical memory location?

In the validation test suite, a read can be done directly via DAP to physical memory to see when the cache page is actually written back. All accesses before that point will definitely be via the snoop channel. A snapshot of one such example is below, in which accesses with EAHB class are through DAP, while the other two are from the cores.

Figure 6  DAP accesses in cache coherency example

Also see:
• Silicon debug challenges and guidelines
• Using emulation to debug software and hardware at the same time