2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

# Achieve Complete SoC Memory Map Verification Through Efficient Combination of Formal and Simulation Techniques

## Equvialent Class Partitioning is used to systematically break down the SoC memory map within the context of multiple modes and a multi master environment

Clemens Roettgermann, Freescale Semiconductor, MCU, Munich, Germany
(Clemens.Roettgermann@freescale.com)

Peter Limmer, Freescale Semiconductor, MCU, Munich, Germany

Michael Rohleder, Freescale Semiconductor, MCU, Munich, Germany

***Abstract*—A fully verified address map on SoC level is one of the challenges in System-On-Chip (SoC) systems. The paper outlines why this is essential in the context of functional safety and security. Existing simulation and formal based verification techniques on their own fail to solve the problem. A method for solving this problem by a unique combination of equivalence class testing, simulation techniques and formal verification techniques is presented. The results prove successful application of the method to SoC verification and give some insight in the type of RTL defects which could be discovered.**

***Keywords—SoC, security, functional safety, freedom from interference, memory map, equivalence class testing, formal verification***

## I. MOTIVATION

A fully verified address map is one of the challenges in System-on-Chip (SoC) verification and a primary goal for all semiconductor devices that are targeted for a safety application and/or including security features. This is driven by some megatrends which are increasing the importance of solving this challenge:

**SECURITY:** An incomplete address decoding or the existence of undocumented registers provides potential backdoors that can be exploited by attackers to circumvent security features (i.e. areas only accessible within a specific lifecycle state, master and mode specific accesses). Mirrored address ranges are known to cause security vulnerability on SoCs throughout the industry.

**FUNCTIONAL SAFETY**: Furthermore, functional safety requires proving the "Freedom from Interference" between software tasks; especially when these tasks are developed for different safety levels (e.g. "quality controlled" software vs. software developed according to an ISO26262 ASIL level). This implies the need to ensure there are no side effects between register accesses but also that the memory usage of different software tasks does not impact other software tasks. "Undocumented registers" and "mirrored address spaces" are on obvious violation of such a requirement.

**MULTI MASTER**: The number of processors, DMA's, encryption engines, Protocol Engines which can act as master within a SoC is constantly increasing. The need to verify the complete memory map for all those masters including any master specific deviation significantly increases the complexity of this task.

**MULTI-MODES**: The increasing amount of security and safety features of SoC's introduces additional "modes" of operation which often influence the memory map.

As a result any memory map access is not anymore a function of the address but becomes a function of the address, the master, and any further SoC mode (e.g. safety,security) having an impact on the memory map decoding:.

$$accessibility = F(address, master, safetyMode, securityMode)$$

.

## II. STATE OF THE ART

State-of-the-art verification methods for the verification of memory and register accesses suffer from the complexity of the resulting address map in combination with involved run-time and capacity issues:

- Checking a single address by simulation requires a significant amount of simulation cycles (to be multiplied by the access sizes, the amount of masters, of the relevant SoC modes). As a result, already the verification of a 16KB address slot for one IP block for one combination already takes hours. Safety/security devices require this check for every master and every possible safety/security mode. Additionally have side effects between registers to be taken into account.
- Formal verification involves the formulation of specific properties. Although possible, this is not trivial due to distributed modes/permissions and involved complexities. The bigger problem however is to get those checks proven on SoC level. This suffers from severe capacity issues that already cause verification at module level to require GBs of memory. The capacity problems result from the state-space-explosion on SoC level. Applying this within a SoC is significantly more complex since it needs to cover the path from the master to the data sink, taking into account the SoC modes, the multiple masters, etc. Getting such checks proven on the SoC level would require lot of abstraction such as abstracting to a single clock domain, abstracting IP, pipeline stages, arbitration etc. Those abstractions potentially cause to ignore relevant aspects and consequently miss a related defect.

