



The Case for Hardware/Software Co-Verification

Can't I Do That With a Development Board?

Because development boards are readily available, many FPGA designers make the mistake of relying on them as their primary embedded processor debug and verification environment. Can you get the job done that way? Well, yes you can, but then you can also dig a trench with a teaspoon – if you have enough time.

Large devices allow you to stuff a whole system into the FPGA, but debugging these complex systems with limited visibility – and a one-day turnaround for synthesis plus place and route – can consume weeks of your precious time.

Hardware/software co-verification has been successfully applied to complex ASIC designs for years. Now available to FPGA designers, this technology brings together the debug productivity of both a logic simulator and a software debugger. Co-verification enables you to remove synthesis and place and route from the design iteration loop, while yielding performance gains 1,000 times faster than logic simulation.

Shortening the Design Iteration Loop

Using a development board in the highly iterative design loop adds major overhead to every design iteration. This overhead comes in the form of logic synthesis, followed by place and route. Although necessary to produce a final design, you can remove these time-consuming steps from the iterative design debug loop by targeting simulation as the verification platform.

With simulation as the verification engine, the only overhead between editing the HDL and verification becomes a relatively quick compile of your HDL. The time you can save on your next embedded FPGA is easy to calculate: How many times did you run place and route on your last FPGA design? And how long did place and route consume your PC for each run?

It's true that simulation runs slower than the real-time speed of a development board. Co-verification provides some innovative ways to dramatically increase the rate at which your embedded software simulates. The increase in a typical system is several orders of magnitude.

Improving Hardware and Software Visibility

To debug your FPGA design, you need full and clear visibility. You need to know what is happening in the hardware and what the software is doing. You need to be able to change a register, or force a signal to a different state. Sometimes you need to be able to stop time and take a closer look. The more visibility you have, the more quickly you can see the problem or prove you have resolved the bug.

Hardware Visibility

Probing inside or even on the pins of your FPGA is a challenge. The ChipScope™ Pro analyzer from Xilinx® or SignaTap® analyzer from Altera® help with this, but in a logic simulator (in addition to viewing every signal) you can also change their values. Working from your source HDL, you can step through the code, view variables, or stop time. For detailed, immediate, and hassle-free visibility, it is very hard to beat logic simulation.

Software Visibility

Software visibility in logic simulation is another item to contend with. Running the fully functional processor model allows you to execute software, but knowing what is in R3 of the processor is almost impossible, if you are given only waveforms.

Co-verification provides an enhanced processor model connected to a software debugger. In the software debugger, you can view and change everything from registers to memory, stack, and variables. A software debugger also provides a source code view with symbolic debug. You can step through code at the source or assembly level and breakpoints can be used to halt execution or run powerful macros.

Some debuggers even provide real-time operating system (RTOS) aware debugging allowing you to view the status of tasks, mailboxes, queues, pipes, signals, events, semaphores, and the memory pool.

Much Faster Than Logic Simulation Alone

Running substantial amounts of software on a standard processor model in logic simulation is not practical; the run times are just too long. However, running this software actually turns out to be one of the most effective verification strategies available. The pay-off for running diagnostics, device drivers, board support package (BSP) code, booting the RTOS, and running low-level application code is huge. It is not surprising that verifying hardware – by putting it through its paces the way the software will actually use it – is effective. Similarly, the software is tested against the actual design (including any external board-level components that are included in the simulation) before the board is actually built.

The challenge has always been to run enough software to really boot the system and do something interesting. Co-verification is able to speed up the run time by taking advantage of one simple observation. Most of the simulation time is spent re-validating the same processor to memory path. Although you need to test your memory subsystem and try several dozen corner cases, you don't need to repeat those same tests over again every time you fetch an instruction from memory. Similarly, you need to verify that the processor can push a value on the stack and pop it off again with the correct result, but repeating this test every time a software function is called would be overkill.

Accesses to hardware peripherals always generate bus cycles in the logic simulation, but instruction fetches and stack operations can typically be offloaded for faster execution. By allowing you to specify which bus cycles are run in the logic simulator and which are not, co-verification allows you to make the performance trade-off. Some co-verification tools let you change this specification at any time during your simulation session. With this ability, you can run through reset with full cycle accurate behavior, and then switch off instruction fetches and stack accesses to boot the RTOS.

Accessing memory through the logic simulator requires several hardware clock cycles. Each clock cycle requires significant work in the logic simulator as it drags along the heavy weight of all the other logic in your FPGA. Using a "back door" to directly access the memory contents instead of running the bus cycle in the logic simulator allows accesses to occur many orders of magnitude faster.

