2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

A Metric-driven Methodology For Firmware Verification

In Simulation/Emulation Environments

Goran SAVIĆ, Elsys Eastern Europe d.o.o., Belgrade, Serbia (*goran.savic@elsys-eastern.com*)

***Abstract*—Contemporary embedded systems have more and more firmware included directly on the die. Verifying it thoroughly before final delivery is sometimes challenging due to lack of appropriate verification tools. An in-house built tool using common (SystemVerilog) and mostly free of charge technologies (Perl, HTML/CSS/JavaScript) demonstrates a metric-driven method for the firmware verification process that is easily reusable in other embedded verification environments.**

Keywords—firmware; code coverage; design verification; embedded systems; EDA tool

I. INTRODUCTION

A typical embedded system consists of one or more processing units (usually an ARM core or other lightweight cortex) which control the system. More and more functionalities of such a device is actually stored in its firmware which implements higher level of controllability of the device, while at the same time extends its flexibility using more complex algorithms and patterns available at a higher software level using standard programming languages like C. Such a firmware inevitably becomes a part of the design and has to be extensively verified along with the hardware.

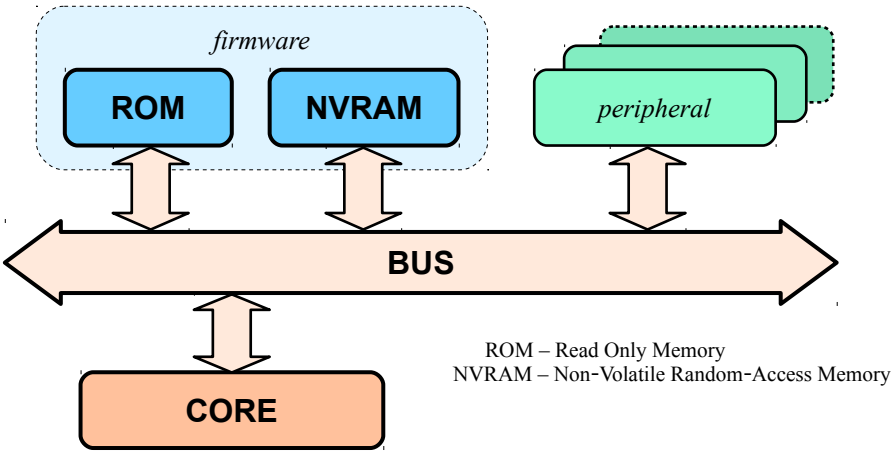

Figure 1 – Typical embedded system

During the verification process, it is important to thoroughly verify this, now crucial component of the design in order to avoid any gaps in the verification process. A common metric for doing this is having the complete software coverage of the firmware performed in order to confirm that every use-case scenario has been verified.

Here is presented a simple, reusable and tool-independent solution based on SystemVerilog [1], Perl [2] and common HTML/CSS/JavaScript [3,4,5] Internet technologies which has been successfully used in a couple of commercial embedded projects based on low-power ARM Cortex. A lightweight variant of this approach may be implemented in the FPGA verification as well.

II. EXISTING SOLUTIONS

Contrary to the functional and code coverage of digital design verification which is fairly well supported by a number of EDA tools, the software coverage usually depends on custom extensions and their implementation is especially required for some parts of the firmware (bootcode, test functions…) where standard firmware coverage tools are not available yet.

Although it is possible to use the built-in SystemVerilog support [1] (`sample`, `coverpoint`, `covergroup`…) for performing coverage on software code as well, in practice it is fairly inefficient and requires that a verification engineer poses a fair amount of software/hardware skills regarding the targeted system just to be able to identify the missing software component.

In Figure 2 is a detail from a typical coverage report of a firmware generated using standard coverage tools.