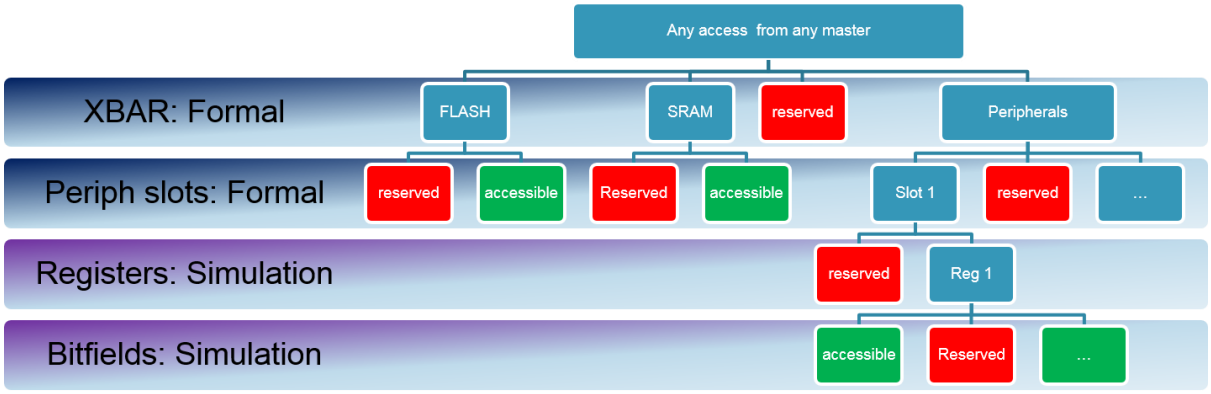
This makes a complete address map verification basically impossible by brute force methods. There are many methodologies for verifying register accesses and behavior in the SoC context. One of them, which is generating register tests based on an IP-XACT database is part of the approach presented within this paper. The generated stimuli can be applied by SoC simulation for validation purposes. In contrast to sub-module IP verification these tests guarantee the register and bitfield access in the context of clock domain crossing, MPU settings, register protection settings and other SoC aspects. Additionally they ensure that the SoC memory map as document in SoC reference manual the SoC specific parameterization of the sub-module IP. Due to simulation time limitations these approaches cannot be applied to large address ranges. On the other side, announcements in [3] mention, that formal methods can be applied for memory map verification but no dedicated method is described. The approaches of OneSpin [2] and formal register verification tookkits target to IP level register verification and are hard to apply on the SoC level. The methods [2] and simulation based methods are regarded as complementary.

The systematic approach we present in the next chapter combines formal and simulation methods by means of equivalent class testing. A brief overview about equivalence class testing can be found in [1].

## III. INTRODUCING A SYSTEMATIC APPROACH : HIERARCHIAL PARTITIONING

The solution we have developed is a systematic break down of the verification space to enable equivalence class testing and the related partitioning. Subsequently the most suitable verification methodology can be selected for every equivalence class. The resulting scheme matches and goes along with the usual address decoding as it is applied in many SoC, and allows as such a natural selection of verification matters for corresponding SoC objects. The method has the capability to ensure that the applied equivalence class partitioning matches the SoC implementation by proving through formal methods; enabling to perform completeness and consistency checks for the performed verification The following picture depicts the achieved hierarchical breakdown of the address space, and the resulting mapping to one of three possible checks:

- accessible memory range
- reserved range
- peripheral register slot (which might be eventually broken down further)




The figure shows how the address range is broken down hierarchically. Both, the accessible memory range as well as the reserved memory range are assumed to be contiguous address ranges providing equivalent access behavior. Peripheral register slots are either broken down further, or covered by simulating verification patterns that are generated from a register database.

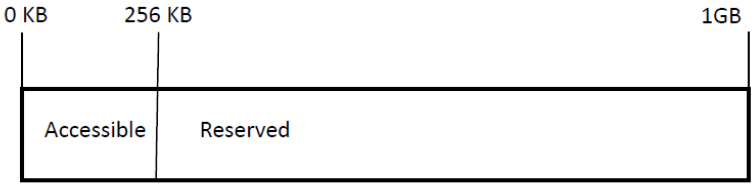
Based on a high-level memory map specification as it is provided within a SoC Reference Manual, the above method enables a partition of the address range into equivalent classes, i.e. [accessible], [reserved] and [other]. For example:

- Memories can have [accessible] and [reserved] equivalent classes.
- Bridge modules can have [reserved] and [other] type of equivalence classes such as [SRAM], [FLASH], [PERIPHERAL SPACE] etc., which usually can be broken down further …

Formal verification is then used to confirm [reserved] and always [accessible] areas. Any ([other]) result to such an address range triggers an RTL bug or a wrong partitioning. Areas belonging to [other] classes can often be further broken down to their expected routing (e.g. a bridge access routed to the Flash) having many alternative access paths stay invariant. With the above methodology it is possible to break down any related address space and decide whether a selected sub-address space is better simulated, verified by formal verification, or its partitioning need to be continued further (new iteration).

