SAME 2012 Conference

Session: Multiprocessing Hw/Sw

Metric Driven Hardware/Software Co-Verification of Interrupt and IPC Driven Firmware Sub-Systems
François Cerisier, Test and Verification Solutions, France
Markus Winterholer, Cadence, Germany

Abstract

As System-On-Chip complexity continues to increase, hardware software partitioning pushes more complex state machines to dedicated processor based sub-systems. The functionalities of such sub-systems are partitioned between hardware blocks and dedicated firmware controlled by inter-processor communication (IPC) channels and interrupts.

Verifying such sub-systems faces the challenge of identifying bugs not only in both the hardware and the software parts, but also at the boundary of the two disciplines.

Co-verification methodologies have proved to be interesting approaches for critical software verification. Existing methodologies are however based on the assumption that the software layer provides APIs or software interfaces. They therefore do not directly apply to the IPC and interrupt driven firmware of these sub-systems.

This paper presents an approach which enables interrupt based firmware verification for such partitioned hardware/firmware sub-systems at the early stages of the projects and before the silicon is available; reusing hardware verification environment and modeling IPC and interrupt events. This methodology has been applied on an industrial complex control sub-system. Hardware event controllability and observability is reached using extended hardware coverage verification techniques and the required software observability is achieved using the monitoring facilities of hardware/software co-verification tools.

Table of Contents

Abstract ............................................................ 1
1 Introduction ............................................... 1
2 Motivations ............................................... 1
3 The hardware software partitions to verify 2
4 Building the test bench ..................................... 2
5 Firmware Verification and Co-simulation ............. 4
6 Dealing with memory space ............................ 5
7 Results ...................................................... 5
8 Conclusion .................................................. 6
9 References .................................................. 6
About the Authors ............................................ 6

1 Introduction

Today’s applications such as portable multimedia devices or smart phones integrate above hundreds of more or less complex blocks within the same System-On-a-Chip (SoC). To face with this ever increasing level of complexity while still improving reusability and flexibility, more and more controls that were previously handled by hardware state machines are replaced by a micro-controller and some dedicated hardware blocks.

The hardware part of such sub-systems is responsible of handling atomic tasks (e.g.: copy data from a memory to a FIFO), while the high level functionality is handled by the firmware functions (e.g.: wakeup the display, the main SoC processor and stop the MP3 player when a USB key is plugged in).

The choice of implementing part of this system in firmware comes from the following reasons:

1- Implementing software in parallel to backend activities and silicon mask realization reduces the time to market of the final product.
2- Critical bugs in software have less impact than in hardware and can be fixed at the late stages of the process delivery.
3- Part of the desired functionalities may not be known at the time of the hardware implementation and need flexibility (standardization still on-going, late definition of specific applications, ...)
4- The same hardware can be used in different configurations or for different applications, augmenting design re-use and reducing design cost.

The main aspect of such firmware is that they are strongly linked to their targeted hardware resources (hardware counters, peripheral registers, hardware FSMs, SoC events, ...) and are controlled by interrupts and hardware events via software interrupt service routines and inter-processor communication channels. This is opposed software that are controlled by API function calls.

2 Motivations

Since the true functionality of the design resides in the firmware controlling the hardware and vice-versa, a complete co-verification using state-of-the-art functional coverage metrics at the hardware/software boundary is necessary.

Verifying the hardware alone only assesses that the basic hardware blocks enable the higher level software features. On another hand, verifying the firmware alone is generally not possible due its strong dependencies on its targeted hardware
In the past few years, hardware/software co-verification techniques have been developed to automate software verification and to report software functional coverage metrics. They are however based on the assumptions that the software is a set of functions defining an API for higher level applications and they do not directly apply to interrupt driven firmware and inter processor communication channels.

We therefore need to adapt the existing methodologies and techniques to solve the interrupt driven firmware co-verification problem.

3 The hardware software partitions to verify

The design we have to verify consists of a complex control subsystem (Figure 1) and its associated firmware.

The hardware part is composed of:

- a 32-bit RISC processor core dedicated to embedded software
- interrupt interfaces and inter-processor communication dedicated hardware
- a set of control registers that the firmware accesses for specific hardware features
- a set of timers and a dedicated interrupt controller
- external interfaces (AHB, APB, I2C …) to communicate with external hardware blocks or the main SoC processor.

