Power Up Hardware/Software Verification Productivity

by Matthew Ballance, Mentor Graphics

Today’s complex designs increasingly include at least one, and often more, embedded processors. Given software’s increasing role in the overall design functionality, it has become increasingly important to leverage the embedded processors in verifying hardware/software interactions during system-level verification. Comprehensively verifying low-level hardware/software interactions early in the verification process helps to uncover bugs that otherwise would be uncovered during operating system or application bring-up – potentially in the lab. Characterizing, debugging, and correcting this type of bug is easier, faster, and thus less expensive, early in the verification cycle.

Creating comprehensive hardware/software verification, however, has some very unique challenges. Automation, in the form of constrained random generation or Intelligent Testbench Automation, is often applied to generation of hardware-centric stimulus. However, the test program running on the embedded processor is written in C or assembly without access to any type of automation for stimulus generation. Consequently, the software tests that can be created are low-productivity directed tests. More challenging, still, is coordinating the activity of verification stimulus applied from outside the design via the design’s interfaces together with the verification stimulus applied by the embedded processor from within the design. Some verification teams have created communication mechanisms to enable the hardware-centric testbench to communicate with and control the software running on the embedded processors. These mechanisms are time-consuming to create, often address challenges specific to a single design, and thus are difficult to reuse and apply broadly.

Questa inFact is a graph-based Intelligent Testbench Automation tool that achieves coverage closure 10-100x faster than random stimulus, and enables users to be more than 10x more productive creating stimulus than with directed tests. inFact recently introduced functionality that enables inFact graphs to easily control software running on embedded processors, as well as the hardware-centric stimulus applied from the design interfaces. This article illustrates how inFact software-driven verification can be applied to the task of verifying power management in a SoC design.

VERIFYING POWER MANAGEMENT HARDWARE AND FIRMWARE

Verification of power management in a SoC is an interesting case that illustrates the intersection between verification of hardware and firmware. Power management is implemented in the hardware domain by power domains, voltage islands, clock gating logic and the power management unit. However, these power management mechanisms are controlled by firmware that runs on the processor and controls the power management unit.

Questa inFact is a graph-based Intelligent Testbench Automation tool that achieves coverage closure 10-100x faster than random stimulus, and enables users to be more than 10x more productive creating stimulus than with directed tests. inFact recently introduced functionality that enables inFact graphs to easily control software running on embedded processors, as well as the hardware-centric stimulus applied from the design interfaces. This article illustrates how inFact software-driven verification can be applied to the task of verifying power management in a SoC design.

Figure 1 - SoC with an Embedded Processor

The power management firmware has two primary sources of input events: application-level requests for the design to transition to a specific high-level power mode, and events from blocks within the system that cause the firmware
to either change the power mode or customize operating conditions to optimize power consumption. For example, the application might request that the design transition to the suspended mode. Then, at some point in the future, the keyboard might trigger the design to transition back to the active mode. Or, during active mode operation, the driver for the 802.11 block might reduce power due to inactivity.

Verification of the power management firmware in conjunction with the power management hardware is complicated by the fact that some power-state transitions are provoked by software while others are provoked by hardware. One first level verification task might be to exercise all valid power state transitions to ensure that the design correctly enters the target power state and behaves as expected. Beyond this basic level of verification, it could be desirable to provoke conflicting requests from the hardware and software. For example, does the design behave correctly if the keyboard issues a wakeup interrupt at the same time that the firmware is initiating entry into suspend mode?

Going beyond this macro level of power state verification, it might be interesting to exercise the combination of software- and hardware-initiated activity that provokes the firmware to dynamically customize the performance and power consumption of various blocks. This would enable verification that the firmware and hardware work together to meet the performance and power consumption requirements in various operating scenarios.

A typical verification environment, shown in Figure 2, divides the test into hardware and software portions. The software portion of the test can provoke some changes in power state by calling functions in the firmware layer. The hardware portion of the test can provoke some changes in power state by injecting stimulus from the design interfaces using the connected verification IP.

However, setting up the types of coordinated tests for power state transitions is difficult if not impossible without simultaneous control over both the software- and hardware-controlled portions of the test.

**QUESTA INTELLIGENT SOFTWARE-DRIVEN VERIFICATION**

Questa inFact is an intelligent testbench automation tool that integrates into many types of testbench environments. When running in a UVM environment and generating stimulus to be applied via the design interfaces, inFact integrates as a UVM sequence or UVM virtual sequence, and controls the verification IP connected to the design interfaces as shown in Figure 3 on the following page.
The Questa inFact intelligent software driven verification (iSDV) feature enables stimulus to also be easily applied via the software running on the embedded processor. As shown in Figure 4, inFact provides pre-built components that integrate into the UVM testbench and the embedded software to enable the inFact graph to control both the verification IP attached to the design interfaces and call functions within the software test. The pre-built infrastructure provided by inFact enables verification engineers to focus on creating the critical-to-verify scenarios rather than building core infrastructure.

