Multi Processor SoC Debug with Synopsys Hardware Software Debug Tool

A. Mark Jesensky
Andy Sha

Analog Devices
Norwood, MA, USA

www.analog.com

ABSTRACT

The purpose of this paper is to review the test code debug method of a BIG.little SoC incorporating two processors and various dedicated and shared peripherals using traditional methods versus the Verdi Hardware Software Debug Tool (HW/SW Debug Tool). The traditional method of using I/O messages and aligning the timestamps for general areas to look in the waveform was cumbersome and time consuming. Errors in the system behavior needed to be reviewed as timestamps, and were then referenced back to a timestamped CPU log-file or through looking at the waveform fetch data, and then referenced again at a compile/instruction loader file to see what C command was causing issues.

Using the Verdi HW/SW Debug Tool we have cut hours of debug time into minutes. The tool has allowed us to align Verdi waveforms directly to assembly and C code for instant reference. The GUI contains core register and C variable values for easy reference, and will single step though the code to show what is changing in the code and have it aligned with the Verdi waveform. It also allows stepping through one core while keeping the code aligned in time with the other core of our system so that race conditions are easy to see.

This paper will outline the benefits of the tool, the implementation of the dual core debug file generation, instantiation of the recorders needed, and some of the issues that we encountered which shows the benefit of the tool.

Table of Contents

Table of Figures

1. Introduction

The Verdi Hardware Software Debug Tool (HW/SW Debug Tool) is a debugging tool using an Eclipse based GUI environment. It integrates information from the software compiler and hardware simulator and provides a user interface similar to common software debug interfaces which makes it intuitive to use and greatly enhances debug efficiency with any SoC design. This efficiency increases exponentially in multicore systems as it tracks and aligns register, variable, and execution information for all cores. As this tool was implemented on a multi-core SoC at ADI, we have seen decreases in debug time of all issues where this is used, especially in multicore test scenarios.

This paper will overview the traditional methods of debug, implementation of the HW/SW Debug Tool, and basic usage of the tool features. Some brief overviews of the types of issues debugged on this project are included in Section 5 of this paper.

2. Legacy Debug Challenges

Before using HW/SW Debug Tool, verification engineers had to manually synchronize waveforms to the C code, which is a very difficult and error prone process. The steps normally taken are:

1. Locate the point where test case failed from runtime message log.
2. Use timestamps to locate failure in the waveform and find the corresponding instruction address in the simulation waves.
3. Open each core's dis-assembled C-file and find the corresponding assembly code and C code matching the instructions address values.
4. The next step is to analyze the code and trace how the program got to this point. With function calls, interrupts, and core interaction this process is extremely slow, and the code that caused the error could have executed much earlier in the simulation and it can take hours to days to ultimately identify the root cause.

While this flow is effective at solving the issues encountered it is difficult to standardize, because the flow uses the instruction and decode buses to debug. The engineer is required to have detailed knowledge about the core structure, which means training and experience. This includes knowledge of mapping C code to the function of assembly codes and the formatting of CPU core log files.

In a complex debug scenerio, we have to manually go through detailed information in both the hardware and software which results in a very inefficient and increasingly difficult work flow.

The HW/SW Debug Tool standardizes this work flow into a simplified and traceable GUI environment. Verdi's waveform view and software view can be synchronized in time. Clicking in the waveform will align the program code for all cores, the associated assembly/opcodes, and all internal core register values. It also provides the call stacks for functions in each core individually for simplified execution status and traceability through a test's C code. While the

initial setup should be done by an engineer with detailed knowledge of the system and core, the tool allows a wide range of hardware and software users to be effective in debugging.

3. Implementation

The implementation of multicore HW/SW Debug Tool requires the generation of a number of debug files from the simulation environment. It requires a single .fsdb waveform file, one compiled .elf/.out file per instruction memory bank with embedded debug information, and one CPU log file per core. Here are the steps to be followed as part of the flow.

A) All debug information for HW/SW Debug is saved in the waveform file using the standard FSDB dumping routines. The debug information is contained within its own hierarchy so only this portion of the design requires waveformdumping. The remainder of the design can be fully optimized by the simulator.

Code example:

```
$fsdbDumpfile("name.fsdb");
$fsdbDumpvars(0, "verdiHwswDebugTop", "+all","+parameter");
```

B) The HW/SW Debug Tool reads a formatted file of core information including instruction pipeline, register loads, and other events within the core. If using ARM® cores, this information is generated by instantiating verilog code included in the ARM® development kit. If using custom cores debug files are created by snooping the internal signals for the PC, instruction pipeline, registers, and other signals of note and creating a timestamp marked file. For support in creating a custom recorder for debug information, talk to your Synopsys rep for some help in supporting this.