![Figure 1: Design under Verification](image)

The firmware’s default execution state is IDLE and only wakes up on IPC and hardware interrupts. The firmware is then responsible for servicing requests from external hardware events or from the main bus interface. The following code example, gives a template of such firmware:

```c
void main() {
    main_init();
    while (1) {
        asm("IDLE")
    };
}

void wakeup_request() {
    ...
};

__attribute__((interrupt))
void isr_3() {
    maskinterrupt_macro;
    if ( cfg.allow_wakeup ) {
        wakeup_request();
    };
    unmaskinterrupt_macro;
}
```

Direct control from the main SoC CPU is therefore only possible via interrupts. Either these interrupts are routed from the main SoC CPU to the DUV, or the SoC triggers the DUV interrupts via specific IPC hardware.

In our case, the communication between the host processor and the DUV is performed in two steps:

a) The main SoC processor writes a message to the data memory (IPC mailbox) with information on the action to be performed.

b) The main SoC processor writes to a specific register (IPC trigger) of the design, which triggers an interrupt request via the dedicated internal interrupt controller.

To this extends, in regards to the rest of the SoC, the DUV therefore looks like it is a pure hardware block and is only controlled by hardware signal events or bus accesses to its registers or memory.

4 Building the test bench

In order to minimize development effort the firmware verification strategy builds on an existing coverage driven hardware verification environment.

Existing C tests are replaced with actual firmware which we trigger by using test bench verification intellectual properties (VIP, in our case Specman eVCs ) to generate external events.

We use Cadence Incisive Software eXtensions™ (ISX) to collect the hardware/software functional coverage metrics necessary to reach our goals and to implement functional checkers on the firmware execution.

Existing Hardware verification

The hardware verification environment is developed using an eRM /OVM functional coverage driven methodology.
The test bench contains all necessary verification IPs (VIP), a generic scoreboard to check the transaction correctness and routing, register models and all necessary checkers and coverage metrics to complete the hardware verification.

Due to the large number of interfaces, complex tests implementation has been eased by the use of a higher level virtual sequence driver able to synchronize transactions between the different interfaces.

It is therefore possible to define high level test scenarios for responding features, or to synchronize register accesses with interrupt events. This test bench capability is actually a key point in reusing the hardware test bench for interrupt driven firmware verification.

Extending the hardware test bench to target the firmware verification

Although the hardware verification environment includes behavioural models of the external interfaces and protocols, additional modeling is required to properly execute the firmware. For example, the firmware, after requesting an external state change, may wait for this state transition. The hardware verification would not have this model if input and output relations were not defined as a hardware protocol linking the request and its result.

To implement this additional modeling, we extend the existing hardware verification environment. The monitors are extended and linked to the virtual sequence drivers to create the required behaviour.

In addition the firmware accesses registers of an external component. The hardware verification environment may not need a complete model of these external registers as its focus is on the interface protocol only. However, for the firmware verification we need to check that the firmware is correctly performing actions depending on these external registers values.

In our particular case, the external component that the firmware is controlling is an IP which had also been verified internally and all register definitions were already available. The strategy is therefore to reuse the register description of this IP, instantiate it in our verification environment with no modification and link it to the AHB slave eVC to model the transfer responses. The checkers therefore automatically verify the validity of the register access. This also allows easier implementation of external state machines which were dependent on these external registers.

Virtual Sequence Library

The virtual sequence library of the hardware verification is further developed to randomize accesses to the hardware resources.

To cope with valid requests for the firmware being executed, the virtual sequence library is extended to provide:

- simple high level firmware request entry points (change state sequence, …)
- synchronization between firmware requests and pure hardware events.
- additional SoC modeling of the external interfaces.

Using a virtual sequence to control all the sequence drivers of the environment eases the hardware verification and becomes a “must have” for the firmware verification to allow productive high level implementation of all possible interactions and requests from the different external interfaces.

