TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MIIKA IHANAJÄRVI

UNIVERSAL VERIFICATION METHOD AND ENVIRONMENT FOR RISC PROCESSOR

Master of Science Thesis

Examiner: Prof. Mikko Valkama
Examiner and topic approved in the Faculty of Computing and Electrical Engineering Council meeting
4th of March 2015

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Signal Processing and Communications Engineering
MIIKA IHANAJÄRVI : Universal Verification Method and Environment for RISC Processor
Master of Science Thesis, 63 pages
May 2016
Major: Digital Communication Circuits
Examiner: Prof. Mikko Valkama
Keywords: processors, verification

This thesis introduces Nokia Co-processor (COP) and universal verification method (UVM) based verification environment for it. COP scales from a very small one threaded 32-bit processors up to a 128-bit and 16 threaded processors. COP alone can be used for example as a direct memory access accelerator, but by adding hardware accelerators, the COP can turn into a very effective engine for many different applications.

COP is a legacy IP from Nokia's previous system on chip (SoC) organization. Recently is has been updated to accommodate the needs of SoCs nowadays. Due to major changes, COP's legacy verification environment became unusable. Thus, a proper verification environment had to be developed.

A proper verification environment is crucial for IP blocks, as it proves that block meets the set requirements. An IP block cannot be taken to new SoC designs if it is not properly verified.

During this thesis project a new UVM based COP verification environment was created and it is introduced in this thesis. The environment has lots of verification IP's (VIP) and some of them are third party VIP's. Many of the parts are developed for this particular verification environment, such as: COP C++ reference model, COP instruction generator, Auxiliary unit VIP and COP assertion module.

COP verification environment is created and lots of test cases are done. The verification is not completely done yet, though. More features must be covered to achieve verification closure.

Most important result of developing a new verification environment for COP is that Nokia Co-processor is adopted to be used in new SoC designs and COP related development and research projects can be initiated.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Signaalinkäsittelyn ja tietoliikennetekniikan koulutusohjelma
MIIKA IHANAJÄRVI: RISC-prosessorin UVM-varmennusympäristö
Diplomityö, 63 sivua
Toukokuu 2016
Pääaine: Tietoliikenteen digitaalitekniikka
Tarkastaja: prof. Mikko Valkama
Avainsanat: prosessorit, verifiointi

Tämä diplomityö esittelee Nokia Co-prosessorin (COP) ja sille tehdyn UVM-pohjaisen varmennusympäristön. COP skaalautuu helposti pienestä yhden säikeen 32 bittisestä prosessorista 128 bittiseen, 16 säikeiseen prosessoriin. Pelkkä COP soveltuu erittäin hyvin esimerkiksi datasiirtimeksi, mutta COP:in saa muokattua helposti monenlaiseen käyttöön lisäämällä kiihdytinyksikköjä.

COP on peräisin Nokian edellisestä järjestelmäpiiriorganisaatiosta ja sitä on päivitetty vastaamaan nykyisten järjestelmäpiirien tarpeita. COP:in vanha varmennusymäristö ei toimi muutosten takia ja vaatisi suuria muutoksia. Tästä syystä oli selvä tarve uudelle kunnolliselle varmennusympäristölle.

Varmennusympäristö on erittäin tärkeä IP-lohkoille, koska varmennuksella pyritään todistamaan, että lohko vastaa sille annettuja vaatimuksia. IP-lohkoa ei voida ottaa käyttöön uusiin järjestelmäpiireihin, jos varmennusta ei ole suoritettu.

COP:ille kehitettiin uusi varmennusympäristö tämän diplomityöprojektin aikana ja diplomityö esittelee kyseistä varmennusympäristöä. Ympäristö sisältää runsaasti kolmannen osapuolen varmennus-IP:itä. Monet muista tämän varmennusympäristön osista on tehty tätä ympäristöä varten. Niistä esimerkkinä COP C++ referenssimalli, COP käskygeneraattori, kiihdytin VIP ja COP assertiomoduuli.

COP ei ole vielä täysin varmennettu vaikka COP varmennusympäristö ja paljon testejä on jo tehtynä. Uusia testejä pitää vielä tehdä, ennen kuin testien kattavuus on riittävällä tasolla, että varmennus voidaan todeta valmiiksi.

Varmennusympäristön tekemisen tärkein tulos on, että Nokia Co-prosessori on voitu ottaa mukaan uusien järjestelmäpiirien kehitykseen ja COP:iin liittyvä kehitystyö on mahdollista.

PREFACE

In the spring of 2014, my studies in Tampere University of Technology was approached the point where Master's thesis was needed to be done. I heard from Professor Mikko Valkama that Broadcom was looking for thesis workers, so I sent my application there.

I received an announcement email during my first working day at Broadcom that the company is going to sell the businesses related to its research and development activities in Finland. I was supposed to make my Master's thesis about ARM AMBA bus bandwidth and latency measurements of a system on chip (SoC) design, but I was forced to find other options due to ramp-down of the site.

I heard rumors that the Tampere site of Nokia Solutions and Networks (NSN) was hiring some Broadcom employees who were getting laid off from Broadcom's Tampere site. I sent an application email to NSN and I was hired as a design engineer, not solely as a thesis worker.

My work at Nokia was initially top level verification of some SoC project. At some point, I started to make verification for a COP in SoC top level verification environment. It was natural to start making master's thesis about verification suite for the COP after the SoC top level verification was completed.

I would like to thank Professor Mikko Valkama for examining this thesis and giving guidance and support. Thank you Anssi Örn for suggesting me this great project and for supporting through. I would like to thank Jari Heikkinen for support and suggestions how to improve this thesis. I would like to thank Samuli Rahkonen who did great job by helping me set the test environment up and creating test cases. Thanks for Mikko Kemppinen, Toni Tarhonen, Janne Mäkitalo, Sami Ahlberg, Janne Helkala and other colleagues for support during the thesis project.

I would like to thank my lovely wife Rea for love, patience and support. Thank you my children Daavid, Reetta, Minttu and Kerttu for giving so much happiness and great moments to my life.

Tampere, 24.5.2016

MIIKA IHANAJÄRVI

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

ADC	Analog-to-Digital Converter
ALU	Arithmetic-Logical Unit
AMBA	ARM Advanced Microcontroller Bus Architecture
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Procesesor
CAD	Computer-Aided Design
CISC	Complex-Instruction Set Computer/Computing
COP	Nokia Co-Processor
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DPI	Direct Programming Interface
DPI-C	Direct Programming Interface for C language
DSP	Digital Signal Processor
DUT	Design Under Test
DUV	Design Under Verification
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite-State Machine
I/O	Input/Output
IC	Integrated Circuit
IP	Intellectual Property
NRE	Non-Recurring Engineering
OVM	Open Verification Methodology

PCB	Printed Circuit Board
RTL	Register-Transfer Level
SoC	System On Chip
TB	Test Bench
VIP	Verification Intellectual Property
XML	eXtensible Markup Language

1. INTRODUCTION

This Master's thesis is done for Nokia Networks in Nokia's Tampere site. Nokia started again, after many years a system on chip (SoC) organization which started designing new SoC designs basically from scratch. There was some legacy components from the old SoC organization but those were outdated in many cases.

There was Nokia Co-processor (COP) amongst the legacy components. To accommodate to the needs of today's SoC architectures, for example bus interfaces of the COP were changed to ARM Advanced Microcontroller Bus Architecture (AMBA) compatible interfaces.

A legacy COP verification suite was outdated as well. It was not possible to compile the verification suite with latest versions of register-transfer level (RTL) simulation tools even though many makes and versions was tried out. The changes in the bus interfaces would have also impacted major changes to the legacy COP verification suite.

A problem is that there is no readily available solution for verification of the updated COP version. Some of the functionality can be verified with an old simulator version and other functionality in integration verification at SoC top level. It is evident, that it can be only a temporary solution which does not work in long term.

COP needs a verification environment which can verify all the functionality of the processor. The verification environment shall support further development of new features to the COP, development of COP peripherals as well as other COP related activities such as software development and high-level programming language compiler development.

In this thesis, a general idea of SoC designs and justification for using SoC concept is provided in Chapter 2.

About Verification concepts, methods and reasoning why verification is needed is discussed in Chapter 3.

Reduced Instruction Set Computing (RISC) processors are introduced in a Chapter 4. The chapter is also introducing the Nokia Co-processor.

Universal Verification Methodology (UVM) test environment creation for COP processor is explained in Chapter 5.

Results and further development and improvement ideas are introduced in Chapters 6 and 7.

2. SYSTEM ON CHIP

A system on chip (SoC) is an integrated circuit (IC) which contains an entire electronic system such as a computer or a more specialized device. Typically SoC designs include central processing unit (CPU), memory, hardware accelerators, inputs/outputs (I/O), and other components on a single chip (see Figure 2.1). In addition to digital components, it is possible to have on-chip mixed-signal components such as analog-to-digital (ADC) and digital-to-analog (DAC) converters and phase-locked-loops (PLL). [3]

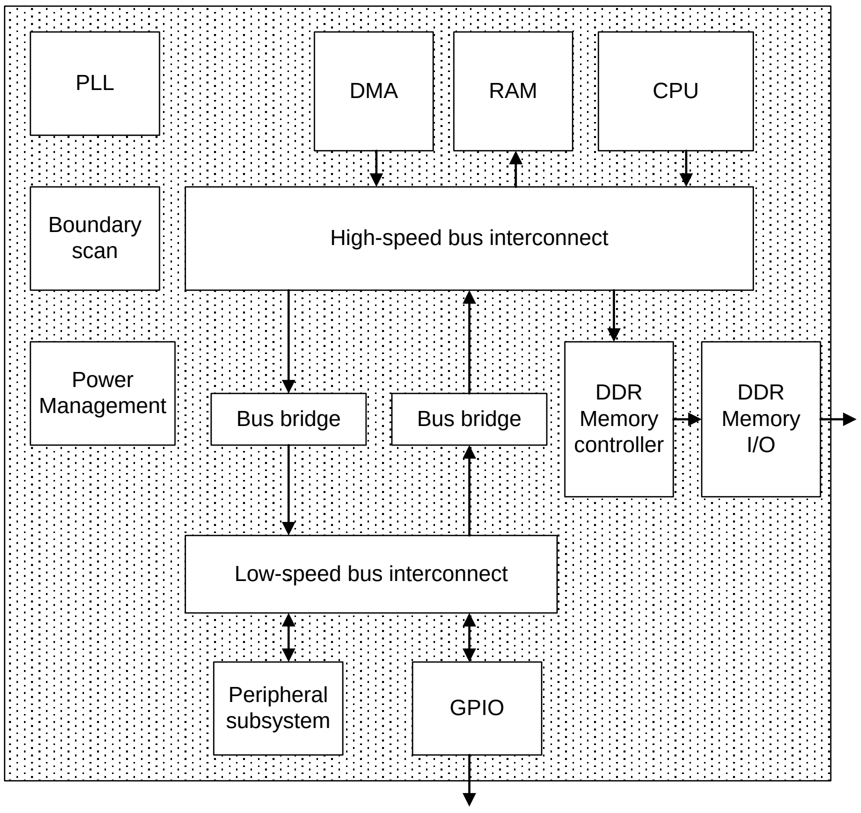


Figure 2.1: Typical SoC block diagram

There are clear benefits of fitting large system inside one chip. For example, price of end product is reduced as there is no multiple chips to be sourced, assembled, and their supporting circuitry such as clocking, power, filtering is not needed. Printed circuit board (PCB) for the design can be also smaller.

Other benefits come from close proximities of each component. The inter-component latencies are smaller and clock frequencies can be higher. Higher clocking frequency improves performance of the system.

SoC's can be made either on Field-Programmable Gate Arrays (FPGA) or on Application Specific Integrated Circuits (ASIC). FPGA is a re-programmable chip whereas ASIC cannot be modified after it is fabricated. In order to make changes, such as bug fixes, to ASIC hardware, a new design spin must be done. One ASIC design spin costs a lot of time and money.

FPGA supports quicker time-to-market but there are limitations with maximum clock frequencies, power consumption and price. For example, it is not possible to make competitive mobile phones with FPGA SoC's due to all of these limitations. ASIC's must be used in that kind of most demanding applications. Comparison between ASICs and FPGAs can be found in Section 2.3.

2.1 SoC Technology Evolution

Silicon foundries keep on shrinking the physical dimensions of silicon structures that can be realized on an IC. The circuit capacity and performance are increased by the shrinkage. Moore's Law states that the number of logic gates which can be integrated in single chip doubles in every 18 months. As more logic gates are added to the devices, the productivity related issues are also present. Increasing demand for productivity affects both design and verification. [3]

The tendency that productivity increase of design engineers can not keep up with the speed of IC evolution, is called design productivity gap (Figure 2.2). IC technologies evolve to smaller logic gates which permits more and more gates into a single chip. New methodologies are needed to help the designers.

Similar productivity gap is between design complexity and verification. Methodology improvements and re-using are keys to get the gap smaller.

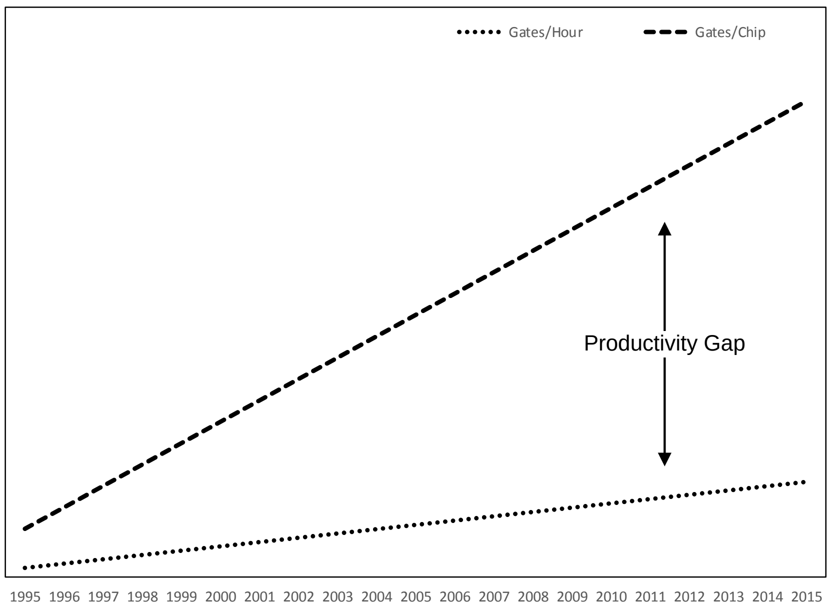


Figure 2.2: Design Productivity Gap

Adding more and more engineers is not a viable option in long term. The design teams can not be expanded too large. At some point the overhead of the needed coordination between each designer approaches the point where design team's head count increase does not increase productivity.