C) Linker files for each memory segment need to be created for each instruction memory. Using the standard gcc/armcc compile flows, adding the -g option to retain the debug information of the source code as required by the HW/SW Debug Tool. If using IAR or similar software on a PC based platform to create loader files, the --debug option needs to be used during the compile. Also the path names need to be matched with the gcc format of source files for retaining assembly to C code references. Combinations of elf file generation from each method is supported, i.e. Core0 uses an IAR pre-created loader file and Core1 uses a gcc simulation runtime compile of code.

D) Add in one SystemVerilog recorder instance per core from the Synopsys supplied recorder libraries for ARM® processor cores or work with Synopsys to create custom recorders for proprietary cores. Specifics for this are unique CPU numbers if they have dedicated memories, unique CPUIDs for the same CPU number if they share a memory, or any combination that your project requires. One .elf file per instruction memory block is required, no matter how many cores share it.

Recorder code example, dual core with separate instruction memories:

```
`include "verdiRecorderCore0.svp"
`include "verdiRecorderCore1.svp"

// Top-level recorder module (must have this name)
module verdiHwswDebugTop;

  Core0_cluster Core0_cluster();
  Core1_cluster Core1_cluster();

    // FSDB dumper: change FSDB file name if needed.
    initial #0 $fsdbDumpvars(0,"verdiHwswDebugTop","+all",
"+parameter");
    // Common init - Leave this at close the end of the module.
    `VERDI_HWSW_INIT_TOP

endmodule // verdiHwswDebugTop

module Core0_cluster();
    verdiRecorderCore0 #(.clusterId(0), .cpuId(0),
.traceFileName("trace_dbg_drv_Core0.log")) cpu0(.cpuClock(1'b0));
endmodule


module Core1_cluster();
    verdiRecorderCore1 #(.clusterId(1), .cpuId(0),
.traceFileName("trace_dbg_drv_Core1.log")) cpu0(.cpuClock(1'b0));
endmodule
```

E) Compile items needed:
   a. Recorder "customHWSWDebugTop.sv"
   b. FSDB dump switch if needed

F) Runtime flags needed:
   a. If using ARM®:
      For Cortex®-M series: `+TarmacR +TarmacI +TarmacB +TarmacM +TarmacE`
      For Cortex®-A15: `+TarmacALL`
      For Cortex®-A7: `+tarmac_on`
   b. Elf file inclusion:
      `+Verdi_hwsw_exe_cluster0=Core0.elf`
      `+Verdi_hwsw_exe_cluster1=Core1.elf`

Detailed usage and features of the tool are outlined in previous papers so the following is a quick overview. Launch Verdi with the appropriate –f file for system fabric source code debug with annotation. Load the .fsdb in nWave or Verdi as the normal debug flow dictates:

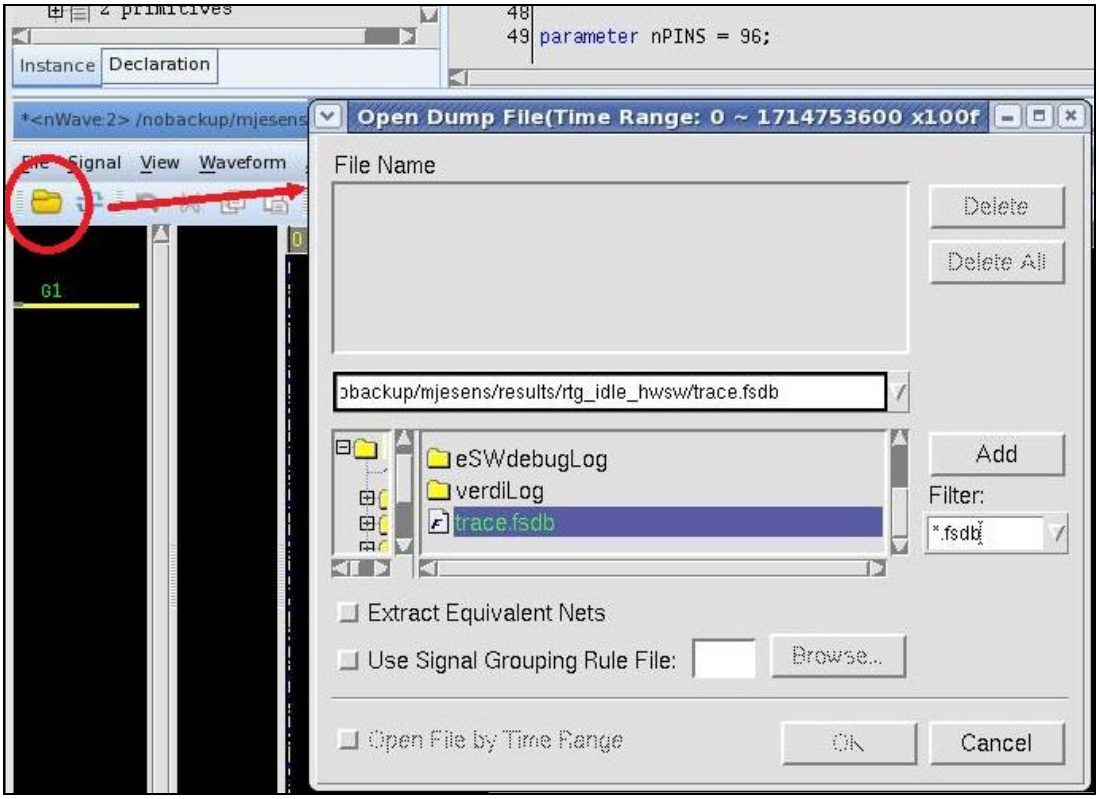

Figure 1-Loading .fsdb in nWave

Once .fsdb is loaded, choose the Tools pull-down and "Invoke HW SW Debug...." from the menu.

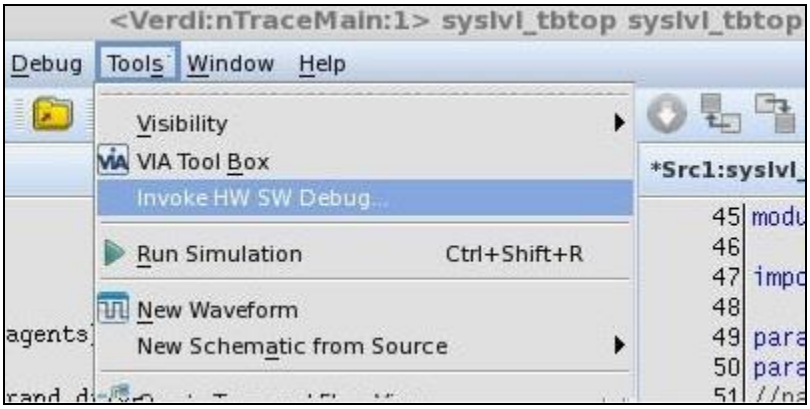

Figure 2-Invoking HW/SW Debug Tool from pull-down menu

The startup screen shows the target cores, initial breakpoint, and file information. The message window will show any issues that are involving the .elf files or recorder information that has been dumped. The active core pull down allows you to set the initial scope of the HW/SW Debug Tool when launched, though you can change the core freely during debug.

