Jump-Start Software-Driven Hardware Verification with a Verification Framework

Matthew Ballance
Mentor Graphics
8005 SW Boeckman Rd
Wilsonville, OR 97070

***Abstract*- Software-driven hardware verification is growing in importance for verifying interactions between hardware and low-level software in today's complex SoC designs. Hardware verification productivity has benefits from verification frameworks such as UVM. Similar verification-focused frameworks don't exist to boost productivity in creating embedded software-driven tests. This paper describes the benefits of having such a framework and proposes key features of a verification framework for software-driven hardware verification.**

I. INTRODUCTION

Software-Driven Hardware Verification has been around for a very long time. Going back to the era when designs were composed of discrete packages wired together on a board, running a small amount of software on the embedded processor was a great way to confirm that the processor and peripheral devices were correctly wired up. Moving forward to the era when processors were still delivered as self-contained packages and connected to custom logic on an ASIC or FPGA, hardware-software co-verification was critical to ensuring that the custom logic correctly implemented the processor interface protocol. Today, as the interactions between software running on multiple processor cores and hardware IP continue to increase in complexity, software-driven verification continues to be relevant as a way to verify integration between the embedded software and the hardware IP blocks in environments as varied as RTL simulation, emulation, FPGA prototype, and first silicon.

II. CHALLENGES IN SOFTWARE-DRIVEN VERIFICATION

The test creation process for software-driven verification environments involves unique challenges. Tests created for a software-driven verification environment are expected to run in a variety of environments, while most other tests, whether software- or hardware-oriented, are intended to run in one or, at most, two environments. Software-driven tests often need to run on a virtual prototype, in RTL simulation, in an accelerated RTL simulation, on an FPGA prototype, and on first silicon. These environments have radically different performance and visibility characteristics, and provide very different supporting software stacks. RTL simulation has low performance, high visibility, and typically has very little in terms of a software stack. In contrast, an FPGA prototype has high performance, low visibility, and likely will include a full OS software stack.

Hardware-oriented verification environments are typically monolithic, which is why it is typical to refer to “the testbench”. In contrast, software-driven verification environments are more like islands of test functionality that need to be coordinated to perform the verification activity. Each processor core in the design can be considered to be an “island”, since it is usually desirable for the test activity running on a core to run loosely synchronized with the test activity running on other processors. Generators of external inputs to the design, such as a hardware testbench environment, host workstation, or lab equipment, can be considered to be other “islands” of verification activity. Here again, the execution of these design-external components must be coordinated with test activity running on the processors within the design to maximize the verification value provided by a given test.

Furthermore, with a software-driven verification approach, the test executes along with the software portion of the design. This is a challenge because execution of the test can now alter execution of the software portion of the design. For example, a test that emits large quantities of debug information may slow down execution of driver software leading either to false failures or masking performance-related issues with the driver’s integration with the hardware.

III. MOTIVATION FOR A FRAMEWORK

Frameworks boost productivity by identifying common domain-specific testing patterns and providing infrastructure and design patterns that make implementing those testing patterns easier. Over the past decade, adoption of verification frameworks has been instrumental in boosting the productivity of hardware verification engineers. Hardware-centric verification frameworks, such as VMM, OVM, and UVM have provided a critical base layer for boosting productivity in building up verification environments. Some aspects of these frameworks, such as the component model and logging APIs are simple but fundamental to creating a verification environment. While, arguably, any verification organization could create their own version of these primitives, providing these primitives in the context of a framework ensures that they needn't be reinvented and re-implemented by every organization. Having a common component model and abstracted communication mechanisms also facilities reuse of IP developed within an organization and eases adoption of IP and automation developed outside the organization.

Frameworks for testing application software also exist, of course. One of the most popular unit-testing frameworks is JUnit [3], which is just one in the xUnit family of unit-testing frameworks [4]. Application-software test frameworks typically focus on managing test collections and making result checking more efficient. There are relatively few broadly-reusable testing patterns that application-software test frameworks can provide, due to the incredible diversity of the software to which these frameworks are applied. In contrast, software-driven hardware verification is about verifying the interaction between hardware IP blocks and between low-level software driver code and the hardware IP it controls. In this regard, software-driven hardware verification is much closer to hardware verification than it is to testing arbitrary software, and consequently benefits from many of the same patterns that benefit hardware verification.

