D3.1 Virtual Platform Architectural Design

V1.0

Document information

<table>
<thead>
<tr>
<th>Contract number</th>
<th>687902</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project website</td>
<td><a href="http://www.SAFEPOWER-project.eu">www.SAFEPOWER-project.eu</a></td>
</tr>
<tr>
<td>Contractual deadline</td>
<td>30/06/2017</td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>CO</td>
</tr>
<tr>
<td>Nature</td>
<td>Other</td>
</tr>
<tr>
<td>Author</td>
<td>IMP</td>
</tr>
<tr>
<td>Contributors</td>
<td>OFF</td>
</tr>
<tr>
<td>Reviewer</td>
<td>OFF</td>
</tr>
<tr>
<td>Keywords</td>
<td>Virtual Platform, Architecture</td>
</tr>
</tbody>
</table>

Notices:

This project and the research leading to these results has received funding from the European Community’s H2020 program [H2020-ICT-2015] under grant agreement 687902

© 2017 SAFEPOWER Consortium Partners. All rights reserved.
## Change log

<table>
<thead>
<tr>
<th>VERSION</th>
<th>DESCRIPTION OF CHANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1.0</td>
<td>Reviewed by OFF and consolidated by IMP</td>
</tr>
</tbody>
</table>
Table of contents

1. PREFACE ................................................................................................................................. 6

2. IMPERAS TECHNOLOGY ............................................................................................................. 7
   2.1. IMPERAS SIMULATION TECHNOLOGY .................................................................................. 8
   2.2. OVP MODELLING TECHNOLOGY ......................................................................................... 9
      2.2.1. Model Generation Tool (iGen) ....................................................................................... 9
      2.2.2. Model Component Library ........................................................................................... 9
   2.3. EXTENSIONS FOR TIMING ANNOTATION AND POWER ESTIMATION ......................... 10
   2.4. USER VISUALIZATION ........................................................................................................ 10

3. ANATOMY OF A VIRTUAL PLATFORM ...................................................................................... 11
   3.1. HARNESS .......................................................................................................................... 11
   3.2. MODULE ............................................................................................................................. 11
   3.3. PROCESSOR COMPONENTS (MODELS) ............................................................................. 12
   3.4. PERIPHERAL COMPONENTS (MODELS) ........................................................................... 12
   3.5. INTERCONNECT ................................................................................................................... 13
      3.5.1. Bus .................................................................................................................................. 13
      3.5.2. Net .................................................................................................................................. 13
      3.5.3. PacketNet ..................................................................................................................... 14

4. SAFEPOWER VIRTUAL PLATFORM ......................................................................................... 14
   4.1. VIRTUAL PLATFORM ANATOMY .......................................................................................... 14
      4.1.1. Harness ......................................................................................................................... 15
      4.1.2. Boards ............................................................................................................................ 15
      4.1.3. Xilinx Zynq .................................................................................................................. 15
      4.1.4. Xilinx Zynq Processing Sub-System .............................................................................. 15
      4.1.5. Xilinx Zynq Programmable Logic ................................................................................. 15
      4.1.6. User I/O and Visualization ............................................................................................. 16
4.2. SAFEPOWER PROCESSING SYSTEM COMPONENTS ......................................................... 16
   4.2.1. ARM CortexA9MPx2 Processor ........................................................................ 18
   4.2.2. Ethernet (Cadence GEM) .................................................................................. 18
   4.2.3. GPIO .................................................................................................................. 19
   4.2.4. I2C ....................................................................................................................... 19
   4.2.5. QSPI/Flash ......................................................................................................... 19
   4.2.6. System Watchdog Timer .................................................................................... 19
   4.2.7. System Level Control Registers ........................................................................ 19
4.3. SAFEPOWER PROGRAMMABLE LOGIC COMPONENTS .......................................... 19
   4.3.1. MicroBlaze Processor ......................................................................................... 21
   4.3.2. AXI Timer ............................................................................................................ 22
   4.3.3. logiCore Fixed Interval Timer ........................................................................... 22
   4.3.4. Network-on-Chip Implementation .................................................................... 22
4.4. SAFEPOWER PROGRAMMABLE LOGIC MODULES ............................................... 23
   4.4.1. BareMetal Single Microblaze PL ....................................................................... 23
   4.4.2. BareMetal Dual Microblaze PL with Shared Memory Communication .......... 23
4.5. BAREMETAL NOC PL NETWORK COMMUNICATION ............................................. 24
5. BINARY INTERCEPTION TECHNOLOGY .................................................................... 25
6. TIMING ANNOTATION .................................................................................................. 28
7. POWER ANNOTATION .................................................................................................. 30
8. SAFEPOWER VLNV LIBRARY COMPONENTS .......................................................... 34
9. GLOSSARY / TERMINOLOGY ..................................................................................... 37
10. ................................................................................................................................. REFERENCES 38
Index of Figures

Figure 1: IMP Technologies .......................................................... 7
Figure 2: Overview of Code Morphing Flow .................................... 8
Figure 3: User Visualization Display ............................................. 10
Figure 4: Block Diagram of Xilinx Zynq Virtual Platform .................. 11
Figure 5: Relationship between API calls, Simulator and Processor Model .... 12
Figure 6: Peripheral Model Block Diagram ..................................... 13
Figure 7: Zynq SOC .................................................................... 14
Figure 8: NoC Implementation as Module in Virtual Platform ............... 22
Figure 9: Programmable Logic single MicroBlaze sub-system .................. 23
Figure 10: Programmable Logic dual MicroBlaze sub-system .................... 24
Figure 11: Programmable Logic MicroBlaze sub-systems with NoC .......... 24
Figure 12: Overview of Intercept Library Flow .................................. 26
Figure 13: Interception used for semihosting .................................... 27
Figure 14: Overview of Timing Model in Interception Library ................. 28
Figure 15: Overview of the location of the Power Interception Library ........ 30
Figure 16: Overview of the components of the Power Interception Library ... 31

Index of Tables

Table 1: Processing System Model Status ......................................... 16
Table 2: Processing System Memory Status ...................................... 17
Table 3: Processing System CPU Model Components ........................... 18
Table 4: SAFEPOWER Specific Model Status .................................... 20
Table 5: Programmable Logic Module Descriptions ............................ 20
Table 6: VLNV Library Components ................................................ 34
1. PREFACE

A Virtual Platform is a software description of a hardware board, including processors, peripherals, memories and interconnects that can execute the same application software as the physical hardware, whilst providing an enhanced environment for software development, debug, verification, analysis and profiling.

