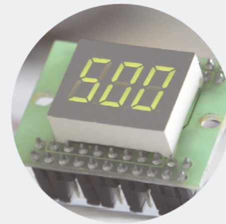
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

isep



Development of a DPI-C Test Environment

ORLANDO PEDRO CARDOSO VIEIRA

novembro de 2018

POLITÉCNICO
DO PORTO

Development of a DPI-C Test Environment

Orlando Pedro Cardoso Vieira

1101421

Tese/Dissertação

Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

2018

This paper satisfies the requirements present on the Subject Sheet of Tese/Dissertação, of the 2nd year of Electrical and Computer Engineering Master Degree Branch of Autonomous Systems

Candidate: Orlando Pedro Cardoso Vieira, N 1101421, 1101421@isep.ipp.pt

Scientific Orientation: Professor Manuel Gericota, mgg@isep.ipp.pt

Company: Synopsys

Supervision: Eng. Rui Manuel de Sousa Ferreira, rui.ferreira@synopsys.com

Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

October 25, 2018

Acknowledgments

Todo o meu percurso académico no Instituto Superior de Engenharia do Porto foi rico em experiencias interessantes e desafiadoras, desde pequenos projetos ao longo do ano, Estágio/Tese e até Erasmus. Todas essas experiências, colegas e professores com que me cruzei, contribuíram para a pessoa que sou hoje.

Para esta Tese, tenho de começar por agradecer ao meu orientador, o Professor Manuel Gericota, por ter sido um elemento presente e indispensável para a conclusão desta etapa. Tenho também de agradecer por ter começado o interesse pelo *Digital Hardware Design* que foi decisivo no rumo da minha carreira profissional. Pela sua paciência, insistência, apoio, conhecimento e disponibilidade até ao último momento.

Tenho também de agradecer à Synopsys, e em especial ao Eng. Antonio Costa e Eng. Rui Ferreira, pelo apoio, disponibilidade e sobretudo por terem acreditado em mim e me terem dado a oportunidade de realizar o estágio nesta fantástica equipa. A todos os meus colegas da Synopsys, em especial Pedro, Silvio, Terra, Andreia, Sá, Antonio Salazar e os restantes membros da equipa que me ensinaram e ajudaram na conclusão deste projeto e que tornaram a minha integração na equipa descontraída e rápida.

A nível mais pessoal, agradeço aos meus pais e família, por estarem presentes, pela paciência, pelo esforço, sacrifícios e apoio a todos os níveis, desde o inicio desta longa jornada.

A todos os amigos que o ISEP me deu, em especial Pedro, Cavadas, Vitor, Tatiana, João, Esteves e a todos de Electro 010, agradeço a amizade, estudo e aprendizagem, descontração e diversão. Aos meus amigos mais antigos, Diogo, Flávio, Ricardo, César, Carvalho, Alves e à ClockWork Orange por todos os momentos ao longos dos muitos anos e por estarem sempre presentes apesar da distância. E ainda à Ivone, Luisa, Ana, Vitor Soares e Diogo Castro e a todos os amigos de Erasmus que estiveram do meu lado

nos bons e maus momentos.

Por último e mais importante, à Eunice, a minha nova família, o pilar que me mais me apoiou ao longo destes quase 10 anos. Agradeço pelo apoio, motivação, por me ensinar a querer sempre mais e melhor, a não desistir e acreditar em mim mesmo quando nem eu acreditei. Pelo amor, insistência, paciência, compreensão e disponibilidade mesmo quando não tinha tempo para mais nada.

A todos, muito obrigado.

Resumo

Fruto da evolução tecnológica, a integração de sistemas com milhares de milhões de transístores num único circuito integrado conduziu a um aumento da complexidade dos projetos digitais. Consequentemente, o esforço/tempo necessário para o desenvolvimento destes é maior, tal como a probabilidade da existência de erros de *hardware*.

De forma a evitar passar esses erros para o silício, a Synopsys, uma empresa líder em Automação de Projetos Eletrónicos (*Eletronic Design Automation - EDA*), e em IP (*Intellectual Property*), e uma das empresas responsáveis por esta evolução tecnológica, tem uma equipa de *IP Prototyping Kit* (IPK), que cria protótipos rápidos dos circuitos IP em sistemas baseados em *Field Programmable Gate Arrays* (FPGA) e outras plataformas.

Durante a integração do sistema em FPGA são usadas simulações / *testbenchs* em *SystemVerilog* (SV) e testadas as plataformas de *hardware* de forma a ajudar no desenvolvimento do projeto e sua depuração.

O IPK permite a integração do *software* e do *hardware* numa etapa inicial do projeto. Contudo, a prototipagem pode ser um processo complicado e demorado. Para auxiliar os engenheiros a acelerar o desenvolvimento, foi proposta a criação de um ambiente de teste baseado em C, batizado de DPI-C (*Direct Programming Interface* em C), compatível com *SystemVerilog* e com as plataformas de *hardware* para testes.

Este ambiente, também chamado de *C-Tests*, usa o mesmo caso de estudo como parâmetro de entrada para as duas plataformas. O objetivo é obter o mesmo resultado em ambas, facilitando a sua comparação e o diagnóstico e resolução de qualquer problema existente.

Palavras-chave

Synopsys, IPK, DPI-C, C-test, *hardware*, validação

This page was intentionally left blank.

Abstract

Due to the technological evolution, chip integration increased, with billions of transistors available in a single chip, and thus the complexity of the digital design. As a consequence, the development effort/time increased and accordingly the possibility of the existence of hardware bugs.

In order to avoid passing these bugs to real silicon hardware, Synopsys, a leading company in Electronic Design Automation (EDA) and Semiconductor IP (Intellectual Property) and one of the responsible companies for this technological evolution, has a IP Prototyping Kit (IPK) team.

The IPK team creates rapid prototypes of these circuits in Field Programmable Gate Arrays (FPGA) systems and other platforms. During system integration in FPGA, it is used simulation/testbenches in SystemVerilog (SV), and tests on real hardware to aid design and debugging.

The IPK allows a software and hardware integration in an early stage of the project. However, the prototyping can be a complicated and slow process. To help engineers to accelerate this development, it was proposed the creation of a C based test environment, named DPI-C (Direct Programming Interface in C), compatible with SystemVerilog and hardware platform tests.

This environment, also known as C-Tests, uses the same test case on both platforms as an input parameter. The aim is to obtain the same results, easing the comparison between them and helping to understand and solving the existing problems.

Keywords

Synopsys, IPK, DPI-C, C-test, hardware, validation

This page was intentionally left blank.

Contents

1 Introduction **1**

1.1 Thesis Context 1

1.2 Objectives 2

1.3 Document Structure 3

1.4 Project Schedule 3

2 Technology Overview **5**

2.1 Verilog Language 6

2.1.1 A Brief History of Verilog 6

2.2 Verilog Language Extensions 7

2.2.1 Program Language Interface 7

2.2.2 SystemVerilog DPI 10

2.3 Random-Access Memory 11

2.3.1 Evolution of DRAM 12

2.4 DDR Protocol 13

2.4.1 DDR SDRAM Architecture 14

3 Synopsys DDR IP solution **17**

3.1 Prototyping team work presentation 17

3.2 DDR uMCTL2 IP Prototyping Kit 18

3.2.1 High-Level Description 19

3.2.2 DDR Memory Subsystem 20

3.2.3 AXI Tunnel & AXI Subsystem 21

3.2.4 HAPS - High-performance ASIC Prototyping Systems 22
3.2.5 ARC AXS101 Software Development Platform 23
3.2.6 DDR Software Tools 23
3.3 Workflow of DDR IPK 26
3.4 Benefits of the Prototyping Kit 28
3.5 Disadvantages/Problems 28

4 Project Development 31
4.1 Project requirements 31
4.2 Test cases 32
4.3 Hardware Abstraction Layer 35
4.4 Simulation 36
4.5 Bare metal and Hardware platforms 37

5 Project Implementation 39
5.1 Background for DPI-C Test Environment 39
5.2 DPI-C Test Environment Architecture 40
5.3 Test Case 43
5.4 C Libraries C-API 47
5.5 Hardware Abstraction Layer 48
5.5.1 Write 51
5.5.2 Read 52
5.5.3 Memory Burst 52
5.6 Bare metal and Hardware platforms 55
5.7 Simulation and DPI-C 59

6 Tests & Results 63
6.1 Tests 64
6.2 Results 65
6.3 Unsolved Problems 67

7 Conclusion and Future Work **69**

This page was intentionally left blank.

List of Figures

2.1 PLI flow 10

2.2 Comparison between SDR and DDR 13

2.3 DDR block Diagram 15

3.1 Memory modules boards available for kits 18

3.2 DDR IPK setup 19

3.3 DDR uMCTL2 IPK Block Diagram 20

3.4 HAPS family of products 22

3.5 DesignWare ARC AXS101 SDP 23

3.6 Segment of log file created by simulation 24

3.7 DDR IPK usage/testing flow 25

3.8 IP Prototyping Kit workflow 27

4.1 High-level flow of the project 32

4.2 Memory initialization flow 34

4.3 Testbench architecture 37

5.1 File tree project directory 41

5.2 Project Organization 42

5.3 Board and configuration defines 43

5.4 MakeFile and options 44

5.5 File tree - C-API 49

5.6 DDR configuration block diagram 56

5.7 DDR configuration block diagram 56

5.8 Example bare metal output log 57
5.9 File tree - bare metal drivers 58
5.10 DDR configuration block diagram 59
5.11 Testbench directory 60

6.1 Testing flow 64
6.2 Testing results from both environments 66
6.3 DDR4 memory test error 67

List of Tables

4.1 C and SystemVerilog compatible variable types [1] 36

5.1 DDR project directory organization 41

5.2 Available user's HAL's functions 49

This page was intentionally left blank.

List of Acronyms

ACC *Access*

AMBA *Advanced Microcontroller Bus Architecture*

ARC *Argonaut RISC Core*

APB *Advanced Peripheral Bus*

ASIC *Application-Specific Integrated Circuit*

AXI *Advanced eXtensible Interface*

BIST *Built-In Self-Test*

BS *Bank Select*

CA *Column Address*

CKE *Clock Enable*

CS *Chip Select*

DCU *DRAM Command Unit*

DDR *Double Data Rate*

DFI *DDR PHY Interface*

DLL *Delay-Locked Loop*

DIMM *Dual In-line Memory Module*

DMA *Direct Memory Access*

DPI *Direct Programming Interface*

DRAM *Dynamic Random Access Memory*

DQ *Data Lines*

DQM *Data Quality Management*

DQS *Data Strobe*

DUT *Design Under Test*

DWC *DesignWare Cores*

ECC *Error Correcting Code*

EDA *Electronic Design Automation*

EEPROM *Electrically Erasable Programmable Read-Only Memory*

FPGA *Field Programmable Gate Array*

HAL *Hardware Abstraction Layer*

HAPS *High-performance ASIC Prototyping Systems*

HDL *Hardware Description Language*

HDMI *High-Definition Multimedia Interface*

HIB *Host Interface Bridge*

HT3 *HapsTrak3*

HVL *Hardware Verification Language*

I2C *Inter-Integrated Circuit*

IDE *Integrated Development Environment*

IEEE *Institute of Electrical and Electronics Engineers*

IP *Intellectual Property*

IPK *IP Prototyping Kits*

IRQ *Interrupt*

ISEP *Instituto Superior de Engenharia do Porto*

JEDEC *Joint Electron Device Engineering Council*

LPDDR *Low Power DDR*

LRM *Language Reference Manual*

MR *Mode Register*

MIPI *Mobile Industry Processor Interface*

OSI *Open System Interconnection*

OVI *Open Verilog International*

P4 *PerForce*

PHY *Physical Layer*

PLI *Programming Language Interface*

PUB *PHY Utility Block*

RA *Row Address*

RAM *Random-Access Memory*

RTL *Register-Transfer Level*

SDP *Software Development Platform*

SDR *Single Data Rate*

SDRAM *Synchronous Dynamic Random-Access Memory*

SoC *System-on-Chip*

SPD *Serial Presence Detect*

SSH *Secure Shell*

SV *SystemVerilog*

TEDI *Tese/Dissertação*

TF *Task/Function*

uMCTL2 *Universal Memory Controller 2*

USB *Universal Serial Bus*

VCS *Verilog Compiled Simulator*

VIP *Verification IP*

VPI *Verilog Procedural Interface*

VTB *Verilog Testbench*

Chapter 1

Introduction

1.1 Thesis Context

This document is referent to the project developed in full collaboration with Synopsys as part of the Tese/Dissertação (*TEDI*) course of the Electrical and Computer Engineering Masters Degree lectured at Instituto Superior de Engenharia do Porto (*ISEP*).

The increasing demand for better and faster electronic equipment motivates the technological companies to a constant development and evolution of their products that lead to a complexity of the digital design.

Synopsys, a leading company in Electronic Design Automation (EDA) and Semiconductor IP (Intellectual Property), is one of the responsible companies for this technological evolution. Synopsys offers its customers products based on a wide range of protocols used in all types of electronic equipment, such as HDMI, MIPI, USB, DDR, PCIe, among others.

One of the main reasons for the success of this company is the quality assurance of all its products. However, the constant technological evolution increases the complexity of its products, making it difficult to assure the same quality without compromising the cost and/or production time.

Since starting silicon production or delivering an IP to a costumer with

hardware problems or defects has highly costs for manufacturers, companies like Synopsys choose to create IP prototypes on Field Programmable Gate Arrays (FPGA) systems.

The prototyping of a system allows the integration of software and hardware at an early stage of the project. In this way, it is possible to verify, test and correct the IP, making it more robust.

For hardware verification it is used simulations in SystemVerilog (SV) and tests on real hardware platforms. Due to the differences of these two test platforms, sometimes the reason for the problem is not so obvious and can make this prototyping process a highly time consuming task.

To shorten the development time, it was proposed to create the DPI-C Test Environment, a C test environment compatible with SystemVerilog and hardware platforms testing.

1.2 Objectives

The main goal is to develop a test environment using the *DPI-C* functionality of the SystemVerilog description language, within the context of the *DDR* (Double Data Rate) Intellectual Property Prototyping Kits (*IPK*).

This test environment allows the use of the same code in a simulation environment (using Synopsys VCS® Tool) as well as its execution on hardware platforms.

To successfully accomplish this goal it is important to achieve some intermediate objectives:

- To learn about the working methods and the tools used by Synopsys;
- To understand Verilog, SystemVerilog and C languages;
- To study and understand the DDR protocol.

Although the DPI-C test environment (referred to as the "C-Tests" throughout this document) was developed and tested using the DDR protocol, the intent is to apply "C-Tests" to every protocol such as HDMI, MIPI,

among others, and ease its development.

1.3 Document Structure

This section serves to explain the structure of this document and the content of each chapter:

- Technology Overview - gives a brief introduction of how memory RAM works, show previous technologies to the DPI-C and other related technologies;

- Synopsys DDR IP solution - introduces the background of this project and its impact in the organization. It also presents the current flow of the Prototyping Kit's development.