Hierarchical design helps to fight the overhead of large design teams. The design is split into small, possibly re-usable blocks and each block has a designated design engineer who is responsible for the block. The hierarchical design (for example in Figure 2.3) consist of layers within the design. Top level is for connecting subsystems and blocks. Subsystems have connections of blocks within the subsystem. When looking the design in one level, the visibility is restricted. With hierarchical design flow, the whole SoC design can be partitioned into meaningful functional subsystems which gives good abstraction.

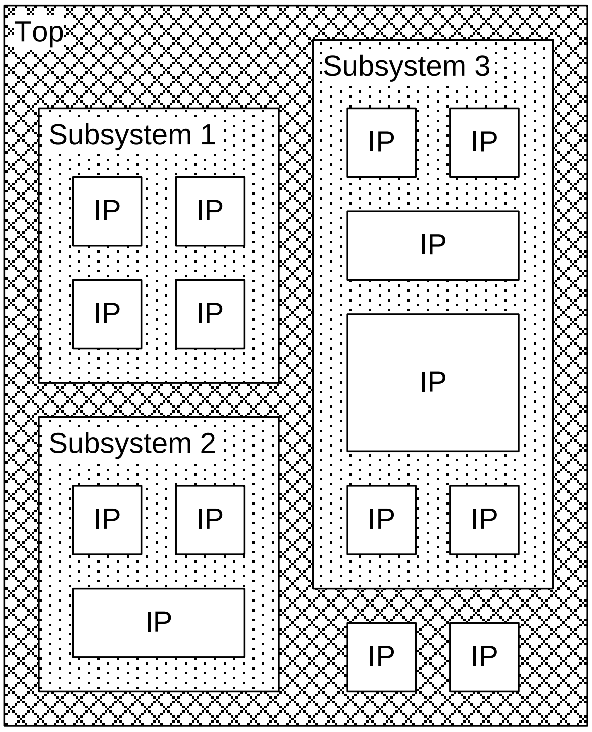


Figure 2.3: Hierarchical design example

Design reuse integrates pre-existing blocks with newly created blocks. This aides the development in two ways. First, since one or more of the blocks have been pre-designed, the amount of original design work is reduced. Secondly, since pre-designed blocks have been pre-certified or validated, they can be viewed as black boxes and need not be revalidated. [3]

Difficult challenges related to the shrinkage of the silicon structures are related to timing closure, capacity and physical properties [3].The timing closure means the process where the design is changed to meet the timing requirements. Timing requirements are tighter when the clock speeds are higher.

As it is possible to integrate tens and hundreds of millions of gates onto a single IC, it introduces significant capacity challenges to many of the tools in the design flow. To manage this level of complexity, the design systems must adopt hierarchical design and design reuse as solutions. [3]

2.2 SoC Development

SoC development cycle (Figure 2.4) starts by collecting customer requirements. The customer can be in the same company or external. General specifications and architecture documentation can be created after the requirements are collected. The requirements should not restrict the implementation too tightly into one specific way of implementation. It is up to the architect which kind architecture the design will have.

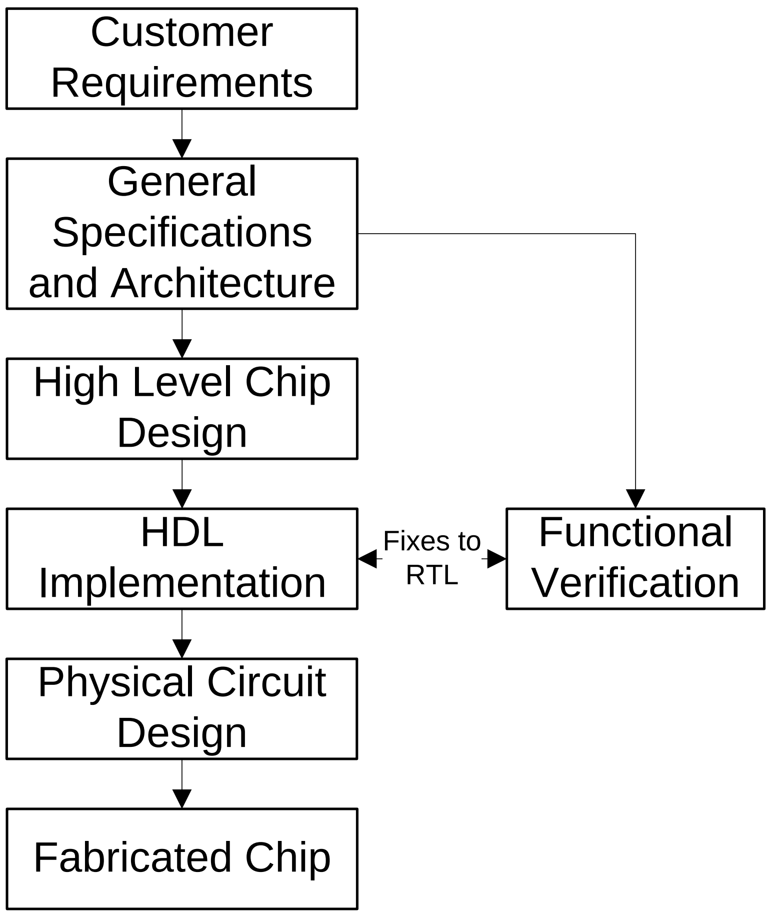


Figure 2.4: Chip Design Process

Components on the SoCs are called IP blocks. IP blocks are developed separately and allow re-usability. General purpose IP blocks can be re-used in more specialized IP blocks, subsystems and design top level. When the same IP block is used in multiple projects, there is no extra verification effort needed.

HDL implementation includes IP block design and verification as well as top level integration and top level verification. There might be additional layers of hierarchies or subsystems which are integrated and verified separately before top-level work.

Functional verification gives feedback about quality and maturity of an IP to the design engineer. When the verification is ready and design is ready, the flow proceeds to physical circuit design. It includes placing of the components, their connections and so on. Then SoC is ready for fabrication. There is also verification involved in physical circuit design and testing in prototyping phase of fabricated chip.

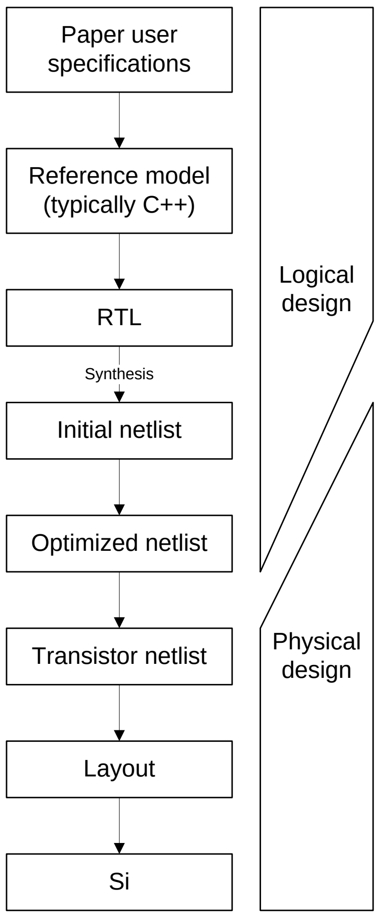


Figure 2.5: Abstraction layers of the design flow [1]

As shown in the Figure 2.5, there are two aspects in the design flow: logical design and physical design. The design flow starts with logical design which describes how the design shall work. The design is entered into a SoC design system. Outcome of this phase is a design entry.

Second phase in logical design is a logic synthesis where hardware description language and synthesis tools produce a netlist. The netlist is a description about logic cells and their connections. This is where physical design starts to come into play. It is still related to logical design as well. System partitioning is needed if the design is so large that it has to be split on more chips than one.[4]

To verify that the timing requirements are met, a physical layout, netlist has to be created. The delays are calculated from the layout. Depending on the distances of IP the bocks the lengths of their connections vary. As the reference clock frequencies get higher, the timing requirements get more difficult to achieve. If the requirements are not met on the first time, more optimizations need to be done on either layout or in IP blocks.

Tape-out is the final result of the design cycle, a spin. The tape-out means that the layout of the SoC design is sent to IC manufacturer.

Then there is a lead time the IC manufacturer needs for producing sample chips after the tape-out. First, a batch of samples are created for testing of the SoC and for prototyping. If there is no fatal bugs found during testing and prototyping in the SoC design, no re-spin of design cycle is needed and the SoC is good to go for higher volume production.

2.3 SoC Economics

In addition to technology limitations of FPGAs, there is also economical aspect which supports ASICs (in certain cases) for SoC usage. FPGAs are relatively cheap to design (Non-Recurring Engineering (NRE)) and the time-to-market is quicker. A drawback is a high part cost. In high volume production, the part cost is where most of the savings can be made, because once happening design cost is divided with large number of parts.

A Figure 2.6 shows that it is possible to find a number of parts when FPGA becomes as costly as ASIC, and ASIC is cheaper after that number. The number is called break-even volume. Depending of the intended volume of a product is can be estimated whether the design should be made on an FPGA or an ASIC.

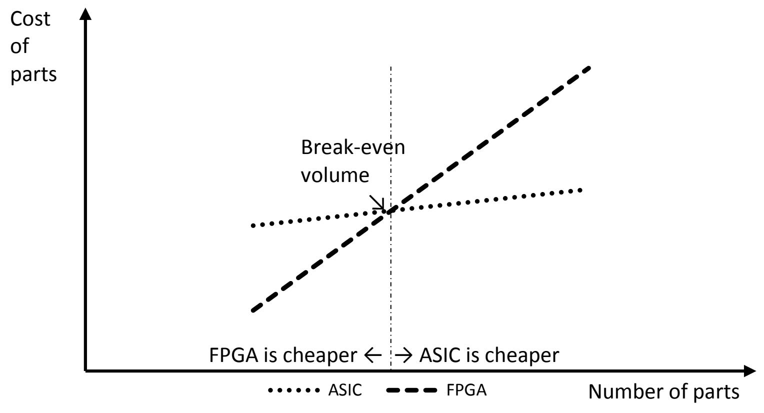


Figure 2.6: Break-even volume

Costs can be divided in two categories. Fixed and variable costs. Fixed costs are there even with one produced SoC unit so fixed costs are initial values in an example Figure 2.6 on both lines. Variable costs are depending on number of sold parts. There is also third category of indirect design costs. They are lead time and time-to-market.

Fixed part costs include of non-productive work such as trainings of design personnel for usage of the design tools, programming languages, methodologies and design flows as well as SoC project related productive work. There is also hardware and software (EDA system) costs as fixed cost. Cost of salaries depend on the number of engineers and their productivity.

Variable part costs are proportional to the number of the sold parts.

Cost of time-to-market can be seen as loss of profit during the time before the product is available in market. Competitors might have sold their alternative products to possible customers. Those customers may not buy other similar product from different producer again in the near future. The effect of weak time-to-market is shown in Figure 2.7. Loss of being late in market is the area between the lines of weak and strong time-to-market. Total sales of late arriving products are expected to be lesser compared to a product which is available in timely manner. [5]

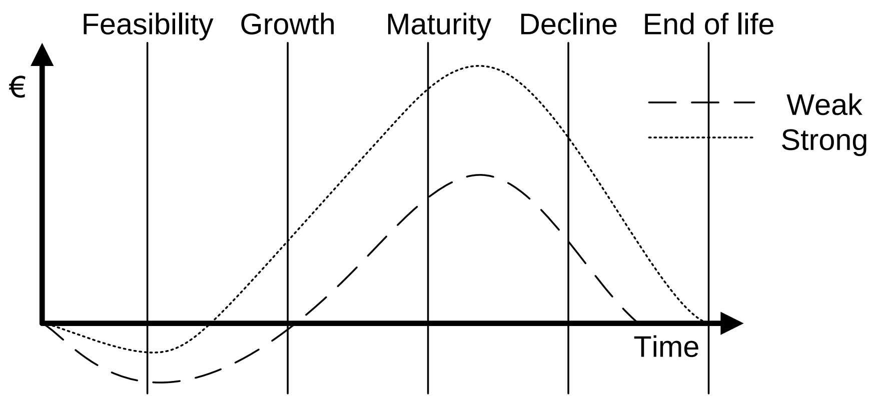


Figure 2.7: Time-to-market

Effect to the cost of lead time is similar as with time-to-market and it contributes as part of time-to-market. Lead time means the time needed for an ASIC factory to produce chip. Lead time is a latency from the time when the SoC design is given to the ASIC factory to the time when the fabricated chips arrive from the factory. Normal lead time is 4 to 8 weeks. FPGAs does not have similar lead time as ASICs have.

The technology selection is a trade-off between many factors. The optimum selection is not always easy to find. Table 2.1 shows where different technologies are good at.

	FPGA	**ASIC**
Tailored Masks		15 or more
Area		Best (smallest)
Speed		Best (fastest)
Power		Best (minimal)
NRE Cost	Best (smallest)	
Per-part Cost		Best (smallest)
Design cost	Best (easiest)	
Time-to-market	Best (shortest)	

Table 2.1: Summary of SoC technologies [2]

In general, FPGAs are better for prototypes and low volume products as there is low NRE, large part cost and small lead time. ASICs are better for large volume products because of high NRE, small product cost and large lead time. [2]

3. VERIFICATION

3.1 Introduction

The ultimate reason for verification is to ensure that the design meets the functional requirements as specified in functional specifications [3].

SoC development cycle (Figure 2.4) starts by collecting customer requirements. It does not matter whether the customer is in the same company or is it external. The general specifications and architecture can be created after the requirements are collected. Verification flow (Figure 3.1) can start as soon as the specification and architecture documentation is available.