IV. KEY SW-DRIVEN VERIFICATION FRAMEWORK REQUIREMENTS

The requirements for a software-driven verification framework have many similarities with the requirements for hardware-centric verification frameworks. For example, a software-driven verification framework must provide a component model to encapsulate functionality and abstracted communication mechanisms to facilitate communication between components. However, several key requirements differ from those of a hardware-centric verification framework.

Software-driven verification is performed in a resource-constrained environment where the test executes within the context of the design. Consequently, software-driven verification environments must be very lightweight – both in terms of memory consumption and in terms of computational requirements. Because of the resource-constrained nature of the embedded-software environment, a software-driven verification framework should provide features that facilitate shifting computation and data storage to the host workstation when possible.

Software-driven tests for today's complex designs are invariably multi-threaded. However, these multi-threaded tests must be able to run in a variety of environments with different threading capabilities – from cooperatively-threaded bare-metal environments to full multi-core preemptive threading running within a full operating system. An abstracted threading API insulates tests from the specific threading environment the test is running in, making tests portable and reusable across execution environments.

V. KEY SW-DRIVEN VERIFICATION FRAMEWORK COMPONENTS

Given the history of the development of verification frameworks for use in hardware-centric verification, it makes sense to leverage any concepts from hardware-centric verification frameworks that are also applicable in the software-driven verification space. The components described below borrow from the OVM and UVM component model and environment-construction concepts, extend from the abstracted communication mechanism used in SystemC, as well cover new ground to meet the requirements of a software-driven verification test. Examples of these framework components are illustrated using the API of an open-source software-driven verification framework named SVF [1].

A. Component Model

Having a component model is critical to enabling reuse. Unlike more-traditional application software, having uniformity in how reusable test software is encapsulated, is constructed, and communicates is critical to simplifying reuse. A component model specifies both a component hierarchy as well as the construction and connection order of components.

The component model used by SVF borrows heavily from the component model used by the UVM library. Component instances have instance names, and the component hierarchy is constructed, connected, and started via a three-step process:

- Build: Construct components within the object hierarchy
- Connect: Connect components in the object hierarchy
- Start: Launch any required threads

```
class producer : public svf_component {
  svf_component_ctor_decl(producer)

  producer(const char *name, svf_component *parent) { super(name, parent); }

  uint32_t get_data() { /* Called by the consumer */ }
}
svf_component_ctor_def(producer)


class consumer : public svf_component {
  svf_component_ctor_decl(consumer)

  producer                  *m_producer;

  consumer(const char *name, svf_component *parent) { super(name, parent); }

  void start() { /* Start 'consumer' thread */ }

  void run() {
    while (true) { m_producer->get_data(); }
  }
}
svf_component_ctor_def(consumer)


class top : public svf_component {
  producer     *m_producer;
  consumer     *m_consumer;

  top(const char *name, svf_component *parent) { super(name, parent); }

  void build() {
    m_producer = producer::type_id.create("m_producer", this);
    m_consumer = consumer::type_id.create("m_consumer", this);
  }

  void connect() {
    m_consumer.m_producer = m_producer; // connect consumer to producer
  }
}
```

Figure 1 - Component Example

Figure 1 shows an example of the SVF component model. In this example, a thread running in the consumer component calls the *get_data()* method on the producer component to get data.

- Instances of the producer and consumer component are created in the build() method of the *top* component. The build() method on each sub-component will be called after the *top* build() method completes.
- After all build methods have completed, the connect() method in the *top* component is called by the SVF library. At this point in time, all sub-components are guaranteed to be constructed, so the consumer can be connected to the producer.

- Finally, the start() method of each component in the hierarchy is called. In this case, only the *consumer* component needs to start a thread.