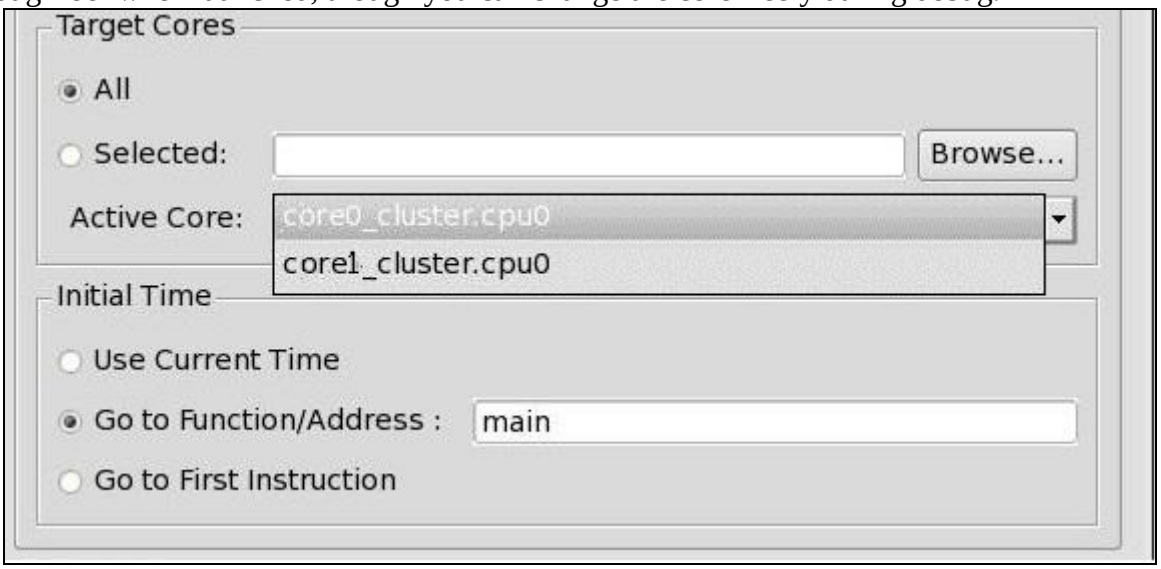

Figure 3-Setting initial scope of HW/SW Debug Tool

4. Tool usage for dual core debug

Once the tool has launched, the Eclipse interface shows each core that was run in the test, including the elf file it is based on and the current execution stack. Figure 4 shows a dual core example where each core started at its own main function, and Core0 is in a function called from main() to send message data to Core1, and Core1 is in an expect message function called from main().

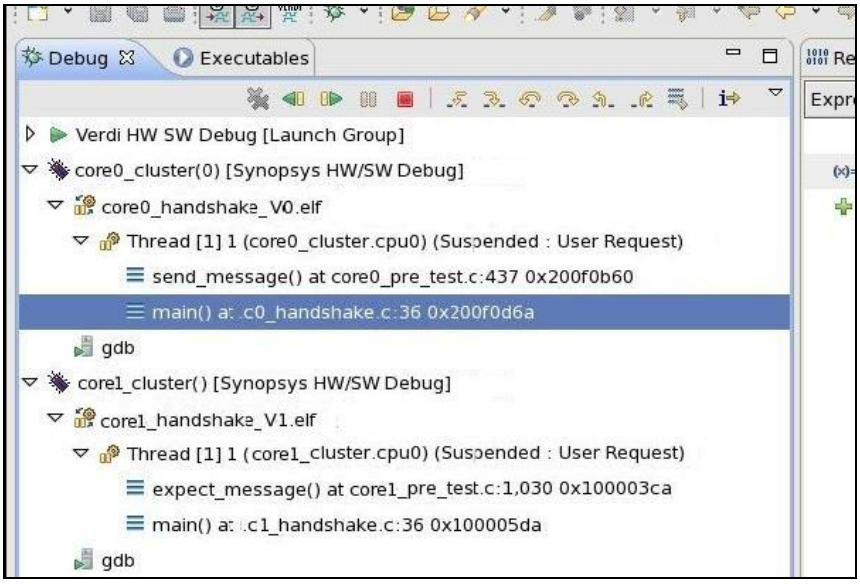

Figure 4-Dual core execution stack

You can click on any stack item for either core to display the C and assembly code at the current execution time. If you were to select the Core0 main() function from the listing in Figure 4 it will highlight the function call to send_message(), while if you selected the send_message() function in the same area, it would show you the execution line within the function.

The stepping functions in the HW/SW Debug Tool shown in Figure 5 will be acted upon the scope that is selected from this debug tab. So stepping when selecting the main() function for Core0 will navigate the code for Core0, but will track the execution and stack of Core1. You can switch scopes at any time to Core1 and continue stepping from that scope. In the example above, you can step into the Core1 expect message function, then switch to Core0 and step into the send message function to verify data is sent out, then switch back to Core1 to see the data come in. As with single core debug, you can optionally choose to step by C instruction or assembly instruction in the scope selected.

Figure 5-Stepping functions

The scope of the displays in HW/SW Debug Tool are still limited by the scope and function. For example in Figure 6 the variables for the selected function ena_svc are shown in the variable scope, while the variables for ena_int are not. Similarly, variables and user defined expressions are limited by the core scope. Only the items for the selected core and shown even though the stack and execution is tracked.