Issues when extending an environment for firmware verification

Extending an existing hardware verification environment for firmware co-verification has been made feasible by the modularity of the primary environment and the availability of a virtual sequence driver which became the main entry point for the firmware verification (Figure 4).
The techniques to extend an existing environment differs depending on the environment language (e.g., SystemVerilog, others) and methodology applied (eRM, OVM, VMM, ...)

Specman e-language implements the aspect orientation using the “extend” keyword. Extension of a ‘e’ verification environment is consequently eased by this feature, but care must be taken while building the original environment. In general, eVCs or other VIPs are generally well-coded, following guidelines and methodologies such as eRM or OVM, making them highly reusable and extendable.

However, due to project planning and pressure, verification environments are less often built in a way that can be easily reused by other projects. In our case, little modification of the original environment is necessary. This mainly relates to:

- hardcoded behavioural models of slave interfaces, which required a switch to set them off
- dependencies between some checkers and the sequence library, not used by the firmware.

As a general rule there should be no general struct/class extension in the verification environment that overrides all instances of the struct. Sub-type should always be used so that there is always a possibility for the extension to define a new sub-type of the original class.

### 5 Firmware Verification and Co-simulation

#### Interrupt Based Firmware Verification

Interrupt based firmware verification is enabled by the verification environment extension described above. Stressing the firmware is then made by creating the right set of sequences, requesting firmware actions via all possible means available.

Functional coverage metrics is also implemented to assure that:

- all global variables of the firmware reaches all their possible transitions,
- interrupt requests occurs while certain functions are being executed,
- contradicting hardware events occurs while firmware is running other tasks.

**Figure 5: Software State Machine**

Figure 5 shows an example of a firmware state machine where a rising edge of sig_A could immediately follow a IPC request to go to state S2. In this case, the firmware may still not be stable and could potentially be still executing the transition function from S0 to S2, performing intermediate actions.

In such cases, we first expect the transition from S0 to S2 to complete, and then immediately followed with a transition to S1.

**Firmware monitors using ISX**

The Incisive Software eXtension from Cadence provides means to monitor function calls and their arguments from Specman environment (via memory mailboxes). This is mainly useful in order to:

- ease the debug by tracing the exact time of the function calls
- collect information to implement functional coverage metrics on the internal firmware execution and cross these metrics with hardware events
- implement checkers between firmware execution and the external model states.

ISX default monitoring is performed using wrapper functions surrounding the function being monitored.

```c
int isr_routine_N_wrapper(t_State newState) {
   isr_routine_N_call (newState);
   int ret_val = isr_routine_N (arg1);
   isr_routine_N(ret_val);
   return ret_val;
}
```

To use such wrapper we need to replace all function calls by function calls of the wrappers. This could be error prone and also becomes impracticable when the calls are performed from assembler routines or from a dynamically modified interrupt vector table.
We therefore need to automate a different instrumentation which does not require any modification of the caller routines and which can be bypassed using appropriate "define":

```c
#ifdef _ISX_MONITOR_ISR_FONCTION_N_
extern int isr_fct_N_core(T v);
int isr_fct_N(T v){
  isr_fct_N_call(v);
  int ret_val = isr_fct_N_core(v);
  isr_fct_N_return (ret_val);
  return ret_val;
};
#else
// Original Non instrumented Firmware function
int isr_fct_N (T v) {
#endif
// Actual isr_fct_N implementation
};
```

As described in Figure 6 below, this implementation is automated by parsing the ISX description file of the monitors and parsing in the C the functions to be monitored.

This approach allows us to create the wrapper on the monitored function instead of replacing all possible calls in the code.

Another big advantage of this implementation is that it allows the Makefile to select which functions are actually monitored by defining (or not) the associated C label. This proved to be very useful when dealing with the memory issue (see below).

### 6 Dealing with memory space

ISX instrumentation requires additional code to be compiled with the firmware.

In our case, the design was providing a memory of 24 kilobytes and the firmware alone was consuming 22 kilobytes. The instrumentation of all functions required additional 5 kilobytes in total. To resolve this, we had to:

- comment/remove all non-functional code such as DEBUG trace
- remove code from ISX source itself, used only for emulators (not used for simulator backdoor)

This had been made feasible by the automatic instrumentation script described in Figure 6.