- Project Development - Describe the project planning and the DPI-C Test Environment expected structure and features.

- Project Implementation - Describe all project implementation based on the chapter "Project Development"

- Tests & Results - shows how test were executed, what results were expected and problems found.

- Conclusion - provides and overview and discussion of this project.

1.4 Project Schedule

This project had 3 distinct stages:

- Learning and understanding the DDR prototyping kit - consisted in research, read datasheets and other documentation available. By the the end of the stage some changes were made to the DDR Config Tool;
- Planning, discussion and development of the test environment - writing test cases, creating the HAL interface and other needed files;

- Test, debug and verification of the C-Tests in the testbench and bare metal with all types of SDRAM - some adjustments in the project were done in this final stage.

Chapter 2

Technology Overview

Electronic devices are in constant change because companies and final customers are always looking for devices with better performance, user friendly interfaces, more memory capacities and others features. This demand for new products, driven by an exponential increase in integration, opens the possibility of creating more and more complex chips, resulting in increased development efforts/times and consequently increased chances of occurrence of hardware bugs.

To keep up with the market demands, technological companies have to improve the quality of their products in an effective and efficient way. One way to develop easier and faster devices is by designing these circuits in Field Programmable Gate Array (FPGA) systems and other platforms, simulate to verify that the design behaviour is as intended, debug and test those designs before passing them into real silicon hardware.

Despite the fact that there are several approaches to accomplish this goal, and each one has unique strengths and weaknesses, in this chapter is not contemplated every mechanism available in the market. It is only presented some theoretical background about the mechanism used in this project.

2.1 Verilog Language

This just means that, by using a HDL, one can describe any (digital) hardware at any level.

Verilog is an *IEEE* (Institute of Electrical and Electronics Engineers) standard Hardware Description Language (HDL) and is a textual format for describing digital systems, such as a network switch, a microprocessor, a memory or a simple flip-flop. The HDL allows describe any hardware component at any level [2].

Applied to electronic design, Verilog is intended to be used for verification through simulation, for timing analysis, for test analysis and for logic synthesis.

2.1.1 A Brief History of Verilog

The history of the Verilog HDL goes back around 1984, when a company called Gateway Design Automation developed a logic simulator, Verilog-XL, and with it a hardware description language. It is said the original language was designed by taking features from a HDL called HiLo, as well as from computer languages like C [3].

Cadence Design Systems acquired Gateway in 1989 and became the owner of the rights to the language and the simulator. Cadence organized the Open Verilog International (OVI) to release the language (without the simulator) into the public domain, with the intention that it should become a standard, non-proprietary language. With that, Cadence avoided the industry to shift to VHDL and at the same time prevent the language from being changed by companies for their own benefit [4].

Several companies start working on Verilog simulators being the Verilog Compiled Simulator (VCS) from Chronologic Simulation the most successful.

The OVI successfully concluded the work and the standard, which com-

bined both the Verilog language syntax and the PLI (Program Language Interface, see 2.2.1) in a single volume. In 1995, the IEEE approved it and it is now known as IEEE Std. 1364-1995. After that, another significantly revised version was published in 2001 and a further revision in 2005, but this latter one only added a few minor changes [3].

The Verilog is maintained by a non profit making organization, Accellera, which was formed from the merger of OVI and VHDL International.

2.2 Verilog Language Extensions

Between the several reasons of the Verilog language success for hardware design is the existence of extensions and high-level hardware modelling languages that increase the capabilities of commercial Verilog simulators. One of this type of extensions that make Verilog a versatile language is the ability to access to functions and libraries written in C/C++ language.

In this section it is presented more than one way to gain access to C/C++ from a hardware model.

2.2.1 Program Language Interface

The Verilog PLI was the first mechanism to allow Verilog code to call functions written in the C programming language. The PLI has been an integral part of the Verilog language since 1985 [5] [6].

The PLI was developed with two main objectives: offer to Verilog-XL users a mechanism to use the C language for tasks such as file I/O; and at the same time provide a mechanism for ASIC foundries to analyse the usage of their standard cell libraries in order to calculate accurate propagation delays within each cell instance. It can also be applied in engineering environments like commercial and proprietary waveforms viewers, design debug utilities, power analysis, RAM/ROM program loaders, custom file readers

and writers, parallel process distribution, and much more [6].

The first generation of the Verilog PLI is referred to as the Task/Function interface (TF interface) and consists in a set of C language functions defined in a library, also known as TF routines, which allow task/function arguments to be passed to C functions. One of the most important aspects that TF routines offer is the ability to synchronize PLI activity with the simulator's event scheduler. Event synchronization is a key part of many types of PLI applications.

The PLI standard later included another library of C functions know as *access routines* (ACC routines). ACC routines provide access to the structural level of a design and were created by request of ASIC vendors specifically to perform delay calculations, power analysis and other types of analysis involving the cells that make up an ASIC netlist. This also made the ACC library very useful for other types of applications like waveform displays such as VirSim and other graphical design debug utilities.

When Verilog language PLI was released to the public domain, the TF/ACC libraries without any new features, was labelled as "PLI 1.0".

As there were a lot of redundancies and limitations in TF/ACC libraries, OVI replaced those old libraries that combined more than 200 routines by a single library of about 25 routines, with a simple and well defined syntax. With this new library, called "PLI 2.0", OVI intended to completely eliminate the TF and ACC libraries and consequently the associated weaknesses and disadvantages. Although some functions and constant names have remained the same as in the old ACC library, their functionality and values have changed, making PLI 2.0 backward incompatible with PLI 1.0. That cause many PLI applications to not work together with PLI 2.0 and therefore it was not widely adopted by electronic design automation companies in their Verilog simulators.

In 1995, when the first IEEE version of the Verilog standard was re-

leased it included OVI's PLI 1.0 version of PLI 2.0 that was rewritten to be backward compatible. As part of that process, the PLI 2.0 routines were renamed to VPI (an abbreviation for "Verilog Procedural Interface").

How PLI Works

The function invoked in Verilog code is called a system call. By using PLI 1.0 or PLI 2.0, the user can create custom system calls, something that Verilog syntax does not allow. In PLI applications, the C code needs to have access to the internal data structure of the Verilog simulator.

A PLI application, like Verilog testbenches, consists in functions written in C/C++ (the name of those functions should start with $), which are compiled together generating shared libs (*.DLL in Windows and *.so in UNIX). Then the user can call those functions in Verilog code (an example of PLI usage can be seen in the figure 2.1). Depending on the simulator, the user needs to pass the C/C++ function details to it during the compile process of the Verilog code - this action is called linking. Some simulators such as VCS allow static linking.

After completing each step, the user only has to run the simulation like in any other Verilog simulation. During the execution of the simulation, whenever the simulator reaches an user defined system task, the execution control is passed to the C/C++ function. [7]

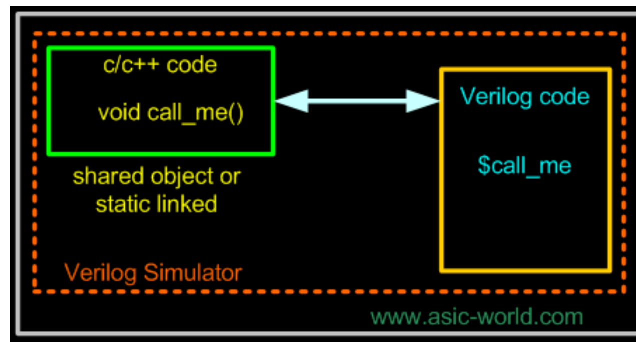


Figure 2.1: PLI flow[7].

2.2.2 SystemVerilog DPI

IEEE expanded the features in Verilog in 2001 but those new features were not good enough for design verification. Therefore engineers had to use two different languages: one for modelling hardware logic and another for verification such as "e", VERA or Testbuilder. [8]

This problem motivated Accellera to developed a single platform for both tasks by combining the verification capabilities of *HVL* with the Verilog simplicity, all that with fewer lines of code. In 2015, this new public standard was named SystemVerilog (*SV*).

Since PLI is part of the Verilog standard, it is also the standard application interface between Verilog and C. However, PLI has some disadvantages, such as being a complex interface that can discourage the learning and sometimes slows down simulation performance. The SystemVerilog Direct Programming Interface (SV DPI) simplified the usage and overcome those disadvantages [9].

DPI allows Verilog code to call the exact names of C functions as if they were tasks or functions defined in the Verilog language. So contrary to PLI, with DPI it is possible to call the function directly from any C library, pass the functions inputs and receive directly the return value of that function. This can be done by simply using an **import** (or export) statement that

contains a prototype of the function name and arguments of the function defined in the DPI interface. Later, the imported C functions are compiled into the Verilog simulator [9].

This Verilog extension include much of the C language, as well as several other features and provides a higher level of modelling abstraction, which enables to model much larger designs and to begin the design process at an architectural level instead of an *RTL* (Register-Transfer Level). SystemVerilog allows the use of C data types such as int, typedef, structs and dynamic data types, classes for object oriented programming, dynamic queues and arrays. It also added new operators and built in methods, semaphores and VPI extensions and reinforced the flow control by using foreach, return, break and continue. Those features makes easier to transfer data between Verilog and C models [8].

All the enhancements provided by SystemVerilog makes possible to design hardware in a high level language. All the similarities between HDL and programming languages narrowed the barrier between software and hardware. This provides flexibility during design and debugging because it is possible to choose how to handle some tasks - by hardware or software.

The most efficient method to integrate HDL-based models and C models is by using Verilog with SV DPI extension. DPI can replace the complex Verilog PLI with a simple and straight forward import/export methodology.

2.3 Random-Access Memory

It is impossible to process anything if it is not possible to store data, whether for human or machine processing. With that in mind we can claim that memory is the core of logic, that is why memory has been an essential component in computer design. One of the types of memory used is a hardware device called Random-Access Memory, or *RAM*.

RAM allows information to be stored and retrieved in a computer. In

this device, the stored information is accessed in a random way and not sequentially, as in a hard drive, for example, which makes the access to the data much faster. However, it is a volatile memory, which means it requires power to maintain the information stored or otherwise all data contained in tRAM is lost.

2.3.1 Evolution of DRAM

In spite of the fact that there were other devices used for main memory functions, it was only in 1970 that the first integrated-circuit RAM chip was commercially available.

The first big evolution was the change from asynchronous to synchronous clocks. Instead of transfer data at any point of the system clock, as in the asynchronous RAM, Synchronous DRAM (SDRAM) send/receive data on a rising or falling edge of the system clock. This change allowed the address and data buses to perform faster than 66-MHz.

In the 90s, most computer systems used one of the first memories that supported synchronous memory architectures: Single Data Rate (SDR) SDRAM. SDR means that the memory only accepts one command and transfers one word of data per clock cycle. The data buses of those chips commonly are 4, 8 or 16 bits wide. However, chips are assembled into 168-pin DIMMs (Dual Inline Memory Module) that enable to write or read 64 bits or 72 bits (if it has ECC - Error-Correcting Code) at the same time, achieving typical clock frequencies of 66, 100 and 133 MHz [10].

In order to take advantage of the high potential bandwidth of DRAM, Double Data Rate (DDR) interface was released in 2000 (see next section 2.4 for more details about DDR) and made SDR virtually obsolete in the computer industry.

Since then the DDR has evolved considerably and other variants were developed, such as the Low Power DDR (LPDDR). The LPDDR, also known

as mobile DDR, is a type of memory used in mobile phones and tablets. It is based on the DDR SDRAM with several changes to reduce overall power consumption. Nowadays, the fifth generation of DDR - DDR5 - is already being developed [11].

2.4 DDR Protocol

Similar to SDR SDRAM, Double Data Rate memories are based on SDRAM design. Most of the addressing and command control interface are similar to the SDR, accepting also those commands once per cycle. However the data is transferred using both rising and falling edges of the clock signal, allowing a rate of twice the clock frequency. A comparison between SDR and DDR data transfer can be seen in figure 2.2.

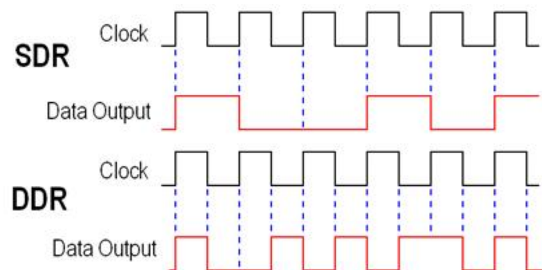


Figure 2.2: The figure shows a comparison on how the data is sampling in SDR and DDR. [12]

Because DDR transfer 2 blocks of data per cycle at the same clock rate, this memories are labelled with double the real maximum clock rate they can actually operate - for example DDR2-800 and DDR3-1066 memories work at 400 MHz and 533 MHz respectively.

This feature can be accomplished by using n-bit prefetch architecture - a prefetch is an array of capacitors where dynamic memories store data [13]. In this case, n bits of data are transferred from the memory cell array to the I/O buffer every clock. The memory cell internal bus has twice the width of

the external bus width and by transferring in both rising and falling edges of the clock (CK) it can achieve a data output rate twice the data rate of the internal bus. As DDR evolves, the prefetch increases and allow DDR memories to work at higher rates than the previous version, for example DDR3 works at higher clock rates than DDR2.

2.4.1 DDR SDRAM Architecture

DDR SDRAM is designed to provide high memory depth and bandwidth. Therefore this architecture is complex and have many different elements [14].

The addressing memory in the DDR SDRAM memories is organized in four group addresses: Chip Select (CS), Bank Select (BS), Row Address (RA), Column Address (CA). Usually there are two memory chips connected in parallel that share address and data lines with a unique chip enable signal each that selects one chip at a time [15].

The Bank Select, Row Address and Column Address are addressing levels within a memory chip. As it is shown in the a block diagram in figure 2.3, DDR SDRAM memory cells are arranged into a two dimensional arrays of rows and columns called banks. To access (read or write) a particular memory cell it is required to specify first the address of the row, also known as page, and then the exact column, isolating the data storage element [16].