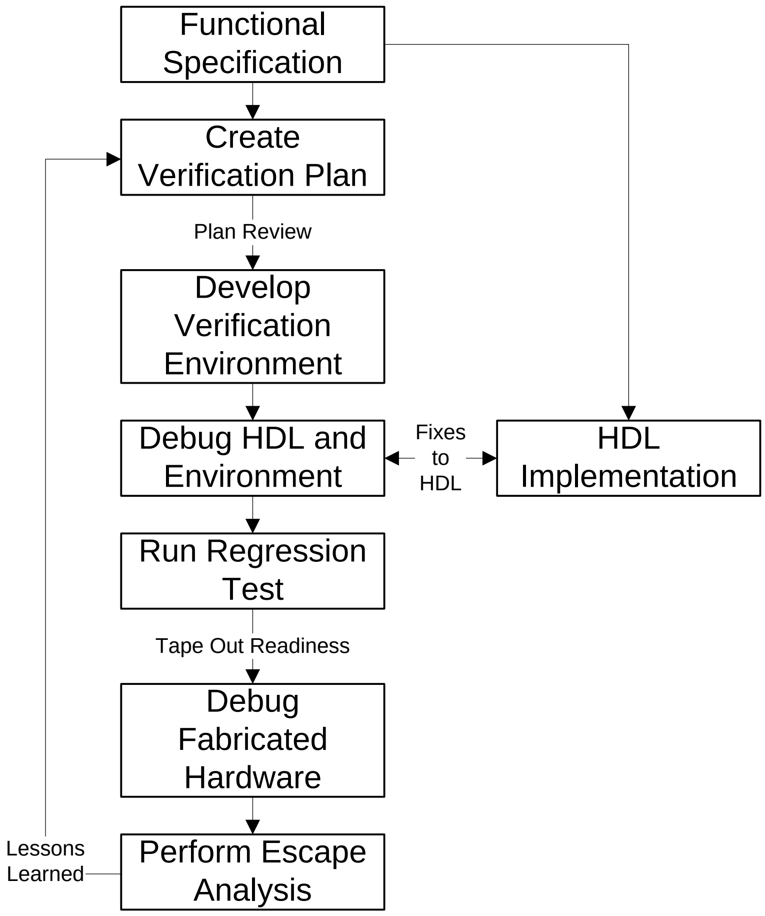


Figure 3.1: Verification Cycle

According to Rashinkar et al [3], the verification of a SoC devices takes usually 40 to 70 percent of total development effort for the design. In time-to-market, the verification plays a very significant role.

Verification work must be well planned and organized. Otherwise it is impossible to say and prove that the verification is ready. Features to be verified shall be stated clearly and completeness of the verification needs to be tracked as well as implementation errors of the design. If either one of two, verification or design, is not ready, the design can not be considered as ready for the tape-out.

The state space of the SoC designs is so huge today that exhaustive verification of everything is not possible. Formal verification of whole design is not possible. Regardless of these truths, the designs need to tape-out and there must be some criteria to be used to determine when the verification is done. [1]

As in Albin's paper [1], some of these criteria can be for example:

- 40 billion random cycles without finding a bug
- directed tests in verification plan completed
- source and/or functional coverage goals met
- diminishing bug rate
- a certain date on the calendar reached

The hardware design of a SoC can't be altered after tape-out. If there is a bug, the system may not boot or the usage can be somehow otherwise limited. Each spin you have to make changes to the design and having a new tape-out is going to cost a lot. Naturally the wasted time during re-spin impacts to time-to-market as well.

In order to make a functional device, all the parts which are intended to be used must work correctly. How can we ensure everything works? It is easy to answer. Everything has to be verified before tape-out. It depends on the application how do we define "the everything" and what exclusions do we allow. The scope of the verification must be set clearly to verification strategy and plan documents.

The needed amount of work and money is reduced when the design bugs are found early. Sooner the better. Best is that all the design bugs are found in IP verification. IP verification must have 100% coverage. When verification closure for an IP is done, the IP is proven bug free. Integration and system level verification are done on subsystem or top level verification.

IP designers can perform lint (static checking for source code) checking and formal checking of the block before making releases to revision control system. That ensures some quality of the design before the verification. Top level integration can benefit from lint checking and formal checking.

A formal verification is a great way to verify that the design works with any stimuli as intended in small designs. With many programmable devices, such as processors, formal verification is too complex as the functionality depends on the program code run on the processor. More information about formal verification in Section 3.2.

According to Albin [1] there are two methods to manage increasing complexity: divide-and-conquer and abstraction. A very common way is to use divide-and-conquer method by utilizing block level verification. Each IP block has a standalone test environment where the functional verification is easy to do.

Computer-Aided Design (CAD) tools allow translation from RTL down to mask data to be checked routinely and reliably. That's one of the reasons why currently most of the integration and verification work is done in a very low abstraction level (RTL, see Figure 2.5).

There are methods to check verification environment quality. One way is to use fault injection to check if the test bench is able to find the generated bugs. For example Synopsys Certitude can be used for verification qualification. [6]

3.2 Formal Verification

Formal verification can prove, in a mathematical sense, that two representations are equivalent. Formal verification software can tell how the representations differ if they are not equivalent. This can be used when translating from behavioral model to structural model of a design.[4]

Formal verification methods do not require test benches for verification and they promise theoretically fast verification time and 100 percent coverage. The flavors of formal verification are Theorem Proving Technique, Formal Model Checking, Formal Equivalence Checking.[3]

The theorem proving technique shows that the design meets the functional requirements by allowing user to create proof of the design behavior using theorems. [3]

Formal model checking verifies behavioral properties of the design. A model checking tool compares the design behavior to a set of logical properties defined by the user. The properties defined are directly extracted from the design specifications. Formal is good for verifying complex control structures, such as bus arbiters, decoders, processor-to-peripheral bridges and so on.[3]

Formal model checking does not require any test benches or vectors. When tool finds an error, it generates a complete trace from initial state to the state where the specified property failed. [3]

Formal equivalence checking is a method to prove equivalence of two different views of the same logic design. It uses mathematical techniques to verify equivalence of a reference design and a modified design. These tools can verify equivalence of RTL-to-RTL, RTL-to-Gate and Gate-to-Gate implementations. Since equivalence checking tools compare the target design against the reference design, it is critical that the reference design is functionally correct. [3]

Any of the previously mentioned formal verification methods does not take timing into account. Timing verification must be done in separate timing analysis tool. Formal model checking uses a lot of computing resources, so usage of it may be limited to small designs only. Another potential problem is that some of the design issues and bugs might not be detected if the design specifications and other definitions are not set correctly to the verification tool.[3]

3.3 Dynamic Verification

Dynamic verification is a verification method where directed or random stimuli is given to the design under verification. Usually dynamic verification is done by using RTL simulator.

Dynamic verification is available in two flavors: randomized and directed tests. Directed tests use “static” stimulus. The result is the same on every simulation run. Directed testing is needed for example for verifying IP block’s configuration sequences. The configuration values are dictated by the verification engineer who made the test case.

Random tests use randomized stimuli. In many cases there are some limits for values which are used as stimulus. The values need to be constrained somehow. This is where a new term “Constrained random” comes into play. With constraints the random values can be generated in such a way that they follow the rules defined by the verification engineer.

In both, randomized and directed, tests the design under verification is connected to a test bench. The test bench provides the necessary signals such as clock, reset and other signals to the design under verification (DUV). Simplified block diagram of the DUV and test bench relation is captured in Figure 3.2.

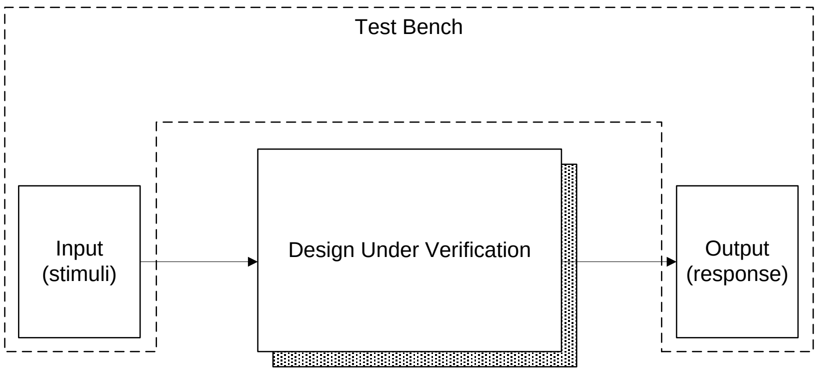


Figure 3.2: Test bench connected to the design under verification

It is good to use a test bench which apply stimuli (input), capture response (output) and compare the result with a golden reference which is known to give valid response for given stimuli (as in Figure 3.3). In directed tests the stimuli is always the same and the golden reference response is always the same. In randomized tests the stimuli is generated by the test bench.

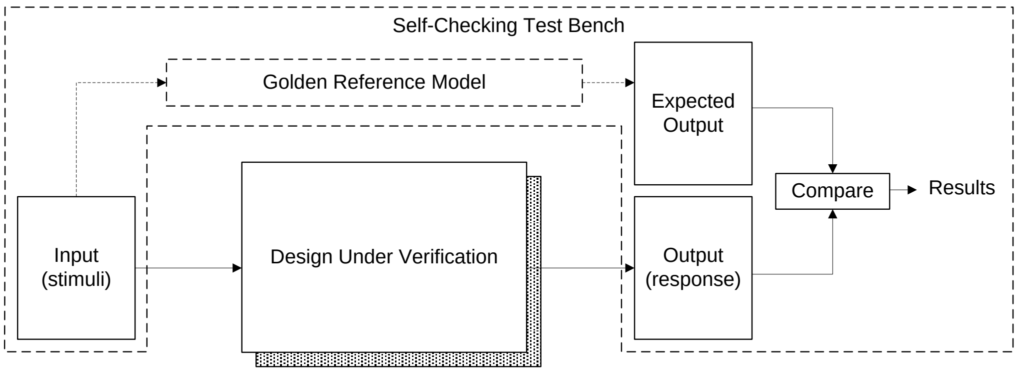


Figure 3.3: Self-checking test bench connected to DUV

When random stimuli is used, the output values are random as well and thus cannot be known beforehand. Some kind of a reference model is needed for generating valid reference output values. The reference model is considered to be perfect, that is why it is called Golden Reference Model.

Depending the DUV, there may be configuration registers which must be configured before using the DUV. The configuration can be made by constraining the randomness of the configuration values. There may be some configuration values which must be inside a given range. In that case, constrained random input can be used instead of dictated values.

3.4 Verification Qualification

Qualication is needed, because it is not usually possible to completely verify designs using simulations. Therefore, the correctness of the design can be assumed if the testbench can be trusted. Different coverage metrics and mutation analysis methods are applied to measuring if enough verification has been done. [6]

There are number of methods to measure the quality of verification environment:

- Number of passed tests in verification plan
- Code coverage
- Bug rate
- Functional coverage
- Test review
- Manual bug injection

None of the previous are actually objective measures of verification environment's quality. Number of passed tests does not address the question of checker's correctness. Code coverage shows the activated source code lines of DUV, but it does not tell whether action in the line propagated to any of the DUV's outputs. Functional coverage is very similar, but it emphasizes checking the executions of important functionalities specified in the verification plans. Test reviews are time consuming and subjective measures of quality.[6]

Test bench's quality can be checked by injecting bugs manually to the design and checking that test bench can detect them. This kind of manual work does not cover all meaningful cases in sensible time.[6]

Mutation-based qualification is quite a new approach to validate verification viroment's checkers. It provides tools for automatical bug injection and verification environment qualification.

In mutation-based qualification tool generate artificial bugs (mutations) to the design. Tool replaces, injects or removes expressions and variables from the code or design. Before tests are run with the design, their correctness has to be checked by running them with unaltered design. If tests fail, they have to be fixed before continuing with mutation analysis.[6]

New coverage metric is the percentage of detected injected bugs after applying the method for all tests. If all mutations are not detected, the verification engineer should consider changing the existing test cases or adding new ones.[6]

The mutation-based qualification seems to be good addition to traditional verification environment quality measurements as it gives feedback on potential flaws of checkers as soon as first test had been entered into verification system.

3.5 SystemVerilog

SystemVerilog is a combination of Verilog (IEEE Std 1364-2005 Verilog Hardware Description Language) programming language and SystemVerilog extensions (IEEE Std 1800-2005 SystemVerilog Unified Hardware Design, Specification, and Verification Language) to the Verilog. The SystemVerilog was defined in two separate IEEE standards which were merged in 2009 to one standard (IEEE Std 1800-2009 SystemVerilog - Unified Hardware Design, Specification, and Verification Language). The reason for merging two standards was to provide one information source for users about syntax and semantics of the SystemVerilog. [7]

SystemVerilog includes support for modelling hardware at behavioral, register transfer level (RTL) and gate-level abstraction levels. SystemVerilog has also support for writing test benches using object-oriented programming.

SystemVerilog Direct Programming Interface (DPI) is an interface which can be used to interface SystemVerilog with foreign programming language. DPI-C is an interface for C language. DPI-C example can be found from Subchapter 5.4.1.

3.6 Universal Verification Methodology

Universal Verification Methodology was developed because the verification engineers needed unified verification framework to enable efficient development and reuse of verification environments and verification IPs (VIP). Simulation tool vendors had their own methodologies which were not directly compatible with each other's.

A standards organization for electronic design automation (EDA) and IC manufacturing, Accellera Systems Initiative, decided to establish the UVM in 2009. Open

Verification Methodology from Cadence and Mentor was chosen as a base where the UVM is built on. UVM is created in co-operation between several companies in verification industry. Nowadays Mentor Graphics, Synopsys, Cadence and Aldec are co-operating in UVM development.

UVM is built on Open Verification Methodology (OVM) which is combination of Advanced Verification Methodology (AVM) and Universal Reuse Methodology (URM). UVM has also some concepts of **e** Reuse Methodology (ERM) and some concepts and code of Verification Methodology Manual (VMM). [8]

UVM family tree is shown in Figure 3.4. It can be seen that UVM consists of parts or ideas from many verification frameworks.

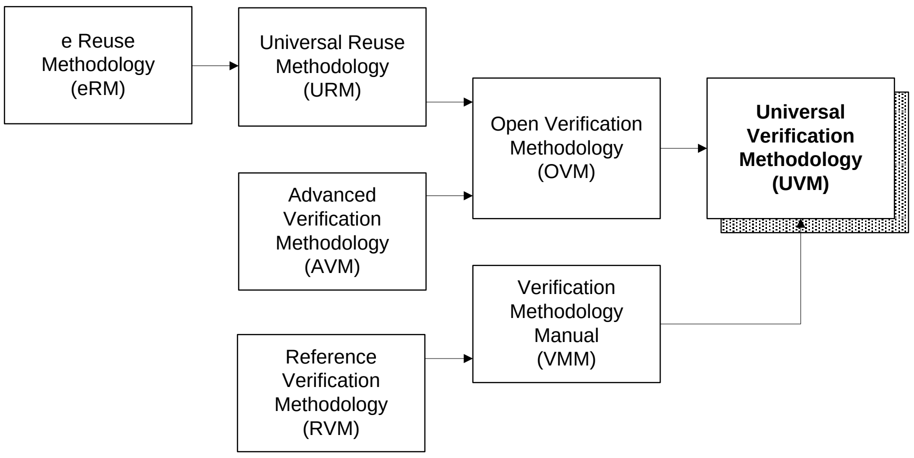

e Reuse Methodology (eRM)
Universal Reuse Methodology (URM)
Open Verification Methodology (OVM)
Universal Verification Methodology (UVM)
Advanced Verification Methodology (AVM)
Verification Methodology Manual (VMM)
Reference Verification Methodology (RVM)

Figure 3.4: Family tree of the Universal Verification Methodology

Universal Verification Methodology is built using of SystemVerilog programming language. It promotes reuse by providing collection of SystemVerilog classes which can be used as a common base for every test bench. The base classes are extended with needed methods. Some of UVM provided base classes and their relations are shown in Figure 3.5.