## IV. EQUIVALENCE CLASSES

Equivalence class verification requires to understand and analyze the implementation. For example, suppose we have a 1GB FLASH address space, in which 256KB of FLASH is accessible.




From a system point of view, the behavior of this FLASH block relates to the following pseudo-code:

```
if ( flash_address < 0x40000 ):
       access_flash(flash_address)
else:
       respond_error
```

At this abstraction level, only two "types" (equivalence classes) of accesses can be observed:

[accessible] : any access between memory address 0 and 0x3FFFF

[reserved]: any access to an address larger or equal than 0x40000

From a behavioral point of view any access to an address within the memory range 0x00000 and 0x3FFFF behaves equivalent; dependent on the specified access properties. Of course the behavior can be further dependent on these properties; e.g. misaligned 32 bit accesses. However any access with equivalent properties to such a range will behave exactly the same. This even holds if the property "Accessible" and "Reserved" is not just a function of the address but a function of other attributes e.g. like a security_mode. In that case would the pseudo code look like:

```
if ( (flash_address < 0x40000) && (mode==unsecured) ):
       access_flash(flash_address)
else:
       respond_error
```

Hence, only two sets (classes) of properties need to be verified. Unfortunately, only two test cases are not enough, since a wrong implementation can introduce additional behavior:

```
if ((addr<0x40000) && (mode==unsecured)):
       access_memory
else if (addr==0x4cafe)
       I_AM_THE BUG
else:
       respond_error
```

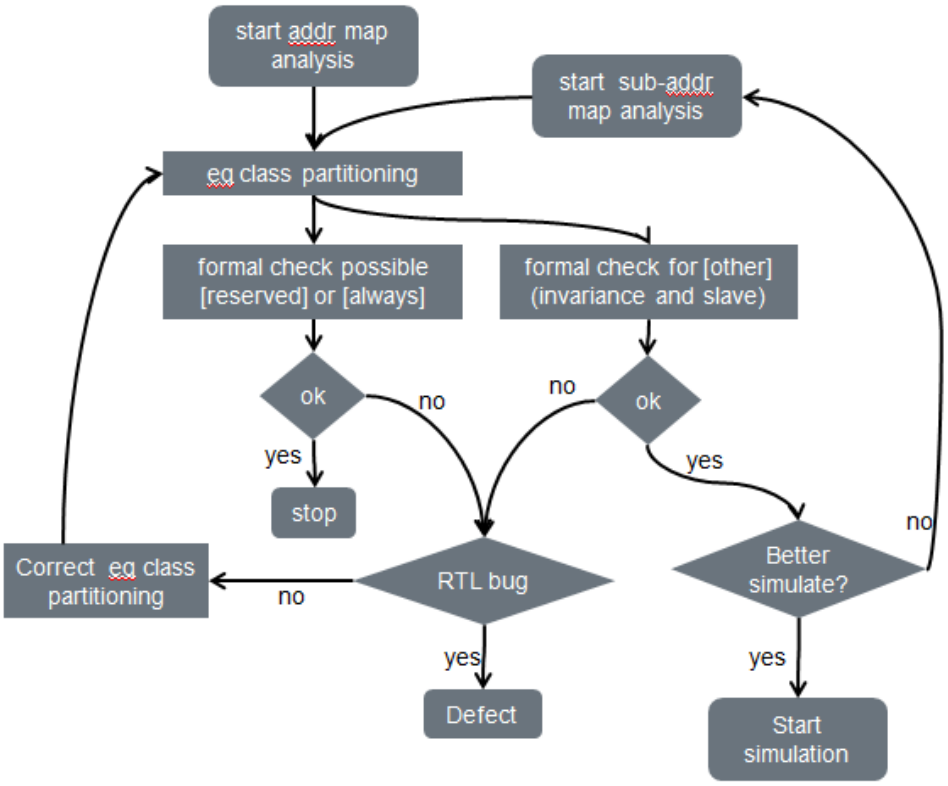
This bug can be discovered in three ways:

1. Brute-force: simulate the design for all addresses. Given, the usual simulation run times, this is unrealistic; especially when different SoC modes need to be considered.
2. By analyzing and reviewing the logic. Due to the involved complexity and the need to comprehend this by the human brain, this is only realistic for small designs with few lines of code. Furthermore it would need to be repeated each time the logic is modified.
3. By exploiting formal methods and prove that all accesses below 256 kB range are accessible if SoC is in unsecure mode (1st property) and all accesses are reserved (2nd property). The formal tool will provide a counter-example if one of the properties does not hold in the design. In fact, the tool analyzes the code and the formal method provides evidence that the assumed equivalent class partition matches the actual RTL implementation.

Once we have such a partitioning we can apply a technique inherited from equivalence partitioning; for which it is enough to test particular cases on the equivalence class to prove correctness (equivalence partitioning plus boundary value analysis). Therefore, what we can apply formal verification to prove existence of a proper equivalence class separation and break down the address map to significantly reduce the number of simulated test patterns.

## V. THE VERIFICATION FLOW

In practice an iterative verification flow is performed which is shown in the following flow.




## VI. INVARIANT CONDITION

Another issue that makes the verification of the complete memory space very complex are the many variations of accesses and access conditions that are in existence. To conquer this problem, a second concept has been applied: invariant conditions. For many elements within a system-on-a-chip, there are access properties, that are invariant – and do not alter between multiple conditions – and can therefore be used to reduce the complexity. Some typical examples for invariant conditions are:

- a multi-core device, where multiple processor cores share a major portion of the address space, having equivalent access to any location within this address space
- a "secure" address range, where only the "security" core is permitted to access the contained memories and/or peripherals; any other bus master cannot access any location within this range
- a "local" memory range, that is providing a dedicated memory for a particular bus master; this memory can only be accessed by this bus master
- address ranges, that can only be accessed in test mode; but also address ranges, where there is no dependency on the selected mode (supervisor/user or test mode)

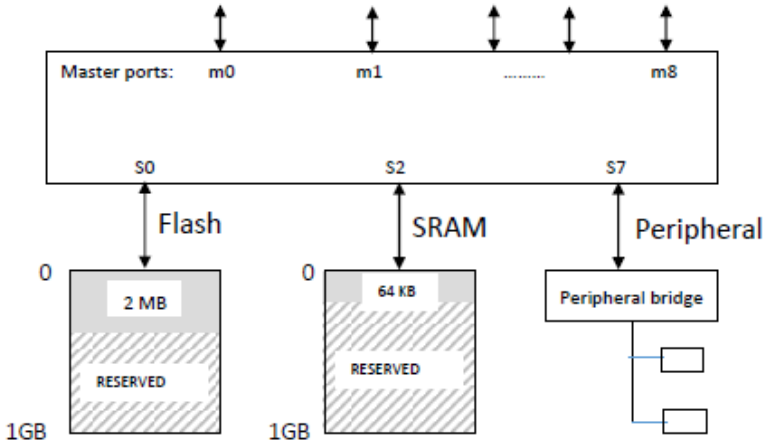
Due to the importance of "invariant conditions" we hosted the master thesis [4] which developed methods for verifying such "access invariants" on bridge modules

Without taking such invariant conditions into account, any possible access condition will result in the need to consider multiple accesses. When it is possible to proof the invariance of such a condition (eventually only for a particular range) for any resulting combination, it is possible to limit the verification to cover only a single combination (for the particular range). As such, invariant conditions are just a specialization of an equivalence class. Due to the complexity of the involved logic, proofing such invariant conditions is often only possible for a particular SoC element, i.e. a cross-bar or a peripheral bridge. However, when applied, such invariances can drastically reduce the verification space, as can be illustrated by the following examples:

- when it is possible to proof that any master connected to a cross-bar behaves equivalent, then only the accesses of a single master need to be verified
- when it can be proven, that any access to a specific address range will be routed to a single slave port on a cross-bar, then only the slave devices connected to this port need to be verified for this address range. Additionally, all other slave ports can safely ignore this address range for the verification
- when it can be proven, that toggling the wire transmitting the supervisor property does not have any impact on a particular set of SoC elements, then the corresponding verification can be done for a single mode; selecting either supervisor or user mode will not make any difference.