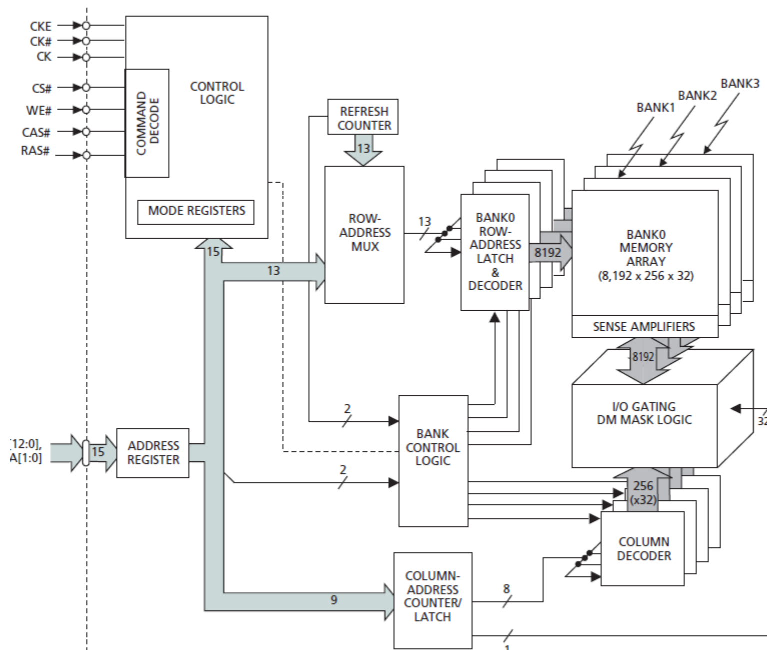


Figure 2.3: Internal DDR SDRAM memory chip block diagram. [16]

Since there are some factors that can limit the maximum attainable frequency, for example, when the data cannot track the clock due to changes in temperature and loading, it is transmitted externally along with the data line (DQ), a bidirectional data strobe (DQS) for data capture, smoothing these limitations and providing high-speed signal integrity. For this purpose is also used a SSTL_2 interface with differential inputs and clocks [15] [17].

This page was intentionally left blank.

Chapter 3

Synopsys DDR IP solution

This chapter introduces the background of this project and its relevance in the organization. It is provided a brief presentation of the Prototyping team work and a high-level description of the DDR IP Prototyping Kit used and of all its essential components.

3.1 Prototyping team work presentation

The DPI-C test environment was proposed by the Prototyping Team. This team is responsible for developing prototyping kits and providing hardware validation of IP titles developed by Synopsys.

Their base work can be subdivided into the following areas:

- Design - this activity produces a synthesizable Verilog code integrating an IP title and PHY (Physical Layer), targeted for a FPGA platform;
- Verification - the design is verified by means of development of a test-bench in Verilog and simulating it on Synopsys VCS;
- FPGA tool flow - consists in using the in-house synthesis tools (Synplify/Protocompiler) aggregated with the platform tools (Xilinx Vivado/ISE) to generate a bitfile;
- Hardware Validation - consists in validate the bitfile on a real hardware platform, using Test Equipment such as scopes, and analysers.

The Prototyping team is working with several protocols and respective kits, such as DDR, HDMI, MIPI and USB.

The DPI-C Test Environment developed can be defined as a verification and hardware validation tool and a proof-of-concept of an incoming flow.

3.2 DDR uMCTL2 IP Prototyping Kit

This project was developed based on the complete DDR IP solution offered by Synopsys: the DesignWare® DDR Universal Memory Controller (uMCTL2) IP Prototyping Kit (IPK).

The DDR uMCTL2 IPK is a hardware and software solution for memory interface. It includes a configurable Verilog source and design files required for FPGA and software with initialization samples in order to reproduce the project. It is compatible with all popular Electronic Design Automation (EDA) and it supports one or more of the broad range of high-performance SDRAMs: DDR3, DDR4, LPDDR3, and LPDDR4 (see figure 3.1).

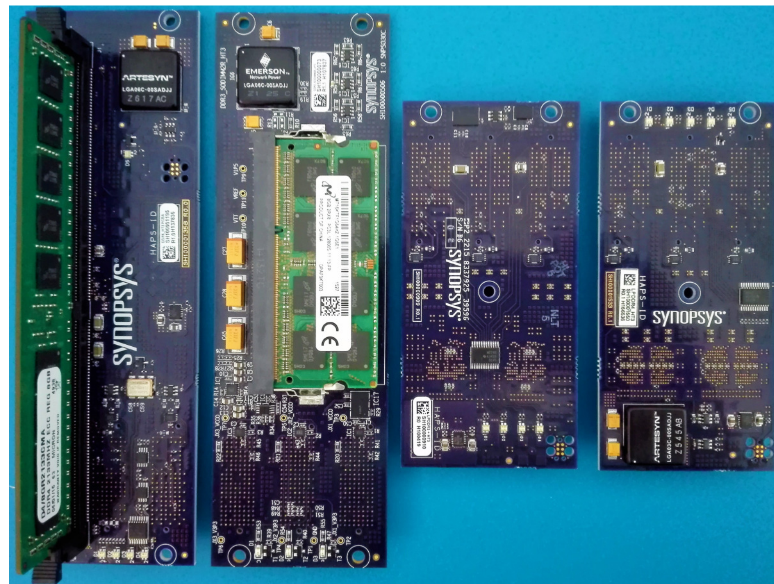


Figure 3.1: Memory modules with HAPS Daugther board available for kits. From left to right: DDR4, DDR3, LPDDR3 and LPDDR4.

The DDR uMCTL2 IPK consists of a validated DDR uMCTL2 configuration, necessary SoC integration, and test logic—implemented in a HAPS FPGA prototyping system (for a more detailed description about HAPS see subsection 3.2.4). The kit, shown in figure 3.2, also includes a SDRAM HAPS Daughter Board, DesignWare ARC processor-based Software Development Platform (SDP) (for a more detailed description about ARC SDP see subsection 3.2.5) and application examples.

The next subsections describe the prototyping kit and some components that are important to understand how the kit works.

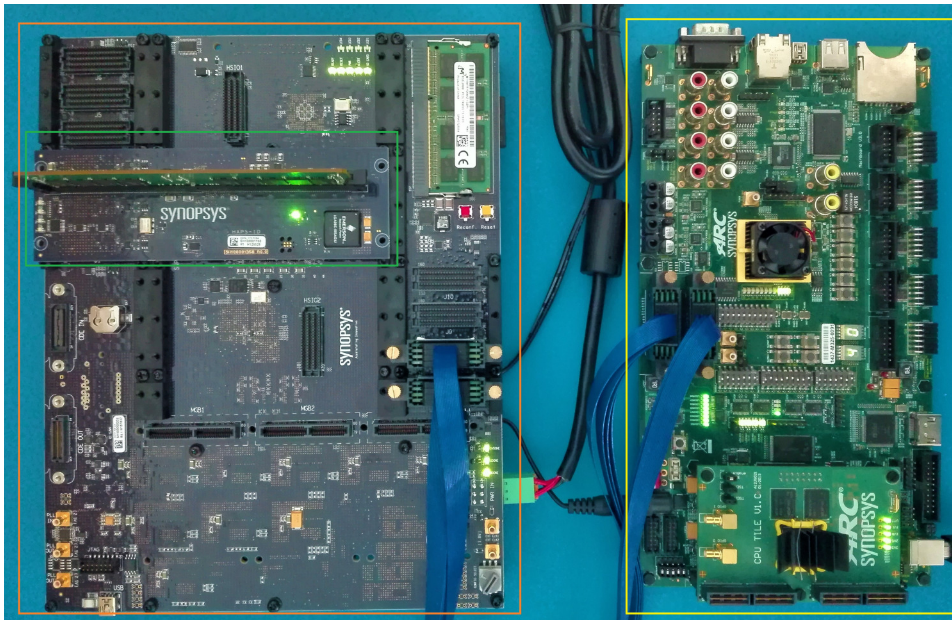


Figure 3.2: DDR IPK setup: HAPS-DX7 FPGA Board (in orange), ARC AXS101 (in yellow) and HAPS DDR4_HT3 Daughter Card (in green).

3.2.1 High-Level Description

Figure 3.3 provides an overview of the DDR uMCTL2 IP Prototyping Kit main blocks in HAPS and ARC AXS101 SDP side, including the DDR subsystem, AXI (Advanced eXtensible Interface) Tunnel and AXI subsystem.

These main blocks are addressed in the next subsections.

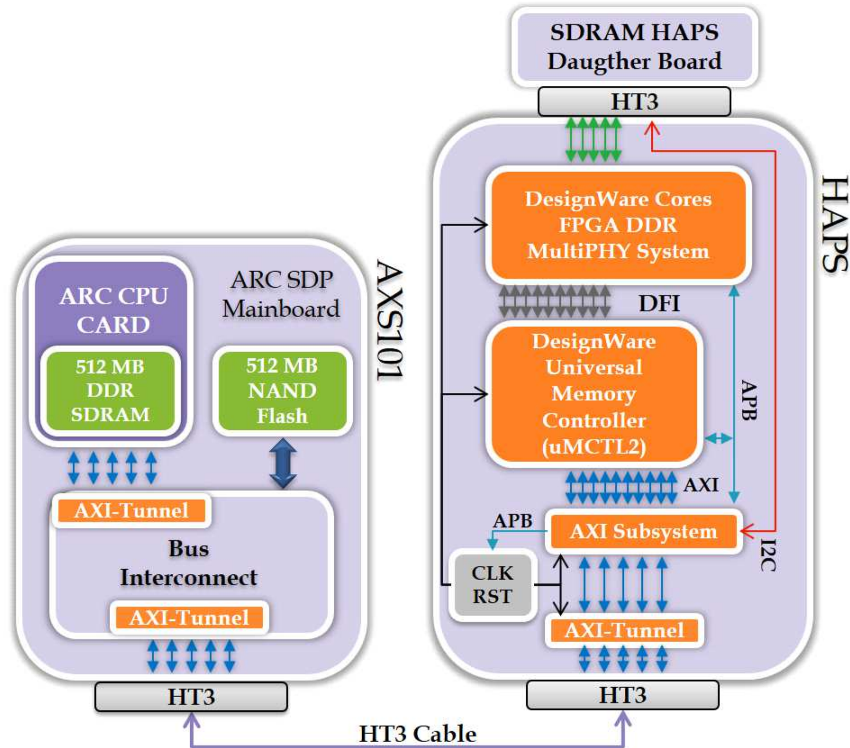


Figure 3.3: DDR uMCTL2 IP Prototyping Kit Block Diagram.

The design used in HAPS FPGA includes instantiations of the corresponding IP, as well as all the complementary blocks for clock, reset and additional logic necessary.

Sometimes it is important to access some relevant interfaces, registers and internal state machine for analysis and debugging, so there is an alternative version of the design described that includes instantiated RTL debugging capabilities for increased observability.

3.2.2 DDR Memory Subsystem

The DDR memory subsystem is the memory interface solution that combines the instantiations of the DesignWare Cores (DWC) Enhanced Univer-

sal DDR Memory Controller (uMCTL2), the DWC FPGA DDR MultiPHY (FPGAPHY), DDR PHY Interface (DFI) and additional support structure.

The uMCTL2 connects to the AXS101 SDP through AMBA elements and to the FPGA PHY through DFI.

PHY stands for physical layer and implements the physical interface between uMCTL2 and the JEDEC standard SDRAM through a DFI bus. The FPGAPHY is a combination of RTL components paired with IO instances from Xilinx FPGA library and it was configured to match the uMCTL2 settings. It also includes Hard Macrocells, PHY Utility Block (PUB) synthesizable RTL block data training, BIST (Built-In Self-Test), PHY configuration registers, and DFI Interface.

The DDR PHY Interface (DFI) is an interface protocol that defines signals, timing and programmable parameters required to transfer control information and data to and from the DRAM devices, and between MicroController and PHY. It is used in several consumer electronics devices, like smartphones, and is applicable to all DRAM protocols including DDR4, DDR3, DDR2, DDR, LPDDR4, LPDDR3, LPDDR2 and LPDDR.

SDRAM HAPS Daughter Board is a board that physically connects through HT3

3.2.3 AXI Tunnel & AXI Subsystem

The main system interface bus for data transfer between the controller and the ARC AXS101 SDP is the AXI interface, constituted by the AXI tunnel and corresponding AXI subsystem.

In order to reduce the pin count in the HapsTrak 3 (HT3) standard connector, both HAPS and ARC SDP have an AXI-Tunnel that physically connects the AMBA (Advanced Microcontroller Bus Architecture) AXI system bus interface. All the necessary logic required to adapt the AXI system bus to the DesignWare IP interface is contained within the AXI subsystem.

The configuration of the DDR uMCTL2 and FPGAPHY blocks is done through an Advanced Peripheral Bus (APB). The AXI and APB buses are versions of ARM Advanced Microcontroller Bus Architecture (AMBA) Specification.

3.2.4 HAPS - High-performance ASIC Prototyping Systems

HAPS is a FPGA-based prototyping solution for ASIC and SoC prototyping that include HAPS hardware components supported by an integrated software tool flow for design planning, FPGA synthesis, and debug.

The HAPS products provide an early hardware and software integration leveraged by design and verification teams to enhance their ASIC design schedules and avoid costly device re-spins.

The available HAPS products are HAPS-80, HAPS-70, HAPS-Developer eXpress (HAPS-DX) (the models are shown in figure 3.4) and a pre-silicon system validation and hardware/software integration called HAPS Proto-Compiler [18].

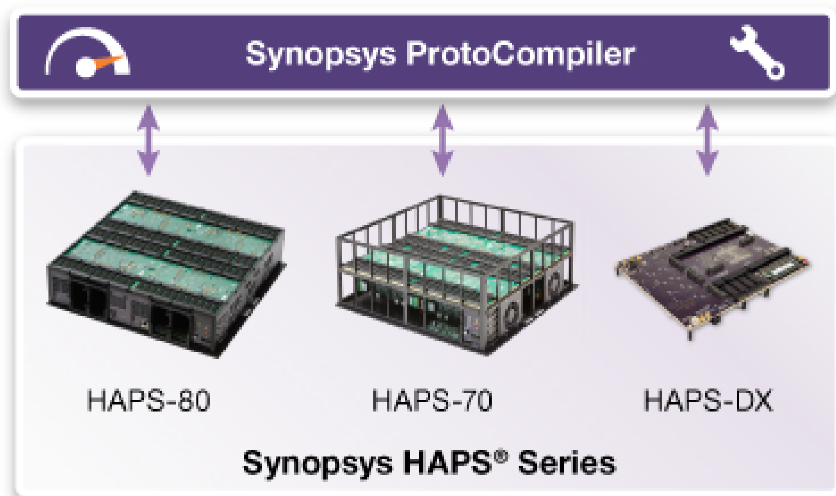


Figure 3.4: HAPS products available in the market.

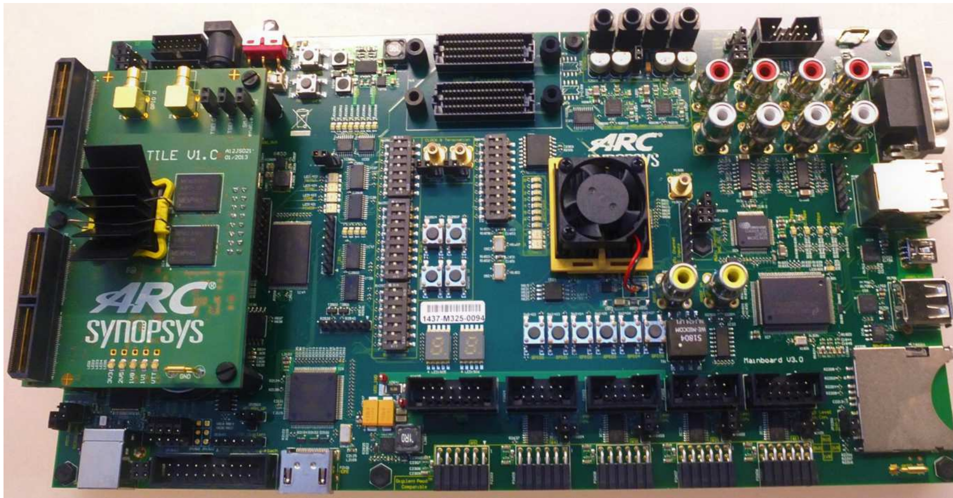


Figure 3.5: DesignWare ARC AXS101 Software Development Platform.

