Formal property verification: A tale of two methods

David Vincenzoni - January 15, 2018

The Formal Property Verification (FPV) methodology often gets used in the last step of verification flow, after much time spent building a complex random constrained UVM (Universal Verification Methodology) environment where some corner cases are still not covered.

Other times, FPV is used when a silicon bug is found that was not raised during the dynamic verification phase.

In a wisely applied verification flow, FPV should be used in the first phase, as soon as the RTL (Register Transfer Logic) code is available.

Here we look at two examples of verification flow:

- A digital block verified through a UVM test bench
- Then, first verified using FPV flow.

Verification of a flash interface

We’ll use example IP of a block that connects embedded flash memory through the AMBA (Advanced Microcontroller Bus Architecture) AHB (Advanced High-performance Bus). We translate the AHB transactions into flash custom transactions in order to perform the read, program, and erase commands.

We have two options to verify it:

1. Build the UVM test bench and develop random constrained tests
2. Formal Property Verification

Here’s the test bench for the UVM option:
Our DUT (device under test) is connected to the flash memory model, and the UVM Verification IP (UVM AHB Active Master) generates the random constrained sequences. A monitor or Passive Slave is used to check the transactions, and it is extended in order to implement a Sparse Memory for checking the data.

The test bench for the second, FPV option is null of course; we need to write the System Verilog Assertions (SVA) in order to check the expected behavior of custom flash interface.

To do this, we need to understand the signals for accessing the memory in read, program, and erase modes, and translate them to SVA. And as the DUT has an AHB interface, we need to avoid illegal transactions. In the Formal tool, we have to add the constraints (namely, SVA assume) for this bus.

**Formal Property Verification**

Our DUT is a block that has a processor bus interface (AHB) and a custom flash interface. We need to verify that the signals of the flash interface comply with the specs on its datasheet. The compliancy must respect the AHB bus protocol.

The SVA assumption for AHB bus can be summarized in these two simple statements:

- **AMBA_STABLE_ADV**: assume property (!HREADYout && HTRANS[1] == 1) |=> $stable(AMBA_BUS)
- **AMBA_HREADY**: assume property (HREADYin == HREADYout);

The first statement says that the bus is stable when it is stalled (when HREADYout) and we have a new transaction (HTRANS[1]=1). The second statement keeps the HREADYin signal coming from slave decoders equal to the HREADYout coming from the selected peripheral.
Of course, the AHB bus has more constraints related to burst and special accesses, but this block does not use that information, so we do not need to add more constraints.

We know that every `assume` constraint is a possible cause of bug masking. The constraint reduces the state space analyzed by the Formal tool. When the `assume` is not true, we do not verify a portion of the state space. For this reason, we have to be sure about the constraint that we set in the Formal analysis. The best way to check these `assume` constraints is to convert them into assertions and check through dynamic simulation. The above `assume` constraints have been checked by using the UVM AHB Verification IP.

![Diagram of Flash read access](Image)

**Figure 2** Flash read access, showing the timing diagram for reading data `Data1, Data2` stored at `Adr1, Adr2`. The memory has control signals and a strobe signal that samples the address and sector value.

Simply, the assertion that checks this read sequence is similar to this:

```plaintext
FLASH_READ:   assume property ((read_access) |->Control)
```

The `read_access` is a condition that describes the AHB read:

```plaintext
assign read_access = (HTRANS[1] == 1) && !HWRITE && HSEL && HREADYin;
```

The Strobe signal is the default clock.

We can then check that the AHB address (HADDR) that defines the read location is correctly translated into flash address (Adr1, Adr2). This can be written as:

```plaintext
FLASH_READ_ADR: assert property ((read_access) |-> Address == HADDR[m:n]);
```
At every read_access transaction, the flash address must be the same AHB address (HADDR). The range [m:n] is selected since the address size of Flash is smaller than the AHB address size.

**Figure 3** shows the waveform for programming the flash. Multiple program sequences can be defined by toggling the Strobe_Prog signal. Programming starts with the rising edge of Prog, and the Control signals are latched. When the Strobe_Prog is high, the Address and Data In must be stable, as well as the Control signals, and held for the programming duration.