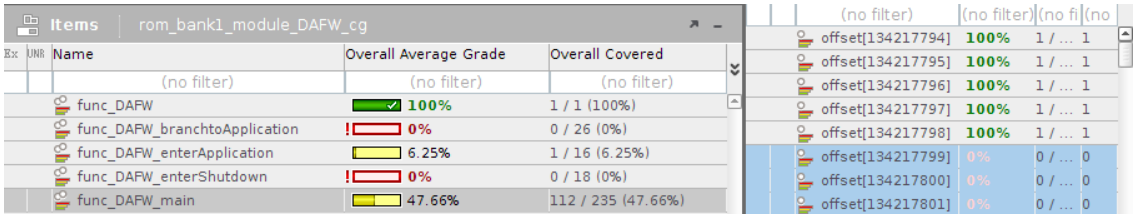

Figure 2 – Typical coverage report
Details from a custom coverage report accustomed for firmware.
On the right are given successive addresses associated with a function (DAFW_main)
showing that the instruction or data element at that address is covered or not.

It is obvious that a verification engineer has incomplete and quite cryptic bits of raw information regarding the executed code. The basic motivation is to provide a set of custom-built tools that may drastically improve the verification process and make it more accessible to a wider range of verification engineers.

III. IMPLEMENTATION

The implementation is done in a low-power System-on-Chip (SoC) embedded design with multiple ARM Cortex-M0+ cores using a standard Cadence verification environment based around the SimVision tool. Each core has its own memory space, firmware and peripherals attached to it.

Firstly, basic tool requirements were identified in order to estimate their feasibility against the simulation environment and potential constraining flaws. Secondly, a set of tools has been chosen with their availability, cost and productivity in mind. Several potentially critical points in the flow had been tested before the actual implementation was started.

A. Basics of Software Coverage

Generally speaking, tracking the flow of a program requires two major components to be monitored:

- the Program Counter register (PC) – it directly follows the program execution; if an instruction address appears in the PC, it means it is being executed (remark: if there is no speculative execution, then instructions may be dropped eventually).
- the dedicated instruction or general purpose data bus (AHB – Advanced High-performance Bus in ARM) – not all the code contains executable instructions, a part of it consists of different data as well, so they have to be taken into account and covered as well; their address locations do not appear in the PC and therefore would not be registered as covered.

Since all the components of the design are usually available to the simulation environment, it is easy to connect these two to a single SystemVerilog interface of our code coverage monitor. The system is sketched in Figure 2.

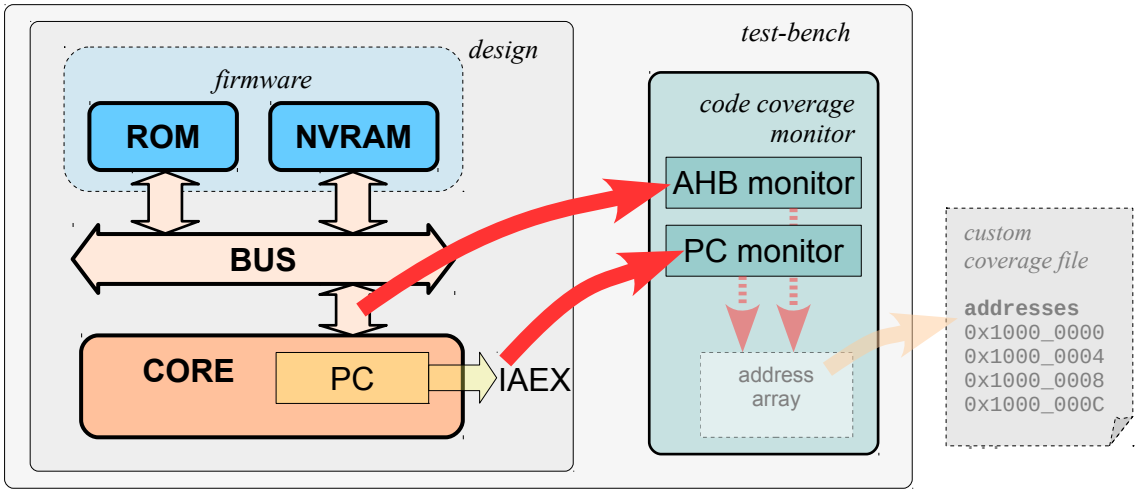

Figure 3 – Monitor interface and points of connection
Data flow from these tracked points to the monitors and their temporary storage (the address array) are marked with red arrows. At the end of a simulation, the collected data are saved as a custom coverage file.