3.2.5 ARC AXS101 Software Development Platform

The DesignWare ARC AXS101 Software Development Platform is a high-speed development platform for software development and validation, debugging, code porting and system analysis.

The ARC AXS101 SDP package is composed by an AXC001 CPU Card mounted in the ARC Mainboard (see figure 3.5) that provides connectivity between the CPU Card and the main infrastructure through an 18-pin header for power supply and two HapsTrak-III connectors for a fast AXI tunnel operating at up to 150 MHz. The HT3 connectors also supply additional clocks, data, and control signals and stablish the connection with peripheral subsystem implemented in the FPGA on the ARC Mainboard.

The platform contains a wide group of interfaces such as audio, USB 2.0 host, HDMI, Ethernet, and several serial protocols.

3.2.6 DDR Software Tools

The software included within the DDR Prototyping Kit is a Linux environment that contains three tools - DDR Initialization Tool, DDR Test Tool

and DDR Configuration Tool - and log files for each type of SDRAM.

The log files are the simulation reports generated by the simulation in VCS and contain all executed operations, registers written/read and respective data and timings. It describes the validated initialization/configuration sequence used in the simulation.

Since there are two ways to initialize the SDRAM module - one through controller and other through PHY - there are two logs for each memory type.

The log file is organized by tasks/sections such as "PHY Configuration", "uMCTL2 DFI Configuration" or "DQ Training". Figure 3.6 shows a small part of a log file of the simulation of DDR3 initialized through uMCTL2. As it is shown, each section has all the operations needed to accomplish its propose. Each line contains the instruction (read or write) to execute, the register's name and address and the value that should be written.

```
    [WRITE]   ADDR_PLAYBACK_ENGINE_CODE_0   ADDR=dxxxxxxx   WDATA=xxxxxxxx
    [WRITE]   ADDR_PLAYBACK_ENGINE_CODE_1   ADDR=dxxxxxxx   WDATA=xxxxxxxx
    [WRITE]   ADDR_PLAYBACK_ENGINE_CODE_2   ADDR=dxxxxxxx   WDATA=xxxxxxxx
    [WRITE]   ADDR_PLAYBACK_ENGINE_CODE_3   ADDR=dxxxxxxx   WDATA=xxxxxxxx
    [WRITE]   ADDR_PLAYBACK_ENGINE_CODE_4   ADDR=dxxxxxxx   WDATA=xxxxxxxx
PHY configuration : End ...

uMCTL2 DFI Configuration : Start ...
    [WRITE]                       DFIMISC   ADDR=dxxxxxxx   WDATA=xxxxxxxx
    [WRITE]                       DFITMG0   ADDR=dxxxxxxx   WDATA=xxxxxxxx
    [WRITE]                       DFITMG1   ADDR=dxxxxxxx   WDATA=xxxxxxxx
    [WRITE]                     DFILPCFG0   ADDR=dxxxxxxx   WDATA=xxxxxxxx
    [WRITE]                     DFILPCFG1   ADDR=dxxxxxxx   WDATA=xxxxxxxx
    [WRITE]                       DFIUPD0   ADDR=dxxxxxxx   WDATA=xxxxxxxx
    [WRITE]                       DFIUPD1   ADDR=dxxxxxxx   WDATA=xxxxxxxx
    [WRITE]                       DFIUPD2   ADDR=dxxxxxxx   WDATA=xxxxxxxx
uMCTL2 DFI Configuration : End ...
```

Figure 3.6: Segment of the log file created by simulation of DDR3 initialized through uMCTL2. Each line contains the operation to be executed, the register name and address, and the value written/read.

The DDR Initialization Tool is a simple and faster option to validate, on a hardware platform, the sequence verified in simulation. It configures the DDR controller, FPGA DDR PHY and runs a very simple memory test to

ensure that the system is correctly configured and initialized. To accomplish this, it uses simulation logs to retrieve software register information and configures a different initialization sequence for uMCTL2 and FPGAPHY.

This tool requires the execution of the DDR Test Tool in order to fully test the DDR subsystem implemented on FPGA. To configure a new system initialization, the user just needs to replace the provided logs by logs corresponding to the new configuration validated by simulation.

The DDR Configuration Tool is a more reliable, and therefore complex, tool. It combines the purpose of both tools, the DDR Initialization Tool and DDR Test Tool, but instead of a log file, it requires INI files as it is shown in figure 3.7. The INI files result from parsing the simulation log by using a script in python that extracts the data of each section of the log and organizes it into different INI files.

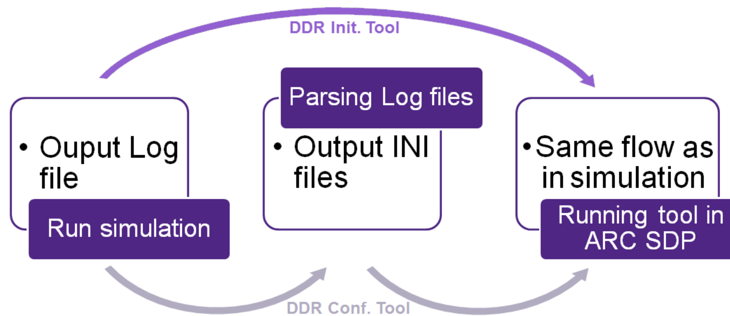


Figure 3.7: DDR IP Prototyping Kit software usage/testing flow for both tools.

For example, the section "uMCTL2 DFI Configuration" shown in figure 3.6 is included in a file named "init_ctrldfi_dwc_fpgaphy_ddr3.ini" and each line has a structure similar to "WRITE UMCTL2 DFIMISC 0x0xxx 0x00000000". If all INI files for all types of memory are in the same folder, the user just needs to run the tool and specify the path to the INI files and the type of initialization (uMCTL2 or PHY).

The tool starts by reading the Serial Presence Detect (SPD) form EEP-

ROM, connected through an I2C bus, to retrieve information about the memory module - such as memory type, number of ranks and memory timings. Depending on the information obtained, it runs the proper predefined sequence, based on the data in the INI files, in order to create, set and verify the configuration of the DDR controller and FPGA DDR PHY, and perform memory initialization. At the end, it runs several hardware tests to verify system functionality. Those tests consist in read/write tests using incremental sequences or patterns - to the memory module on HAPS and ARC platform.

3.3 Workflow of DDR IPK

At a high-level, the workflow is common to all prototyping kits, including the DDR IP Prototyping Kit. The workflow followed is represented in figure 3.8 and explained throughout this section.

The development is done by two distinct teams, Hardware and Software. The hardware team should start their work by setting up all environment variables and build the FPGA design for the given IP. The FPGA synthesis alone can take more than one hour and based on configuration options it updates the necessary files, updating the FPGA image build and simulation flows.

The next step is to create a testbench with different test cases that should cover all features required and use them as input to the simulation in order to validate the design and IP configuration. The simulation time may vary depending on the kit. For the DDR Prototyping Kit, the duration is 2 to 4 minutes.

At the same time, the software team should configure the Buildroot - a set of Makefiles and patches that offers a simple and efficient tool to generate embedded Linux systems through cross-compilation - for the given platform. It should create the drivers and tools needed to use the IP. The tool used in

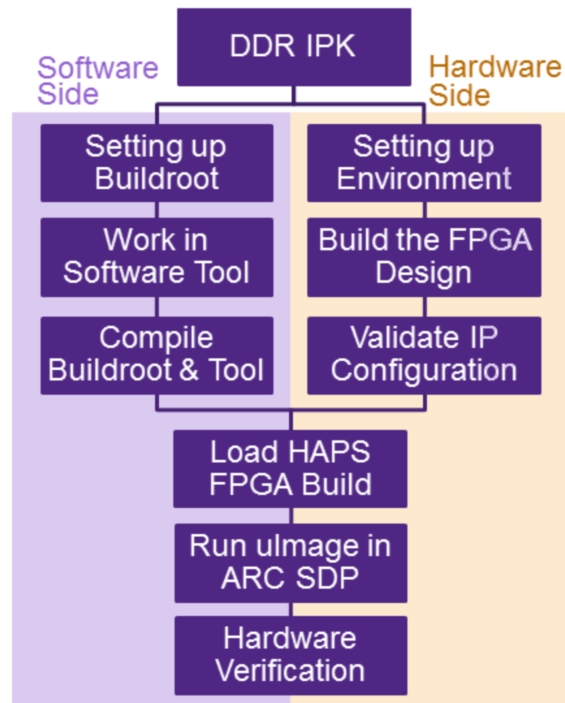


Figure 3.8: IP Prototyping Kit development workflow.

the DDR kit is addressed in subsection 3.2.6.

When the design and IP configuration is approved in simulation, the Buildroot and tool are compiled together to generate the uImage Linux that should be used in the ARC SDP. When the design is synthesized, it generates a bitfile that is loaded to the HAPS FPGA. The ARC SDP boots the uImage with the tool that performs the configuration and tests, verifying that the real hardware platform works as expected.

This method of work requires both teams to be synchronized and in constant contact, since they are dependent on each other work to accomplish most of their individual goals.

The tools used in the DDR IPK (explained in subsection 3.2.6) help the hardware and software teams to keep track of the changes in the workflow, allowing the hardware team to provide similar operations sequence to the

tool in the Linux environment. In that way, the developer can compare the results from the simulation with the results of the hardware platform, analyze signals from both sides and debug, in case of any error occurs.

3.4 Benefits of the Prototyping Kit

The Prototyping Kit was designed to quicken the integration of the IP within a SoC, optimize the IP configuration and develop drivers and software applications.

This kit allows hardware engineers to validate IP configurations and explore design trade-offs for an application. It accelerates the bring-up of the design and the availability of a prototype for software development.

Software developers can speed up debugging and system integration of firmware and device drivers using the provided sample drivers and applications in a software development platform with debugging capabilities, and hardware signal observability. Moreover, developers do not need an in-depth understanding of the hardware when developing applications or drivers.

Finally, SoC integration teams can reduce their integration risk by using a proven controller and PHY with interoperability and reference design.

3.5 Disadvantages/Problems

In spite of the fact that the Prototyping Kit has some strong advantages, it also has flaws.

The workflow used as part of the development of the prototyping kits, depicted in the section 3.3, even with the improvements done in the tools in the DDR IPK, it is not good enough for debugging and validation.

The development of the design and software are extremely time-consuming tasks, plus the time required for debugging, synthesis, simulation and testing. Altogether, it can take days, weeks or even months until the prototyping

kit is ready for shipping.

The simulation module used in VCS sometimes is not exactly the same as the real hardware, therefore its behavior can differ from the real one - for example, the memory module used in VCS is slightly different. Adding the fact that simulation and hardware are not tested strictly in the same way, the developer can obtain different outputs without any error in the tests executed.

Moreover, the goals and tasks performed by the two teams are distinct and sometimes they have different aspects to test and different approaches to do it.

Occasionally, the source of the errors is not clear. In those cases, it can be necessary to use test equipments to understand if it is a software or hardware problem. This type of debugging and test can take hours or even days.

All of these issues can make the development process more difficult, slower and consequently raise the cost of development and the final price of the product.

This page was intentionally left blank.

Chapter 4

Project Development

This section introduces the changes that this project brought to the DDR IP Prototyping Kit to improve it and eliminate the disadvantages referred in the previous chapter (see section 3.5).

Throughout this chapter, it is described the requirements for the new test environment for DDR IPK, a high-level description of the project as a whole and all components needed to achieve the project requirements.

4.1 Project requirements

The focus of this project, as mentioned in chapter 1.2, is to create an interface in C capable of running the same test cases on VCS simulations as on physical hardware. In this way, the hardware engineer can use the exact same data input in simulation/real hardware and if the design is correct, the same output from both can be expected.

Quality assurance in hardware and software requires a wide range of approaches to testing. Although it is not possible to test every aspect of the system, by using dynamic and specific test cases the results can be close to that. The DPI-C test environments should allow engineers to try different test combinations and discover the right test cases for their systems. This

feature should help engineers to overcome the difficulties that exist during development and debug of the design, accelerate its availability for software team and improve the quality of the products.

In order to fulfill this objective and simultaneously make the test environment easily adaptable to other protocols - such as MIPI, HDMI, USB - the C-Test was organized in three distinct parts (see figure 4.1):

- Test cases (see subsection 4.2);
- Hardware Abstraction Layer (HAL) (see subsection 4.3);
- Testbench/Bare metal (see subsections 4.4 and 4.5).

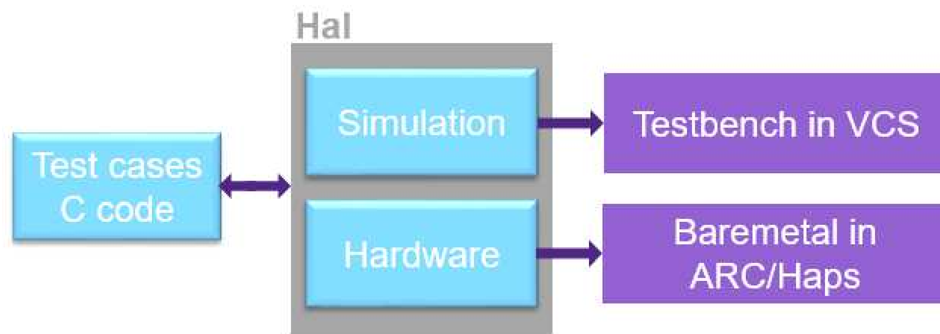


Figure 4.1: High-level flow of the project

4.2 Test cases

A test environment must be structured for debugging and avoid false positives (untested functionalities). To do so, the test must achieve functional coverage, anticipated cases (with error injections), unanticipated cases (random tests) and be robust, reusable, and scalable. The process of verification starts with the definition of verification goals. The best way to reach 100% coverage is to start by doing a preliminary verification, then execute a broad spectrum verification. After the test is stable, it should focus the corner-cases, outside of the normal operation parameters, that has not yet been tested. These steps are described in the test cases.

By definition, a test case is an executable test that analyses a set of inputs, execution conditions and outputs of a system. It verifies a particular feature or functionality and gives a point by point depiction of the steps that should be executed. It can be used as a technical explanation and as a reference guide for systems [19].

In this specific project, all test cases were developed in C language and were based in the existing DDR testbench used in the simulation.

The verification goal for DDR is to test several SDRAM from different brands and types - DDR3, DDR4, LPDDR3 AND LPDDR4. Also, it is required to test different configurations and initialization of PHY, controller, and SDRAM to find out the best initialization sequence for the kit.

