Method of Free C++ Code Migration Between SoC Level Tests and Standalone IP-Core UVM Environments

Fedor Putrya
R&D Center ELVEES
fputrya@elvees.com

Abstract

Common way for IP-Core standalone verification assumes UVM based environments and tests development. At the same time, IP-core integration verification at the SoC level and hardware-software co-verification as a whole, requires development of the code running on the embedded CPU (usually written on C/C++). When C/C++ tests and software are developed it is desirable to reuse IP-Core standalone level verification code, presented in the form of UVM sequences library. However, porting of UVM sequences to C++ is difficult because of differences in the organization of UVM and C++ programs. Moreover, SoC-level tests development often requires a lot of time for debugging and simulation. This paper proposes method of free verification code migration between the standalone IP-Core and the system level SoC environments, which increases code reuse efficiency and accelerates SoC level tests debugging.

1. Introduction

The following tasks, associated with IP-Cores verification and maintenance can be highlighted:

- Development of test environments and tests for standalone IP-Core verification
- Development of tests for FPGA or emulator based IP-Core prototype verification.
- Development of IP-Core integration tests at SoC level and tests of IP-Core interaction with other elements of SoC
- IP-Core drivers and software development and debug
- SoC level faults found in silicon (or model of whole SoC) restoration on standalone IP-Core test environments (in order to fault cause analysis and regression tests base update)

Development of the code for each of these stages is a labor-intensive task. Therefore, the ability to reuse code between the stages of IP-Core and SoC development is a significant task.

Currently the most popular way for creating standalone IP-Core test environments and tests is the UVM methodology [1] based on SystemVerilog language [2]. This approach perfectly solves the problem of code reuse for agents - test environment building blocks. Such standalone test environments are quite fast, as they include only IP-Core's RTL model and UVM agents emulating activity of whole system.

UVM test sequences may be reused in SoC level for IP-Core integration tests. In this case CPU model is replaced by the UVM agent imitating CPU activity by executing of UVM sequences.

These UVM sequences also can be used for IP-Core verification in case of hardware emulator. However, at the moment it can be only possible in conjunction of a workstation and the emulator when the IP-Core model is placed into the emulator and test actions are formed on the workstation by UVM environment simulation. But simulation speed of such conjunction is limited, and the hardware and software tools aimed to accelerate such environments are very expensive.

Most of SoCs have an embedded CPU as their part. In this context, the following reasons that prevent the effective reuse of UVM sequences can be highlighted:

- It is still necessary to create test programs executed directly on the model of embedded CPU (commonly C/C++ code) for a complete SoC verification
- Using of UVM test sequences for topological netlist and especially for silicon prototype verification is complicated
- It is difficult to use UVM for IP-Core prototype verification without expensive hardware and EDA
- SoC level fault recovery on standalone IP-Core environments requires modifications of the code that caused the error
- The number of experts who know how to use UVM is still significantly less in comparison with C/C++ programmers for embedded systems

2. Related work

Solution proposed in patent US6539522 B1 [3] is closest to verification code reuse problem solving. It's allows to reuse verification code, written in a high level language (such as C) for both standalone IP-Core environment level, and the SoC level. In this solution verification code is divided on a high-level (test application) and low-level parts (drivers). However, this solution has several disadvantages:

- Reuse is possible only for the test application code, driver code still had to be rewritten
- It is not possible to create code for standalone master IP-Core verification, in the same style as it is usually done for SoC level software
- There is no hardware interrupt controller emulation for standalone environments
- OS simulator is required for standalone IP-Core verification (development or purchase of such simulator is needed)

Thus, full verification code reuse between all stages of IP-Core verification is still complexly realizable task.

3. The use of IP-XACT and IP-Core driver classes unification

The lowest level of the verification code is an IP-Core driver, interacting with the hardware resources of IP-Core by using of register structures. Nowadays IP-XACT IP-Core descriptions are growing in popularity [4]. Such descriptions may be a source of data for header files generators for SystemVerilog tests (such as UVM_RGM register map generator) and C/C++ programs (including drivers). Such register structures can be generated automatically for both standalone level and SoC level IP-Core drivers. But high level test code for UVM and SoC level tests are still very different.

To simplify code reusing it is necessary to unify low-level drivers for standalone UVM and SoC level C/C++ tests. In proposed approach IP-Core driver is created by using object oriented programming. IP-Core driver base **class** which provides a unified interface for IP-Core registers and memory access is generated automatically. **IP-Core driver class** itself is developed by using inheritance from generated driver base class and expanding by manually written IP-Core configuration and control functions.