![Figure 3 Flash programming cycle](image)

We can write SVA that describes the timing diagram in **Figure 3**, checking the stability of the Address and Data In during the Strobe_Prog:

```
FLASH_PROG_ADDR: assert property ((write_access) |=> ##1 !Strobe_Prog[*1:n] ##1 $rose(Strobe_Prog) ##1 $stable(Adress)[*p:q] ##1 $fell(Strobe_Prog));
```

The above assertion checks that at every AHB write cycle, the signal Strobe_Prog remains low for a defined number of clock cycles (from 1 up to n), then it “rose”, and the Address must be “stable” for a defined number of clock cycles (from p to q), after which, Strobe_Prog must “fell”.

**Figure 4** shows the timing diagram for flash erase. In this case, a page of flash is erased according to the value of Address. Also for this diagram, we can write an SVA assertion that checks the stability of Address and Control, and the sequence between Strobe_Erase and Erase signals, as we did for flash programming.
Other useful assertions related to the checks of setup-hold between the control signals and the Prog/Erase can be written. From Figure 3, we see that the control signal must be set, at least, in the same clock cycle of Prog rise.

Translated in SVA:

PROG_CORNER: assert property ($rose(Control) |-> not($past(PROG,2) == 0 && $past(PROG,1) == 1));

That means that every time I have a rising edge of Control, I never before had a rising edge on Prog.

The same can be done in case of erase (Figure 4), by changing Prog with Erase.

Others assertions can be written for checking the setup-hold for the data and address.

We verified the first version of this block using the UVM test bench, but for the second release of RTL code, we applied the Formal methodology.

Since the functionality has not changed, the random constrained regression tests have been run and successfully passed in this latest version.

Since the code coverage was very close to 100%, we should not expect bugs in this new release of RTL.

Surprise! Even after running the SVA assertions seen before, we discovered a bug for a corner case. In particular, the corner case that checks the setup-hold conditions (assertion PROG_CORNER) failed. When an assertion fails, we can debug it as we normally do when a dynamic test fails. We check the waveforms and RTL code, and we try to understand why the waveforms are not what we expected.
Figure 5 shows the counter-example coming from the Formal tool. We see the failure of the property; the Prog can rise before the rise of Control signal. Going deep into the analysis of this failure, we see that a legal AHB sequence access generates this violation. Figure 6 shows the AHB sequence. We discovered that there were possibly two back-to-back writes: the first one (hadr1) produced the rise of Prog, and the second one (hadr2) made a change of Control signals. These accesses can be generated by the AHB Master (e.g., microprocessor), and the program command will fail. This is a dangerous sequence that must be corrected. Once we found the root cause, the bug was fixed, and the property passed successfully.

Figure 6 shows the counter-example with AHB sequence.

From these examples, we gleaned two important observations:
• Even though the UVM test bench – formed by more than 400 random constrained tests with a high degree of coverage – was complete, it still did not raise the bug inserted in the second release of the block.
• FPV found a legal sequence that fails the property in just a few seconds.

Both environments run in about 6 hours total, but the big difference between them is the exhaustive proof of FPV.

Conclusions

We’ve looked at a case of digital block verification using the Formal Property Verification methodology versus the same block tested in Universal Verification Methodology. We saw the worth of the FPV flow: its ability to discover a bug introduced in the second version of the RTL. The UVM regression tests did not stimulate a particular legal sequence that violated the flash protocol, while the FPV found it in a few seconds of proof of SVA.

It required a week to write the SVA assertions. Much less time is needed to build a UVM test bench and develop the tests.

These two methodologies must be used jointly, because the dynamic simulation is also necessary to validate the SVA assume. But using formal verification first allows you to clean the RTL code, and to check corner cases that can require a lot of effort when the random constrained flow is used.

Related articles:
• Don’t over-constrain in formal property verification (FPV) flows
• Reuse UVM RTL verification tests for gate level simulation
• How formal verification saves time in digital IP design

—David Vincenzoni is R&D Design Manager at STMicroelectronics, responsible for the design and verification of new chips for Broadband Power Line Modems and for new families of devices for industrial applications.