To achieve most of the functional coverage, it was discussed and created a test plan with the R&D team responsible for the DDR project. The final version of the test plan was based in the testcases already in use in DDR IPK and should be applied to DDR3, DDR4, LPDDR3 and LPDDR4. It consists in the follow tests:

- Verify if controller, FPGAPHY and SDRAM registers are correctly configured and accessible;
- Test and improve the sequence that initializes the system through the uMCTL2 controller;
- Test and improve the sequence that initializes the system through the FPGAPHY;
- Execute basic tests to SDRAM;
- Execute stress tests to SDRAM.

It is important to verify that all registers exist and can be read to confirm that the setup is well configured and there is no error in any device. The basic test to SDRAM should be a sequence of single write/read operations

to 4 different sections of the memory under test, followed by a burst write to one of those sections and single reads to the addresses of the same section to validate the previously written data. The stress test to SDRAM should be an extensive combination of single write/reads and burst write/reads to several sections of the SDRAM with different sizes. This test should be executed during several hours consecutively.

The testcase for the initialization should be a sequence of write and read operations to FPGAPHY, uMCTL2 controller and the memory module under test. The initialization sequence to be used in the C-Tests is depicted in figure 4.2.

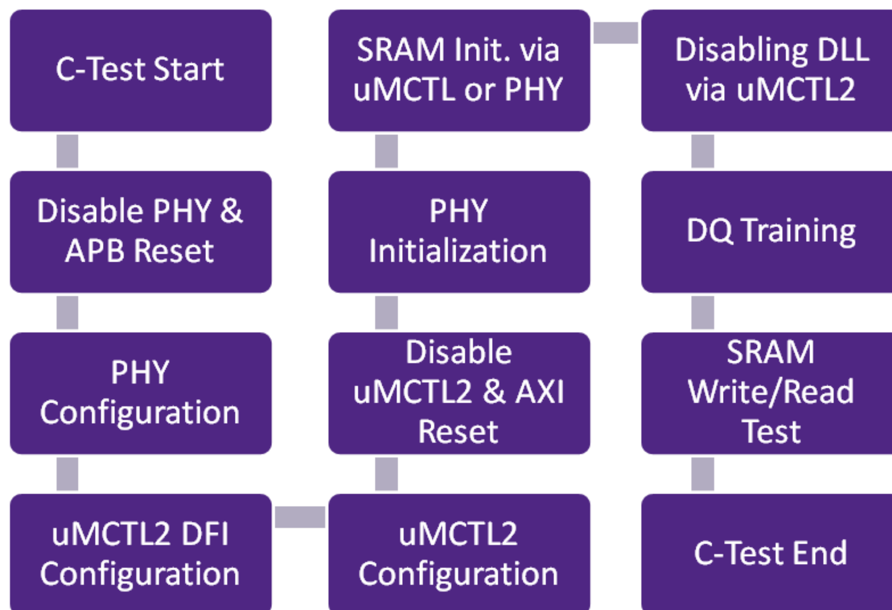


Figure 4.2: Memory initialization flow

4.3 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) is a compilation of generic functions compatible with simulation and hardware platforms. This compatibility is accomplished by using C and the DPI-C functionality of SystemVerilog.

The HAL creates the link between the test cases functions with the tasks and functions used in simulation or in real hardware platforms.

Direct Programming Interface C (DPI-C) allows SystemVerilog to call C/C++ functions just like any other native SV function/task by using *import "DPI-C"* declarations and at the same time allows C/C++ functions to use SV functions/tasks by using *export "DPI"* declarations.

The HAL is implemented in two different files:

- HAL.c – contains the C functions to be used in test cases, including SV task calls.
- HAL.sv – contains the implementation of the SystemVerilog task to be used in HAL.c.

In the following code sample (listing 4.1), it is shown how the functions are structure in the HAL.C. Each different function in the HAL source code is divided, by using C directive ifdef/endif, into VCS tasks (for simulation, see line 2 from code below) and bare metal functions (line 4 from code below).

```
1 void log_msg(char str[256]){
2   #ifdef VCS
3     printf("%s", str);
4   #else
5     DBG_print(str);
6   #endif
7 }
```

Listing 4.1: Example of a function implementation in the HAL.c.

This division allows the pre-processor to remove some unnecessary code depending on which condition is true. In listing 4.1, the function depicted shows a message in simulation if the parameter *VCS* is defined or in bare metal if the same parameter is not defined.

To pass a value through DPI the type definition of SV and C variables must match and this is a responsibility of the user. The types which are compatible in both languages are presented in the table 4.1.

Table 4.1: C and SystemVerilog compatible variable types [1]

C type	SystemVerilog type
char	byte
int	int
long long	longint
short int	shortint
double	real
float	shortreal
void*	chandle
char*	string

4.4 Simulation

The DDR uMCTL2 IPK package provides a verification environment, with the top-level FPGA design (described in section 3.2) as a Design Under Test (DUT).

As depicted in figure 4.3, the simulation design includes the uMCTL2 core instantiation, FPGA DDR MultiPHY (DWC_fpgaphy), DW AXI interconnector components, the memory model to be tested and the testbench components and framework.

The AXI VIP is a bus functional model that replaces the ARC CPU in order to control the cores the IPK elements through the AXI bus.

The existing verification environment is written in Verilog and SystemVerilog. As in the DDR IPK flow, the memory initialization in sim-

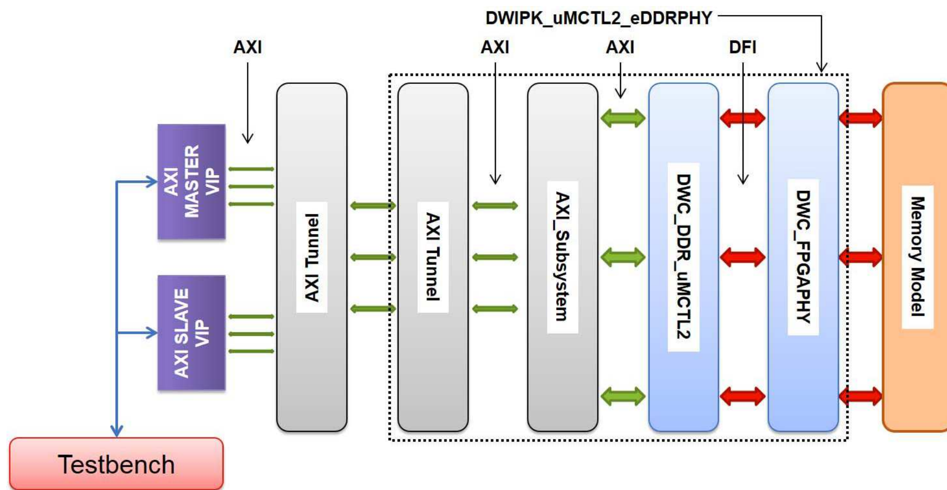


Figure 4.3: Illustrates the testbench architecture of the DDR uMCTL2 IP Prototyping Kit.

ulation can also be performed by the controller or the PHY. The test case used follow the next main steps:

- AXI Tunnel Verification
- I2C Module Verification
- Core and SDRAM Configuration
- Memory Testing
- Result Verification and Error Reporting.

By using DPI-C, testbench variables can be passed directly to/from C/C++ functions and most of the existing simulation code can be reused.

4.5 Bare metal and Hardware platforms

The DDR IPK uses a Linux OS with DDR drivers and applications running in an ARC platform. The software used can take a lot of time to compile and if there is any error it needs to be debugged and recompiled again. For hardware engineers this can be a problem when they need to test their designs. For testing propose, the engineers can use a bare metal application

to run the same test case from simulation and in this way the test process would be faster and easier.

A bare metal environment is a computer system or network installed directly on hardware rather than within the host operating system. The term "bare metal" refers to a hard disk on which a computer's OS is installed. In this project, it means that the test case is executed directly in the ARC processor, without the need of any operating system.

The AXC001 CPU Card, that is part of the DesignWare ARC AXS101 Software Development Platform (see section 3.2.5 for more details) used in this project, supports bare metal applications thanks to the drivers included in the package. The package also contains bare metal example applications, peripheral drivers, and the corresponding makefiles.

To use the bare metal package is advised to installed and use the Metaware toolchain (compiler/linker/debugger), a tool based on Eclipse, developed by Synopsys. It allows the development, debugging and compilation of the application. It supports all processor from ARC family with several configurations for the ARC and other extensions. The possibility to use the IDE and command line to execute every task needed makes this a complete tool for development of drivers and applications.

Chapter 5

Project Implementation

In this chapter is described the behaviour and structure details of the test environment and how it was developed. It is explained the changes done in the existing workflow and tools used in the DDR IP Prototyping kit.

5.1 Background for DPI-C Test Environment

Through the years, the DDR IP Prototyping kit had multiples PHYs such as DDRMPHY, GEN2PHY, LPDDR4MPHY, depending on which SDRAM were being used. Since DDR IPK was expanding, to maintain those solutions was no longer viable and took too much effort to support. The best option was the implementation of a PHY in a FPGA, named DWC FPGA DDR MultiPHY (FPGAPHY), compatible with all SDRAM under test. By using FPGA, the team could make a quick update or change as needed and easily test it. The problem with the FPGAPHY implementation was that after upgrading the hardware in the DDR IPK there was no software to support it.

So before implementing the DPI-C in DDR IPK, there was a need to update the existing software to support FPGAPHY. To do so, it was required to understand the hardware, the DDR IPK flow and software (already de-

scribed in section 3.2).

As described in section 3.2.6, the input of DDR Tool is the output log files of the simulation adapted to the real hardware. Since the FPGAPHY was already running in simulation, the first task was to update the python script that parsed the log files into *ini* files and organized the information to meet the DDR Tool's requirements.

After the support for the new feature was added to the ddr_cfg_tool, it was tested in real hardware. These tests revealed that even with the tool changes and improvements, it still had some bugs and was too complex and slow since it was designed to support several PHYs. After the simulation and after passing all tests this task was consider finished and the development of the DPI-C environment started.

All this work allowed to gather enough information and knowledge about the DDR IPK to ease the implementation of the DPI-C as depicted in the following sections.

5.2 DPI-C Test Environment Architecture

The main objective was to create a test environment that complemented the existing IPK flow, therefore the file structure and content for DDR IPK was kept unchanged and a C-Test folder with all data necessary for DPI-C Test Environment was added to the directory.

In order to understand where the DPI-C Test environment fits in the DDR project, figure 5.1 and table 5.1 describe the original DDR project directory plus the *c_test* folder were the DPI-C Test Environment was stored.

From the original DDR project directory, only the content of the *test* folder should not be used since the testbench and simulation were included in the c-tests. To create the synthesis environment, the Hardware/R&D Engineer still needs to use the remaining information located in the *fpga* folder, the RTL files of the design written in Verilog in *src* folder, scripts

for environment and controller configuration located in *config*, *config_vp* and *script* folders.

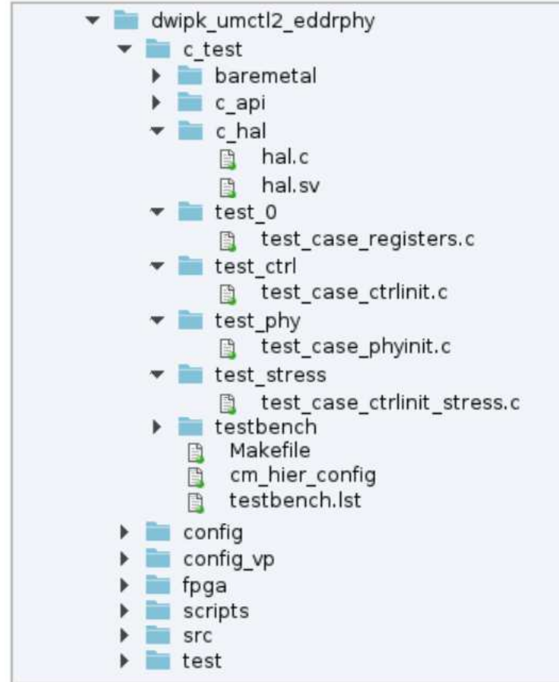


Figure 5.1: File tree from project directory

Table 5.1: DDR project directory organization

Folder	Content Description
c_test	DPI-C Test Environment - testbench and bare metal files
config	DWC uMCTL2 core configuration for HAPS system.
config_vp	Configuration files to create the workspace through coreConsultant.
fpga	Synthesis file for DDR design - constraints files, pin mapping files
scripts	Scripts to setup the environment, load tools and variables
src	Design files (RTL) related to the subsystem
test	Testbench and simulation files, log files and simulation waveforms

As shown in figure 5.1, the *c_test* directory contains several folders. Each of these subdirectories will be discussed in this chapter in the homonym subsection.

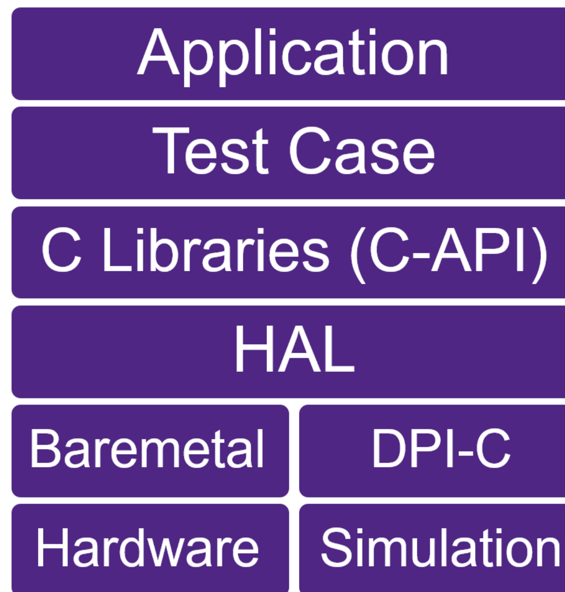


Figure 5.2: Project Organization

One of the most important requirements of the DPI-C Test Environment is that it needs to be protocol independent and easily applicable to all. To accomplish this requirement, the test environment was divided in layers, similar to the Open Systems Interconnection (OSI) model, as shown in figure 5.2. The three higher layers cover the protocol specific scripts and parameters that need to be changed by the user.

The lower layers do not require user intervention and consist in the C-API created and corresponding C Libraries and simulation/hardware design.

The application layer consists in a makefile, auxiliary scripts and files such as the *proto_vars.mk* under *config* directory that contains some input parameters required by the Makefile as is shown in figure 5.3.