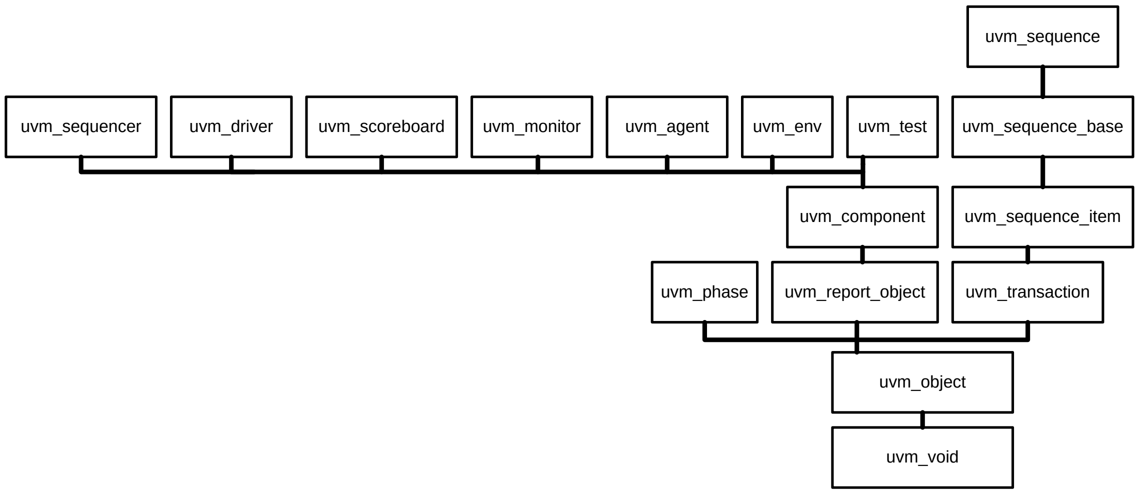


Figure 3.5: UVM class relationships

UVM provides base for environment configuration, phasing, sequencers and sequence items to mention few. As UVM test benches are composed from reusable verification IPs, it is also possible to use third party VIP's in the test bench to speed up the verification test bench development. The most important benefit for UVM is that IP verification environment (if designed wisely) can be included and re-used in subsystem or SoC top level.

3.6.1 UVM Agent

Figure 3.6 shows an UVM Agent (or VIP) which is configured as active. It means, the agent has driver fuctionality. The driver receives transaction items from sequencer and translates them to pin-level activity.

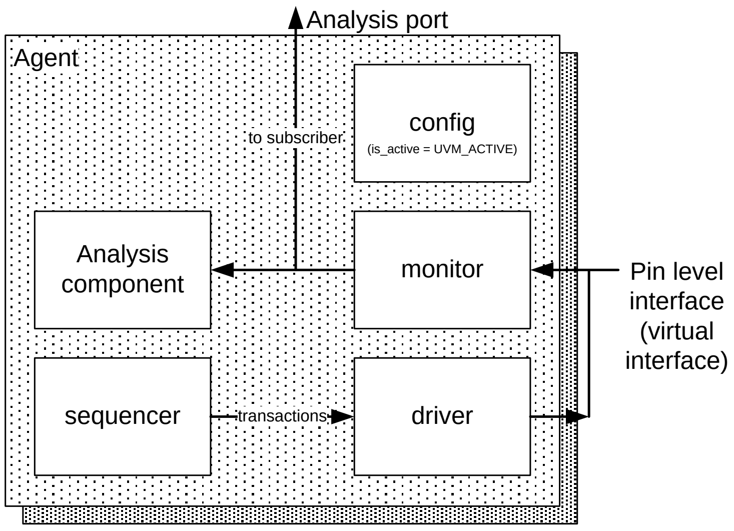


Figure 3.6: UVM Agent configured as active

Passive agent is only for monitoring and for creating transaction items from pin level activity. Analysis ports are used to perform non-blocking broadcasts of connected components' transactions. Other components such as scoreboard or coverage monitor can be set as subscribers for analysis ports.

The agents sequencer can be controlled from outside with virtual sequencer. One virtual sequencer can handle complex sequencing with multiple active agents.

3.6.2 Example UVM Environment

In the following example, there is DUV with two separate interfaces (Figure 3.7). One verification IP (agent) is instantiated for each interface. Predictor is a reference model of DUV behavior. Stimulus is given with virtual sequencer as transaction items to the agents. Agents communicate with DUV via their interfaces.

Agents monitor and translate the activity on interfaces to transaction items. The items are then transmitted to scoreboard and coverage monitor as needed. The items are compared to predicted and results analyzed.

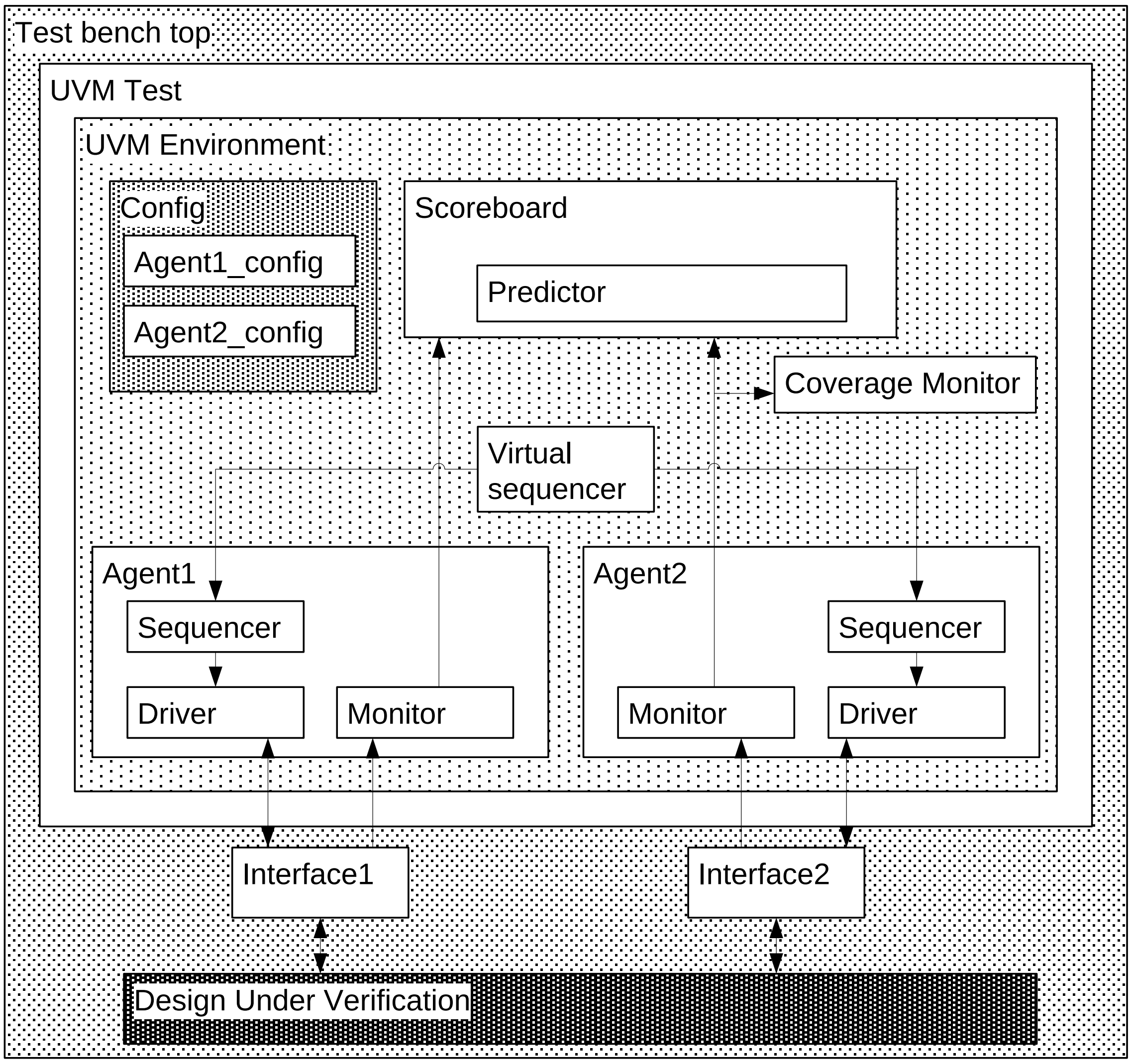


Figure 3.7: UVM verification environment example

The design under verification and its SystemVerilog interfaces are instantiated in UVM verification environment example's Test bench top SystemVerilog module. The interfaces are assigned to appropriate ports of DUV and they are added to uvm_configuration_db. After that, UVM Test class is called.

UVM Test configures and creates the UVM Environment. UVM Environment configures the agents according the configuration given by UVM Test. The SystemVerilog interfaces assigned to DUV are given to agent configuration. The UVM Environment class configures Scoreboard and Predictor classes. Scoreboard is configured as subscriber for agents analysis ports. New test cases can be added by extending test base class.

It is possible to create hierarchical environments where there is several environments within other environments. For example, at SoC top level verification, the UVM environment contains all subsystem verification environment classes, and subsystem environments contain IP level environments. To promote re-usability, the lower level environments should provide convinience methods to ease the configuration in higher level environment.

3.6.3 UVM Register Abstraction Layer

UVM Register Abstraction Layer (RAL) can be generated from IP-XACT model of the register or memory layout of the design under verification (DUV). UVM RAL can be used with bus master agents which transforms RAL accesses to bus transactions. The UVM RAL enables usage of UVM built in register tests such as bit bash test and reset test. The reset test checks the default values of the registers. Bit bash test toggles all bits of all registers.

IP-XACT standard describes an eXtensible Markup Language (XML) schema for documenting intellectual property (IP) used in development, implementation, and verification of electronic systems. The schema provides a standard method to document IP that is compatible with automated integrating techniques and a standard method (generator) for linking tools into a system development framework. The standard is independent of any specific design processes. It does not cover behavioral characteristics of the IP that are not relevant to integration.[9]

3.7 Processor Verification

Processor is a central component in SoCs. Usually there is either control CPU, digital signal processor (DSP), application specific instruction set procesesor (ASIP), co-processor or set of many processors.

The design and verification effort of a custom CPU core is significant and its benefits over an off-shelf CPU core must be justified [10]. Typically the costs associated to the verification of a processor are very high [11], and costs of failed verification and re-spin even higher.

For example in 1994, Doctor Thomas R. Nicely discovered a bug in Intel's Pentium processor's floating point unit (FPU) during mathematical research. Nowadays, the bug is known as Intel FDIV bug and it affected wrong decimal results during complex calculations. In response of the discovery, Intel announced recall of the flawed processors. In 1995, Intel announced "a pre-tax charge of 475 million dollars against earnings, ostensibly the total cost associated with replacement of the flawed processors".[12]

The requirement to support legacy software and/or a high level programming language is a driving factor when selecting a CPU core for many applications. This requirement typically limits designer flexibility to a set of binary compatible cores with available software tools.[10]

The processor verification can be done in theory using formal or simulation-based methodologies. As formal methodology is good forsmall blocks[11], it can be used for verification of arithmetic-logical unit and other sub-blocks. However, formal cannot be used for complete processor verification, as formal methods does not take timing into account. At some point of processor verification, simulation based verification must be taken into use.

A processor presents a unique verification challenge, particularly one with pipeline, jumps, branches, multi-cycle instructions, and hazards[13], [10]. The chosen verification methodology needs to:

1. have quick ramp up time
2. randomly generate instructions
3. steer randomly generated instructions into interesting corner cases
4. use functional coverage to stop the test bench once functional coverage is obtained[10]

A simplified processor verification environment example is shown in Figure 3.8. The Figure is generalized version of a figure in Becvar's and Tumbush's paper [10] which presents verification environment of CPU core and hardware accelerator unit for image processing application.

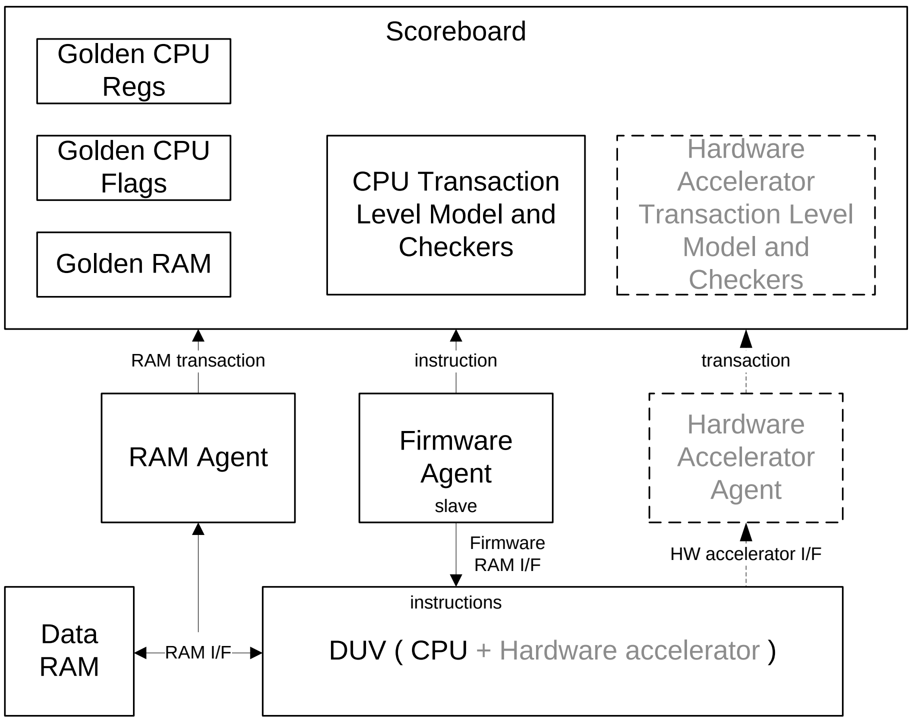


Figure 3.8: Processor verification environment

If the hardware accelerator is omitted, the needed verification components for simulator-based verification are:

1. Agents
    (a) RAM agent
    (b) Firmware agent
2. Scoreboard
    (a) Transaction models
    (b) Predictors
    (c) Checkers
3. Environment

RAM agent monitor RAM interface and provide transaction items of each RAM access to the Scoreboard. The Firmware agent is used for generating instructions to the DUV. The generated instuctions are also given to the Scoreboard.