Notice that in case of a multi-core design, the bus and PC of each core need to be tracked separately. In order to distinguish between different firmwares, beside tracking addresses it has also to be registered which core and/or memory space it is dedicated to.

B. Additional Design Requirements

The PC and AHB must be accessible to the simulation environment, i.e. they must be readable from the SystemVerilog-based test-bench. However, some systems have their specifics, hence the tool shall be aware of these deviations from the commonly accepted behavior.

For example, in an ARM-based system the actual value of PC points to a location 4 bytes above the memory address of the instruction currently being executed. Furthermore, in Gate Level Simulation (GLS) some bits of a register/bus might be omitted due to optimizations during the synthesis phase, so instead of accessing the PC register directly, it is considerably more efficient to use exposed values available at the specialized DWT IAEX bus of the Cortex itself – it points to the address of the actual instruction and it works just fine in GLS as well.

C. Collecting Coverage Data

Addresses being accessed during a simulation run are collected in a SystemVerilog associative array in the Code Coverage monitor component. The reason for using associative arrays is to allow a full range (32-bit) of memory addresses to be tracked.

Each element in the array associated with an address has three counters:

- **read accesses** – how many read requests was performed from that address;
- **write accesses** – how many write requests was performed to that address;
- **executed** – how many times the instruction at that address has been executed.

Accessed/executed addresses are collected concurrently from AHB and PC and new entries are added to the array if the address has not been registered before. Each new read or write transaction on AHB will increase the appropriate read or write access counter, while any new value of PC will increase the executed counter for that

address. The address resolution should be at least 2-byte since the 16-bit ARM Thumb instruction set is used.

This way the monitor is able to distinguish particular 16- or 32-bit instructions, as well as accesses to 16- or 32-bit data. The resolution of data access can be extended even further to the byte level, but it is generally not required for this task.

D. Generating Collected Output

At the end of a test-case run, the collected addresses with the number of read/write/executed accesses are saved to the directory of the test-case in a simple textual format. The disk space consumption is not critical, so we have chosen the following textual format depicted in Figure 4.

Figure 4 – Custom coverage file format

```
# block: User ROM Bank 1
# core: 0
# base: 0x10000000
# offset r_count w_count x_count
0 r1x1
2 r0x1
4 r10x10
...
# block: User RAM
# core: 0
# base: 0x20000000
2040 r12w17
2044 r17w17
```

Each accessed address is represented through its offset in a predefined memory block which has its base address, then the number of read accesses (after the "`r`" character), the number of write accesses (after "`w`") and finally the number of executions (after "`x`"). Notice that a memory block may be associated with a different memory space and/or core, so this information has to be provided (see "`# core: N`" shown in Figure 4) along with the addresses covered in order to allow multiple memory spaces.

E. Collecting and Connecting the Collected Data

When the coverage option is enabled in simulations, each test-case run (even in regressions) in its run directory collects the required data within the custom coverage file previously described. Collecting all the data created in a regression and their later processing is performed using a custom Perl script with a number of features which shall generate a coverage report at the end.