The speedup is very significant. For example, the following data is from a typical design configuration with a PowerPC running the Nucleus RTOS on the

Xilinx Virtex-II Pro FPGA. Booting the Nucleus RTOS in logic simulation alone requires 12 hours and 13 minutes. The same task with these techniques employed accomplishes the task in only six seconds, 7,330 times faster.

Using this technique, co-verification can maintain one coherent view of memory contents through a "back door" into on FPGA RAM memory models or any other memory device. So if your DMA controller drops something into memory that the processor later executes, it will still all work together correctly. And if the processor generates a large data packet and instructs hardware to transmit it using DMA, there are no data inconsistencies.

Identifying Processor Bus Bottlenecks

The performance of your FPGA platform can be seriously impacted by the memory structure of the design. What should be located in cache versus BRAM or external memory? Where are the bottlenecks? Do other bus masters starve the processor? Questions like these are important, but getting the answers can be difficult without real data from your hardware/software application.

System profiling tools can gather performance data from the simulation and display it graphically (Figure 1), enabling you to identify:

- Which functions are consuming most of the CPU time
- Unexpected lulls or bursts of activity
- Cache efficiency and memory hot spots
- Code execution and duration at the function level
- Bus utilization and bus master contention

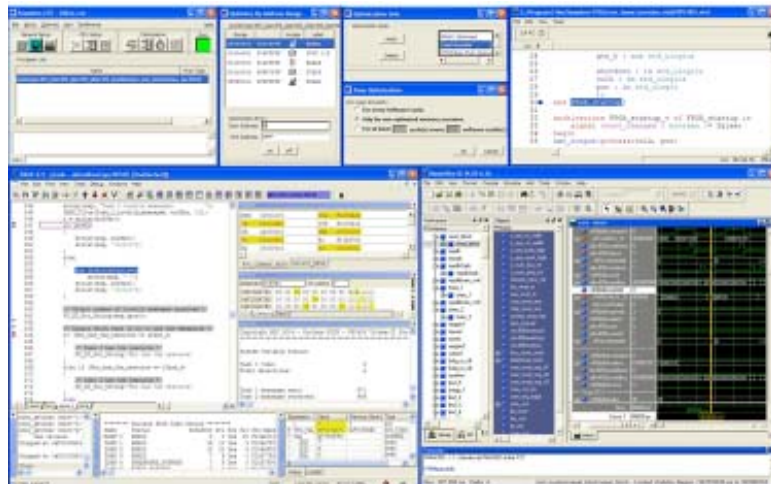


What Will It Take To Co-Verify My Design?

Co-verification is generally easy to use and set up, because you have already entered the data in the FPGA vendor's platform creation tools like Platform Studio from Xilinx. This allows an FPGA co-verification tool to automatically configure itself to co-verify your design. You already know how to use your logic simulator, and co-verification generally leaves the full functionality and user interface unchanged. The software debuggers use many of the same menu icons for operations like step, step over, and run.

If you have two or more processors in your design, you will have additional software debugger windows, one for each processor.

Once the logic simulator and software debugger have been invoked (Figure 2), you are ready to verify your design. In the logic simulator, enter any stimulus commands needed – typically this is reset and clock, plus any design specific stimulus – and then run. In the software debugger, you are ready run or to start stepping through your embedded code. By default, all bus cycles are routed to the hardware simulation.



To increase software execution speed, "optimizations" can be applied to direct the co-verification tool to access memory contents through a "back door" without requiring the logic simulator to run every bus cycle. Optimizations can be applied to all instruction fetches, or to any number of memory address ranges. When accesses use the back door, you can either choose to keep advancing the logic simulation in lock step with the software, or remove that requirement.

Optimizations can be applied from the user interface, or through macros attached to breakpoints in the software debugger or the logic simulator. Furthermore, optimization settings can be changed at any time on the fly during a simulation session. This allows you to quickly run to a certain point in your software, and then enable all bus cycles for detailed cycle-accurate verification.

Conclusion

With large FPGA designs employing embedded processors, it's not possible to complete a design in a few weeks. These designs are very sophisticated and, unfortunately, so are the bugs that you must track down and resolve to produce an effective system on schedule.

Software content in your FPGA can bring lower system costs, higher configurability, and increased functionality. But software doesn't execute alone – it interfaces with hardware, and the hardware/software interface often stretches across disciplines and design teams.

Co-verification bridges the hardware/software gap with a productive software and hardware debug environment that provides the visibility to find bugs and performance bottlenecks efficiently. And once you have fixed them, you can quickly turn the fix and verify it, without having to wait for your PC to rumble through place and route for hours on end.

For more information, visit www.seamlessfpga.com or e-mail seamless_fpga@mentor.com.

by Ross Nelson, Seamless FPGA Product Manager, Mentor Graphics Corporation, ross_nelson@mentor.com

November 8, 2005