Virtual platforms lower software development costs, increase quality and reduce the risks involved with the software development side of delivering advanced electronic systems. This is accomplished by enabling:

- Ability to run the physical hardware executables on the virtual platform
- Early start of software development and hardware-software integration
- Accessibility of the virtual platform for the entire development team, no matter the location
- Full visibility and controllability of the simulation environment for improved software testing
- Repeatable, deterministic simulation makes debug easier
- Flexibility of the virtual platform to easily accommodate changes in the specification
- Ability of the virtual platform to connect to real world resources

IMP made their virtual platform simulation solution public and freely available under the auspices of Open Virtual Platforms (OVP™).

The Virtual Platform, the models of processors and peripherals and interconnects can be created using the APIs published by OVP.

IMP provides additional technologies, including SlipStreamer™ (the Binary Interception technology) that enables verification, analysis and profiling of software applications executing on the virtual platform. This technology allows an independent software library to monitor operations in the software execution and to call simulator API functions to modify the operation and interact with models.

The binary interception technology is also used to provide timing annotation to the virtual platform so that the execution time of the application on the Virtual Platform is extended to approximate the execution time of the real hardware. The technology also allows analysis of the execution of an application and its interaction with hardware within the platform so that power estimations may be made.

This document introduces, in chapter 2, the core technologies and tools provided by IMP that which been used in the SAFEPOWER project. Chapter 0 contains a description of the constituent parts that make up a virtual platform. The Binary Interception technology is introduced in chapter 5 and its usage is described to implement timing annotation, chapter 0, and power estimation, chapter 7.
2. IMPERAS TECHNOLOGY

IMP has provided a number of technologies to enable the creation and use of Virtual Platforms in the development of embedded software applications.

The Virtual Platform contains models that are executed using IMP simulation technology that can provide simulation speeds of 100s of millions of instructions per second and is fully deterministic i.e. without external influence two executions of the same VP will give the same results.

Models, provided by IMP and Open Virtual Platforms (OVP), are implemented using the OVP APIs with templates generated using the iGen generation tool.

An integrated development environment is multithread / multicore / multiprocessor aware allowing application software for all processors and peripheral model behavioral code to be debugged together. It is also a 3-dimensional debugging environment (Spatial, Temporal, Abstract) allowing the debug of firmware, OS kernel, drivers and applications concurrently.

The simulator includes the IMP binary interception technology (SlipStreamer™) that supports the IMP Verification, Analysis and Profiling (VAP) tools and is used to create custom tools such as the Timing and Power estimation tools described in chapters 0 and 7.
2.1. Imperas Simulation Technology

IMP simulation technology enables very high performance simulation of software applications executing on platforms that may contain multiple processor and peripheral models.

The IMP simulation technology is provided as an OVP simulator, OVPsim, which supports the execution of virtual platform simulation (a sub-set of the OVP APIs) and as the IMP Professional Simulator, CpuManager, which supports the complete OVP APIs that includes binary interception technology, an integrated development environment and 3-dimensional debug.

The simulation technology is based upon Just-In-Time code morphing technology that generates host code as it is needed for the simulator to be able to execute a target processor cross compiled application instruction.

The Code Morphing is performed by a processor model that provides a description, using calls to the VMI API functions, of the stages required to perform the required behavior.

The flow from target processor cross compiled instructions to the code executed on the host processor (Intel x86) that provides the equivalent operations during the simulation execution is shown in the following figure.

![Figure 2: Overview of Code Morphing Flow](image)
A code morphing Just-In-Time simulator is instruction accurate i.e. the state of the simulation on an instruction boundary will accurately represent the state of the real hardware, however, the state within an instruction will not be represented e.g. no pipeline or cache effects.

All models are individually created and stored, in binary form, in a library from which they are dynamically loaded by the simulator. This ensures that a verified model behavior will not alter and that the model can be incorporated multiple times into one or more virtual platforms.

### 2.2. OVP Modelling Technology

The models used in a virtual platform initially created using a model generation tool. The behavior is added to the model using the OVP APIs, VMI API for processor models and the BHM and PPM APIs for the peripheral models. The models are stored within a VLNV component library.

#### 2.2.1. Model Generation Tool (iGen)

The generation of component templates and virtual platforms is aided by the Model Generator tool, iGen. This tool executes scripts making calls to the iGen Command API and using TCL (Tool Control Language) for structure to create C templates for simulation models and intercept libraries and to create virtual platforms and harness.

Documentation references to specific uses of iGen:

- iGen Model Generator Introduction [4]

#### 2.2.2. Model Component Library

The component models utilize a library structure for model source and for the compiled binary output to provide a unique identifier for all components using Vendor, Library, Name and Version (VLNV) specifiers.

Using a VLNV library allows individual models to be compiled and verified and to remain unchanged in the library. The VLNV gives each model a unique reference. All the components that are used in the SAFEPOWER project VP are listed, by VLNV, in chapter 8.

A pre-defined VLNV component library is provided with the OVP and IMP products.

The model source library is located at

```
$IMPERAS_HOME/ImperasLib/source
```

With corresponding output binary library located by the `IMP_VLNV` environment variable which is set to
A user can create their own VLNV library in which to create new components and modules to be used with the VP. The new library can then be specified using the `vlnvroot` command line argument or by adding the path to the IMP_VLNV environment variable.

### 2.3. Extensions for Timing Annotation and Power Estimation

The VMI API has been extended and the simulator enhanced to allow for timing information to be annotated into the simulation from an Interception Library.

An intercept library is used to analyze the operation of a software application on a virtual platform and then generate timing information that can be used to drive APIs controlling timing in the Virtual platform.

Without the addition of the intercept library providing feedback the Virtual Platform executes instructions at a configured, fixed, rate and the busses used to interconnect components and memory are assumed perfect.

### 2.4. User Visualization

A new API has been created to allow a user to interact with peripheral models. These are typically devices monitoring switch positions for input and generating outputs to simple display devices such as alphanumeric displays and LEDs. A typical display is shown in the following figure.

![User Visualization Display](image)

*Figure 3: User Visualization Display*
3. ANATOMY OF A VIRTUAL PLATFORM

The Virtual Platform (VP) is created as a combination of a harness, module(s), components and interconnects. These elements are described in more detail later.

Figure 4: Block Diagram of Xilinx Zynq Virtual Platform

The figure shows one possible SAFEPOWER VP configuration and the relationship between a harness, a zc702 Xilinx board and a Zynq FPGA containing the Processing Sub-system (PS) and the Programmable Logic (PL) depicting the default PL without any components.