It is not unusual that an embedded device has more than one part of the firmware (functional API, bootstrap, test functions...). Usually they are internally distributed in the Executable and Linkable Format (ELF [6]) files which are naturally the major and obligatory input to the script containing all relevant information related to the firmware whose functionality is verified. Once these ELF files are disassembled to a textual format (using `fromelf` or `objdump` command-line tools and eventually cached to speed up the process), each generated file is parsed line by line and different sections, functions and individual instructions/data associated to a function are identified and stored in an appropriate data structure for later processing. A sample from raw output of the `fromelf` tool is given in Figure 5.

```
3951     main
3952 ;;; test_soc/sbl_bootloader_entry.test/sbl_bootloader_entry.c
3953 ;;;25     int main() {
3954         0x00001c34:    b510        ..      PUSH     {r4,lr}
3955 ;;;26         // prepare MSP and initialize SBL TB
3956 ;;;27         __set_MSP( SBL_MSB_ADDRESS );
3957         0x00001c36:    4846        FH      LDR      r0,[pc,#280] ; [0x1d50] = 0x20000400
3958         0x00001c38:    f3808808    ....    MSR      MSP,r0
3959 ;;;28         sbl_test_init();
3960         0x00001c3c:    f7ffff70    ..p.    BL       sbl_test_init ; 0x1b20
```

Figure 5 – Example of a disassembled ELF file.
Raw textual format, function main() disassembled with its interleaved source shown.

The parsed firmware data are usually stored in the form of associative arrays which form a tree structure at ELF, section and function levels, thus allowing quick looking up based on their names. Finally, each function node in this tree contains an associative array of instructions and/or data elements associated with their physical addresses, so they are uniquely identified for that memory space/core. Each ELF/section/function/instruction node in this tree also contains other parameters (memory start/end, number of instructions/data...) in order to ease their later processing.

Another functionality of the script is to collect previously generated firmware coverage either in a single-run simulation or a regression. Parsing the custom coverage files (in the format given in Figure 4) from each test directory is straightforward and collected addresses with the number of different read, write and executed accesses to each address by different test-cases is easily performed. Matching the overall number of accesses to an address of a memory space/core with an instruction or data element from the associated disassembled ELF file shows whether the instruction/data has been covered by that set of test-case(s). Nevertheless, this is optional, so an ELF file alone may be also inspected in this visual environment without coverage information.

Finally, all the collected data are finally saved as the final HTML report. A reasonable amount of care should be taken here since large quantities of data may affect final HTML report performances. For example, the HTML format has a lot of redundant data in its format specification, so it would make the generated report file enormous. Providing only raw data along with dynamically created HTML to graphically represent the data ensures that both memory footprint of the application and final user experience are quite acceptable.

F. Graphical Report

The final report may be presented in a raw textual form readable in any text editor. Nevertheless, to increase usability of the generated report even further, it is better to represent the collected data in a more readable form. Using a common and all-present HTML/CSS with additional scripting is a good choice – the tools are widely available, free of charge and extensible further with good overall support.

The presented graphical interface shown in Figure 6 can represent the collected data at several levels.