The key point is similarity of IP-Core driver class interfaces for C++ and SystemVerilog (equivalent set of register structures, control and configuration functions). Afterwards, UVM test sequence or C++ test should be created by using the functions provided by the IP-Core driver class.

According to examples (Figures 1-3), C++ program and UVM test sequences that uses unified driver classes are similar. Thus UVM test sequence can be easily ported to C++ with test algorithm structure saving.

However, code porting from UVM to C++ as well as from C++ to UVM is partly automatic process and it's involves manual work and time for code rewriting and debugging. Moreover in case of third party IP-Cores standalone verification is not strongly required. In this case most of effort is spending on SoC level integration tests debugging.

```
class DMA_REGS{
  volatile unsigned int CSR;
  volatile unsigned int RUN;
}
class DMA_BASE{
   DMA_REGS * REGS;
}
class DMA_DRIVER public  DMA_BASE{
  int wait_not_run();
}
```

Figure 1. C++ IP-Core register structure generated from IP-XACT, IP-Core driver base class and extended IP-Core driver class for DMA IP-Core as an example.

```
DMA_DRIVER.REGS->CSR=SET_ENABLE(1);
DMA_DRIVER.wait_not_run();
```

Figure 2. Access to the register and function call example for C++ test of DMA IP-Core.

```
DMA_DRIVER.REGS.CSR.ENABLE=1;
DMA_DRIVER.REGS.CSR.write()=1;
DMA_DRIVER.wait_not_run();
```

Figure 3. Access to the register and function call example for UVM test of DMA IP-Core.

4. C++ program as test sequence for standalone IP-Core test environments

4.1. IP-Core model register and memory access

There is a software-hardware co-verification method which involves main program running on workstations CPU instead of SoC models CPU [5].

In the proposed approach, this method is extended by using the capabilities of the SystemVerilog DPI interface and the idea of using IP-Core driver classes. DPI interface which is a part of SystemVerilog, makes it easy to implement UVM test environment controlled by C++ program. In this case SystemVerilog DPI export functions are used to provide read and write tasks to the specified addresses in standalone UVM environment. Thus, main C++ program, that performs control operations over standalone IP-Core might look like in an example in Figure 4.

```
int wait_not_run(int timeout){
   while (dpi_export_read(DMA_RUN_REG));
}
```

Figure 4. Register read through the dpi_export_read task exported from SystemVerilog.

Such an implementation has a serious problem. Porting code between standalone IP-Core environment and SoC level tests requires wide code rework (DPI function calls replacing by regular register and memory access code and vice versa).

As mentioned above, in the proposed approach access to registers is performed through generated register structures with members of unsigned int type. After initialization of a pointer to such structure its members are mapping on the address space of real IP-Core registers. Feature of the proposed solution is register structure for standalone IP-Core tests environment. Such register structure is generated with same register layout as for SoC level, but with members of specialized register class type (**reg_class**, see Figure 5) instead of **unsigned int**. Register class reg_class has overloaded function set enough to provide register access operations from C++ program (dereference, casting, address taking, etc.).

```
class DMA_REGS_BASE{
  reg_class CSR;
  reg_class RUN;
}
```

Figure 5. Register structure class for standalone IP-Core tests.

In this case, C++ program that controls a standalone environment will be absolutely the same as C++ IP-Core test program for the SoC level shown on Figure 2. But, unlike the program running at the SoCs CPU, in case of program running on workstations CPU register access results in chain of function calls. Register access code execution causes call of appropriate overloaded function in reg_class. Reg_class member function causes calls of DPI functions exported from SystemVerilog and performing register or memory read/write operations within SystemVerilog test environment address space, including IP-Core model address space (Figure 6). In UVM test environment read or write task call produces sequence, that puts to master agent, which drives system IP-Core interface signals.

4.2. IP-Core requests handling

The foregoing is applicable for slave IP-Core. But IP-Core may also be a master and generate memory access transaction by itself. When C++ program is used as test sequence, data arrays intended to IP-Core can reside in address range allocated for the C++ program. Therefore it is important to provide access to the C++ program memory from IP-Core model. For this purpose two modifications of the standalone environment is needed: 1 – functions which can read and write given address in C++ should be imported into a SystemVerilog test environment (dpi_import_read, dpi_import_write). 2 - UVM agent connected to the host interface of IP-Core should be connected to these imported functions.

In this case when IP-Core tries to access addresses allocated in C++ memory through its host interface, UVM slave agent generate transaction, which executes appropriate imported DPI memory access function call.

4.3. IP-Core interrupt handling

Generally IP-Core is the source of interrupts which should be handled by C++ test program. If IP-Core interrupt occurs, UVM IRQ agent catches it and calls global IRQ handler function imported through DPI to SystemVerilog test environment. In proposed approach an IRQ handlers' registration interface is similar to that which is used in the OS and in SoC level tests. Thus, global interrupt function itself determines which of the registered user IRQ handlers must be called for current type of interrupt.