3.1. Harness

The harness is a C file using the OP API that provides a command line user interface and controls the loading and configuration of one or more modules that describe the hardware.

3.2. Module

A module can contain other modules, components, memories that are joined together by interconnects (see later section).

In the SAFEPOWER project the important modules, described in chapter 4.3, were created for the end-users but any module can be described and generated using the iGen tool.
3.3. Processor Components (Models)

A library of processor models is provided by IMP as part of a product installation.

A processor model template is generated using the cpuGen tool from a list of the instruction decodes and its available modes of operation.

The VMI Morph Time (MT) and Run Time (RT) APIs are used to describe the effect on the state of the processor of each instruction executing, in addition to synchronous events e.g. timers and asynchronous events e.g. external interrupts.

The processor models are the key component within the simulation. As the simulation is instruction accurate all actions within the flow are related to the execution of instructions on the processors.

The following diagram shows the execution flow, from the initialization of the processor model to the execution of instructions. The initial execution of an instruction causes the simulator to call into the processor model so that it can create a block of native code representing the target instruction to be executed. This is then held in a code dictionary from where it can be executed multiple times.

![Diagram showing relationship between API calls, Simulator and Processor Model]

3.4. Peripheral Components (Models)

A library of peripheral models is provided by IMP as part of a product installation.

The peripheral components allow slave and master port accesses onto any address mapped region in the virtual platform. The template is created using the iGen tool and the behavior created using the BHM and PPM APIs in C/C++ code.
A peripheral model can contain both registers (detailed to the bit field level) and memory regions. It will typically become active by the use of callback functions activated on a read or write to a register or memory region, on a change to an input signal or on timed events.

The structure of a peripheral model is shown in the following diagram. The behavior of a peripheral model can be realized as PSE code or as native code.

![Peripheral Model Block Diagram](image)

*Figure 6: Peripheral Model Block Diagram*

The PSE code utilizes the BHM and PPM APIs and executes within the simulator framework. This provides a ‘protected’ environment that restricts accesses.

The native code element allows the peripheral model to use native libraries and interact with host resources. It may utilize the VMI RT API functions to interact directly with the peripheral model PSE and the simulator.

### 3.5. Interconnect

#### 3.5.1. Bus

A bus connection provides a memory mapped interconnect between devices and memories.

A bus can be defined with a width that controls the maximum address that can be accessed by any device connected to it.

A bus connection is the typical connection used between components.

#### 3.5.2. Net

A net connection is primarily used for carrying signals between one and one or more components. It is used to signal reset, interrupts and other events in a system.

When a net is written ALL components that have registered a callback to listen to the net are called.
A net can carry a 32 bit integer value so, as well as being used to signal events, can also carry data.

3.5.3. PacketNet

The packetnet connection provides a mechanism to notify one or many components that there is a data structure available to transfer. The data is not transferred directly between components; instead a pointer is transferred to indicate the location of the data structure defined by the source component. This allows any of the components to update fields in the data structure prior to it being read by the next component. In this way an indication of the data being consumed by a component can be indicated.

The packetnet connection is used to implement networks such as CAN or NoC.

4. SAFEPOWER VIRTUAL PLATFORM

4.1. Virtual Platform Anatomy

The Xilinx Zynq virtual platform consists of a Processing Sub-system and Programmable Logic, as shown in the following Xilinx product diagram.
This is modeled in the Virtual platform as the following components and interconnects:

4.1.1. Harness

The harness selects and loads a Xilinx board, either zc702, zc706 or the SAFEPOWER PCB and may load a module which represents the Programmable Log.

4.1.2. Boards

The Xilinx zc702 and zc706 boards and the SAFEPOWER PCB are provided as modules. The modules provide the instantiation of the Zynq module. The modules are also where the user visualization is defined. Included within the module is C code using the OP API to connect GPIO from the modules to the visualization interface.

4.1.3. Xilinx Zynq

The Zynq module is the Zynq-7000 device and instantiates the Zynq_PS and the Zynq_PL modules and the interconnect for bus accesses, GPIO and interrupts. At this level the Zynq_PL that is defined is the default, Zynq_PL_Default. The actual PL that is loaded may be controlled by the harness.

4.1.4. Xilinx Zynq Processing Sub-System

The PS is defined using standard OVP module design. The design includes all the components of the physical hardware, including the Cortex-A9MPx2 processor. However, some components are included as register only models or dummy models, as shown in Table 1: Processing System Model Status

A register only model correctly defines the registers of the component and the read/write capabilities of those registers. However, they do not include any behavior that may be associated with a register access.

A dummy model provides an area of memory equivalent in size to the registers of the actual component.

The OVP ARM processor model implements some of the core features that are described as components in the Zynq specification and so will not be found in the Xilinx Zynq_PS module definition.

4.1.5. Xilinx Zynq Programmable Logic

The PL is defined using standard OVP module design. The module may include any OVP components, processors, peripherals, memory etc. and be connected using any bus architecture. The PL module can communicate with the PS via the defined memory interconnect interface.
There may be many PL modules representing alternative hardware designs that can be loaded into the programmable device. Each is defined and stored in the IMP VLNV library and can be loaded to represent any programmable elements by selecting the module as a Zynq platform command line argument.

4.1.6. User I/O and Visualization

The hardware boards provide certain User I/O in the form of switches and LED’s. These can be represented in the virtual platform using the visualization technology.

The visualization technology allows the definition of different user interactive devices, for example switches, LED’s, alphanumeric displays etc. that are provided in an HTTP browser with a socket connection to the virtual platform.

The platform provides the switch and LED definition from the board to allow user interaction.