B. Abstracted Communication

Existing verification frameworks, including AVM, VMM, OVM, and UVM, have provided specific features for abstracted verification. These hardware-centric frameworks have focused on transaction-oriented abstracted communication, and provided constructs that allow a component to send or receive transactions without caring what produces or consumes the transactions.

Transactions, however, are not a natural, general-purpose software concept. Software modules interact via APIs. Several existing frameworks provide mechanisms for a component to export an API that is implements, though with varying degrees of complexity for the user. A software-driven framework must certainly provide a similar way for a component to export an API that other components can call.

API-based interfaces are often bidirectional, however, with the software modules involved being able to call each other. Consequently, a software-driven verification framework must provide a way for modules to interact in an abstract manner via bi-directional APIs. SVF provides a bi-directional API port and export that allows software modules to provide and consume an API.

```
class irq_if {
  virtual void irq_event() = 0;
}

class data_transfer_if {
  virtual void transfer_data(uint32_t data) = 0;
}

class transfer_gen : public public irq_if {
  svf_bidi_api_port<irq_if, data_transfer_if>    drv_port;
  uint32_t  m_data;

  virtual irq_event() {
    drv_port->transfer_data(m_data++);
  }
}

class driver : public data_transfer_if {
  svf_bidi_api_port<data_transfer_if, irq_if>    drv_port;

  virtual transfer_data(uint32_t data) {
    // start data transfer
  }

  void hw_irq() {
    // Notify the connected component of the interrupt
    drv_port->irq_event();
  }
}
```

Figure 2 - Bidirectional API Port Example

Figure 2 illustrates the bi-directional API port construct with a simple example of using the API port to implement interrupt-driven data transfers. In this example, the *driver* component implements a driver for a hardware IP block that transfers data. Various implementations of the test program will program the IP block to transfer data, and must be notified when the transfer is complete. The *transfer_gen* component implements the *irq_if* API that allows the *transfer_gen* component to be notified when a transfer is complete. The driver component implements the *data_transfer_if* API that allow the *transfer_gen* to request transfer of data. When the data transfer is complete, as indicated by a call to the hw_irq() method, the driver component signals the *transfer_gen* by calling the irq_event() method of the connected component.

C. *Threading Primitives*

Software to help verify today's complex systems is almost invariably multi-threaded. Consequently, a verification framework must support the notion of threads. Tests written against a verification framework will run in many environment types, with different thread implementations. For example, early system bring-up is done in simulation. Minimizing the overhead of the software stack is critical, both because the speed of simulation is low and because the complexity of debugging failures grows as more software, and more complex software, is running. Consequently, simulation-based bring-up tests run in a bare-metal (without an OS) environment, and often start with cooperative threading. As the system stability is proven, and perhaps as tests begin to run in emulation, preemptive threading may be introduced – still in a bare-metal environment for efficiency. Later, when an OS is introduced, extended versions of the same tests run in simulation and emulation may be run in the OS-based environment.

This need to run the same types of tests across an array of bare-metal and OS-based software environments is supported by a verification framework that provides threading constructs that can be implemented in environment-appropriate ways.

The SVF framework provides object-oriented threading primitives for thread, mutex, and condition that are modeled on the constructs provided by the POSIX threads library [2]. A few implementations of the threading API provided with the library are:

- Bare-metal cooperative threading for ARM cores
- POSIX threads, running on top of the Linux OS
- SystemC threads, for a virtual-prototype environment

D. *Logging*

The ability to log messages and data is key to getting visibility into the operation of a test environment. Message logging also represents an area of high overhead – almost independent of context. Consequently, even high-speed environments provide ways to control the verbosity of messages for performance reasons. Efficiency around logging is especially critical for software-driven verification, given the constraints imposed by the environments in which these tests run.

When a software-driven test runs in simulation, one of the primary constraints is the effective speed of embedded software execution. Due to the low instructions-per-second execution speed, message-formatting code can easily dominate the run time of the test. Simulation environments, however, will often provide high-bandwidth communication paths, such as shared memory, between the embedded processor and the host workstation. This helps to compensate for the low embedded software-execution speed.