The Makefile options are:

- To run the C-Test in simulation environment:
  - *sim* - run without database (no waveforms output file);
  - *dbg* - run with dabatase;

```
1  ##################### Generic #################
2  #---------------------------------
3  # Board and config define
4  #---------------------------------
5  export BOARD            = HAPS_DX7
6  export INTERFACE        = arc
7  export DRAM_TYPE        = ddr3
8  export PHY_TYPE         = fpgaphy
9  export PROTO_NAME       = dwipk_umctl2_eddrphy
10 export TOP_NAME         = ${PROTO_NAME}
11 export CFG_NAME         = DWC_ddr_umctl2
12 export PROTO_PATH       = ..
13 export CORE_ROOT_DIR    = ../../cores
14 export COMMON_PATH      = ../../common
15 export INCLUDE_PATH     = ..
```

Figure 5.3: *proto_vars.mk* and parameters for project configuration

- *cov* - run with full coverage during compilation for debug purpose;

- To run the C-Test in hardware environment:
  - *baremetal* - run the C-Test in the hardware platform. Although this option exists in Makefile, its desirable to run through MetaWare GUI to establish the connection to ARC board;
- To clean the directory:
  - *baremetal_clean* - clean the bare metal directory (baremetal/apps/ddr_case);
  - *clean* - clean simulation database, logs and other files created during simulation.
- *help* - lists descriptions of the most commonly used compile-time and run time options.

Figure 5.4 shows an excerpt of the makefile described above, containing the input parameters require to run the simulation.

5.3 Test Case

The test case is a protocol dependent layer and it requires user intervention and protocol knowledge. For this reason, the test cases created were only

```
#########################################################################
#                        Environment Variables                          #
#########################################################################
include ../scripts/proto vars.mk
#########################################################################
#                           VCS variables                               #
#########################################################################
ifeq ($(strip $(DRAM_TYPE)),ddr3)
export DDR_MODEL_LST=./testbench/tb_hardware/ddr_models/ddr3.lst
endif
ifeq ($(strip $(DRAM_TYPE)),ddr4)
export DDR_MODEL_LST=./testbench/tb_hardware/ddr_models/ddr4.lst
endif
ifeq ($(strip $(DRAM_TYPE)),lpddr3)
export DDR_MODEL_LST=./testbench/tb_hardware/ddr_models/lpddr3.lst
endif
ifeq ($(strip $(DRAM_TYPE)),lpddr4)
export DDR_MODEL_LST=./testbench/tb_hardware/ddr_models/lpddr4.lst
endif

ifeq ($(TC),)
TESTCASE=$(wildcard test_ctrl/*.c)
else
TESTCASE=$(wildcard ${TC}/*.c)
endif
```

Figure 5.4: Excerpt of the Makefile and options available

DDR dedicated.

To maximize the coverage percentage during the test phase it was created and discussed a test plan during the meetings with hardware and software engineers that worked in DDR projects and based on previous tests and new features.

In a high-level, the test flow for DPI-C follows the same flow as in the previous testbench, already described in figure 4.2. The test covers every possible boot configuration and tests several memory blocks in different columns in different ranks and banks of the memory under test.

The test cases were organized in 4 main tests:

- test_0
  - test_case_registers.c - check if every register from PHY and uMCTL2 exists and are readable;
- test_ctrl
  - test_case_ctrlinit.c - runs the flow for initialization through uMCTL2 and a basic test to memory under test;
- test_phy

  - test_case_phyinit.c - runs the flow for initialization through FPGAPHY and a basic test of the memory under test;

- test_stress
  - test_case_ctrlinit_stress_test.c - similar to test_ctrl with additional stress tests.

All the test cases follow the same template shown in the listing 5.1. The example is a shortened version of the test_ctrl and most info, such as registers and data written/read, was edited out. It starts by loading the C header files needed for VCS (simulation) or bare metal. As global function/task to start the program, it will be *test* if it is VCS or *main* if not. Then it proceeds to write to screen a header to the report with a beginning timestamp and it will close the report with a end time stamp so the user can keep track of the execution time of the test. This header also contains the type of test that is being executed based on makefile information, for example, "DDR3 - Initialization through uMCTL2".

```
#ifdef VCS
    #include "../vc_hdrs.h"
#else
    #include "board.h"
    #include "dw_gpio.h"
#endif
#include "../c_api/define.h"
//(...)
#ifdef VCS // in VCS define test function as "test"
void test()
#else
void main(int argc, char **argv)
#endif
{  //Application
  log_msg("\n\r\n\r** DDR TEST - ");
  #ifdef VCS
    log_msg("Simulation/Testbench **\n\r");
```

```
    #else
      log_msg("Baremetal Application **\n\r");
20  #endif
    log_msg("C-Test Start at ");
    log_msg(__DATE__); log_msg(","); log_msg(__TIME__);

    #ifdef ddr3
25    log_msg("DDR3 - Initialization through uMCTL2\n\r");
    #endif
    #ifdef ddr4
      log_msg("DDR4 - Initialization through uMCTL2\n\r");
    #endif
30  #ifdef lpddr3
      log_msg("LPDDR3 - Initialization through uMCTL2\n\r");
    #endif
    #ifdef lpddr4
      log_msg("LPDDR4 - Initialization through uMCTL2\n\r");
35  #endif
    log_msg("\n\tDisable PHY reset\n\r");//previous value 0
    wr_val(ADDR_BASE_RESET_APB+0x0024,0x05,TRACE_DISPLAY);
    hal_sleep(100);
    log_msg("\tDisable APB reset\n\r");//previous value 0
40  wr_val(ADDR_BASE_RESET_APB+0x0020,0x03,TRACE_DISPLAY);
    hal_sleep(100);
    init_PHY();  /** PHY CONFIGURATION **/
    init_uMCTL2();  /** uMCTL2 CONFIGURATION **/
  //  (...)
45  /*************** Memory READ/WRITE ***************/
    // WRITE/READ AXI normal capabilities testing section
     log_msg("\n\tuMCTL2 Write/Read Test: Start ...\n\r");
     mem_test_basic(INST_AXI_UMCTL2_P0);
  //  (...)
50   mem_test_basic(INST_AXI_UMCTL2_P3);
     // WRITE/READ AXI Burst capabilities testing section
     log_msg("\n\tuMCTL2 Write/Read Burst Test\n\r");
     mem_test_burst(INST_AXI_UMCTL2_P0);
```

```
55      sprintf(strng,"\tERROR REPORT:\n\r\tTotal Error(s):%d\n
            \r",err_cnt); log_msg(strng);
        log_msg("\nC-Test End at ");
        log_msg(__DATE__); log_msg(","); log_msg(__TIME__);
        #ifndef VCS
            while(1);
60      #endif
    }
```

Listing 5.1: Excerpt of test_case_ctrlinit.c

5.4 C Libraries C-API

The third layer C Libraries, called C-API, contains the auxiliary C functions and header files needed to run simulations/bare metal applications. The DPI-C environment was built with small C functions such as *init_PHY()* and *init_uMCTL2()* (line 42 and 43 in listening 5.1). These modules help the user to quick and easily create a test case.

Similar to what was done in the test case layer, all the modules and header files used in this layer are specific to the DDR protocol and are stored in the folder *c_api*, as presented in figure 5.5.

Files such as *addr_fpgaphy.h*, *addr_umctl2.h* and *define.h* contain mostly defines to allow the usage of register names instead of addresses or parameter names throughout the DPI-C test environment. This can ease the development and fasten the debug. Instead of:

```
wr_val(0xDF0011b0, 0x00000000, TRACE_DISPLAY);
```

it can be used:

```
wr_val(ADDR_BASE_UMCTL_APB + DFIMISC, 0x00000000, TRACE_DISPLAY);
```

In this way, even if the user does not have the full knowledge about the offset used in the system, s/he will be capable to easily identify that the register DFIMISC of the uMCTL2 will be written.

There are initialization files, specific to the memory type, named **_Initialization.c*, that defines a homonym function that runs the initialization sequence for that memory type. By calling this function, the software will write to several registers of the PHY with a certain order and data.

Included in the *c_api*, there are also functions to run tests to memory modules. The *mem_test_basic.c* define a function to perform single write/read transactions accessing different Rows, Columns, Banks and Ranks. The *mem_test_burst.c* defines a function similar to the one previously described but is capable of running burst write/read operations. It runs three tests with different memory block sizes. Each test writes the memory block with the given start address in single operations. Then it reads the written addresses as a block and writes the same information to a new start address as a block as well. To verify if it was successful, it reads each address and compares the content with the values expected in single operations.

5.5 Hardware Abstraction Layer

The Hardware Abstraction Layer was designed to be a generic interface with the hardware. It is composed by six general functions required by most of the prototyping kits and are described in the table 5.2. Each of these functions is divided in simulation (VCS) and bare metal, as previously described in section 4.3, by using preprocessor defines macros.

The operation "write" means that the data is transferred from the host memory to the DUT and in the operation "read" the data is transferred from the DUT to the host memory. In this project, the DUT is the setup with controller, PHY and memory to be tested, either virtual or implemented on FPGA.

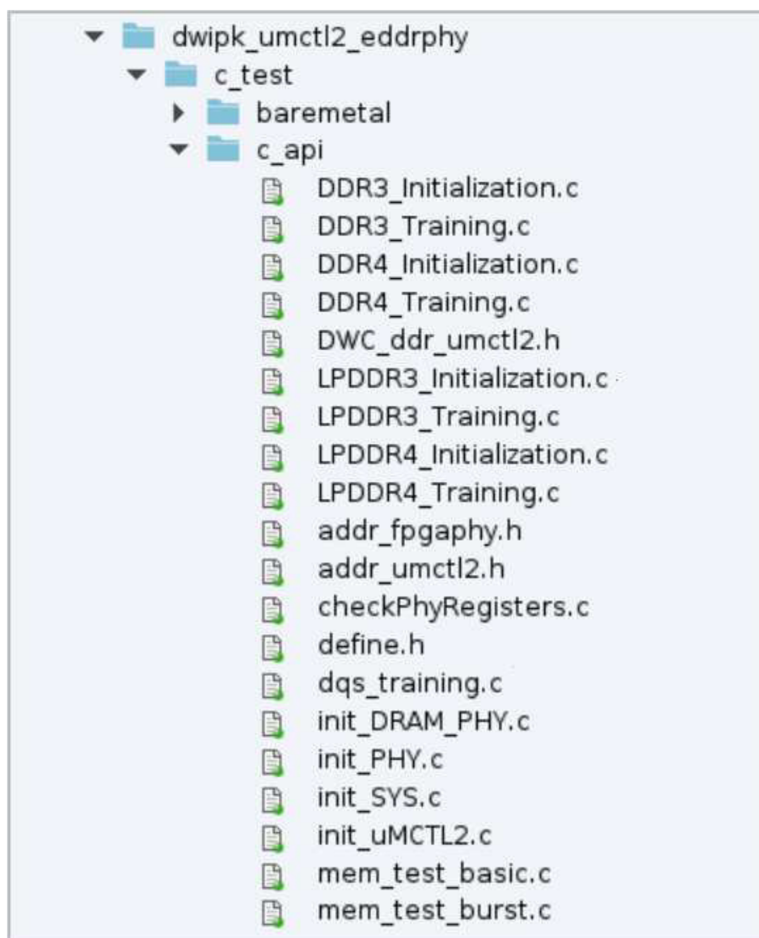


Figure 5.5: File tree from C-API directory

Table 5.2: Available user's HAL's functions

Function name	Description
wr_val	Write single value directly (AXI Master)
rd_val	Read single value directly (AXI Master)
wr_mem	Write block of memory directly (AXI Master)
rd_mem	Read block of memory directly (AXI Master)
hal_sleep	Standardized sleep in nanoseconds
log_msg	Standardized log for developer messages

The code samples in listing 5.1 from *HAL.c* and 5.2 from *HAL.sv* represent the HAL implementation of a delay function in SV and C to show the

interaction between these two layers.

As shown in line 3 and 5 of *HAL.c*, each function in the HAL source code is divided into VCS tasks (simulation) and bare metal functions. This division allows the software compiler to filter the code for each application. It is only possible to call the task *sv_sleep* (line 4) of *HAL.c*, if the task is exported and implemented as in *HAL.sv* code sample.

The line 2 of the *HAL.sv* code exports the task *sv_sleep* and makes it available for usage in the *HAL.c* in line 4.

```
1 //HAL.C
2 void hal_sleep(uint32_t delay_ns){
3   #ifdef VCS
4     sv_sleep(delay_ns);
5   #else
6     uint32_t cnt;
7     cnt = delay_ns/40; // clock 25 MHz -> 40 ns
8     while (cnt > 0){ cnt--; }
9   #endif
10  }
```

Listing 5.2: C function hal_sleep.

```
1 //HAL.SV
2 export "DPI-C" task sv_sleep;
3
4 task sv_sleep(input int delay_ns);
5   #(delay_ns * 1ns);
6 endtask
```

Listing 5.3: SystemVerilog task sv_sleep.

In the following sub sections it is described the write/read functions present in the *HAL.c*.

5.5.1 Write

To send data from the host to the DUT, it should be used the function *wr_val()*, presented on listing 5.4. It requires that the user specify the address and data to be written. In addition, there is the input parameter *u_int8_t level*, that enable/disable the write of a message to the log file with the status of the operation. This parameter is common to all functions in HAL. An example of this messages is the following string:

```
[WRITE]   ADDR = 0xdf004024   WDATA = 0x00000005
```

In VCS environment, it is created a empty memory array that is filled with the data to be written. This, together with the target address, are the input for the SystemVerilog task *wt*.

In bare metal, the function just executes a simple write in C to a specific memory location.

```
static int mem[32*256];
void wr_val(u_int32_t addr,u_int32_t data,u_int8_t level){
  u_int8_t resp=0;
  #ifdef VCS
    memset(mem,0,16*256*sizeof(u_int32_t));  // clean mem
    mem[0] = data;
    wt(addr, mem, sizeof(int)*8,&resp);
  #else
    *((unsigned long *) addr) = data;
  #endif

  sprintf(strng, "\t\t[WRITE] ADDR=%08x  WDATA=%08x\n\r",
      addr, data);
  if(level == 1)//TRACE_DISPLAY
    log_msg(strng);
  check_error(resp);
}
```

Listing 5.4: *wr_val* function that is part of the HAL.

5.5.2 Read

The function shown on listing 5.5, *rd_val()* is responsible for retrieving data from a specific address of the DUT to the host memory. To do that, in VCS it is used the task *rd* from *HAL.sv*. In bare metal is used a similar process to the previous function but in the opposite direction. It reads a single word from a target DUT address and returns it to an address memory in the host side.

```
u_int32_t rd_val(u_int32_t addr,u_int8_t level){
  u_int8_t resp=0;
  memset(mem,0,16*256*sizeof(u_int32_t)); // clean mem
  #ifdef VCS
      rd(addr,mem,sizeof(u_int32_t)*8,&resp);
  #else
      mem[0] = *((u_int32_t *) addr);
  #endif

  sprintf(strng,"\t\t [READ] ADDR=%08x RDATA=%08x\n\r",
      addr, mem[0]);
  if(level == 1) //TRACE_DISPLAY
    log_msg(strng);

  check_error(resp);
  return mem[0];//read_result;
}
```

Listing 5.5: *rd_val* function that is part of the HAL.

5.5.3 Memory Burst

Some protocols, like DDR, support burst transfers. Hence, a requirement for HAL was a burst mode to allow the transfers of full memory blocks of a given size from a memory position to another or between different memories.