To create a golden model or predictor requires a designer's level of knowledge about the DUV and can take very long term to create. Only model as low level as absolutely necessary. [10]

For processor verification, a random program generator is needed. The random generated program can be written to instruction memory before execution of the program or instructions can be generated during the simulation and by injecting the instructions to instruction bus. Kamath [11] suggests following programming rules to follow:

1. The generator must produce existing instructions with valid (legal) arguments
2. The target address of a branch instruction must be a valid program location
3. A number of branches or subroutine calls must not create an infinite loop
4. After a call to subroutine there must be a return from subroutine
5. A software loop must have a branch back to the loop start
6. No attempts to pop elements from an empty stack or push elements on a full stack

Kamath suggested programming rules are valid for random program generator which generates instructions and stores them to instruction memory before execution. If generated instructions are injected to the instruction bus on-the-fly, instruction can be different even if the instruction is "fetched" from the same address as earlier in the same test. Therefore, with on-the-fly injection, there is no need to worry about infinite loops or invalid program locations.

4. RISC PROCESSORS

4.1 RISC

The early computer systems were hard coded to perform specific tasks. Later, the computer systems handled more tasks which were executed according to instructions fetched from a memory. The instructions determined the function and parameters used by the computer. An instruction listing, a program was made for that specific computer and was not portable. The program was written in a machine language or assembly. Later on, compilers and better programming languages were developed to support program compilation for different target processors. [14]

Basically, there are two kinds of computer architectures: Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC). An idea of CISC is that hardware performs given task faster than a software, so amount of run instructions is reduced. RISCs trusts in the quick execution of small instructions.[15]

A performance equation (Equation 4.1) is one approach to calculate performance of a processor in a manner the performance can be compared between processors [15]. The same program is run on two processors and the used time on both processor shows which one is better on that specific program. The performance equation can be also used for comparing CISC and RISC processors.

$$\frac{time}{program} = \frac{time}{cycle} \times \frac{cycles}{instruction} \times \frac{instructions}{program} \tag{4.1}$$

In CISC, a complex algorithm is compressed to a single instruction. In early days of computers, memory was expensive, so it was worth of it to compress the program size. CISC processors have only one bus for accessing both instructions and data so they have von Neumann architecture (Figure 4.1). Using only one bus is a bottleneck as instruction and data cannot be accessed concurrently and it is is difficult to implement pipelining effectively.

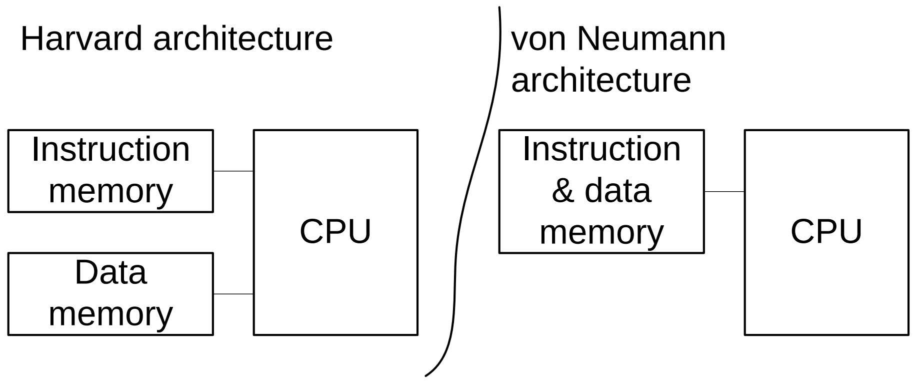


Figure 4.1: Harvard vs. von Neumann architecture

Execution of CISC instruction may take lots of clock cycles and decoding may have side effects, such as fetching data from memory before the execution is completed[15]. Completion of simplest instructions takes many clock cycles due to a complex instruction decoding. Decoder is part of the paradox of CISC: even though algorithms run faster on the hardware than on software, the decoder of CISC processor is more complex and thus slowing execution speed down. Higher amount of logic gates contributes also to higher power usage[14].

An acronym RISC stands for Reduced Instruction Set Computing. The word "reduced" does not mean that an amount of executed instructions is reduced. Instead, a single RISC instruction contains less functionality than one CISC instruction. Thus, more RISC instructions needs to be executed in order to perform a same algorithm as with a CISC instruction. With RISC processor one instruction contributes less, the software is responsible of more complex functionality.[15]

In contrast to CISC, RISC utilizes harvard architecture which have separate buses for instruction and data memories. The RISC's memories do not have to share same specifications. For example, instruction memory can be read-only type and the data memory read-write type memory.

RISC reduces the number of cycles spent on each instruction by having small number of simple instructions in its instruction set. Due to less time consuming instruction decoding the reference clock speed can be greater than for a CISC processor with complex decoding.

RISC	**Benefit**
Uniform instruction format	Less decoding
Simpler design	Faster clocking
Simpler design	Less power / clock cycle
General purpose registers	Less memory accesses
Harvard architecture: separate data and instruction bus	Pipelining
Separate memory load/store and arithmetic operations	Simpler design
Most operations are done in same amount of clock cycles	Predictable
The operations are usually performed in one clock cycle	Predictable

Table 4.1: Benefits of RISC architecture

RISC architecture benefits can be seen from Table 4.1. If both RISC and CISC processors are running with the same reference clock freequency, it depends on complex/simple instruction ratio needed by CISC processor whether one specific program is faster or slower than RISC processor. So for more fair comparison, the benchmarking should be made with intended clock rates which are supported by the silicon technology to be used for the target design.

4.2 Nokia Co-Processor

4.2.1 Overview

Nokia Co-Processor (COP) is a highly configurable RISC processor which is easily scalable from small machine word size to a large machine word size. Internal functional blocks of the COP are shown in Figure 4.2.

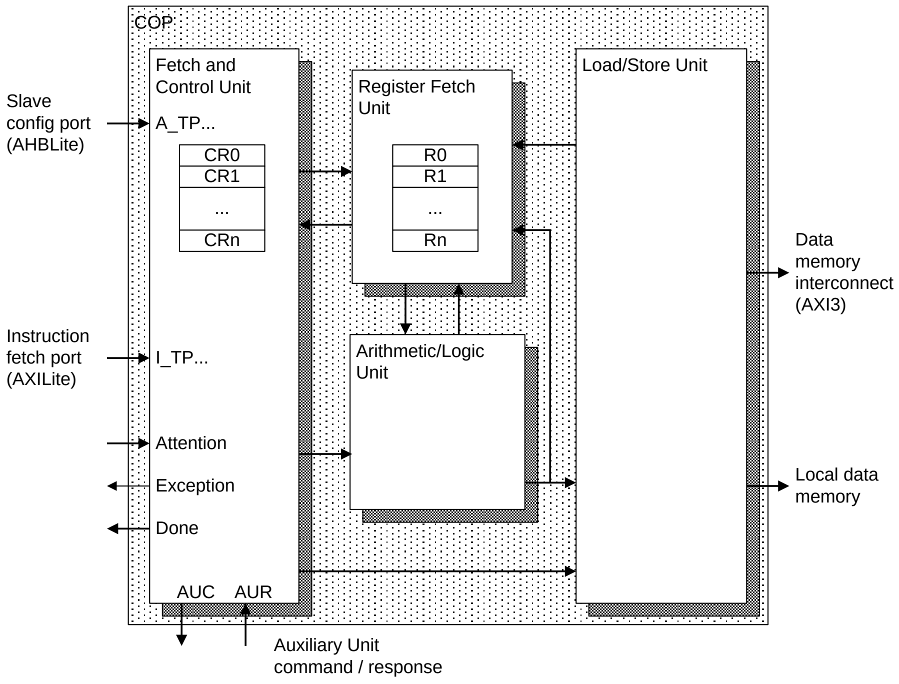


Figure 4.2: COP Microarchitecture

Some features of the COP:

- Relatively large architectural register space (max 64 general registers / thread)
- Generic word length (32 / 64 / 128 bits)
- Byte–addressing memory semantics with support for byte, short (16 bits), long (32 bits) operands in addition to native machine word.
- Simple interlocked instruction execution pipeline (address, fetch, decode, execute).
- Asynchronous execution of memory references and auxiliary arithmetic/logical operation.
- Hardware support for fine–grained multithreading.
- Support for additional, possibly application–specific, arithmetic/logic processing units through an Auxiliary Unit interface.

COP is a little endian fixed-length instruction word load-store machine which has relatively large architectural register space (64 addressable registers / thread).

Instruction word length is typically 32 bits. At the moment the software development is done in assembly language, but compiler project is on-going.

The instructions use byte-addressing memory semantics which supports byte, short (16-bit), long (32-bit) operations in addition to native machine word. The machine word length is scalable in $32 \leq 2^n \leq 128$ bits. It is also possible to configure COP with smaller machine word length than 32 bit, when many of instructions which are intended to be used with wider data size, becomes unusable.

As other RISC processors, the COP is based on harvard architecture so it has separate instruction and data buses. The COP has simple interlined execution pipeline (address, fetch, decode, execute).

The COP supports multithreading. It can be configured to have up to 16 threads.

There is support for additional, possibly application specific, arithmetic/logical processing units through an Auxiliary Unit interface. Instruction set contains AUX instructions which can be used for writing data, controlling, and reading data from an Auxiliary Unit.

The COP has an attention port which can be used as interrupt like signaling. Attention signals can trigger for example DMA transfer or other predefined function.

Nowadays COP has ARM AMBA compatible interfaces (Table 4.2).

Interface	**Use**
AHB-Lite Slave	Module configuration
AHB-Lite Master	Instruction memory communication
AHB-Lite Master	Local data memory communication
AXI4 Master	Main interconnect connection

Table 4.2: COP interfaces

4.2.2 History

The COP development started in 2000's at Nokia Mobile organization.

COPs have been in use as standalone DMA engines (as in Figure 4.3). A plain COP without auxiliary units is an ideal engine for DMA usage as it have a small footprint and it have the necessary memory load and store operations in its instruction set. COPs have also been used as a cryptography engines.

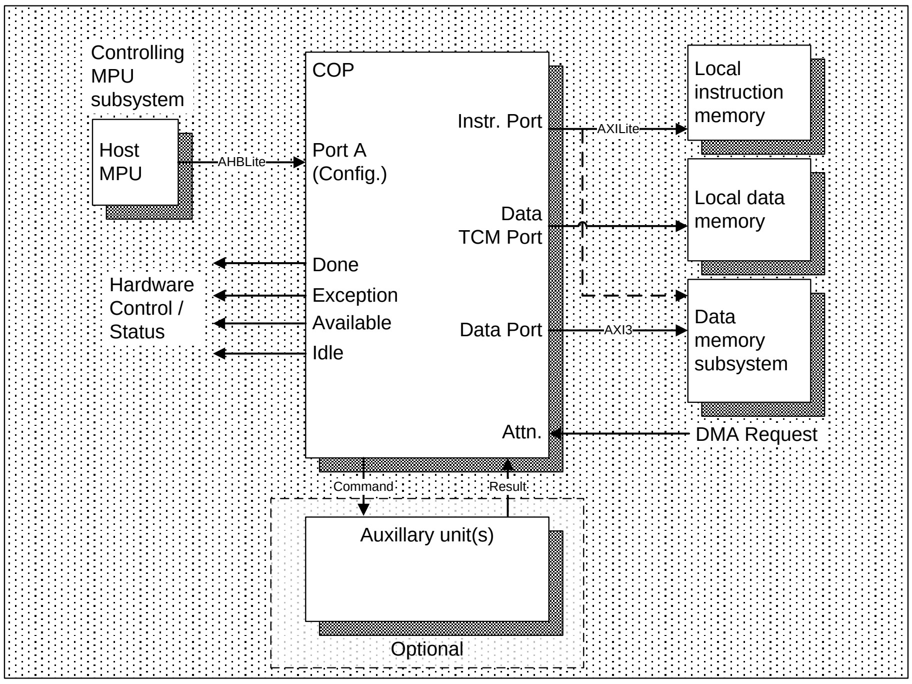


Figure 4.3: COP application example

Sometimes COP have been included in the designs as a backup so it can be taken into use if workarounds are needed for certain hardware bugs. It can be programmed to perform needed algorithms after tape-out even though hardware modifications cannot be made anymore.

COP can initiate multiple operations on buses, it sustains latency. With multi-threading, COP can change context when thread execution stalls for waiting read response.

4.2.3 Legacy COP Verification Test Bench

Back in the years when the old COP was developed and used in Nokia, a VHDL test bench (called legacy test bench) was created. While studying the legacy test bench, the legacy test cases were run and coverage was gathered. The coverage results were very good. A drawback of the legacy test bench was that it used old bus interfaces which are not compatible with later AMBA bus interfaces.

Another drawback is that the legacy test bench is not compatible with simulator software used nowadays. Latest versions of available simulators were not able to compile and run the legacy test bench and the test cases successfully. One problem is also that the legacy test bench is very complex. It is difficult to update by verification engineers who are using SystemVerilog as programming language for other test benches.

During studies on the legacy test bench, it proved to have good toggle, FSM and line coverage. Almost everything is checked within the legacy test bench.

One drawback is that the legacy test bench was made with one bus architecture in mind, so it is not easily re-usable with other bus architectures. The legacy test bench is complex to learn and update. It is also difficult to re-use as it does not have much in common with current verification environments at Nokia.

Another small detail, which can cause trouble is that the verification test bench was made by the same person who has designed the processor. There is possibility that some design issues are duplicated to the verification test bench.

A problem is that there is no readily available solution for verification of the updated COP version. Some of the functionality can be verified with an old simulator version and other functionality in integration verification in SoC top level with the latest simulator version. It is evident, that it can be only a temporary solution which does not work in long term.

The COP needs verification environment which can verify all functionality of the COP. The verification environment shall support further development of new features to the COP, development of COP peripherals as well as other COP related activities such as software development and high-level programming language compiler development.