When applied, it can also be used to verify differences between the used bus protocols, to ensure the involved translations are not affected by an actual access.

The following example can be used to illustrate a possible break down of the address space, in its corresponding invariant conditions. Assuming a cross bar, providing multiple master ports to three slave, as shown in the following:




First it might be possible to proof that all masters behave equivalent for any access to one of the master parts. Subsequently only a single master needs to be selected for the remaining verification steps. Secondly it might be possible to proof that any access to a certain address range selects slave port S0 (Flash), to another address range selects port S2 (SRAM), and to yet another address range selects port S7 (Peripheral). All other address ranges are forbidden to access (equivalent to [reserved]) and shall result in an error condition. This can be formally proven.

The next level might now break down the corresponding address ranges; e.g. the Flash or SRAM address range might provide a certain valid range (related to the [accessible] condition), while the remaining portion of this address range cannot be accessed again (corresponding to a [reserved] condition). Furthermore write accesses to the Flash range may be forbidden or may be ignored, which provides another invariant condition that can be used to reduce the complexity of the verification task further. Similar applies to the peripheral space, that is usually broken down into peripheral slots, which are either [reserved] or provide access to a single peripheral.

As it is described above, it is possible to avoid needless repetition of the same verification steps for different masters/slaves/SoC elements. By exploiting the address map hierarchy we can select the most appropriate method for a particular verification task. In fact, register and bitfield spaces are, on SoC level, better handled by generated stimuli because formal analysis would explode for accesses that come from a top-level module in the memory hierarchy (a core for example) and go down to a peripheral slot in the SoC. On the other hand, formal methods are very efficient to verify address ranges with repeating or equivalent access patterns.

## VII. RESULTS

In the following we show some examples of problems which have been identified and fixed:

- Nested mirroring of address range could be identified. By nested we mean for instance a mirror within a 16K peripheral slot which multiplies with a mirror of the complete peripheral area.
- One safety relevant feature is the “Register Protection” which allows to lock configuration registers after configuration. It uses the register address within a 16K peripheral slot. One mirror within the peripheral slot allowed to bypass that mechanism.
- One security relevant feature is a restricted programming of specific flash blocks. The mechanism uses an absolute address in conjunction with specific security modes. A global mirror allowed to bypass that protection logic.

Overall we could identify several critical address map problems and fix those before silicon fabrication. A few of those even presented security vulnerabilities or affected safety mechanisms. Furthermore we provide better confidence for “freedom from interference” arguments which are based on the documented SoC address map.

## VIII. SUMMARY AND CONCLUSION

During the verification process of SoCs we were able to develop the described methodology and perform formal verification of high-level address decoding by proving existence of equivalence classes. Additional verification has been performed by simulation of several hundreds of generated stimuli to target registers and bitfields.

In conclusion, we can say that this methodology is an unique combination of formal verification and auto-generated simulation based test cases, which can be applied in conjunction with equivalence class testing and proof of invariance conditions to achieve an increased confidence in the equivalence partitioning applied to reduce required simulation times. This methodology may allow for an increase in the quality of memory map verification and of documentation delivered to customers and to avoid security vulnerabilities and to confirm ‘freedom of interference’ required for safety applications.

## IX. REFERENCES

[1] http://en.wikipedia.org/wiki/Equivalence_partitioning

[2] http://www.onespin-solutions.com/index.php/register-map-verification-solution.html

[3] http://www.design-reuse.com/articles/27424/static-formal-verification-for-system-level-verification.html

[4] Michele Di Giorigo, “Enhancing the verification of safety and security aspects of a hiearchical memory map in a System/on/Chip with formal methods”, EURECOM,, March 2015

.