To accomplish this, the HAL has two functions:

- *wr_mem()*: executes a burst write from source to destination address (see listing 5.6);
- *rd_mem()*: executes a burst read from the DUT to the host memory (see listing 5.7).

Both function are identical and require the same input parameters. Similar to what is done on the functions *wr_val()* and *rd_val()* previously described, the user should specify the destination address and the address where the data is currently stored. Another required parameter is the *_size* that defines the size of the memory block to be copied.

To write in burst mode in VCS, the functions sets a memory array and uses a for loop to fill it with the data stored in the source address until it completed the number of words defined by the *_size*. Then, the created memory array and the other input parameters are passed to the task *wt* that will use SystemVerilog to copy the data through the AXI Master to the DUT. To read in burst mode the process is similar, but the first step is to use the task *rd* to retrieve the memory block from the DUT and then use the for loop to copy it, word by word, to the destination address.

```
void wr_mem(u_int32_t dst_addr, u_int32_t *src_addr, int
    _size){
  u_int8_t resp=0;
  #ifdef VCS
    memset(mem,0,16*256*sizeof(u_int32_t)); // clean mem
5   u_int32_t cnt;
    for(cnt = 0; cnt < _size; cnt++){
      mem[cnt] = src_addr[cnt];
      hal_sleep(1000);
    }
10  wt(dst_addr, mem, _size*sizeof(int)*8,&resp);
  #else
    memBurst((uint32_t) src_addr, dst_addr, _size);
  #endif
```

```
15    sprintf(strng, "\t\t[DMA-BURST WRITE]\t From = 0x%08x>0x
          %08x \tTo = 0x%08x>0x%08x\n\r", src_addr, src_addr +
          (_size), dst_addr, dst_addr + (_size*4));
      if(level == 1)//TRACE_DISPLAY
        log_msg(strng);
      check_error(resp);
    }
```

Listing 5.6: *wr_val* function that is part of the HAL.

The main IP blocks inside the FPGA are all DMA (Direct Memory Access) capable and do not require an external DMAC (DMA Controller) to transfer data to/from the main memory. However, the serial connectivity peripherals in the peripheral subsystem do not have DMA capability. Therefore, a DMA controller must be provided within the drivers for ARC SDP bare metal.

To use the bare metal write/read in burst mode it is necessary to use the ARC SDP drivers and DMAC. Therefore, it was created another function, named *memBurst*, to execute the data block transfers between memories. This function starts and configure de DMA, prepare the channel and the data block and finally enables the DMA, starting the transfer. It copies a full data block from a given source address to a destination address, either DUT to host or between different positions on the same memory. The *memBurst* is used for both write and read bursts and both the source and destinations addresses are opposites.

It is recommend to copy from a memory position that contain valid data. To guarantee this, the user should write single words to all addresses of the memory block to be copied.

```
void rd_mem(u_int32_t dst_addr,u_int32_t *src_addr,int
    _size){
  u_int8_t resp=0;
  #ifdef VCS
    u_int32_t cnt;
5   memset(mem,0,16*256*sizeof(u_int32_t)); // clean mem
    rd(dst_addr,mem,sizeof(u_int32_t)*8*_size,&resp);
    for(cnt=0;cnt<_size;cnt++)
      src_addr[cnt] = mem[cnt];

10 #else
    memBurst((uint32_t) src_addr, dst_addr, _size);
  #endif
  sprintf(strng, "\t\t[DMA-BURST READ]\t From = 0x%08x>0x
      %08x \tTo = 0x%08x>0x%08x\n\r", src_addr, src_addr +
      (_size), dst_addr, dst_addr + (_size*4));
  if(level == 1)//TRACE_DISPLAY
15   log_msg(strng);
  check_error(resp);
}
```

Listing 5.7: *rd_mem* function that is part of the HAL.

5.6 Bare metal and Hardware platforms

As shown in figure 5.6, the DPI-C is composed by two main parts:

- The Host Processor running bare metal on ARC SDP board by using the DesignWare® ARC® MetaWare Development Toolkit;
- The design under test on fpga platform - HAPS-DX7.

The ARC® MetaWare IDE, shown in figure 5.7, allows to run the test case directly on ARC processor without Linux support as usual. The HAL passes the information through the AXI Tunnel that connects to the Host

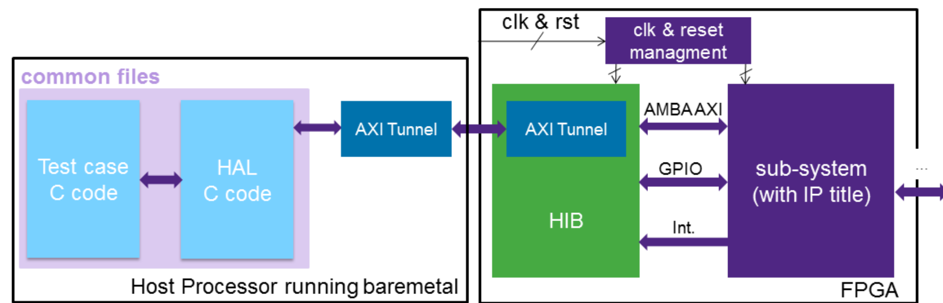


Figure 5.6: Test environment bare metal side

Interface Bridge (HIB) in the DUT with the memory under test. The MetaWare IDE allows the user to debug the application in runtime by using breakpoints or ”run line-by-line” option.

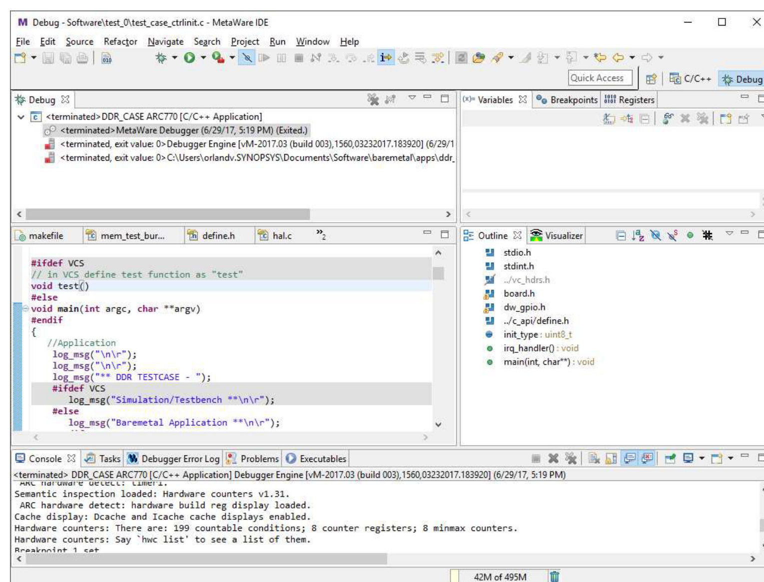


Figure 5.7: Test environment bare metal side

Along with MetaWare, the user also needs to access ARC via SSH by using Putty in order to get the output of the terminal during the executions, shown on figure 5.8. The output of the bare metal application for the same configuration should be the same as the simulation output.

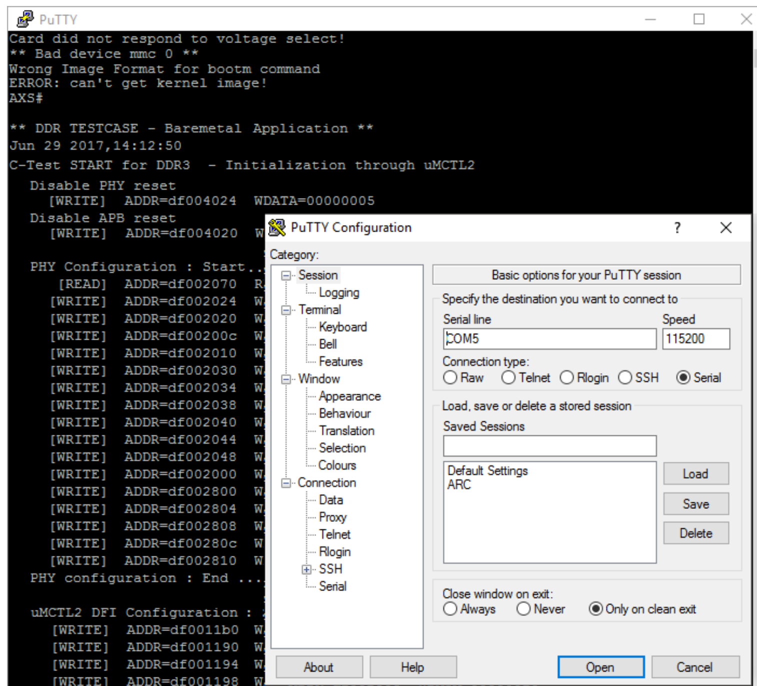


Figure 5.8: Example of a bare metal output log.

Although bare metal does not need Linux OS it still needs drivers to provide a software interface to ARC's hardware. The bare metal drivers used are part of a deliverable software package containing a pre-built Linux image, U-Boot, pre-built MQX Real-Time Operating System (RTOS) binaries, bare metal drivers and application examples that was developed to made easy to use the ARC MetaWare Development Toolkit.

In figure 5.9, it is presented the file tree of the bare metal directory with the following content:

- /apps - contains individual sub-directories for all application examples;
- /board - contains board-specific include files. Particularly, it contains linker files that include the definition of the memory map, located in the /board/axs101/src/ folder. Two linker files are included, map_axs101_ddr.met and map_axs101_ram.met. They are used for building the code for AM or local SRAM respectively;

- /inc - contains a general include file for type definitions;
- /io - contains all basic drivers for the AXS101 SDP, including specific drivers for IP located in the peripheral subsystem on the Mainboard, for peripherals on the AXC001 CPU Card and generic drivers suited for peripherals located on the ARC SDP Mainboard or a CPU Card (or both);
- /project - contains the files related to the "gmake" build flow. In particular rules.mk with all makefile rules and options.mk with more compiler/linker related options;
- /project_arcmw - contains files related to the MetaWare IDE flow.

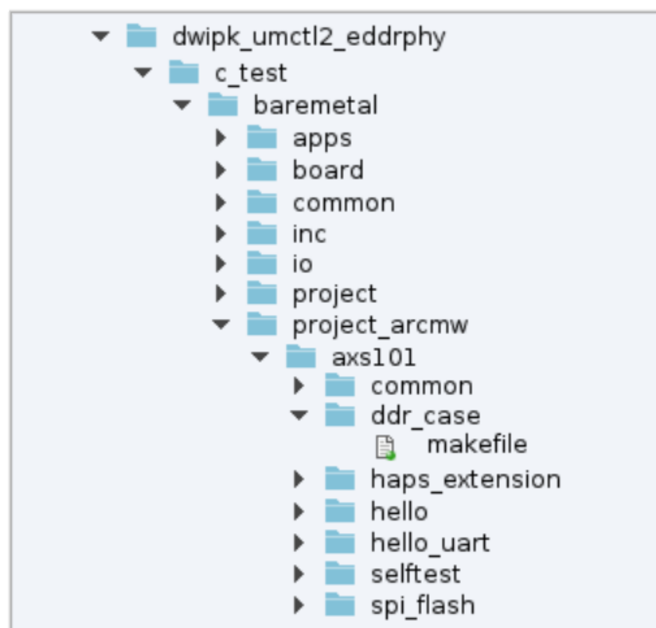


Figure 5.9: Bare metal drivers directory file tree

5.7 Simulation and DPI-C

The purpose of the simulation testbench is to model the FPGA hardware prototyping as closely as possible. In this way, the top-level FPGA design is used as a design under test in the simulation. Therefore, the simulation environment is similar to the bare metal environment described in the previous section 5.6.

Since the DUT is "virtual" and does not require any physical connection to any device, it is not possible to use AXI Tunnel on HIB. Instead it uses a DW AMBA VIP (Synopsys IP) that acts as AXI Bus Master. Another difference that can be seen on diagram of figure 5.10 is that the testbench needs the DPI-C.

The DPI-C is a set of SystemVerilog tasks that acts as a bridge between C functions on *HAL.c* and DUT. These tasks are defined in the *HAL.sv*.

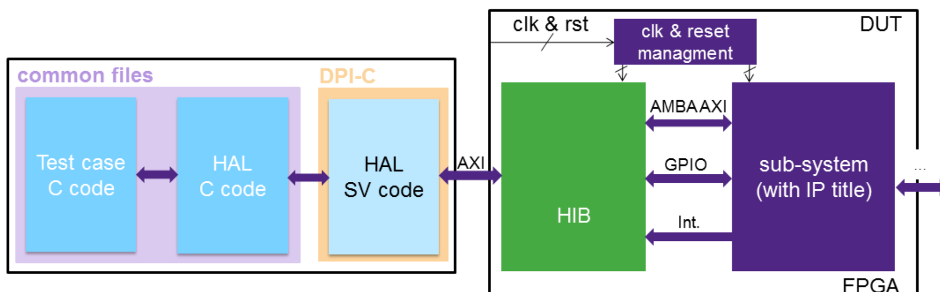


Figure 5.10: Test environment testbench side

The testbench directory (figure 5.11) is organized in the following way:

- config - contains definition of parameters used through testbench flow;
- tb_hardware - contains the digital models (RTL) of DDR3, DDR4, LPDDR3 and LPDDR4. It also contains the Synopsys DesignWare Synthesizable Components DW_axi Master and Slave, for AXI 3 and 4 with differents data sizes;
- tb_instances - contains the instantion of the top-level FPGA design

(*DUT.v*), of AXI Master and Slave, of the DDR model under test and clock generators;

- tb_signals - defines the testbench connections signals used to connect de DUT to DDR and AXI devices.

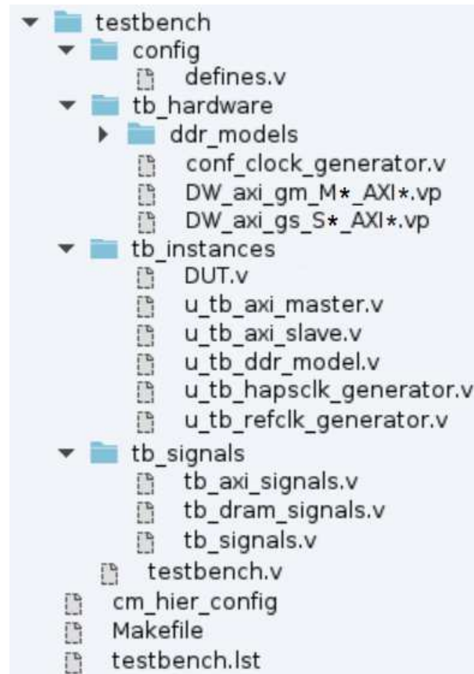


Figure 5.11: Testbench directory content

When the command *make sim* is executed it starts the simulation flow. It begins by sourcing the *testbench.lst* that includes all files from Xilinx modules, testbench, C-API and source files of the design, needed for a successful simulation. Then the *testbench.v* - the main file of this flow - is executed. If the user chooses it, it starts by configuring the waveform dump for debugging purposes. Then it resets the testbench and the whole system and executes the test case, described in section 5.3, by calling the task *test()*.