In higher-speed environments, message-formatting time is much less of a concern. However, the link between the embedded processor and the host is frequently a low-bandwidth serial link. Consequently, even a moderate volume of messages transmitted across the link can overwhelm the available bandwidth.

The SVF library provides a logging API based that provides features to address both the concern of limited effect speed of the embedded processor and limited bandwidth on the link between the embedded processor and the host workstation. The SVF logging framework allows the work of formatting messages to be offloaded from the embedded processor to the host workstation, and minimizes the per-message information that must be transferred from the processor to the host.

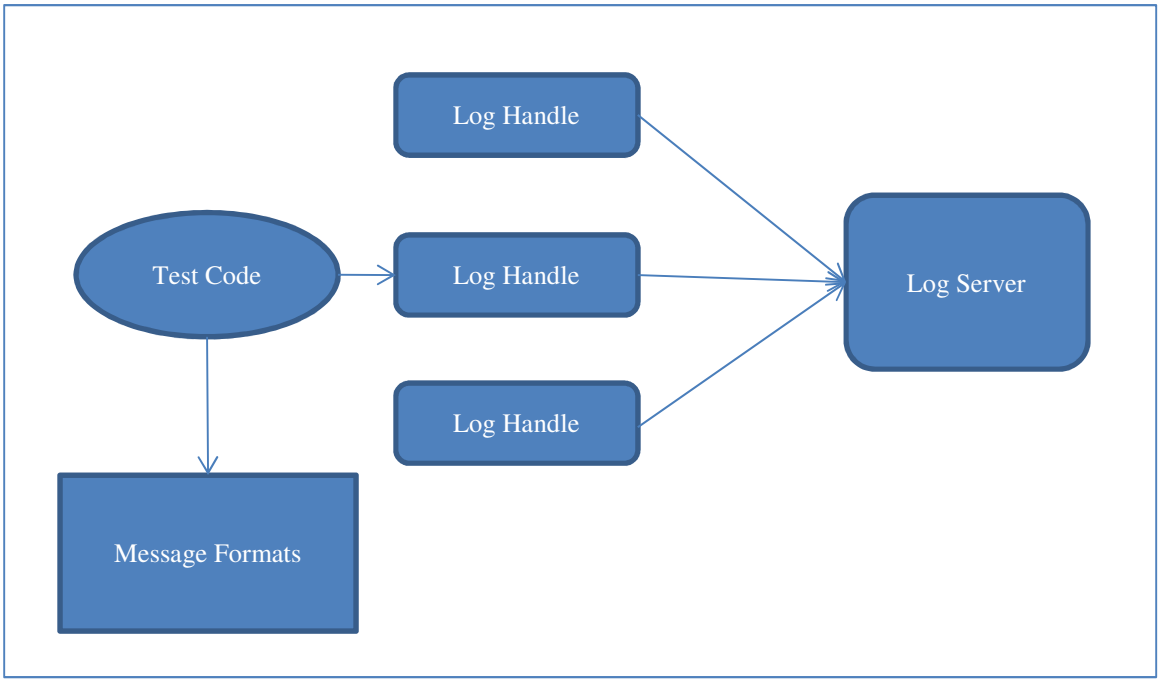

Figure 3 - SVF Logging Framework

Figure 3 shows the structure of the SVF logging framework. Test code uses a log handle to emit messages. The log handle, like those in most logging frameworks, allows the test code to be aware of the current logging verbosity. This allows the test code to only produce messages at the current verbosity level or above. Messages are routed to the log server that, depending on the environment, may display the message, route the message to the host workstation, or buffer the message to be displayed in case an error is encountered.