A need of development of new COP test bench or test suite was evident when the decision of further usage and development of COP was made. The new COP verification environment is presented in the Chapter 5 .

5. NOKIA CO-PROCESSOR VERIFICATION SUITE

The COP Verification process is explained in this chapter. At first, strategy document (Section 5.1), then verification plan document (Section 5.2) are presented. After that, the implementation of the test bench is explained in high level (Section 5.3).

5.1 Verification Strategy

A verification strategy describes the design under verification (DUV) on high level and decisions regarding the verification project of an IP block, a subsystem or a SoC depeding of the scope of the strategy. It also describes which verification methods are to be used. Verification strategy document is written before implementation of a test bench.

At Nokia SoC development, the verification strategy document has for instance following chapters:

1. Introduction
2. Released documents
3. Verification targets
4. Issue tracking
5. Regression management
6. Releasing practices

In addition to those chapters, test environment is also presented in high level without constraining too much the test environment implementation.

The introduction chapter contains basic information about the DUV. In this case it contains almost all the same content as in Nokia Co-processor Overview subchapter 4.2.1.

Released documents chapter introduces reader to design specifications and other useful documents regarding the DUV. Information (such as hyperlinks) where the referred documents can be found, are listed in references chapter.

In the Verification Strategy document, verification targets chapter shows the decisions what the targets for verification activity are. Verification targets chapter is split into coverage targets, test case targets, verification qualification targets, unverified configurations/features. In COP case the main target is to verify that RTL implementation of COP core fulfills requirements set in COP Design Specification and the COP core instruction functionality is following COP Programming Guide.

For COP verification, all external interfaces are selected to be verified. The interfaces are listed in Table 5.1.

Port name	**Protocol**
Configuration port	AMBA AHB Lite slave
Status outputs	Output
Attention request	Input
Auxiliary unit port	AUX Command & Response
Clock reference	Input
Asynchronous reset	Input
Local data memory	AMBA AXI Lite / RAM
Instruction memory	AMBA AXI Lite / RAM
AXI 3 Lite master	AMBA AXI 3

Table 5.1: External interfaces to be verified

The Coverage targets are shown in Table 5.2. The aim is to get 100% coverage result with exclusions. An initial target is to get same coverage as with legacy COP test bench.

Coverage targets
Line
Toggle
Statement
Expression
Feature: instruction set

Table 5.2: Coverage targets

Other targets for the COP verification are listed in Table 5.3. In that table functional tests with data bus width configuration of 8, 32, 128 bit. As said in COP overview chapter (Chapter 4.2), with 8 bit machine word width, the instructions with other than byte semantics are not allowed to be used.

Target	Comment
Instruction set functionality	All instructions
8bit, 32bit, 128bit machine word width	
Thread FSM	
LSU FSM	
CPU FSM	
Use case: data transfer	
Use case: AUX port hardware acceleration functionality	

Table 5.3: Other targets

Issue tracking chapter describes how bugs and issues are tracked during the process. The issue tracking is very important tool to track which bugs are fixed and which are still potentially present. It gives visibility to outside of the design project about maturity of the design. Issue tracking ensures that all reports of fixed bugs will be closed as soon as they are verified and the design proved to be working right. In COP case, Attlassian Jira software is used as issue tracking.

Regression management chapter tells how regression testing is managed and implemented. The regression system can have some subset of tests for design sanity checking. There is also need for running complete list of test cases in order to track the verification maturity and status. Each test case is back annotated to a verification plan so that regression results show which test cases are passing and which ones are failing. For COP IP verification a Nokia made, XLS file and Perl script based regression management was selected.

Releasing practices chapter gives guidelines when and how releases of the test environment are done. The chapter tells what revision control system is used. Other practices of naming convention can also be described. It also describes how the verification environment can be fetched from the revision control system and taken into use. COP IP verification uses subversion revision control. Verification releases are done for each COP design release to provide clean code base for each design release if old design versions are taken into use and needed to be verified.

Verification qualification is done using Synopsys Certitude tool. The tool uses mutation analysis to inject the design with artificial faults and checks how well the verification environment activates, propagates and detects the faults. Certitude computes metrics based on the numbers of different fault statuses. The targets are Certitude's default scores:

- Activation score: activate faults / all faults = 95 %
- Propagation score: propagated faults / activated faults = 80 %
- Detection score: detected faults / propagated faults = 95 %

Also two most important injected fault types (output stuck and reset stuck faults) shall be 100 % detected.

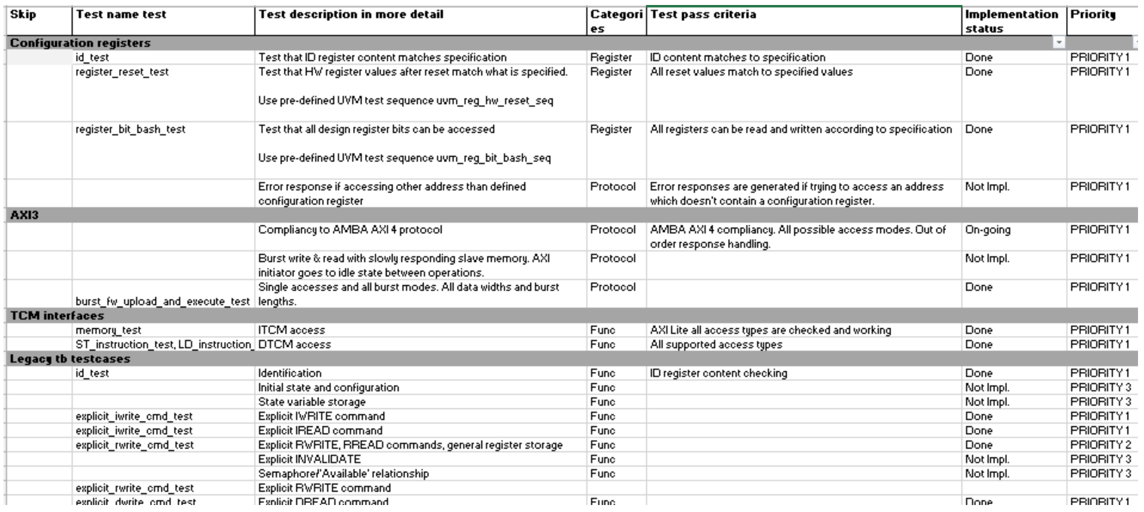
5.2 Verification Plan

The IP verification process is split to small verification tasks in order to ease the verification process. Verification plan describes each verification task. Each test case should return pass/fail status. The statuses are back annotated to the test plan in such a manner that test creation status and IP quality can be easily checked.

Feature list:

- List of all features which need to be verified
- Features can refer to requirements of the design
- Feature verification can be divided to sub-tasks
- Easier to read than plain list of test cases

Not much effort was spent for creating feature list for this verification plan as test case listing of legacy test bench was available. Old test cases were taken and some new test cases added to cover known issues in the design. Snippet from COP verification plan can be seen in Figure 5.1. Non-legacy test bench test cases are related to changes in memory interfaces.



Skip	Test name test	Test description in more detail	Categories	Test pass criteria	Implementation status	Priority
Configuration registers						
	id_test	Test that ID register content matches specification	Register	ID content matches to specification	Done	PRIORITY 1
	register_reset_test	Test that HW register values after reset match what is specified. Use pre-defined UVM test sequence uvm_reg_hw_reset_seq	Register	All reset values match to specified values	Done	PRIORITY 1
	register_bit_bash_test	Test that all design register bits can be accessed Use pre-defined UVM test sequence uvm_reg_bit_bash_seq	Register	All registers can be read and written according to specification	Done	PRIORITY 1
		Error response if accessing other address than defined configuration register	Protocol	Error responses are generated if trying to access an address which doesn't contain a configuration register.	Not Impl.	PRIORITY 1
AXI3						
		Compliancy to AMBA AXI 4 protocol	Protocol	AMBA AXI 4 compliancy. All possible access modes. Out of order response handling.	On-going	PRIORITY 1
		Burst write & read with slowly responding slave memory. AXI initiator goes to idle state between operations.	Protocol		Not Impl.	PRIORITY 1
	burst_fw_upload_and_execute_test	Single accesses and all burst modes. All data widths and burst lengths.	Protocol		Done	PRIORITY 1
TCM interfaces						
	memory_test	ITCM access	Func	AXI Lite all access types are checked and working	Done	PRIORITY 1
	ST_instruction_test, LD_instruction_	DTCM access	Func	All supported access types	Done	PRIORITY 1
Legacy tb testcases						
	id_test	Identification	Func	ID register content checking	Done	PRIORITY 1
		Initial state and configuration	Func		Not Impl.	PRIORITY 3
		State variable storage	Func		Not Impl.	PRIORITY 3
	explicit_iwrite_cmd_test	Explicit IWRITE command	Func		Done	PRIORITY 1
	explicit_iwrite_cmd_test	Explicit IREAD command	Func		Done	PRIORITY 1
	explicit_rwrite_cmd_test	Explicit RWRITE, RREAD commands, general register storage	Func		Done	PRIORITY 2
		Explicit INVALIDATE	Func		Not Impl.	PRIORITY 3
		Semaphore/'Available' relationship	Func		Not Impl.	PRIORITY 3
	explicit_rwrite_cmd_test	Explicit RWRITE command				
	explicit_dwrite_cmd_test	Explicit DREAD command	Func		Done	PRIORITY 1

Figure 5.1: Verification plan example

List of test cases have column for test case name, which is added as soon as the creation of the test case is on-going and test name is known. Test case is described in more detail, in next column. Test pass criteria, implementation status and other information is also for each test case in their own columns.

5.3 Verification Environment

The COP verification environment (Figure 5.2) is programmed in SystemVerilog language and by using UVM classes as a base. An exception to SystemVerilog is COP C++ model which is written in C++ language and it has C wrapper for DPI-C usage. The COP is instantiated in the top level and all needed VIPs and agents are connected to the COP via SystemVerilog interfaces.

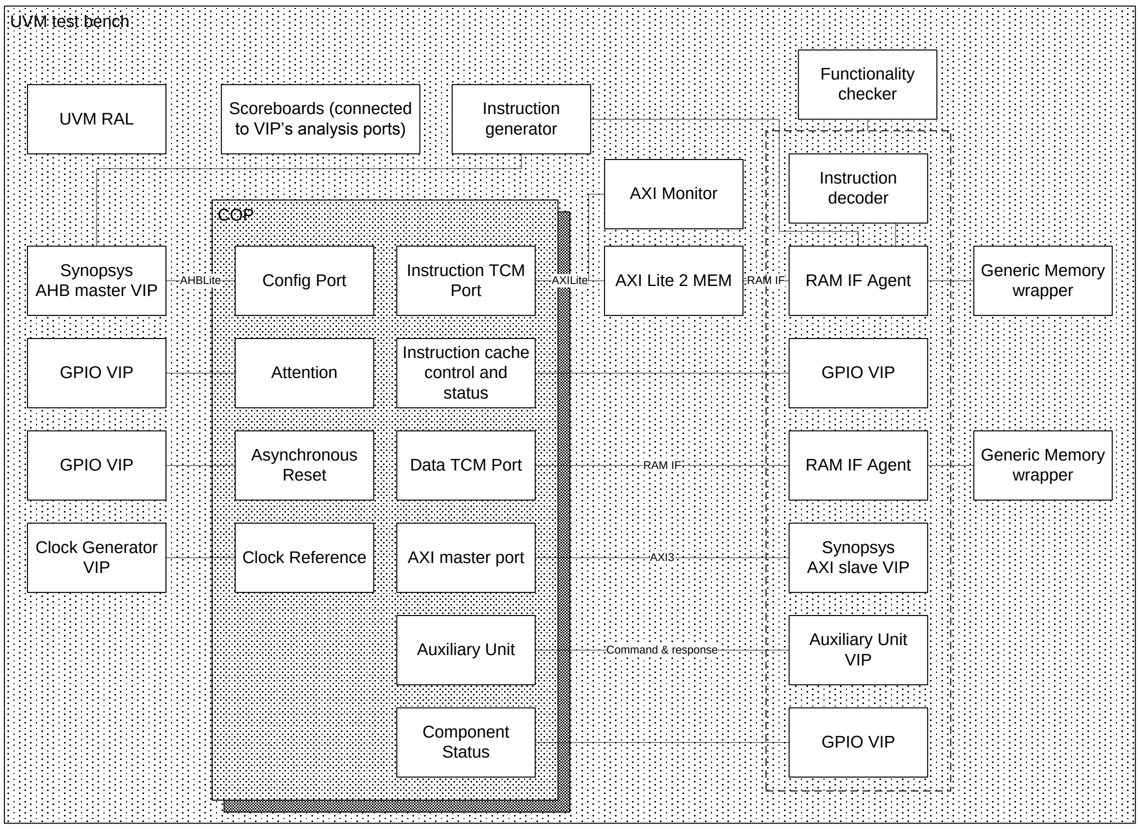


Figure 5.2: UVM Test Bench

The functionality checking is done by instantiating SystemVerilog assertion module and connecting its interface to COP wrapper module's interface. The reference model keeps track on the internal register state and performs the same operations as the RTL version of the COP. The reference model's register values are checked every time they are visible in any of the COP outputs.

When the verification strategy was written, all needed Nokia in-house developed AMBA VIP's were not available. The unavailability of some in-house VIPs lead into the decision that all needed AMBA VIP's are taken from Synopsys AMBA suite. As long as verification environment has one VIP from Synopsys AMBA suite, one Synopsys license is needed. There is no extra cost of having more than one Synopsys VIP.

Needed AMBA VIPs are AHB Lite master and AXI slave. AHB Lite master is connected to COP's Configuration Port. Synopsys AHB Lite master is used with UVM RAL model of COP configuration registers. Adapter function is needed to transform RAL operations to AHB Lite master's sequence items. An example of the adapter was found from Synopsys' SolvNet.

Synopsys AXI slave is connected to COP's AXI master port interface. The VIP can be used with all COP supported read and write modes. Synopsys' AMBA Suite has built-in AXI protocol checker which is enabled.

Top block has interface for connecting the TB and the DUT. As the processor design is written in VHDL, it needs to be instantiated in a SystemVerilog wrapper module before it can be connected to UVM test bench.

SystemVerilog assertions are part of functionality checking. The assertions describe how each COP's ports shall behave. Assertions are specified inside a SystemVerilog module to which the interfaces are given as a parameter.

Functionality checker includes SV assertions and C++ COP model which is connected to the test bench via DPI-C. More about C++ model and DPI-C in Subchapters 5.4 and 5.4.1.