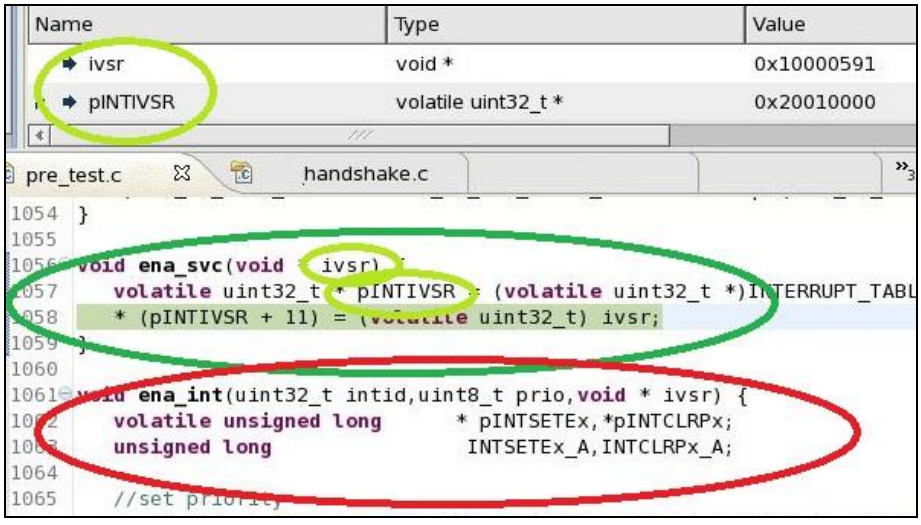

Figure 6-Variable scope

Finally, the Verdi or nWave viewer will track the execution time while stepping through the code, no matter which core is in scope. Since there is only one fsdb file, then it will update on the core selected, even when changing scope and stepping. Clicking on an event in the wave window will also advance the C and assembly code to that point in time in the scope selected as well, the same as with a single core debug effort.

5. Issues found using the tool

There were a number of issues that were found with this GUI that would have been much harder to debug with the traditional methods. There are three issues that I will describe here which cover a testbench issue, an RTL design issue, and a compiler issue. The first is a data addressing alias register issue, where one processor accessed an address that was aliased to the memory space of the other processor's instruction memory. The second is an execution flow of the processor handshaking through a shared mailbox memory space that was out of sync. The third was a runtime issue of code that was due to different core compiler commands.

In the first issue, the problem manifested itself as Core0 executing an output to some flags on the processor to track execution flow. At some point in the flow, the execution became corrupted and the simulation execution had become invalid. Depending on how much additional code was added for debug, the test would pass correctly, pass prematurely, or fail as before. Using the HW/SW Debug Tool, the test was stepped through Core0 and the PC was monitored to see where the failure occurred. The debug window was flagged with an unexpected memory location change, which detailed the time at which an address in the instruction memory was changing. Switching scope to Core1 it was seen that the code executing there was a data memory access, which had been incorrectly overlaid onto the Core0 instruction memory space, causing the corruption.

In the second issue a handshaking protocol between Core0 and Core1 was locking up. The debug messages originally printed in the log file showed that traffic in one direction worked properly while a transaction in the other direction did not. Waveforms showed a race condition between the mailbox valid flag that was causing it to incorrectly display a ready signal. From this, both cores went into their respective expect information stage waiting for the other to write. Starting from the locked up state, each core could be seen in the expect information function. Using the step back and uncall functions, the code went back to the transaction that caused them to both get into the state and the ready flags could be seen in the waveform and RTL source for debug.

The third issue started as a failure of a legacy test on Core0, but passed on Core1. While using a peripheral, status bits were checked to control program flow. While a bit was checked and PC advanced on a bit being set, Core0 hung waiting for the bit to be set. A first instinct was that the status bits were not correctly implemented in the Core0 subsystem, however stepping through the code with the tool, the bit was read correctly into Core0. Tracing through the compare function, the data was seen to truncate during compare, and looking at the assembly code for each core a difference in code was seen at the compare. It was found that due to compiler differences between the cores. An optimization switch was set which pre truncated data to 8 bit compares that the core supported instead of multiple shift/compares to do a 16 bit compare.

6. Conclusions

Our conclusion is that this tool will help us reduce debug times of multicore issues by approximately 80%. What used to take a half day of tracing through 4 files plus waveforms can be accomplished in a few minutes with two GUI windows. The tool still has its flaws including passing variable data between debugger and waveform, multicore variable/expression displays being limited by core scope, and no display/uvm_info information while stepping to mimic the simulation run without a custom recorder. But the net benefit of the tool makes it a good fit for SoC development.

This tool is already in use by various digital design and DV groups within ADI, and it is branching out to other groups including those working with mixed signal designs with embedded processors and algorithm and benchmark developers for SoC products. The ease of use from a hardware viewpoint align with SoC designer needs without requiring a translation of C code though a black box core. The stepping functions and display windows keeps the software developers in their comfort zone so that issues in their domain can be resolved quickly.

7. References

Synopsys datasheet and video for HW/SW Debug Tool:

http://www.synopsys.com/Tools/Verification/debug/Documents/verdi3-hw-sw-debug-ds.pdf

http://www.synopsys.com/Tools/Verification/debug/Pages/verdi-hw-sw-debug-video.aspx

"Efficient Embedded Software Debug" by Andy Sha, SNUG 2014 China:

http://www.synopsys.com/news/pubs/snug/2014/china/efficient_embedded_software_debug.pdf