Main thread of test application should be stopped while interrupt handling is executed. So when imported global interrupt function is executed critical section should be started. Similar critical sections are started in case of register access from main thread. So there is a guarantee that the main thread of test application code will be stopped in moment of IP-Core register access until completion of interrupt handler.

In the issue test program for standalone verification of master IP-Core, which generates an interrupts, becomes more familiar with the terms of the system programmer.

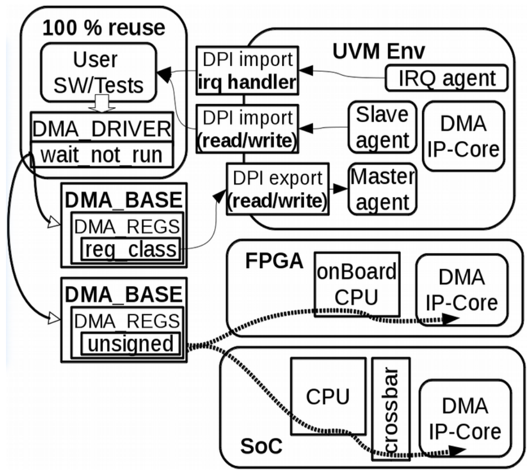

Figure 6. C++ test application and driver code reuse for a standalone IP-Core test, FPGA based prototype and SoC level tests.

5. Video encoder verification example

Example of the proposed approach using is hardware video encoder verification. Main test program for video encoder is video codec, which is a complex program written in C++. Some steps of this program should be performed by hardware encoder resources. SoC model is extremely slow for debugging software and hardware such as video encoder.

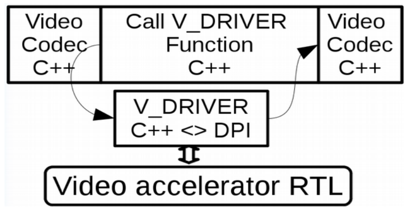

Figure 7. Video encoder RTL model testing with driver classes using.

In proposed approach C++ test program runs on a workstation, and hardware encoder RTL simulation initiated only in the case of a hardware encoding stage calls from main program. This reduces simulation time and helps to verify both hardware and software of encoder even without model of whole SoC. Moreover, C++ code could be reused (in any direction) on a stand-alone video encoder environment, SoC level environment, FPGA prototype (if it has onboard CPU) and later in firmware for encoder.

6. Conclusion

Proposed solution simplifies the task of UVM code reuse and provides an ability of complete reuse of C++ code across all stages of the IP-core verification process.

SystemVerilog provides a huge advantage in creating random coverage-driven tests. Therefore, UVM tests are necessary for IP-Cores which are under development. In case of using proposed approach to port UVM test sequence to C++ low-level driver should be rewritten in part of user-defined methods (register structures are generated automatically), but the high-level test application can be ported in semi-automatic mode. If IP-Core driver port is already done, porting of average tests may take only couple of hours against days of full code rewriting in common case.

In case of C++ code test application code is 100% reusable. And unlike the method from patent [3] IP-Core driver methods and code with direct register accesses is reusable too. Moreover interrupt handling mechanism and master IP-core programming for standalone IP-Core environment is exactly the same as for common SoC software. So C++ code written once according to proposed approach can be used in all stages of IP-core verification process (days of time savings for each test). On the other side SoC level code related to a particular IP-Core can be debugged on a fast standalone IP-Core environment.

Another advantage of the approach is tool independence. HDL simulator is only tool needed to run IP or SoC environments (unlike [3] there is no need for ISA or OS simulators). In case of SoC chip or FPGA only toolchains (like gcc) for appropriate CPU is needed.

Finally approach provides opportunities to create stand-alone IP-Core test scenarios for a wider range of C++ programmers.

7. References

[1] S. Rosenberg, K. Meade "A Practical Guide to Adopting the Universal Verification Methodology (UVM)"

[2] IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language / IEEE 1800(TM) -2005

[3] R.J. Devins, P.J. Ferro, "Method of developing re-usable software for efficient verification system-on-chip integrated circuit designs", U.S. Patent US6539522 B1, Mar. 25, 2003

[4] Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows, IEEE Std 1685 -2009, Published 18 February 2010

[5] A. Jason, "HW/SW co-verification basics: Part 2 - Software-centric methods", http://www.embedded.com/design/debug-and-optimization/4216264/1/HW-SW-co-verification-basics--Part-2---Software-centric-methods