One Synopsys' AXI monitor is added for instruction memory AXI lite bus protocol checking.

5.3.1 UVM Test Bench Top Level Module

On the top level of the test bench there is a SystemVerilog module. All components related to the test environment are instantiated in this module.

The design under verification and verification IP's plus supporting logic are instantiated. VHDL components need a SystemVerilog wrapper module which instantiates the VHDL component. Interfaces are declared and given to each module and intermodule connections are made in the top level module.

When the interfaces are created they need to be added as virtual interfaces to UVM configuration database.

An UVM test package is also declared at top level module. The test package contains a base test and test case specific files. More about UVM tests in subchapter 5.3.2.

In addition to COP instance, few VHDL components are instantiated on top level. SystemVerilog wrappers and interfaces were made for them:

- Local memories (instruction and data)
- AXI Lite to generic RAM interface adapter

- One delta cycle delay component for clock signal

Memory wrappers are instantiated to hide any RAM memory specific port naming or functionality changes. The memory wrapper is used to hide technology specific signals. Generic memory wrapper looks same whether there is generic or silicon technology specific RAM models in use.

AXI Lite to Generic RAM component translates AXI signaling to generic RAM wrapper compatible signaling.

SystemVerilog modules made for this environment and instantiated at top level were:

- ByteStrobe2BitStrobe conversion
- RAM bus multiplexer (2 to 1)
- COP assertion module
- Auxiliary unit VIP

COP has byte strobes at local memory bus interfaces, but interface of RAM wrapper bus has bit strobes. ByteStrobe2BitStrobe conversion module shown in Algorithm 1.

Algorithm 1 Byte strobe to Bit strobe conversion

```
`ifndef _BYTESTROBE2BITSTROBE_SV
`define _BYTESTROBE2BITSTROBE_SV

module bytestrobe2bitstrobe (ram_if ram_if_m, ram_if ram_if_s);

  assign ram_if_s.CK       = ram_if_m.CK;
  assign ram_if_s.A        = ram_if_m.A;
  assign ram_if_s.D        = ram_if_m.D;
  assign ram_if_s.CS       = ram_if_m.CS;
  assign ram_if_s.CS_N     = ram_if_m.CS_N;
  assign ram_if_s.WE       = ram_if_m.WE;
  assign ram_if_s.WE_N     = ram_if_m.WE_N;
  assign ram_if_s.WRENZ    = ram_if_m.WRENZ;
  assign ram_if_s.WRENZ_N  = ram_if_m.WRENZ_N;
  assign ram_if_m.Q        = ram_if_s.Q;
  assign ram_if_s.PW       = 4'h0; //tied to zero

  //Byte strobe to bit strobe conversion.
  always @(ram_if_m.WRENZ) begin
    foreach(ram_if_m.WRENZ[u]) begin
      ram_if_s.BS[u*8 +: 8] = (ram_if_m.WRENZ[u]) ? 8'hFF : 8'h0;
    end
  end

endmodule

`endif //_BYTESTROBE2BITSTROBE_SV
```

RAM bus multiplexers are instantiated between COP and both local memories. Multiplexer has select signal for choosing either memory model or instruction generator. For example, in the instruction memory's bus the multiplexer is used for injecting generated instructions to COP's execution.

COP assertion module is connected to COP ports via SystemVerilog interface. The assertion module contains SystemVerilog assertions which for example describe the protocol of local data memory interface and local instruction memory. The need for assertions was found during mutation based verification environment qualification.

Auxiliary Unit VIP is created for functional verification of COP's AUX instructions and auxiliary unit command and response port. Auxiliary Unit VIP monitors the auxiliary unit command port and depending on the command, it generates response to auxliary unit response port. More about Auxiliary unit VIP in Chapter 5.3.5.

5.3.2 UVM Tests

Base test is a class which is extended by each test class. It contains the basic configuration of UVM environment and methods used by the tests.

Many test cases are used to verify that the verification environment and COP are set up properly. Sanity tests include configuration register accesses and test that all memories are accessible.

UVM has built-in methods to verify registers for which a register abstraction layer (RAL) is generated. UVM built-in tests are easy to take into use in early phase of test bench development.

For sanity checking, there is a verification firmware which can be used for checking that the verification environment and COP are set up properly. During a test, the firmware and reference data are uploaded to the COP's instruction memory and data memory, respectively. The COP is configured, and a thread is started. During the execution of the verification firmware, all COP supported access sizes and burst lengths are used for moving the reference data to different places in the external memories. Always, the destination of previous transferred block is used as a source for next transfer. Thus, after the last transfer, only the last written data block is needed to be compared against the reference. The test returns pass status when all data is checked as passed.

Instruction set tests run a subset of instructions in all threads and by accessing all available registers for each thread. The checking is automatic if instruction functionality for the given instruction is created in the COP reference model.

Implementation of new instruction tests mean adding new instruction decoding and implementation functions to the COP reference model. Instruction generator must be updated with an information on how to form a new instruction. A new UVM test must be added which generates instructions to an instruction queue.

Most of instruction set tests follow same form:

1. Select instruction injection via instruction memory bus

2. Initialize COP's registers with random values
3. Generate instructions with randomized parameters and store them to an instruction queue
4. Give the instruction queue to a sequence which gives instructions to COP processor and COP model
5. Start the COP instruction execution sequence
6. The COP ports are monitored during the execution of the sequence and the responses are compared with COP reference model's responses. In case of any mismatch, an error report is generated and error count increased.
7. After the sequence stops, COP physical registers, exception and fault registers are read and compared with COP model's registers.
8. Pass/Fail status is given in the end of simulation. Pass status is given if there were no errors (error count == 0) during the test.

All tests shall return pass or fail status because the status information is needed while back-annotating regression results to the verification plan.

5.3.3 Instruction Generator

Instruction generation is an essential part of processor verification test bench as it is used for creating stimuli for the processor under verification. For this test bench an instruction generator SystemVerilog function was made. The purpose of the instruction generator is to generate valid instruction words with randomized parameters. Depending on the instruction type there are different parameters to randomize.

COP has fixed length (32 bit) instruction words. Instruction prefix field length varies depending on number of needed bits for other parameters. An instruction which needs less bits for other parameters can have longer instruction prefix.

Each instruction word can contain multiple bit fields which determine the properties of each instruction (see an example of an instruction word from Table 5.4). Instruction prefix determines the type of the instruction. Each instruction prefix is unique and it is used for decoding the rest of the instruction word. Similar instructions are grouped under the same instruction prefix.

	Prefix	Subtype	Dest	Source	Immediate
Assembly		addi	r5,	r10,	20
Binary	000	00110	000101	001010	000000010100
Comments	Register-immediate	ADD			
Size (bit)	3	5	6	6	12

Table 5.4: ADDI (Add immediate) instruction

In an ADDI example (Table 5.4) instruction prefix is 3 bit long and subtype is 5 bit long so there is 24 bits left for determining the functionality of ADDI instruction. It is split in the manner that any of 64 logical registers can be used as destination and source of the values. A 12 bit long field for immediate value is available.

Subtype field value is reusable as there are many variations of instructions. For instance in COP, there is number of ADD operation types: Register-immediate, Register-register scaled and Register-register vector. All of those have their own instruction prefixes but subtype field values are exactly same.

Instructions with Register-immediate prefix instruct the arithmetic logical unit (ALU) to execute their function with value from source register and immediate value. Result is then stored to destination register. Register-register scaled instructions execute functions with two source registers. An immediate operand which determines scaling of second source register (number of shifts to the right). Register-register vector instruction can be used to avoid using loops while there is need to perform same operation multiple times a row.

5.3.4 Usage of Instruction Generator

The instruction generator is a function which can be called from the test. It can be used calling the following function:

```
function logic[31:0] instruction_generator(
          instruction_t instruction,
          int param1 = 0,
          int param2 = 0,
          int param3 = 0,
          int param4 = 0,
          int param5 = 0 );
```

The instruction generator returns generated instruction value. If there are illegal parameters, the function notifies with an UVM_ERROR message.

There is a maximum amount of instruction parameters within the function prototype. Only those which are non-zero, needs to be specified in the function call. For example, a halt instruction, which does not have any parameters, can be called easily with an enumerated type instruction_t such as:

```
instruction_queue.push_back( instruction_generator( HALT ) );
```

There are two use cases for instruction generator:

1. The instructions can be written to instruction memory in the beginning of the simulation by writing them via configuration port (AHB Lite).
2. Instructions can be generated "on the fly" by injecting the instruction words to bus.

In both cases the instruction words are given to the COP C++ model every time the instruction is fetched by the real COP.

There is a bus multiplexer which is used to decide whether the instructions are intended to be fetched from instruction memory or from instruction generator. Multiplexer has select signal which can be toggled from the test.

When COP C++ model is used, the generated instructions are also given to COP model. COP model keeps track on the expected addresses in the instruction bus. Addresses on the instruction memory are compared to the reference model generated reference addresses.

5.3.5 Auxiliary Unit VIP

Auxiliary Unit VIP (AUX VIP) was created because there was need for an auxiliary unit model for generating needed AUX responses in order to verify COP's AUX instructions.

One of Nokia's in-house developed VIPs was taken as starting point for Auxiliary Unit VIP development. Original functionality is removed and simple finite-state machine (FSM) added in place.

The auxiliary unit VIP supports all auxiliary unit commands given from COP's auxiliary unit command port and responds to them via auxiliary unit response port.

AUX VIP is instantiated at COP test bench top level module and configured in UVM environment class' method.

5.4 Co-Processor C++ Model and DPI-C

COP C++ model is a software written as part of this Master's thesis project. Main purpose of COP model is to serve as reference model against which the RTL representation of the COP is verified. The functionality is added while creating test cases to the verification environment. New instructions are added to the instruction decoder of the COP model and instruction generator as soon as there is need for them in tests.

The COP model consists of several files. Cop_model.cpp and Cop_model.h are the most important ones. The internal functionality of COP is created in them. The COP model utilizes some queue classes which are used for storing the output port states. COP model also uses COP disassembler to print the instructions in assembly language representation to a simulator log while software is being run in the COP during simulation. COP disassembler was also implemented as part of this thesis project.

There is also a shell script which is needed for making a C header from COP VHDL generic value listing. That generic value listing describes the instruction encoding and decoding, internal register addresses and so on. Because COP is highly configurable through generics, it is crucial to have the VHDL generics and COP C++ model aligned. In case there is something changed in the COP design, it is translated to the C++ model without an effort.

As DPI-C does not support C++ language as such, C wrapper had to be created. More technical details about DPI-C, C wrapper and C++ usage is given in Chapter 5.4.1.

COP model development

As in making of any reference model, the internal functionality of the modelled design has to be taken into account. COP modelling is not an exception. The COP C++ model work started by studying the functionality of the COP.

Before a thread in COP can be spawned, the COP shall be configured properly. Configuration of the COP model is made via configuration port access methods. Implementation of COP model started by creating the *configuration register* model and making access methods for it.

There are some special registers such as an ID register which behave differently than any other register in the COP. The contents of the ID register is associated with an array of COP specific values such as component ID code, and some VHDL generic values which have been used for creating that particular component. Other generics which can be read via the ID register are values such as total number of physical registers and maximum number of threads which are configurable while instantiating the COP to a SoC design.

The ID register is associated with a pointer which is incremented after each ID register read operation. Write to the ID register changes the value of the pointer. Each read returns a value of the array item pointed by the pointer value and then increments the pointer. All ID register array's values can be thus read by applying consecutive reads to one address. An example of ID register contents shown with pointer values in Table 5.5.

Pointer value	**Name**
0	Sync word 0
1	Sync word 1
2	Component ID
3	Some parameter
...	...

Table 5.5: ID register content

There are also some set-clear type registers such as Acknowledge-exception. The COP notifies the programmer with exceptions, in some cases COP exception causes fault condition which triggers an interrupt request for host processor. Host processor can clear exception state by accessing either acknowledge or exception register.

In addition to configuration registers, there is also general purpose registers which are configurable to be thread specific or register access areas assigned for threads can be overlapping in a way that threads can communicate through them.

Physical register model contains all general purpose registers. The total number of general purpose registers is defined by generic values while instantiating the COP model. A thread can see only *logical registers* which are assigned for them. Thus, there is need for threadwise logical register to physical register mapping.

Instruction specific methods and an instruction decoder which calls the instruction specific methods were created. The instruction specific methods implement the similar functionality as COP, access reference register models and creates transactions to queues which are used as reference for comparing the behavor of COP.

For quick checking of the instructions added to the COP model, a testing application which instantiates and configures the processor model was created. It reads stimuli from a hex file and gives the instructions from it to the COP model. For testing purposes, a file containing assembly language instructions, was created. It is also easy the check the COP C++-model's disassembler against a known reference in the file.

COP model usage

In order to use COP model, it has to be instantiated in the intended use environment. If the model is used in C or C++ environment, it can be instantiated without any wrappers. When the model is used in SystemVerilog environment, DPI-C and C wrappers are needed. The wrapper contain functions for accessing COP model's methods and are provided with the C++ model.

In SystemVerilog environment instantiation of the COP model, a `chandle` (like C pointer) to the new COP model instance is created. The chandle must be provided as a parameter for each COP model's method access function.

Create method of COP model calls `initialize()` method which initializes all register values and clears queues. The COP is then ready for configuration. The configuration is similar as with accessing UVM register abstraction layer. All the same configurations shall be written to the COP model as to "the real COP" in order to get comparable behavior.

When the COP model is configured, thread execution can start. The COP model instance can take instructions as soon as a thread is spawned. Execution is triggered by calling method `execute( uint32_t instruction )`.

The execute methods calls COP disassembler's `disassemble( uint32_t instruction )` method. If the instruction is valid, the disassembled instruction is printed in COP's assembly language. An error count is incremented and notification about non-valid instruction is given if there is no disassembler function for given instruction.

After disassembling, the execute method calls `instruction_decoder( uint32_t instruction )` method. The instruction decoder decodes the instruction's type field and passes the instruction for further analysis to appropriate decoder method. All parameters are then extracted from the instruction and all the symptoms related to that particular instruction is written to registers and when needed, written to output queues. Also an instruction pointer is updated depending on the executed instruction. If the instruction format of the given instruction is not valid, an illegal instruction fault flag is raised as it would be raised in “the real COP”.

There are also methods for accessing the output queues. It comes handy while verifying the instruction set because then COP written data values can be checked against reference model’s output queue values.

5.4.1 C++ Model Integration to UVM test bench with DPI-C

DPI-C does not support directly C++ programming language, but it supports C language. Because C++ classes and methods can be accessed from C program, a C wrapper with access functions for C++ class methods can be created. In the following example, a method is provided for accessing C++ class’ methods through C functions within SystemVerilog in UVM environment.