### 7 Results

**Co-Verification versus SoC level simulation**

Both this technique and directed tests at the SoC level were applied. However, due to the lack of controllability and the simulation overhead of the complete SoC, the SoC level simulation only targeted general use cases and were not directed for corner cases and stress tests.

The following table compares both techniques, in terms of simulation time and coverage metrics.

<table>
<thead>
<tr>
<th></th>
<th>Co-verification</th>
<th>SoC simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boot time</td>
<td>3 minutes</td>
<td>24 minutes</td>
</tr>
<tr>
<td>FSM state coverage</td>
<td>100% automated from generation</td>
<td>100% based on directed test coding</td>
</tr>
<tr>
<td>FSM state transitions</td>
<td>100%</td>
<td>&lt; 30%</td>
</tr>
<tr>
<td>State transition source coverage</td>
<td>100%</td>
<td>&lt; 5%</td>
</tr>
<tr>
<td>Number of tests</td>
<td>27 random scenarios, derived in 1950 different random simulations</td>
<td>35 tests</td>
</tr>
<tr>
<td>Development effort</td>
<td>6 man month</td>
<td>4 man month</td>
</tr>
</tbody>
</table>

**Functional cross coverage**

This coCo-verification allows to full verification fully verifying of all software state machine transitions, in random orders and with random delay.

Additionally, being able to monitor function execution and crossing this information with hardware event occurrences, allows defining functional coverage metrics crossing the hardware and software boundary (e.g.: a software function is being executed when a hardware event occurs).

**Instrumentation over-head**

Code instrumentation requires available spare CPU memory both for data and for code.
ISX implementation is using mailboxes data memory, of a minimum of 16 words, plus 1 word per monitor variables and 1 word every 16 monitored functions.

Originally, in our case, ISX instrumentation consumed an additional 28 assembler instructions per function. Further optimizations has reduced it to 10 instructions.

The code instrumentation required to monitor N functions of the firmware is therefore consuming the following memory space:

\[
\text{MailboxSize} = 16 \times 4 + N \times 4 / 16 \\
\text{CodeSize} = N \times 10 \times 4
\]

Monitoring variables and function arguments also consumes a pointer for each variable being monitored, although this should not be necessary since variables are already present in memory.

For a firmware containing 83 functions, this gives an additional 85 bytes of data memory and 3320 bytes of instruction memory.

In our industrial case, monitoring 83 functions with their arguments, consumes 53 words (212 bytes) for the ISX mailbox and 4280 bytes for the code instrumentations, leading us to split the monitors in different executable depending on the test categories to be run.

8 Conclusion

The methodology described in this paper requires has been built on top of an existing hardware verification environment which was highly modular so that extensions are feasible with minimal modification. Additional modeling work are necessary to model interactions between the firmware and the external world.

On the other hand, this methodology enables the verification of interrupt based firmware during the early stages of the project, prior the Silicon is available and potentially prior the integration at SoC level is performed. Using pseudo random generation techniques similar to hardware verification also allows to stress the firmware states and requests. ISX has additionally brought the capability to better monitor the internal execution of the firmware, allowing to write firmware checkers and crossing functional metrics between hardware events and firmware dynamic execution.

9 References

[8] Winterholer M.: Metric Driven Validation, Verification and Test of Distributed Embedded Software; Embedded World Conference, Nuremberg, Germany 2010

About the Authors

François Cerisier has an Engineering Diploma in Digital Signal Processing from Polytech’Sophia, University of Nice-Sophia-Antipolis and over 12 years of experience in verification of IPs, CPUs and System-On-Chips and in hardware/software co-verification. François gained verification methodology expertise from industrial projects of major semiconductor companies (including Infineon, Broadcom, ST-Microelectronics, ST-Ericsson) and EDA startups. François is also lecturer at Polytech’Sophia where he teaches Design Verification to undergraduate students.

Markus Winterholer has a Diploma in Computer Science from University of Tübingen. He is working since seven years for Cadence Design Systems in the research and development department. As Solutions Architect he is responsible for embedded systems test and verification tools. Before he joined Cadence, he worked five years as a freelancer offering consulting services for hardware and software development and verification for various industry leaders.