4.2. SAFEPOWER Processing System Components

This section describes some of the models that are included in the Zynq Processing sub-system. The full list of components and the status of them is shown in the following table

<table>
<thead>
<tr>
<th>Component</th>
<th>Model Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM Cortex-APMPx2</td>
<td>Full complete and verified</td>
</tr>
<tr>
<td>uart0/uart1</td>
<td>Supports standard I/O and interrupt generation.</td>
</tr>
<tr>
<td></td>
<td>Does not model physical aspects of the device e.g. baud rate</td>
</tr>
<tr>
<td>slcr</td>
<td>Register reset values, allows reset of CPU cores and controls OCM mapping.</td>
</tr>
<tr>
<td>devcfg</td>
<td>Register only</td>
</tr>
<tr>
<td>USB/USB1</td>
<td>Register only</td>
</tr>
<tr>
<td>I2C0/I2C1</td>
<td>Model Implemented</td>
</tr>
<tr>
<td>SPI0/SPI1</td>
<td>Register only</td>
</tr>
<tr>
<td>CAN0/CAN1</td>
<td>Register only</td>
</tr>
<tr>
<td>Component</td>
<td>Model Status</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>GPIO</td>
<td>Model Implemented</td>
</tr>
<tr>
<td>ETH0/ETH1</td>
<td>Model Implemented</td>
</tr>
<tr>
<td>Quad-SPI</td>
<td>Model Implemented – single Flash type provided</td>
</tr>
<tr>
<td>Static Memory Controller (SMC)</td>
<td>Register only</td>
</tr>
<tr>
<td>System-Level Control registers (SLCR)</td>
<td>Register only</td>
</tr>
<tr>
<td>Triple Timer Counter0 (TTC)</td>
<td>Model Implemented</td>
</tr>
<tr>
<td>Triple Timer Counter1 (TTC)</td>
<td>Model Implemented</td>
</tr>
<tr>
<td>DMAC when secure (DMAC S) (DMAC)</td>
<td>Register only</td>
</tr>
<tr>
<td>DMAC when non secure (DMAC NS) (DMAC)</td>
<td>Register only</td>
</tr>
<tr>
<td>System Watchdog Timer (SWDT)</td>
<td>Model Implemented</td>
</tr>
<tr>
<td>DDR Memory Controller (DDRC)</td>
<td>Register only</td>
</tr>
<tr>
<td>Device Configuration Interface (DEVCFG)</td>
<td>Register only</td>
</tr>
<tr>
<td>AXI HP0 (AFI)/ AXI HP1 (AFI)/AXI HP2 (AFI)/AXI HP3 (AFI)</td>
<td>Dummy</td>
</tr>
<tr>
<td>eFuse</td>
<td>Dummy</td>
</tr>
<tr>
<td>SDIO0/SDIO1</td>
<td>Register only</td>
</tr>
<tr>
<td>ARM L2 Cache Controller (PL310)</td>
<td>Register only</td>
</tr>
<tr>
<td>Top level interconnect and Global Programmers View (GPV)</td>
<td>Not modeled</td>
</tr>
</tbody>
</table>

**Table 2: Processing System Memory Status**

<table>
<thead>
<tr>
<th>Component</th>
<th>Model Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDR</td>
<td>Simulator Memory</td>
</tr>
<tr>
<td>On-chip Memory (OCM)</td>
<td>Simulator Memory</td>
</tr>
</tbody>
</table>
ARM processor model CBAR\(^1\) value is set to 0xf8f00000 to base the CPU Model internal register address mappings correctly for the following:

**Table 3: Processing System CPU Model Components**

<table>
<thead>
<tr>
<th>Component</th>
<th>Model Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCU Control and Status</td>
<td>CPU Core Model</td>
</tr>
<tr>
<td>Interrupt Controller CPU</td>
<td>CPU Core Model</td>
</tr>
<tr>
<td>Global Timer</td>
<td>CPU Core Model</td>
</tr>
<tr>
<td>Private Timers and Private Watchdog Timers</td>
<td>CPU Core Model</td>
</tr>
<tr>
<td>Interrupt Controller Distributor</td>
<td>CPU Core Model</td>
</tr>
</tbody>
</table>

### 4.2.1. ARM CortexA9MPx2 Processor

The arm processor model can be configured to support a great many variants, for example it can provide the behavior of processors from ARM7TDMI to Cortex-A53 MP cores. The processor model provides the instruction set and exception modeling and also provides the following features of the Zynq platform:

a) SCU Control and Status
b) Interrupt Controller CPU
c) Global Timer
d) Private Timers and Private Watchdog Timers
e) Interrupt Controller Distributor

### 4.2.2. Ethernet (Cadence GEM)

A new model created for the SAFEPOWER project to represent the Cadence Gigabit Ethernet Module (GEM) as used on the Zynq Processing Sub-system. The model provides both the Ethernet MAC and PHY registers providing Tx and Rx communication. Internally the model makes use of the bhmEthernet* API functions to communicate with the real world using the host Ethernet connection. This allows the application running on the Zynq VP to communicate in exactly the same way as it would on the physical hardware.

---

\(^1\) In the ARM processor implementations this corresponds to the value on the PERIPHRESET input pins.
4.2.3. **GPIO**

The GPIO peripheral of the Zynq allows the input and output of signals between the PS and PL and can also be selected to be used within the user visualization.

4.2.4. **I2C**

An I2C interface provides the register set and allows communication to other devices mapped onto the IIC bus.

When modeling an interface of this type the specific hardware signaling characteristics are ignored and a model is created for the functional behavior. In this way the IIC bus is a memory mapped region used to represent the addressing of components onto the bus. There is, therefore, no modeling of the bit level characteristics of an I2C interface.

4.2.5. **QSPI/Flash**

The QSPI interface is dedicated, on the Zynq platform, to allow access to SPI Flash memory. In the same way as the I2C device does not model the full hardware characteristics of the I2C bus, for the QSPI it was decided to combine the QSPI and Flash model so that a more optimized simulation model could be provided and the physical aspects of the SPI are not included.

The QSPI interface provides the register interface and can be configured for one of two Flash devices used on the Zynq boards.

The default Flash memory device modeled is the Micron n25q128a11esf40g device. The Spansion S25FL128SAGMFIR01 is also provided.

4.2.6. **System Watchdog Timer**

A simple watchdog timer is created that can be programmed to generate an interrupt after a pre-defined period.

4.2.7. **System Level Control Registers**

These registers are modeled to the extent needed to provide the reset control of the ARM Cortex-A9 processor cores and to allow control of the clock rate of the ARM processor.

4.3. **SAFEPOWER Programmable Logic Components**

The PL can include any IP blocks. As such, many components can be added and included, from processors to the NoC. There is a library of models provided by OVP that can all be used in the VP PL even if they are not available in the Xilinx IP library. The OVP peripheral models are provided as source code so can easily be copied as the basis for creating a new model.

---

2 It is intended that this will be extended so that the ARM can be suspended and the clocks to other components, including those in the PL, disabled.
model. The open APIs, the documentation and the iGen tool provided by OVP make it simple for a user to create new models to be used as part of the PL.

The full list of SAFEPOWER specific components and the status of them is shown in the following table.