**VERIFYING POWER MANAGEMENT WITH iSDV**

Questa inFact can easily be applied to setting up the types of verification scenarios described earlier in this article because it can control both the software running on the embedded processor and the stimulus applied from the design interfaces. Let’s see a simplified example of how inFact iSDV might be applied to verifying power management in a SoC design.
In this simplified example (Fig. 5), we can put the system into a low-power active state, a suspended state, or a hibernation state by calling a 'C' function called `set_power_state` supplied by the firmware. When the system is in the low-power active state or the suspended state either a keyboard interrupt or a real-time clock (RTC) interrupt can restore the system to active state. When the system is hibernating, only a keyboard event can restore the system to active state.

The power management API supplied by the firmware is shown in Figure 5. The test program can obtain the current power management state by calling `get_power_state`, and can set the next power management state by calling `set_power_state`. The test program can request an interrupt from the real-time clock after a specified number of clock ticks by calling `set_rtc_interrupt`.

```
typedef enum {Active, ActiveLow, Susp, Hib} ps_t;

ps_t get_power_state(void); // Get the current power state
void set_power_state(ps_t ps); // Set the current power state
void set_rtc_interrupt(int ticks); // Request RTC interrupt after 'ticks'
```

```
rule_graph pm_state_exerciser {
    action init, pre_fill, post_fill;
    set ps[enum Active, ActiveLow, Susp, Hib];
    meta_action_import curr_state[ps];
    meta_action next_state[ps];

    meta_action rtc_event[0..1000];
    meta_action kbd_event[0..1000];

    pm_state_exerciser = init repeat {
        pre_fill
        curr_state // Obtain current power state from the firmware
        next_state // Specify next state

        if {next_state == Hib} {
            kbd_event // Ensure a keyboard event is issued
            (rtc_event | eta)) | // Optionally create an RTC event
            if {next_state == Susp}
            (kbd_event rtc_event) // Issue both a RTC and keyboard event
            if {next_state inside [Active,ActiveLow]} {
                (kbd_event | eta) // Issue a keyboard, RTC, or no event
                (rtc_event | eta)
            }
        }
        post_fill
    }
}
```

Figure 5 - Firmware-provided Power Management API

Figure 6 - Example inFact Rules for Power State Verification
Our goal in verifying the power state is to provoke a series of power-state transitions triggered from both the hardware and software. The verification scenarios of interest are specified to inFact as a set of textual rules. The inFact rules for verifying power management are shown in Figure 6 on the previous page, and request the current power state from the software running on the processor, specify the desired next power state, then determine what events (if any) should be scheduled to affect the selected power state. The test software could be instructed to schedule a RTC interrupt after some interval. The verification IP attached to the keyboard interface could be instructed to initiate activity after some interval.

While the hardware/software stimulus we wish inFact to create is specified using textual rules, we can view these rules as a graph. The graph that results from the textual rules is shown in Figure 7. We can see the process that inFact will take when generating stimulus, and the conditions that will alter the testing process.

The graph does, of course, need to be connected to the hardware and software testbench environments. In the software environment, inFact injects stimulus by calling 'C' functions within the embedded software and passing the value associated with the variable in the graph. In this case, the 'C' functions that inFact calls will invoke the power-management functions provided by the firmware.

The code snippet in Figure 8 shows the function corresponding to the `curr_state` graph variable that inFact invokes to obtain the current power state. Note that the body of the function invokes the `get_power_state` function provided by the firmware, and returns the provided value.

The function in Figure 9 shows the function corresponding to the `next_state` graph variable, that inFact invokes to set the next power state. Note that the body of the function invokes the `set_power_state` function provided by the firmware.

Finally, for the software side of this simple environment, the `rtc_event` function calls the `set_rtc_interrupt` to schedule the real-time clock to issue an interrupt at some point in the future based on the value selected by inFact, as shown in Figure 10.

Within the hardware-centric testbench environment, the inFact graph is integrated as a UVM virtual sequence. The graph variables that result in action within the hardware...
portion of the testbench environment are implemented as SystemVerilog tasks within the virtual sequence.

inFact calls the task shown in Figure 11, which corresponds to the kbd_event graph variable. The task creates a keyboard-event sequence, specifies the inFact-selected delay for injecting the event, then launches the keyboard-event sequence on the keyboard verification IP sequencer.

CONCLUSION

Questa inFact, now with intelligent software driven verification (iSDV) capability, enables verification engineers to easily describe and realize comprehensive hardware/software verification scenarios. The infrastructure and automation provided by inFact enables inFact’s advantages of achieving coverage 10-100x faster than random stimulus and 10-100x more comprehensively than directed tests to be applied to verification of SoC designs where software must cooperate closely with hardware in order to realize expected behavior. Verification of these hardware/software scenarios early in the verification cycle enables hardware/software integration bugs to be located early when they are easiest to debug and correct, rather than being uncovered late – perhaps even in the lab – during operating system and application bring-up. inFact’s iSDV capability provides reusable infrastructure and automation, allowing verification engineers to focus on describing the desired verification scenarios rather than spending time building special-purpose infrastructure. The result is a true boost in hardware/software verification productivity.