The simulation outputs a waveform file *test.vpd* and *test.log*. An excerpt of the log file is presented in listing 5.8. It has the same structure of the terminal outputted by the bare metal application on figure 5.8 and if the

simulation is successful it should also have the same content.

```
** DDR TEST - Simulation/Testbench **
C-Test Start at Jul 27 2017,20:02:28

Test  DDR3  - Initialization through uMCTL2
Disable PHY & APB reset
  [WRITE]  ADDR=df004024  WDATA=00000005
  [WRITE]  ADDR=df004020  WDATA=00000003
PHY Configuration : Start ...
  [READ]   ADDR=df002070  RDATA=00000200
//(...)
PHY configuration : End ...
//(...)
uMCTL2 Write/Read Test: Start
MCTL2 Single Write/Read Test

Accessing Memory in Single Transaction - Port 0: Start
  Different columns, Bank 0 ...
  Different columns, Row 1 ...
  Different columns, Rank 1, Bank 8 ...
Memory Write/Read test in Single Transaction : PASSED
//(...)
uMCTL2 Write/Read Burst Test
Accessing Memory in Burst Transaction: Start ...
  [DMA-BURST WRITE] From=0x006deb00>0x006deb40 To=0
      xd0000000>0xd0000040
//(...)
Memory Write/Read test in Burst Transaction : PASSED ...
ERROR REPORT:
  Total Error(s): 0
C-Test End at Jul 27 2017,20:02:28
End simulation
```

Listing 5.8: Excerpt of the *test.log* created by the simulations

This page was intentionally left blank.

Chapter 6

Tests & Results

The test phase was the last stage of the development of the DPI-C Test Environment. The goal at the end of the development process is to get an application ready to use, user friendly, efficient and without bugs.

To accomplish this, the testing phase should not be perceived as a simple verification. Rather it should check that all features are working in accordance to the specified requirements.

Each test cases created (see section 5.3) is a compilation of several small tests, for example, the function *init_uMCTL2()* contains a total of 7 tests being some of them the uMCTL2 registers programming to initialize the DFI (for each memory type), uMCTL2, memory module, and address map.

This project's development phase followed an Agile methodology, like the flow showed on figure 6.1. Once the small tests (functions) reached a preliminary version the test phase began. The function was tested in simulation and testbench, the results were analysed and some feedback from others engineers involved in the DDR project were retrieved. If the tested function passed on both environments and on all configurations, another function was added to the previous one. If not, after debugging and discussing the problem, the modifications were implemented and retested. This iterative process was repeated until the test case met all the initial requirements,

enabling the final release of the DPI-C Test Environment.

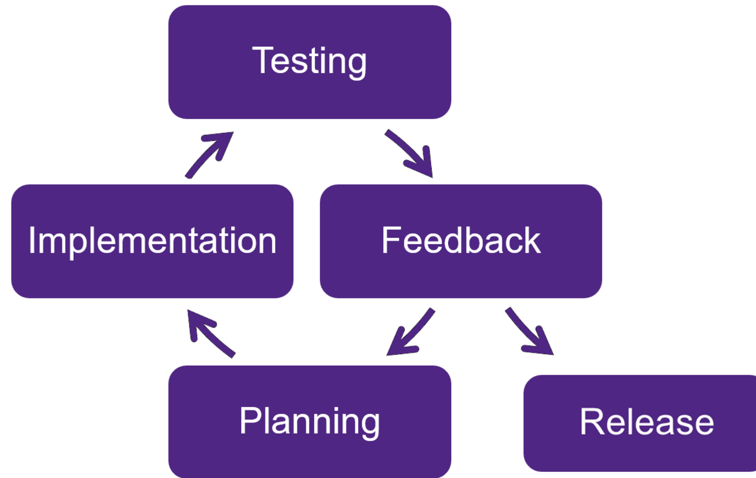


Figure 6.1: Testing flow for DPI-C Test Environment

In this way, it was possible to change the test case throughout the development to better meet the needs of the DUT and to detect early bugs, improving each test individually for the complete and complex test case.

6.1 Tests

The DPI-C Test Enviromnent was built to autonomously test the DUT by using the test cases. The user only needs to execute the test. If the setup supports network connection for remote access, the user may use an automation tool and schedule tasks to run the test.

To accomplish this level of autonomy and efficiency, the test cases comprise several tests. For example, the test case *test_case_ctrlinit.c* contains:

- 23 tests for configurations and initialization;
- 20 test for training;
- 16 single memory write/read tests:

  - 1 test for 4 different memory address offsets.
    * each one write/read 4 memory blocks in different columns, banks and rows;

- 1 burst memory write/read test to 4 memory blocks in different columns, banks and rows of one memory address offset:
  - single write all registers of a given memory block;
  - writes in burst mode from the written block to a new block;
  - single read all registers of the new block and compares to the data written to the first block.

If the test case executed is the *test_case_ctrlinit_stress_tes.c*, the memory tests are executed inside a *for* loop with size of 1000. But for each single memory write/read test, a stress test is executed. The stress test is another *for* loop with a size of 1000 - that executes a single memory write/read test for each address of a given block. Due to the size of this test, usually it was ran overnight and at the end of each report an error report with the number of tests executed and the total and percentage of errors was added.

6.2 Results

The first tests executed were to the DDR Configuration Tool. It was discovered there was a problem with the calculation of a register value of the DDR4 initialization. Consequently, the offset of the base register of the controller was wrong, which caused the tool to stop working when it started to write to the wrong address - this was only seen after the update to the FPGA PHY.

This early debug obligate to review the initialization sequences used for all SDRAMs, controller and PHY. These sequences suffered improvements during the whole development of the DPI-C Test Environment.

Besides the bare metal console outputs, there was a possibility to instrument additional signals of the prototype design. This instrumentation allows the user to record the state of each desired signal of the RTL for debug purpose. The output of the instrumentation is a waveform file similar to the one created in simulation.

Since the DPI-C uses the exact same input data for both simulation and bare metal environments, the output resulting from both platforms should be the same and easily comparable using both logs or waveforms files (see figure 6.2).

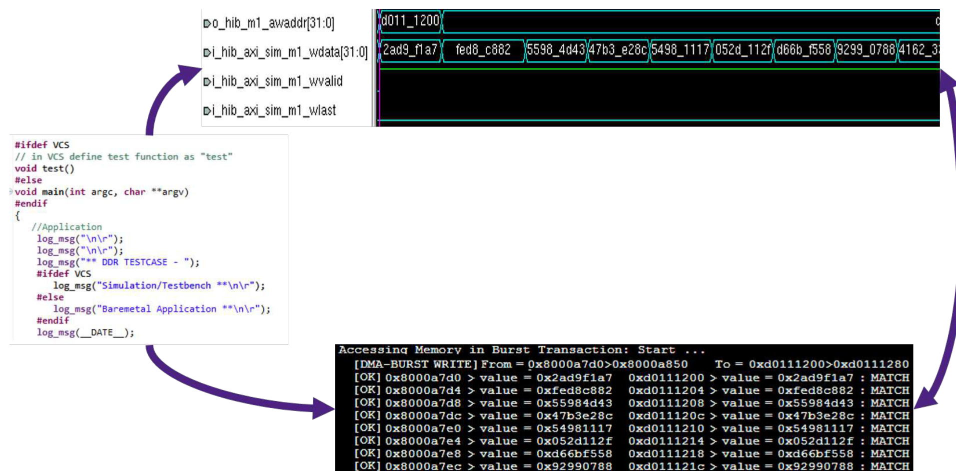


Figure 6.2: The test environment uses the same test case on both platforms and outputs log files and waveform to be compared.

During the development a lot of errors were found. Naturally, most of them were software related and easy to solve. Yet, some of them were DUT and sequence initialization related. Thanks to the knowledge of the hardware engineers assigned to DDR projects and with the tests output all errors, except two of them, were debugged and corrected.

6.3 Unsolved Problems

During the test phase several problems were found on different sources. The software problems were all debugged and corrected. Until the end of the project only two problem were left unsolved:

1. When executing a burst read through DMA drivers, i.e. from the memory under test to the ARC RAM, the first 2 to 4 addresses of the block copied were not correct;
2. During DDR4 memory test, it was found that after writing several consecutive addresses, some specific addresses overwritten all previously recorded data. See figure 6.3 for more details.

Figure 6.3: DDR4 memory test error.

After several tests and different controller and PHY configurations, the problem remained. The DDR4 was tested again with the previous tool and the same problem was observed.

The problems were reported to the teams responsible for the bare metal drivers (regarding the first problem described) and to the IP DDR and PHY teams (regarding the second problem).

After the end of this project, the second problem was solved. The prob-

lem was in the controller design where a data mask bit had a wrong value. Because of this mistake the data mask used during the write operation was the wrong one, causing data to overlap the content in certain register banks of the memory under test.

Chapter 7

Conclusion and Future Work

This last chapter provides an overview and discuss the development of the DPI-C Test Environment during the internship.

Synopsys has a long history of being a global leader in electronic design automation (EDA) and semiconductor IP and is also growing its leadership in software security and quality solutions. It was only possible to reach this level of quality products by constantly improving their solutions. One of the Synopsys' product that ensures the quality of the IP is the prototyping kits. These kits accelerate software development and, as a result, time to market. It also provides an example of applications to clients that can modify or implement their own systems in the kits.

Although, the development of the prototyping kits - design and software - can be extremely time-consuming. The DPI-C Test Environment enables to ease and accelerate the debugging, simulation and testing of the IPKs.

As presented during this document, the DPI-C Test Environment is a complex platform capable of running a test case in a testbench and in a real hardware platform allowing the user to easily compare results between both environments.

During the development of this project, the benefits of the C-Tests in comparison with the old tools became evident. It is faster in both platforms,

output reports are more organized and detailed, and it allows other features that were not possible in the old tools, such as the burst mode. Contrarily to the old tool that needed to simulate, parse its result to a log and ini files to feed the DDR Config Tool, the C-Tests can be a bit time-consuming at the beginning but once the test case is done, it can be changed and rerun quickly. Also, there is no need to compile the OS every time that the software requires an update/fix, saving time.

As described in section 6.3, two main problems were found in the DDR Prototype, which have never been reported before. Therefore, the Test Environment has proven that it is effective and useful for the development and debugging of prototyping kits and that it meets all initial project requirements.

The next step, based on the executed tests that served as proof-of-concept, is to include the C-Tests in the verification and Hardware Validation flows for others protocols. To accomplish this, the DPI-C Test Environment should be even more robust, specially in the drivers for burst write/reads in bare metal applications. There should be more info about the setup/equipment under test in the reports and the C-API should be modified to add more library files for other protocols.

Bibliography

[1] Doulos KnowHow Developing & Delivering KnowHow. *SystemVerilog DPI Tutorial*. `https://www.doulos.com/knowhow/sysverilog/tutorial/dpi/`, last accessed 28 April 2017.

[2] Deepak Kumar Tala. *Introduction to Verilog*. February 09, 2014. `http://www.asic-world.com/verilog/history.html`, last accessed 5 May 2017.

[3] Deepak Kumar Tala. *History of Verilog*. February 09, 2014. `http://www.asic-world.com/verilog/history.html`, last accessed 28 April 2017.

[4] Doulos KnowHow Developing & Delivering KnowHow. *A Brief History of Verilog*. February 09, 2014. `https://www.doulos.com/knowhow/verilog_designers_guide/a_brief_history_of_verilog/`, last accessed 28 April 2017.

[5] Stuart Sutherland. *Using PLI 2.0 (VPI) with VCS (Yes, it really works!)*. SNUG San Jose, 2002. `http://www.sutherland-hdl.com/papers/2002-SNUG-paper_VCS_and_PLI2.0.pdf`, last accessed 23 April 2017.

[6] Stuart Sutherland. *The Verilog PLI Is Dead (maybe) - Long Live The SystemVerilog DPI!* SNUG San Jose, 2004.

`http://sutherland-hdl.com/papers/2004-SNUG-paper_Verilog_PLI_versus_SystemVerilog_DPI.pdf`, last accessed 23 April 2017.

[7] Deepak Kumar Tala. *Verilog PLI Tutorial - Part-I*. February 09, 2014. `http://www.asic-world.com/verilog/pli1.html#Introduction`, last accessed 28 April 2017.

[8] Deepak Kumar Tala. *Introduction - System Verilog*. February 09, 2014. `http://www.asic-world.com/systemverilog/intro.html`, last accessed 28 April 2017.

[9] Stuart Sutherland. *Integrating SystemC Models with Verilog and SystemVerilog Models Using the SystemVerilog Direct Programming Interface*. paper presented at Synopsys 2004 SNUG Europe Conference. `http://sutherland-hdl.com/papers/2004-SNUG-Europe-paper_SystemVerilog_DPI_with_SystemC.pdf`, last accessed 2 May 2017.

[10] Sameer Shaikh. *Complete Computer Hardware Only*. `https://books.google.com/books?id=WSYjdR425p4C&dq=circle+strafing&source=gbs_navlinks_s`, PediaPress. p. 290. Retrieved 2017-04-20.

[11] Wayne Manion The Tech Report. *DDR5 will boost bandwidth and lower power consumption*. March 31, 2017. `https://techreport.com/news/31673/ddr5-will-boost-bandwidth-and-lower-power-consumption`, last accessed 1 May 2017.

[12] *Advanced Features of High-Speed Digital I/O devices : Double Data Rate*. June 03, 2014. `http://www.ni.com/white-paper/7284/en/#toc1`, note=`http://www.asic-world.com/verilog/pli1.html#Introduction`, last accessed 1 May 2017.

[13] Gabriel Torres. *Everything You Need To Know About DDR, DDR2 and DDR3 Memories - Prefetch.* August 27, 2009. http://www.hardwaresecrets.com/everything-you-need-to-know-about-ddr-ddr2-and-ddr3-memories/5/, last accessed 29 April 2017.

[14] Benny Akesson. *An introduction to SDRAM and memory controllers.* http://www.es.ele.tue.nl/premadona/files/akesson01.pdf, last accessed 29 April 2017.

[15] Altera. *The Benefits of Altera's High-Speed DDR SDRAM Memory Interface Solution* . May 2004. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp_stratix_ddr.pdf, last accessed 1 May 2017.

[16] Inc. Micron Technology. *256Mb: x8, x16 Automotive DDR SDRAM Features - Datasheet.* July, 2011. , last accessed 29 April 2017.

[17] Inc. Micron Technology. *General DDR SDRAM Functionality* - tn4605.p65 – rev. a; pub. 7/01. July, 2001. , last accessed 30 April 2017.

[18] Synopsys. *HAPS® Prototyping Solutions.* https://www.synopsys.com/verification/prototyping/haps.html, last accessed 1 May 2017.

[19] Business Dictionary. *What is test case? Definition and meaning.* http://www.businessdictionary.com/definition/test-case.html, last accessed 25 April 2017.