*Table 4: SAFEPOWER Specific Model Status*

<table>
<thead>
<tr>
<th>Component</th>
<th>Model Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>MicroBlaze</td>
<td>Full complete and verified</td>
</tr>
<tr>
<td>TTEL Network Interface</td>
<td>Complete model configurable for NoC implementation. Does not provide network timing</td>
</tr>
<tr>
<td>Nostrum Network interface</td>
<td>Complete model configurable for NoC implementation. Network messages transferred on a heartbeat</td>
</tr>
<tr>
<td>logiCore Fixed Interval Timer</td>
<td>Complete model</td>
</tr>
<tr>
<td>AXI GPIO</td>
<td>Complete model</td>
</tr>
<tr>
<td>AXI Timer</td>
<td>Complete model</td>
</tr>
</tbody>
</table>

Any component from the VLNV library can be included and used in a module for the PL. the following are the example modules that have been created for the SAFEPOWER project.

*Table 5: Programmable Logic Module Descriptions*

<table>
<thead>
<tr>
<th>Module</th>
<th>Module Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zynq_PL_SingleMicroblaze</td>
<td>A single MicroBlaze with local memory</td>
</tr>
<tr>
<td>Zynq_PL_DualMicroblaze</td>
<td>Two MicroBlazes, each with local memory. A shared memory can be access by either MicroBlaze or by the ARM processor in the PS</td>
</tr>
<tr>
<td>Zynq_PL_NoC_node</td>
<td>A single Microblaze with local memory and a Network interface</td>
</tr>
<tr>
<td>Zynq_PL_NoC</td>
<td>A Network Interface connected to an address space that can be accessed by the ARM processor and three Zynq_PL_NoC_node sub-systems all with connections onto the NoC</td>
</tr>
</tbody>
</table>
### Module Description

<table>
<thead>
<tr>
<th>Module</th>
<th>Module Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zynq_PL_TTELNoC_node</td>
<td>A single Microblaze with local memory and a TTEL Network interface</td>
</tr>
<tr>
<td>Zynq_PL_TTELNoC</td>
<td>A TTEL Network Interface connected to an address space that can be accessed by the ARM processor and three Zynq_PL_TTELNoC_node sub-systems all with connections onto the NoC</td>
</tr>
<tr>
<td>Zynq_PL_NostrumNoC_node</td>
<td>A single Microblaze with local memory and a Nostrum Network interface</td>
</tr>
<tr>
<td>Zynq_PL_NostrumNoC</td>
<td>A Nostrum Network Interface connected to an address space that can be accessed by the ARM processor and three Zynq_PL_NostrumNoC_node sub-systems all with connections onto the NoC</td>
</tr>
</tbody>
</table>

The following are components that have been provided by IMP to be used in the Zynq VP PL.

#### 4.3.1. MicroBlaze Processor

The Xilinx MicroBlaze processor model can be configured to support a number of versions, from an ISA model providing the instruction set to MicroBlaze versions from 7.0 to 10.0.

As the MicroBlaze is a configurable soft-core there are many processing blocks that can be included and/or configured, for example, hardware multiplier, barrel shifter, floating point unit. The MicroBlaze model, by default, provides a set of features depending upon the version selected. All other elements are controlled by configuration parameters.

The Microblaze configuration parameter are detailed in the Parameters section of the version specific documentation that is found in the VLNV library with the source, for example the V9_50 version is described in the document

*OVP_Model_Specific_Information_microblaze_V9_50.pdf*

that can be found in a release at

*IMPERAS_HOME\IMPLib\source\xilinx.ovpworld.org\processor\microblaze\1.0\doc*

The parameter is modified by adding an override onto the execution command line. For example, if the MicroBlaze instance is in the Zynq_PL and is called ‘cpu1’ we could enable the barrel shifter by using

```
--override zc702/Zynq/Zynq_PL/cpu1/ C_USE_BARREL=True
```
4.3.2. **AXI Timer**

A Xilinx IP block is the AXI timer. This has a memory interface to access and program 2 timers. There is no modeling of the AXI interface signals.

4.3.3. **logiCore Fixed Interval Timer**

A Xilinx IP block is the logiCore Fixed Interval timer. This device has no memory interface and generates an interrupt at a fixed rate defined by a parameter.

4.3.4. **Network-on-Chip Implementation**

The Nostrum and TTEL NIs and NoC have been created as models in the PL. The following figure illustrates the defined system that is used in the examples to create a 2x2 network, 3 microblaze sub-systems and the ARM sub-system each having a Network Interface (NI).

This is one possible configuration that is defined as a PL module. The OVP modeling environment allows other configurations to be quickly and easily defined and generated as further PL modules. There is no limitation on the number or complexity.

![Figure 8: NoC Implementation as Module in Virtual Platform](image)
Each sub-system depicted comprises an NI peripheral, a MicroBlaze processor and a local memory.

The NI used can be the Nostrum NI, safepower.ovpworld.org\peripheral\NostrumNode\1.0, the TTEL NI, safepower.ovpworld.org\peripheral\TTELNode\1.0 or any other NI.

The Nostrum and TTEL NI components provide programming interfaces that are the same as the physical hardware allowing the same application binaries to be used on both the VP and the physical hardware. They support the message passing methods to allow data messages to be exchanged with other nodes in a NoC.

The OVP ‘packetnet’ communication link is used to transfer data between nodes in the network.

4.4. SAFEPOWER Programmable Logic Modules

The components have been used to create a number of example modules, each of which provides a particular PL configuration, which are described briefly in the following sections.

4.4.1. BareMetal Single Microblaze PL

The PL module contains a single Microblaze processor with local memory and is used to run stand-alone baremetal applications.

4.4.2. BareMetal Dual Microblaze PL with Shared Memory Communication

The PL contains two MicroBlaze sub-systems, based upon the Baremetal Single MicroBlaze PL. Each sub-system contains a MicroBlaze processor with local memory and an address range mapped to a common address space. The same common address space is also

Figure 9: Programmable Logic single MicroBlaze sub-system
mapped to allow the ARM processor in the PS access. The common address space contains a shred memory.

Figure 10: Programmable Logic dual MicroBlaze sub-system

4.5. BareMetal NoC PL Network Communication

The PL contains 3 MicroBlaze sub-systems. Each sub-system has a Microblaze processor with local memory and a NoC node peripheral connecting to the ‘network’. There is a further NoC node peripheral which is within the ARM address map and allows applications on the ARM processor access the network.

Figure 11: Programmable Logic MicroBlaze sub-systems with NoC
Variations of this PL module are provided that utilize the Nostrum and the TTEL Network Interface nodes.

## 5. BINARY INTERCEPTION TECHNOLOGY

IMP provides a fundamental concept: *binary interception*. Using IMP technology, it is possible to define binary interception libraries that are loadable shared objects (on Linux) or dynamic linked libraries (on Windows).