The SVF logging framework provides a mechanism to pre-define the format and arguments of messages. This capability provides a significant boost in logging efficiency. Messages are composed of two classes of data: the message-format data that never changes, and the message-argument data that does. Separately defining these two pieces of data up front enables implementations of the log server to deal with the elements of a message individually, instead of only dealing with the completed message as a whole.

```
svf_msg_def<uint32_t>   data_received_msg("Received Data %d");

class monitor :  public data_listener_if {
   private: svf_log_if      *m_log; // Log handle

   public: virtual void data_received(uint32_t data) {
     data_received_msg.msg(m_log, data); // Send the message
     // Process data …
   }

};
```

Figure 4 - Defining and using Message Formats

Figure 4 shows an example of defining and using a message with the SVF library. In this example, the *monitor* component will receive data produced by some other component. In addition to processing that data in some way, the *monitor* component displays a message indicated the data value that was received. First, the message format is defined using the *svf_msg_def* class. The message format specifies the type of the arguments accepted by the message: in this case, a single 32-bit integer. The message format object also specifies the format string for the message: in this case, "Received Data %d". Sending the message to the log server is done by calling the 'msg' method of the message format and supplying the message parameters – in this case, the data value received.

Splitting the message into message-format and message parameters gives the log server flexibility in how and where the message is stored, processed, and displayed. In an environment with a bandwidth-limited channel between the processor and the host, this flexibility allows just the message ID and the message parameters to be sent to the host. In the example above, sending the fully-formatted message would require sending at least 16 bytes, while just sending the message ID and message parameter value would require sending at most 8 bytes (possibly less with compression). This compression in the amount of data that needs to be communicated between the embedded processor and the host workstation, coupled with offloading message formatting from the embedded processor can substantially improve the throughput of a test.

E. Island-Bridging Constructs

As mentioned earlier, software-driven verification environments tend to be more distributed than verification environments for hardware. Each processor core can be considered to be an independent island that executes autonomously, but may wish to exchange data with the test activity running on another processor core. Possibly the most common islands in a software-driven verification environments is the activity running on the processor core and the activity running on the host workstation. Often, the test running on the processor core needs to communicate information, such as log messages, back to the host workstation for storage. Meanwhile, the host workstation (especially in simulation- or emulation-based environments where the effective speed of the embedded processor is low) may take an active role in generating verification stimulus.

It is helpful to have a common means of communicating, regardless of the source or destination of the data, and independent of how the data is actually transported. The SVF bridge construct provides a means for message-based multi-channel, bi-directional communication. The SVF bridge structure is shown in Figure 5. In this example, a bridge in an embedded-software environment is communicating with a bridge in an environment running on the host workstation. A bridge manages multiple message channels, represented by a *socket* in each environment.