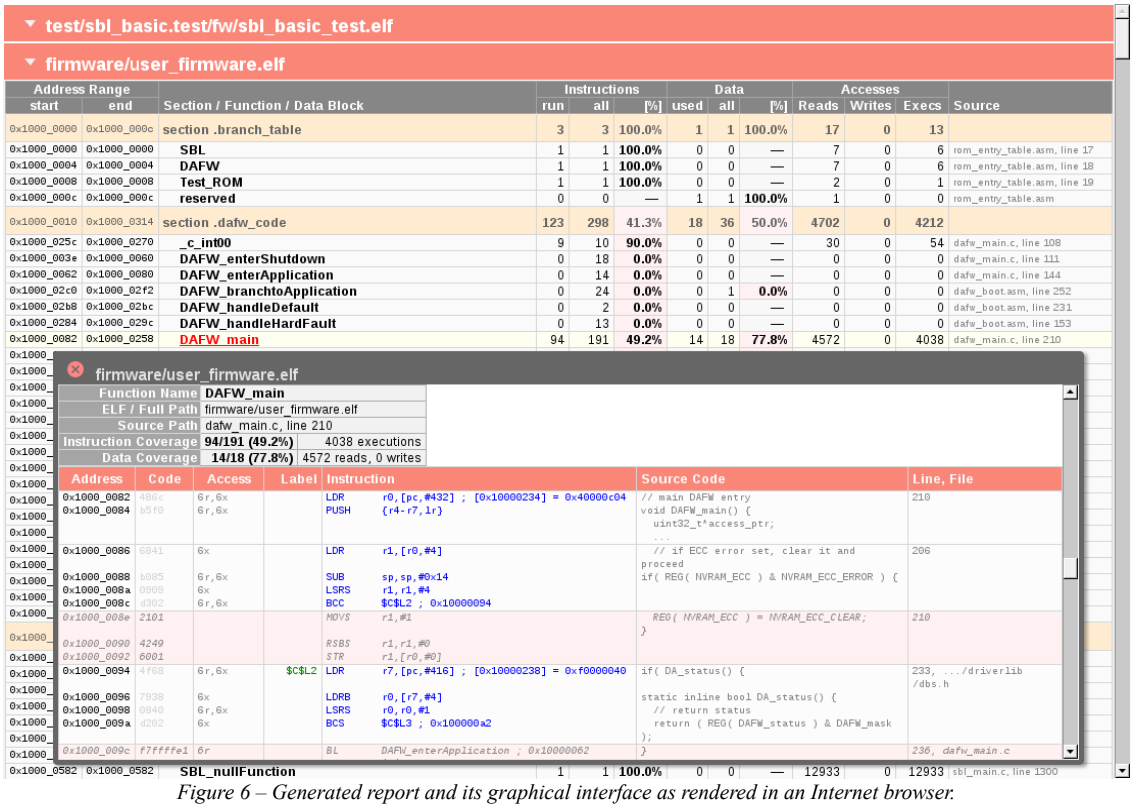

Figure 6 – Generated report and its graphical interface as rendered in an Internet browser.

Disclaimer: the code shown here is for demo purposes only, it does not reflect the actual implementation.

It shows coverage statistics and number of accesses at the ELF, ELF section, function and instruction/data levels. One may easily track the assembly code against its C-language source code leaving the choice of finding verification coverage gaps and related non-executed code to a low-level verification engineer or a high-level firmware designer.

Notice that almost all data available from an ELF file are preserved in this graphical form, but displayed in a more acceptable manner and just one or two mouse clicks away. For example, an incomplete (<100%) instruction or data coverage of a function from the list will be marked red, so it may be further inspected by clicking the link which opens a detailed report for that block of code in a sub-window. Labels in the assembly code are also links and hence makes navigation through it with ease.

The detailed report contains the actual instruction codes in address ascending direction, their assembler mnemonics and corresponding C sources followed with their originating files and line numbers. In the Access column are given the number of reads and/or writes for a data element, or the number of executions of an instruction. An instruction/datum line with no expected access will be marked red (for example, the instruction was only read, but not executed) meaning it is not properly covered by the battery of tests. It is almost straightforward clear which part of the code was not accessed, thus allowing a verification engineer to cover the corner as well.

G. Encountered Implementation Issues

Despite doing a decent amount of research on using different software components, several obstacles during its development have been encountered, in range from dealing with minor differences in formats of the C source code delivered by different software developers involved in the project to browser incompatibilities in implementing HTML/CSS standards along with rendering incapabilities of servers, for instance.

H. Additional Features

The executable code and data are associated with their appropriate addresses in the memory map of a core. Since the memory addresses being tracked are not associated with memories only, one may also track accesses to all addresses in the memory map. For example, one may track whether all the registers of a memory module have been accessed and the type of access (read, write), as well as the number of accesses in a set of tests and/or regression.

I. Lightweight Implementation in FPGA

An embedded system is usually verified on an FPGA-based test-board as well. While the method introduces way faster simulations and additional availability of some common software tools to the software development team, it still lacks the coverage information due to the nature of the verification method itself. Nevertheless, it is possible to implement a lightweight software coverage verification component for the FPGA platform as well, at little or no cost at all.