A binary interception library can cause the simulation engine to call it at certain points in the simulation, including:

- At simulation construction and destruction
- Before each instruction is morphed
- When a specific instruction type is executed
- When a specific address is executed
- When specific memory address ranges are read or written
- When a specific Programmers View event occurs
- When a user invokes a command defined by the library
- After a specific number of instructions have been executed

Once called, a binary interception library may use the VMI Run Time API to query and control the simulation state, including:

- Examine the state of the processor including general purpose and system registers
- Examine Programmers View objects
- Examine the simulation environment
- Replace the simulated behavior for an instruction
- Access symbolic and debug information for the application program being run
- Use GDB to evaluate expressions for the currently running application program
- Add or delete simulator callbacks to the library
- Communicate with other binary interception libraries

Multiple binary intercept libraries may be used at the same time on a single processor, or multiple processors.

This provides a powerful and efficient capability to perform a wide range of functions including modifying or adding processor instructions, performing verification, analysis or other functions on a simulated application - all without the need to modify the application.
Non intrusive instrumentation enables, amongst other features, the ability to add code coverage and code profiling without the need to modify the application source, or recompile the application binary with alternative libraries. So this means that the analysis is performed on exactly the same application binary and not something that is only related by common source code.

Along with the ability to non intrusively instrument the system to gather data the intercept library also has the ability to alter the system. This can be triggered on the occurrence of a system or timed event and provides the ability to inject faults into the system and then to monitor if or how the system is perturbed.

The following figure illustrates the binary Interception Intercept Library, operating in a mode that is monitoring the code executed by a processor model.

![Diagram of Interception Intercept Library Flow](image)

*Figure 12: Overview of Intercept Library Flow*

The standard operation for the simulator is to call into the processor model to morph code for each instruction as it is required to execute. The binary interception library, if it has registered the request, is called prior to the processor model and may perform other actions.

As well as non-intrusively adding instrumentation to monitor the execution of an application program, complete functions can be replaced by the intercept library. This is used for
‘semihosting’ of host resources to allow, for example, functions such as `read` to access stdin or the host file system.

The use of binary interception for ‘semihosting’ is shown in the following figure which illustrates the replacement of the library ‘open’ function with the functionality of the intercept library.

On the left we see the code flow without the intercept library and on the right the flow is shown with the intercept library.

![Image](image_url)

**Figure 13: Interception used for semihosting**

When the intercept library is enabled, none of the instructions within the open function are executed. Instead the intercept library function, extracts the arguments and data for the open function from the processor model and/or memory, and uses them to perform the open on the host. The intercept library then returns the correct return code to the application, indicating the success or otherwise of the function and allows the processor to continue executing.

The binary interception technology is fully described in the IMP document *IMP Binary Intercept Technology User Guide* [7] that is provided with the SDK installation package. The document describes the examples provided in an installation at *Imperas/Examples/BinaryInterception*.

The Binary Interception technology is used to enable Timing and Power annotation as described in chapters 0 and 7.
6. TIMING ANNOTATION

The VMI API provides functions that can be used to annotate timing information to the simulation. The timing annotation is done to reduce the timing error that occurs by using OVP’s simple MIPS (millions of instructions per second) timing model. Normally the processor frequency of the physical system is chosen as value of the MIPS rate. That means the light-weight timing model of OVP approximates that one instruction is executed in one cycle. This introduces an error since each instruction is not executed in exactly one cycle on the physical system. Reference 10 describes a timing model for the Xilinx MicroBlaze processors that uses the previous and deprecated ICM API of OVP, which is anchored in the platform space of the simulation. For SAFEPOWER this timing model is adapted for the VMI API, which is located in the interception space of the simulator. As described in the following paragraphs, the timing model is created as an interception library, which can be dynamically loaded to each MicroBlaze processor model in by the simulator. Figure 14 illustrates an overview of the interception library that includes the timing model for the MicroBlaze (uB) processor model.

The timing model consists of five main parts:

**Core Registration and Callbacks** – This component is the main entry point of the timing model. Each processor model of the virtual platform is registered and callbacks to trace each fetched instruction are announced at the simulation kernel.

```c
void vmirtAddFetchCallback ( 
    memDomainP domain ,
    vmiProcessorP processor ,
    Addr lowAddr ,
    Addr highAddr ,
    vmiMemWatchFn writeCB ,
    void* userData );
```
This method registers a callback function, which is called for every instruction fetch of the processor model, which occurs in the defined address area of the memory domain. So this module gets each fetched operation code and passes it to the processor model. The processor model returns the calculated cycles for each operation code. So this method is able to update the platform time by skipping the execution of instruction slots.

```c
void vmirtAddSkipCount (
    vmiProcessorP processor,
    Uns64 skipCount);
```

This method skips the instruction slots in the simulation kernel, while the time moves forward for the processor model. If an instruction needs \( x \), \( x > 1 \), cycles, \( x-1 \) cycles are skipped.

**Instruction Object** – This class serves as a datatype that can store all needed information about the instructions: name, bitwidth, instruction class and type, pipeline return stage, required registers, base and extra cycles required, is branch operation, is integer division. All these data is used in the cycle and pipeline model to estimate the needed cycles of each instruction.

**Processor Model** – This model includes an instruction decoder, which gets the current operation code and returns the corresponding instruction object. The data of this object is used in the cycle model and the pipeline model. The results of both models are added and returned as final value.

**Cycle Model** – The cycle model analyzes if an instruction needs its extra cycles or not, e.g. an integer division by zero needs less cycles, also branch instructions have variable cycle counts if they are taken or not. For that, the processor’s status registers are read by using to further VMI API methods. First one returns a pointer to the register by searching its name, second uses this pointer to read register’s value.

```c
void vmirtGetRegByName (
    vmiProcessorP processor,
    const char *name);
```

```c
bool vmirtRegRead (
    vmiProcessorP processor,
    vmiRegInfoCP regDesc,
    *void result);
```

**Pipeline Model** – The pipeline model approximates pipeline stalls, which may occur in the five different pipeline stages of the MicroBlaze Processor. This is done by modeling a virtual mirror of the five-stage pipeline that stores all registers needed by the current indexed instructions, which searches for conflicts and returns the penalty cycles.

Furthermore, the timing model has the ability to add extra cycles by an external function call. This is used to pass the time forward in the case of intercepted interface calls, e.g. intercepted I2C communication. By reading the number of transferred bytes, the required processor cycles are approximated and skipped by the timing model.
For the MicroBlaze the remaining overall mean error in 28 benchmarks is at 0.16%, with a maximum and minimum deviation of +0.71% and 0.0%. Bottleneck is the simulation slowdown from 1409 simulated MIPS without timing model to 14 simulated MIPS with timing model attached (factor 100x).