Please note, the following example has only essential parts from DPI-C, C++ model, C wrapper and Makefile. The example does not have all the necessary files to compile, but it can be used as reference.

A C++ class header used in the following example is shown in an Algorithm 2. There is only a constructor for object and one access method called `method()` in order to keep the example as simple as possible.

Algorithm 2 Example_class.h file

```
//Example_class.h

class Example_class {
        public: Example_class() {};
        int32_t method( int32_t value );
}
```

To create a C wrapper for a C++ model there is two headers to be added. First, a SystemVerilog DPI-C header called "svdpi.h" and a C++ model’s class header. See Algorithm 3 on the next page. The svdpi.h is provided with UVM.

C++ compiler shall use C linkage for functions. Extern “C” is used for forcing C linking. Object instantiation is made via C function which returns a pointer to the created object. The pointer is given to SV variable of type `chandle`. Chandle is passed in every C++ model method call.

Each C++ method access function shall take the object pointer value as a parameter.

Methods are called via the object pointer : `instance->method( input );`

Algorithm 3 Example C wrapper

```
#include "svdpi.h"
#include "Example_class.h"

extern "C" {
        void * new_instance(){
                return new Example_class();
        }
        int32_t call_Example_class_method(
                Example_class *instance,
                int32_t value ){
                        return instance->method( value );
        }
} /* extern "C" */
```

DPI-C imports:

Object creation function shall return chandle (pointer) for created object as in Algorithm 4. Chandle is needed for accessing the methods of the object.

Algorithm 4 DPI-C SV file snippet 1

```
/* Class instantiation */
import "DPI-C" function chandle new_instance();
```

Accessing a method from SystemVerilog can be done via a function shown in Algorithm 5.

Algorithm 5 DPI-C SV file snippet 2

```
import "DPI-C" function int call_Example_class_method(
        input chandle instance,
        int value
        );
```

Now imported C functions can be used in SystemVerilog code. Algorithm 6 how the object is instantiated and instance's method can be used in .sv code.

Algorithm 6 DPI-C provided object instantiation in SV

```
int input_value;
int output_value;
chandle example_class_instance0;
input_value = 7;
example_class_instance0 = new_instance();
output_value = call_Example_class_method(
                        example_class_instance0 ,
                        input_value);
```

Compilation of C++ reference:

Variable initialization in a Makefile is shown in Algorithm 7. Use `g++` with `-c` flag to create unlinked object files.

Algorithm 7 Makefile Variable initialization

```
###----------------------------------
### DPI C and C++ files
###----------------------------------
MODEL_DIR = ../sw/cpp_model_dir
CPP_FILE = Example_class.cpp
DPI_C_FILE = Example_class_wrapper_dpi.c
DPI_O_FILE = Example_class.o Example_class_wrapper_dpi.o
INC_OPT = -I. -I$(VCS_HOME)/include -I$(MODEL_DIR)
COMP_OPT = -c
```

Compilation of Example_class C/C++ files is shown in Algorithm 8 on the following page.

Algorithm 8 Makefile: Compilation of example class C/C++ files

```
###———————————————————
### EXAMPLE CLASS
###———————————————————
example_class_cpp: #$(CPP_FILE)
        g++ $(INC_OPT) $(COMP_OPT) $(MODEL_DIR)/$(CPP_FILE)

example_class_wrapper_dpi_c: #$(DPI_C_FILE)
        g++ $(INC_OPT) $(COMP_OPT) $(MODEL_DIR)/$(DPI_C_FILE)
```

After the object files are created, they can be included to simulator compilation.

Vcs elaboration command example shown in Algorithm 9.

Algorithm 9 Makefile: VCS elaboration

```
###———————————————————
### Elaborate
###———————————————————
elaborate: example_class_wrapper_dpi_c example_class_cpp
        ${VCS_HOME}/bin/vcs ${VCS_OPTS} $(DPI_O_FILE) \
                +lint=TFIPC-L +lint=PCWM ${TOP_LIB}.${TOP_NAME} \
                -o ${EXE_DIR}/${EXE_NAME}
```

6. RESULTS

The reason, why not final results are not presented in this Master's thesis is that verification closure is not achieved yet. This have been such a long project that allocated time for Master's thesis have almost been used twice.

Following subchapters present results of the COP verification environment development such as verified features and found bugs. In the last subchapter, the spent effort is estimated.

6.1 Verified Features

In verification planning phase, the list of test cases was prioritized to three categories. The most important features were selected as high priority. Those features addressed register model, exceptions, faults, and external interfaces.

Verified Priority 1 features:

1. Configuration register accesses
   - ID register test
   - Read and write tests (bit bash)
   - Reset value test
2. Local instruction memory accesses
   - Read and write accesses via configuration port
   - COP instruction fetch
3. Local data memory accesses
   - Read and write via configuration port
   - COP read and write accesses by using COP's load and store instructions

4. Physical register accesses
    - Read and write via configuration port
    - COP thread access to physical registers
5. External memory accesses
    - COP read and write access by using COP's load and store instructions
        - All access sizes: 8bit, 16bit, 32bit, and machine word
        - All burst lengths: 1, 2, 4, 8, 16
    - Memory bus protocol compliancy
6. Auxiliary unit command and result port
    - All AUX instructions
    - Auxiliary unit responses
7. Exceptions and faults
    - Some exception and fault tests are not implemented yet

Priority 2 and 3 features are still to be verified. Most of Priority 2 and 3 cases are arithmetic-logical instructions. The reason why arithmetic-logical unit (ALU) specific test cases have lower priority is, that the arithmetic-logical unit is not updated, the legacy test bench's tests can be trusted. There is also jump and branch instructions to be verified. High number of ALU instructions are already implemented to COP C++ reference model, but instruction generation and tests are not implemented.

6.2 Bugs

Bugs found while setting up the UVM environment:

- In very early testing with UVM RAL and configuration port default values, following design issue was found:
    - Data_n register generation was coupled with thread generation. If number of threads is low, there is possibility that not enough registers are not generated for configuration register bank for data_n. For example, 2 threads and 128bit machine word width should have generated four 32bit data registers. Only two 32bit registers was generated due to design issue.

- IP-XACT bugs:
  - IP-XACT which is the source for RAL and C-header generation, had mistakes. They were discovered by using RAL model.
    - Wrong width of the RAL generation for couple of registers lead to port/signal mismatch error in analysis of the test bench
- Register aliasing bug
  - Occurs when misusing the register model with odd numbered thread.
  - Writing to a logical register outside allocated register range.
    - No exception triggered. Register addess gets aliased. Write succeeds.

Bugs found while setting up COP C++ Processor Model:

- Instruction fetch bug
  - COP was trying to initiate write to instruction memory while fetching commands
  - Caused glitch to instruction bus
  - VIA to AXILite component problem
- Store posted triggers COP protocol exception
  - COP protocol fault
  - Problem in COP's internal PSI to AXI conversion
  - There is no posted writes in AXI 3 protocol, but COP supports posted writes. COP does not expect response, but AXI slave gives the response
- Documentation improvements to COP Programming Guide

There are other bugs, found in SoC top level verification which are not included in this list. The fixes are confirmed by running test cases in this verification environment.

6.3 Spent Effort

The verification project of COP IP was started at June 2015 with two verification engineers who had no previous experience of setting up UVM test benches. During the time there have been activities for previous and new projects on-going for both engineers. Summer vacations and paternity leave took place in July-August 2015 timeframe.

Verification strategy and plan reviews were held in August 2015.

It took 3 to 4 months to get the verification environment up with all third-party and Nokia in-house VIP's. By mistake, an old broken version of Synopsys AMBA Suite was used in the beginning of verification environment set up, which caused delay. An estimate of monthly spent effort is shown in Table 6.1 (R: Review, V: Vacation, PL: Paternity leave). As both of verification engineers were making test bench for first time, the learning and problem solving took time.

	2015							**2016**			
	June	**July**	**August**	**September**	**October**	**November**	**December**	**January**	**February**	**March**	**April**
Verification Strategy	1	0,5	0,5R								
Verification Plan		0,5	R								
Environment		0,5	1,5	1	1						
Qualification				0,5							
Reference model					0,5	0,5	0,5	0,5	0,5	0,5	0,5
Test cases				0,5	0,5	1,5	1,5	0,5	0,5	0,5	0,5
Overlapping project	**1**	**0,5V**	**0,5PL**					**1**	**1**	**1**	**1**

Table 6.1: Estimate of spent effort

Calculated from Table 6.1, overall 16,5 man-months have been spent for creating the UVM test suite for COP.

As there is no data available for how long it took to develop COP design, it is hard to compare the design effort to the verification effort. I suppose, the verification has taken already longer than the design effort of the COP. So, the verification effort is going to be much higher than 50% of overall design+verification effort.

For processor reference model creation, the designer level knowledge is needed and the needed effort for reference model implementation can be very similar to the design implementation. In addition to reference model, the verification environment, checkers, test cases, and verification components has to be created.

7. CONCLUSIONS

Importance of proper verification environment is huge for IP blocks. An IP block cannot be taken into use for new system on chip designs if it is not verified properly. It is possible to purchase verified IP blocks, but they are expensive and cannot be customized as freely as in-house designs. It is also good that there is continuity for each in-house developed IP block on the SoC designs. The development stays active and knowledge about each IP block stays at high level.

The chapter 4 introduced RISC processors in general and Nokia Co-processor. There are various reasons why it is good to have Nokia Co-processor IP available. First of all, it is very versatile. It scales well from very small footprinted, one threaded 32-bit processors up to 128-bit and 16 threaded processors. COP alone can be used as direct memory access accelerator, but adding hardware accelerators, the COP can turn into a very effective engine for different applications.

The COP IP is a legacy IP from Nokia's previous SoC organization. Bus interfaces of that were updated to be ARM AMBA compatible as part of the SoC development. Nokia Co-processor had a legacy verification environment which become mostly outdated when the external bus interfaces were changed. The verification of COP instruction set was done in the legacy verification environment, but bus interface functionality was verified in SoC top level verification for that specific SoC.

Nokia Co-processor needed proper verification environment before it can be taken into new designs with confidence.

The new UVM based verification environment can be seen as an enabler for many activities:

1. Usage in new SoC designs
    - COP can be taken into use for new SoC's with confidence
2. COP high-level programming language compiler development (output assembly language and binary validity)

3. Auxiliary unit development
4. Further development of the COP
5. Software development
    - There was no environment available for COP software development
    - COP disassembler can be used for debugging

COP verification environment was introduced in chapter 5. The environment has lots of verification IP's (VIP) and some of them was third party VIP's. Many of the parts was developed for this particular verification environment:

- COP C++ reference model
- DPI taken into use with the reference model
- COP instruction generator
- Auxiliary unit VIP
- COP assertion module

The COP verification environment is completed and lots of test cases have been done. The verification is not completely done yet, though, but the main features have been verified. More features must be covered in order to get verification closure.

Most important result of having the verification environment for the COP is that Nokia Co-processor has been taken into use for new SoC designs and the development around COP is currently very active. Other results are shown in Chapter 6.

7.1 Further Development

The COP development and development activities around COP are currently very active. Thus, there are new features to be added to the verification environment.

One "improvement" would be replacing the Synopsys' AMBA suite VIPs completely with Nokia's in-house VIPs. All the needed in-house developed VIPs were not available at the time when the COP test environment project was started. Now all Synopsys VIP's used in COP verification have an alternative implementation which doesn't require Synopsys' licenses, making verification environment costs smaller.

There are still many test cases to be done to cover all of the functionality of the COP before verification closure can be done.

It would be good to study other DPI use cases. It is possible to use shared code at driver software development and prototyping. Another DPI use case can be at SoC top level verification. It might be possibile to speed up simulation by executing co-simulator tests with DPI interface and UVM bus master VIP so processor model is not needed.

REFERENCES

[1] K. Albin, "Nuts and Bolts of Core and SoC Verification," 2000. [Online]. Available: https://www.cs.york.ac.uk/rts/docs/SIGDA-Compendium-1994-2004/papers/2001/dac01/pdffiles/16_2.pdf

[2] Tampere University of Technology. TKT-1426 Lecture 3: FPGA and ASIC. (2010). [Online]. Available: http://www.tkt.cs.tut.fi/kurssit/1426/S10/Lectures/TKT-1426_lect_4b.pdf

[3] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip Verification.* Kluwer Academic Publishers, 2001.

[4] M. J. S. Smith, *Application-Specific Integrated Circuits.* Addison-Wesley, 2005.

[5] IBM. IBM Global Business Services. The perfect product launch. (2006). [Online]. Available: http://www-935.ibm.com/services/at/bcs/pdf/scm-perfect-prod-launch.pdf

[6] S. Rahkonen, "Mutation-Based Qualification of Module Verification Environments," Master's thesis, Tampere University of Technology, 2016. [Online]. Available: http://urn.fi/URN:NBN:fi:tty-201512111824

[7] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2009*, pp. 1–1285, Dec 2009.

[8] Accellera. Accellera Systems Initiative Inc. Universal Verification Methodology (UVM) 1.2 Class Reference. (2014, 6). [Online]. Available: http://accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf

[9] "IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows," *IEEE Std 1685-2014*, 2014.

[10] M. Becvar and G. Tumbush, "Design and Verification of an Image Processing CPU using UVM," 2013. [Online]. Available: http://www.tumbush.com/published_papers/DVCon_13_Tumbush_paper_final.pdf

[11] A. G. K. Kamath, "Automatic Verification of Microprocessor designs using Random Simulation," 2012, Master's Thesis, Uppsala University. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-179273

[12] T. R. Nicely. Pentium FDIV flaw. (2011, 9). [Online]. Available: http://www.trnicely.net/pentbug/pentbug.html

[13] S. Billava and S. M. Hondwadkar, “A framework for verification of Program Control Unit of VLIW processors,” 2014. [Online]. Available: https://dvcon-india.org/sites/dvcon-india.org/files/archive/2014/proceedings/dv-papers/D2M2-3-1-DV_Framework_for_Verification_VLIW_Paper.pdf

[14] J. Berezinski. Northern Illinois University. System architecture 463 - lecture notes - chapters 4 & 6 - instruction set architecture - risc/cisc. (2016). [Online]. Available: http://faculty.cs.niu.edu/~berezin/463/lec/05/risc01.html

[15] C. Chen, G. Novick, and K. Shimano. Stanford University. RISC vs. CISC. [Online]. Available: http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/