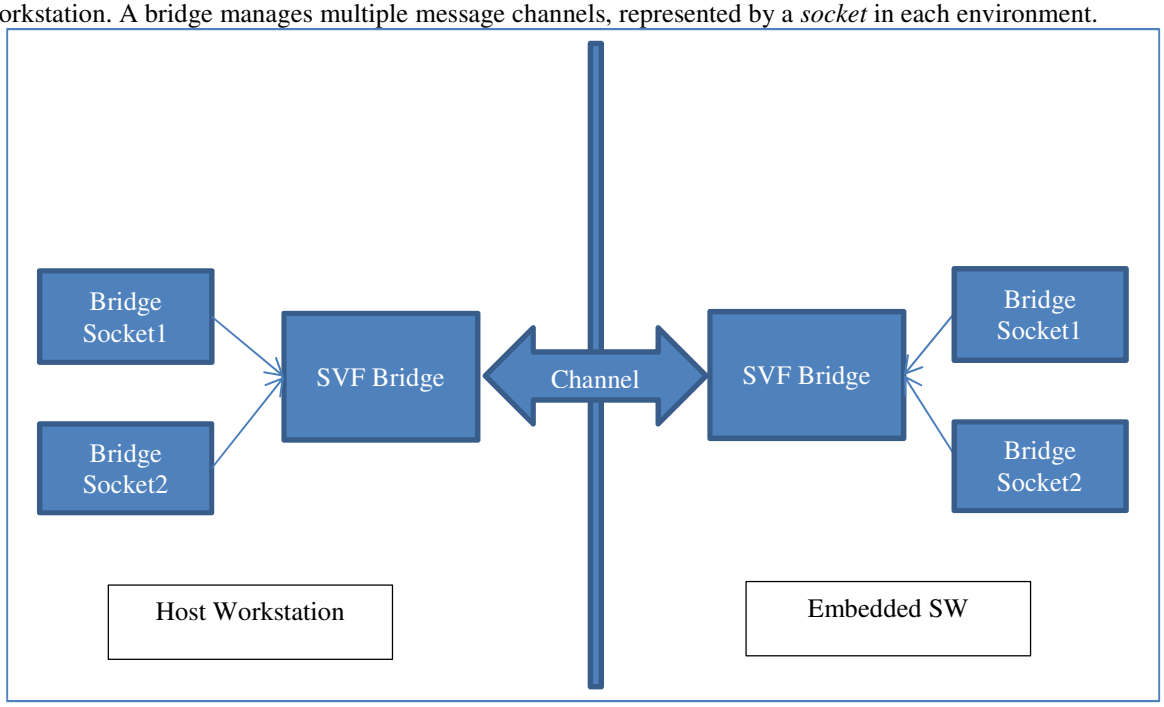

Figure 5 - SVF Bridge Structure

Different bridge implementations support communication between different types of testbench environment 'islands'. Below are listed a few bridge implementations

- Simulator-based shared-memory bridge from the host to embedded software running in simulation
- Emulation memory transactor bridge from the host to embedded software running on an emulator
- Unix sockets bridge from the host workstation to a Ethernet-based connection to the embedded software environment

VI. RESULTS

Applying the design patterns implemented by a software-driven verification framework such as the SVF results in the creation of modular test environments. All components have the same expectations about construction and elaboration order, and have well-defined interfaces between the internals and externals of the component. This well-defined boundary makes it easy to swap out the component implementation using the factory without impacting the rest of the test. This enables test variants to be easily created with minimal code changes.

Modular test environments also facilitate bringing new stimulus-creation techniques into test environments not originally designed with those techniques in mind. Accellera recently formed the Portable Stimulus Specification Working Group [5], with the goal of defining a stimulus-specification language that can be retargeted to various test environments. Modular software-driven tests, such as those enabled by the SVF, will allow the implementation of a portable stimulus model to be substituted for an existing stimulus generator component with no impact on the rest of the components in the environment.

The design patterns implemented by a library such as SVF also enable more automation in setting up test environments. Code templates and file generators were found to be immensely helpful in setting up SVF tests. Less code created 'free-hand' results in fewer mistakes and boosts productivity.

A software-driven verification framework needn't be resource-intensive. The current SVF implementation is quite lightweight, consuming approximately 16KB when compiled for a 32-bit ARM target. By comparison, the GNU libc implementation of printf consumes 8KB. This small footprint allows the SVF to be used in all but the most resource-constrained systems.

VII. CONCLUSION

Verification frameworks have been critical in boosting the productivity of hardware-verification engineers, by providing base infrastructure and design patterns, by easing the reuse of verification components and IP, and by easing the integration of automation tools into testbench environments. As embedded software-driven verification gains importance in tacking tough SoC-verification challenges, a verification framework focused on the challenges of software-driven hardware verification can provide these same benefits by providing pre-built infrastructure for creating components, abstracting the communication between components to facilitate reuse, providing platform-independent thread primitives, and providing communication infrastructure that allows processing and data storage to be performed in the optimal location. This base layer of reusable technology and methodology really can jump-start productivity in software-driven hardware verification.

REFERENCES

[1] M. Ballance, "SVF, a software-driven verification framework", https://github.com/mballance/socblox/tree/master/svf
[2] B. Blarney, "POSIX Threads Programming", https://computing.llnl.gov/tutorials/pthreads/
[3] JUnit, http://junit.org
[4] M. Fowler, "Xunit", http://www.martinfowler.com/bliki/Xunit.html
[5] Portable Stimulus Working Group, http://www.accellera.org/activities/committees/portable_stimulus/