No timing model will be implemented for the ARM processor as it has been for the MicroBlaze. Since the ARM Cortex-A9 Dual-Core is a super-scalar processor, with big and complex pipeline structures, in addition to instruction reordering, a detailed timing model would be too extensive to develop. Reference [2] introduces a very simple timing model for an ARM Cortex-A9 Dual-Core. The authors also evaluated that one core is able to execute 1.8 instructions per cycle in average. Furthermore, some penalty cycles are added for multiplication or memory read and write instructions. If needed a similar timing model will be implemented in future work for the ARM cores.

### 7. POWER ANNOTATION

So far, a power interception library was included by OFFIS to enhance the VP so that it is power aware providing power consumption figures and virtualizing the monitoring devices used in SAFEPOWER project (D3.3). This will help to analyze the state of the low power techniques that can be applied especially for the ARM cores and the voltage rails. Like the above described timing model for the MicroBlaze it is located in an interception library, which is dynamically loaded at run-time of the simulation. In that way the virtual platform itself stays unmodified. Figure 15 illustrates the location of the power interception library, while Figure 16 shows an overview of its components.

![Figure 15: Overview of the location of the Power Interception Library](image)
It consists of five main components. Two instances of core models, a platform model and a tracing engine for generating value change dump (VCD) files and a model of the voltage regulators, which implements the communication behavior of the physical PMBus interface. Excluding the virtual voltage regulators, all other components in the framework communicate via Timed Value Streams (TVS) [3]. TVS are particularly suitable for transferring values of extra-functional properties from one component to another. A TVS is a FIFO that saves tuples of the pushed data and the correct simulation time. If the source component adds new values, the data sink receives a notification to read the new tuple. The core models are the main entry point for each simulation cycle, since the entire data collection is carried out via the processor models of the virtual platform. As mentioned earlier, the processor models are executed alternately by the simulation kernel. As a result, only one core model is active at each time so we do not have to consider the effects of mutual exclusions. The core models deliver the following data for each core: number of accesses to the memory and the AXI bus, accesses to the processor’s frequency register as well as to the memory’s frequency and processor’s power state register. To check the suspend state of the cores, the executed instructions are analyzed for occurrence of \texttt{wfe} or \texttt{wfi} instructions. Finally, the core model defines a timer that executes a function periodically, to calculate the CPU load at a defined interval. If the callback functions detect a change outside this periodic interval, a separate update of the metrics at the current simulation time is made. This ensures that no event is missed by the core models. All core related metrics are processed within the core models, the CPU load, memory and AXI read/write rates, the suspend and power states, and transferred via TVS to the VCD trace engine. All other platform related metrics, all frequency changes and also changes in power states are directly given to the
platform model. In the case that not all data is required for the current analysis, the data acquisition can be switched off individually to improve performance of the simulation.

The platform model processes and stores the received data from the core models and also controls access to them as well as to the simulation kernel. E.g., if a frequency is changed by the user application, only one core model detects this change and notifies the platform model, which then notifies all other core models to update their local values for the global state. The local values must be kept up to date for the correct calculations regarding CPU loads or read/write rates values which depend on the frequency. Furthermore, the platform model provides a monotonic global simulation time, which can be used by all components in the framework to push their values acquired data in the TVS with the correct event time.

Closely connected to the platform model, the virtual voltage regulators give the possibility to the power interception library to recognize changes of the platform voltages. Accordingly, the executed applications can use the provided PMBus interface as per the physical ZC702 board or the SAFEPOWER PCB. The model transfers the voltage values to the platform model since they are platform related. Also the virtual interface supports transmission of the current values in Amperes for each voltage rail. Later in future work, an attached power model can easily fill them.

The VCD trace engine is notified at each push of its connected TVS and pulls the data to update the trace output file as a VCD trace. This file can be viewed at run-time or stored for further analysis. Further output formats are easily produced by using the Timed Value Streams. To use the power interception library the only requirement is to configure the simulation kernel to load this specific intercept library dynamically onto the processor models at simulation run-time. Thus it is easy to provide the power interception library to other OVP users as well as to maintain updates of the power interception library without the need to modify the virtual platform description itself.

The following passages describe implementation details, and the VMI API functions utilized.

The power interception library is written in an object-oriented style, implemented in C++ 14, whereas OVP and the intercept library are written in GNU C. We have implemented a C wrapper to enable the usage of the library in OVP. Since the processing system of the Zynq-7000 contains an ARM Cortex-A9 dual-core processor, the intercept library itself is dynamically instanced twice: once for each processor model. In this way, two separate instances of the power interception library would be created. However, this is not what we want. To counteract this, the top-level class implements a singleton pattern. This singleton object dynamically instances one core model for each constructor call of a processor model, the platform model, the VCD trace engine and the virtual voltage regulator as well as connecting the TVS at the very end of the initialization phase. As mentioned previously, the power interception library uses the VMI RT API of OVP. The core model uses most of the API calls since it acquires most of the observed values. To collect the memory and AXI read and write rates, the core model registers read and write callbacks to the corresponding address ranges of the processor and counts the number of accesses:
To capture changes in the processor’s or memory’s frequency register as well as the processor’s power state register three more write callbacks apply to the particular addresses. Furthermore, a fetch callback can be registered to observe the occurrence of wfe and wfi instruction in its callback function. The signature of the read, write and fetch callback function is:

```c
#define VMI_MEM_WATCH_FN(_NAME) void _NAME( 
    vmiProcessorP processor, Addr address, \ 
    Uns32 bytes, const void *value, \ 
    void *userData, Addr VA )
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

Especially the address and the value are helpful parameters of the read, write and fetch callback function. With these information it is possible to analyze the data on the read, written or fetched address without further effort.

The already mentioned model timer uses of a periodic callback to calculate the CPU load or the read and write rates and is configured for each core model with the following functions.