An FPGA platform usually has a number of built-in RAM (Random Access Memory) blocks available for synthesizing/emulating different memory models (RAM, ROM, NVRAM…). One of them may be connected to a rather simple custom controller which is connected to PC (IEAX port) and AHB. Tracking the values on these buses is done in a similar fashion as already described in the design verification, where one may associate one bit of the memory to one 16-bit instruction or data element to mark whether the address has been accessed or not. For a 128-KB firmware, an 8-KB block RAM is enough to provide coverage on all the firmware instructions/data. This is an acceptable trade-off between required FPGA resources, simplicity of the implementation and finally usefulness of the collected coverage. For test regressions, the content of RAM should be preserved between different test-cases. At the end, the collected data in this block RAM may be easily acquired through a general debugging interface and analyzed later with the same tool.

IV. CONCLUSION

The custom-made software tool for managing firmware coverage is a relatively inexpensive tool to develop (refer to Section *Implementation Code Complexity* below), but it justifies the investment shortly after through its new value added to the set of existing tools. It allows easy identification of coverage gaps during the firmware verification available to a wider range of engineers involved in the verification process, while making it metric-driven. Relying its development on freely available and cross-platform tools and technologies makes it fully platform independent, regardless of the simulation environment or final report consumer.

The flexibility of its realization allows many further improvement where some of them are listed in Section *Further Improvements* below.

J. *Implementation Code Complexity*

For creating a tool like this one, an estimated effort is summarized in Table 1.

Table 1 – Estimated development cost of particular tool components

Component	Language	Number of lines
Code Monitor	SystemVerilog	100–200
Report Generator	Perl	400–800
Generated Report	HTML/CSS	50–250
	JavaScript	150–300

Of course, these numbers are quite approximate and depend on many factors like used coding style, list of functionalities implemented etc.

K. *Achieved Results and Limitations*

The exposed methodology has been used in two projects so far, covering only specific firmware components in the simulation environment where the targeted coverage was 95–100% depending on the component. Despite the ability of the tool to precisely handle software coverage, it is usually constrained only to a set of critical firmware components which have to be extensively verified. A typical firmware development process is usually based on FPGA test-boards which offer way greater execution speeds and more comfortable environment to software developers, so it is natural and more efficient to do the full firmware coverage on the FPGA along with its development and software unit tests. Providing coverage in both environments additionally ensures that there is no loopholes in any environment, especially where the critical software components are tightly related to hardware requiring finer details of the design with specific stimuli.

Since all design elements (software/hardware) are accessible to the simulation environment, there is practically no limitations other than the speed of simulation. The load introduced to the simulation environment is less than 1%, both in simulation performance and memory requirements. Nevertheless, for a firmware of modest size (128 KB) the generated HTML report is about 2–6 MB in size, mostly dependent on whether C source code is included or not, so it may impact the user experience in a particular browser.

L. *Further Improvements*

Color coding for improved code readability, graphical analysis, multi-browser support, switch from propriety `fromelf` to open-source `objdump`, filtering and sorting results in the report are amongst many possible improvements of the tool that might take place in the future.

ACKNOWLEDGMENT

The author would like to thank the entire Texas Instruments Freising Security design and management team for their support during the implementation of aforementioned solutions.

REFERENCES

The following references are given in the form of currently active Internet links to their actual standards.

[1] SystemVerilog 1800-2012 IEEE Standard, https://standards.ieee.org/findstds/standard/1800-2012.html

[2] The Perl Programming Language, http://www.perl.org

[3] HyperText Markup Language (HTML), https://www.w3.org/TR/html/

[4] Cascading Style Sheets (CSS), https://www.w3.org/TR/2011/REC-CSS2-20110607/

[5] JavaScript / ECMAScript, http://www.ecma-international.org/publications/standards/Ecma-402.html

[6] Executable and Linkable Format, http://elinux.org/Executable_and_Linkable_Format_(ELF)