```c
vmiModelTimerP vmirtCreateModelTimer( 
    vmiProcessorP processor, vmiICountFn icountCB, 
    Uns32 scale, void *userData );
```

The callback function is called every time the model timer hits a specified value, with the given signature:

```c
#define VMI_ICOUNT_FN(_NAME) void _NAME( 
    vmiProcessorP processor, vmiModelTimerP timer, \ 
    Uns64 iCount, void *userData )
typedef VMI_ICOUNT_FN((*vmiICountFn));
```

The model timer works with the MIPS parameter of the processor model, to generate a callback when the instruction count of the processor model reaches the defined value. For easier handling the timer can be configured by specifying a delta value:

```c
void vmirtSetModelTimer( 
    vmiModelTimerP modelTimer, 
    Uns64 delta );
```

To calculate the relative CPU load or the read and write rates the MIPS parameter is needed, since OVP interprets this value in a way similar to the “frequency” of the processor:
For computing the memory and the AXI loads, the number of callbacks is used with their current frequencies and the interval of the model timer. To process the CPU load OVP provides two API calls: \texttt{vmirtGetICount} returns the amount of all available instruction slots, whilst \texttt{vmirtGetExecutedICount} returns the number of effectively executed instructions:

\begin{verbatim}
Flt64 vmirtGetProcessorIPS(
    vmiProcessorP processor);
\end{verbatim}

When executing an application such as Linux, the values can differ significantly, since many \texttt{wfe} and \texttt{wfi} instructions could occur. By calculating the value’s differences to the last timer callback and dividing the values, the result is the relative CPU load. When a frequency change of the processors is recognized, the processor model should adjust its MIPS parameter and thus, the virtual “frequency” of the processor. For that, OVP offers the possibility to specify a percentage value, called the derate factor:

\begin{verbatim}
Uns64 \{vmirtGetICount/vmirtGetExecutedICount\}(vmlProcessorP processor);
\end{verbatim}

If the value is set to 0.0 \% the processor model runs at its configured MIPS rate, else if it is set to 100.0 \% the processor model no longer executes no instructions.

\begin{verbatim}
void vmirtSetDerateFactor(
    vmiProcessorP processor, Flt64 factor);
\end{verbatim}

The last scenario is useful if the core is turned off or in dormant mode. In that circumstance the virtual processor halts completely.

One of the remaining future works is especially the implementation of a power model to fill the current values of the virtual power rails as well as to output the power values in the tracing engine. To do so, an interface is already prepared as an own module, which has access to all captured data of the cores and the platform. Especially for the programmable logic of the Xilinx Zynq MPSoC, further observance capabilities of the power management techniques will be implemented and added to the Power Interception Library.

8. **SAFEPOWER VLNV LIBRARY COMPONENTS**

The following are the components that are provided in the VLNV library and used on the SAFEPOWER project.

\begin{table}[H]
\centering
\begin{tabular}{|l|l|l|l|}
\hline
**Vendor** & **Library** & **Name** & **Version** \\
\hline
SAFEPOWER.ovpworld.org & module & Zynq\_PL\_TTELNoC & 1.0 \\
& & Zynq\_PL\_NostrumNoC & 1.0 \\
\hline
\end{tabular}
\caption{VLNV Library Components}
\end{table}
| xilinx.ovpworld.org | Zynq_PL_DualMicroblaze | 1.0 |
| | Zynq_PL_NoC | 1.0 |
| | Zynq_PL_TTELNoC_node | 1.0 |
| | Zynq_PL_NostrumNoC_node | 1.0 |
| | Zynq_PL_SingleMicroblaze | 1.0 |
| | Zynq_PL_NoC_node | 1.0 |
| peripheral | TTELNode | 1.0 |
| | NostrumNode | 1.0 |
| | node | 1.0 |
| module | Zynq | 1.0 |
| | zc702 | 1.0 |
| | zc706 | 1.0 |
| | Zynq_PS | 1.0 |
| | Zynq_PL_Default | 1.0 |
| Peripheral | zynq_7000-tz_GPVsecurity | 1.0 |
| | zynq_7000-gem | 1.0 |
| | zynq_7000-swdt | 1.0 |
| | zynq_7000-sdio | 1.0 |
| | zynq_7000-usb | 1.0 |
| | zynq_7000-dmac | 1.0 |
| | zynq_7000-tz_security | 1.0 |
| | zynq_7000-ttc | 1.0 |
| | zynq_7000-ddrc | 1.0 |
| | zynq_7000-devcfg | 1.0 |
| | zynq_7000-slc | 1.0 |
| | zynq_7000-ocm | 1.0 |
| | axi-timer | 1.0 |
| | zynq_7000-qspi | 1.0 |
| | zynq_7000-can | 1.0 |
| | logicore-fit | 1.0 |
| | zynq_7000-gpio | 1.0 |
| | zynq_7000-iic | 1.0 |
| | zynq_7000-spi | 1.0 |
| | axi-gpio | 1.0 |
| | zynq_7000-qos301 | 1.0 |
| processor | microblaze | 1.0 |
| semihosting | microblazeNewlib | 1.0 |
| arm.ovpworld.org | L2CachePL310 | 1.0 |
| | SmartLoaderArmLinux | 1.0 |
| processor | arm | 1.0 |
| semihosting | armAngel | 1.0 |
| | armNewlib | 1.0 |
The library ‘semihosting’ contains a special form of binary interception library that is used with the execution of a bare metal application to map the use of library functions such as open, read, write etc directly to host system resources. This allows, for example an application running on a processor in the VP to directly access resources on the host system. It could open a file on the host file system to read test data from it/to write results to it. It can allow the application to use the printf function to display information directly to the host stdout or allow a user to interact with the application through the host stdin.
9. GLOSSARY / TERMINOLOGY

**OVP** - Open Virtual platform organization that was created by IMP to make the virtual platform modeling APIs freely available.

**iGen** - IMP productivity tool that has a powerful script based function API that is used to create models and templates.

**OVPsim** - Simulator for Open Virtual Platforms that executes platforms and models coded in the OVP APIs

**CpuManager** - IMP commercial simulator. This is a super-set of the OVPsim simulator, fully supporting all OVP defined APIs.

**OP API** - OVP Platforms API - C API used for creating and controlling virtual platforms. 2nd generation API, replaces ICM API.

**VMI API** – C API used in processor modeling and intercept libraries.

**VMIR T** – the specific VMI API functions used at Run Time

**VMIMT** – the specific VMA ALI functions used at Morph Time

**BHM** – BeHavioral Modeling API used in the creation of internal structure and actions in Peripheral models.

**PPM** – Peripheral Platform Modeling API used in Peripheral models to create access within the virtual platform.

**VP** – Virtual Platform

**VLNV** – The library structure used to provide a unique identifier based upon the Vendor, Library, Name and Version

**TVS** – Timed Value Streams

**VCD** – Value Change Dump, a file format which can be used to show how signals or values change over time and can be displayed graphically using commonly available players.
10. REFERENCES


