MEEP

MareNostrum Experimental Exascale Platform

# D4.2 - FPGA RTL revision 1 release. Complete verification environment (M18)

Version 1.1

## Document Information

| | |
|---|---|
| **Contract Number** | 946002 |
| **Project Website** | https://meep-project.eu/ |
| **Contractual Deadline** | 30/06/2021 |
| **Dissemination Level** | Public (PU) |
| **Nature** | Other (O) |
| **Author** | Teresa Cervero (BSC) |
| **Contributors** | Iván Vera (BSC), Alberto Muñio (BSC), Karim Charfi (BSC), Avani Bharadava (BSC), Tejas Limbasiya (BSC), Daniel J. Mazure (BSC), Bachir Fradj (BSC), Alexander Fell (BSC), Leon Dragiç (UNIZG), Said Seferbey (TÜBITAK), Mario Kovaç (UNIZG), Alp Sarkisla (TÜBITAK), Igor Piljic (UNIZG), Nakul Tandon (BSC), Alexandar Duricic (BSC), Andrés Almarcha (BSC), Monika Raj (BSC), Prashant Ahuja (BSC), Borja Pérez (BSC), Mariano Benito (BSC), Rahul Shrivastava (BSC) |
| **Reviewers** | John Davis (BSC), Francesco Minervini (BSC), Alireza Monemi (BSC), Oscar Palomar (BSC), Nehir Sonmez (BSC), Peter Wilson (BSC), Adrià Armejach (BSC), Filippo Mandovani (BSC) |

The MEEP project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 946002. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Croatia, Turkey.

© 2020 MEEP. The MareNostrum Experimental Exascale Platform. All rights reserved.

## Change Log

| Version | Description of Change |
|---|---|
| V 1.0 | Merge into only one document the three parts of the WP4 M18 deliverable |
| V 1.1 | Adjust format |

# Chapter 1: FPGA RTL revision 1 release

## Chapter Information

| | |
|---|---|
| **Author** | Iván Vera (BSC) |
| **Contributors** | Teresa Cervero (BSC), Alberto Muñio (BSC), Karim Charfi (BSC), Avani Bharadava (BSC), Tejas Limbasiya (BSC), Daniel J. Mazure (BSC), Bachir Fradj (BSC), Alexander Fell (BSC), Leon Dragiç (UNIZG), Said Seferbey (TÜBITAK), Mario Kovaç (UNIZG), Alp Sarkisla (TÜBITAK), Igor Piljic (UNIZG) |
| **Reviewers** | John Davis (BSC), Francesco Minervini (BSC), Alireza Monemi (BSC), Peter Wilson (BSC) |

## Chapter Change Log

| Version | Description of Change |
|---|---|
| V 1.0 | Initial draft for internal review |
| V 1.1 | Internal reviewers feedback |
| V 1.2 | All the information related to Coyote is moved to a new document (D4.2.c) |
| V 1.3 | Restructure of the content into different sections |
| V 1.4 | Final version after internal review |
| V 1.5 | Adding pointers to the Gitlab repositories |

# Chapter 2: Verification Strategy

## Chapter Information

| | |
|---|---|
| **Author** | Nakul Tandon (BSC) |
| **Contributors** | Aleksandar Duricic (BSC), Monika Raj (BSC), Andrés Almarcha (BSC), Prashant Ahuja (BSC) |
| **Reviewers** | Teresa Cervero (BSC), John Davis (BSC), Pete Wilson (BSC), Nehir Sonmez (BSC), Oscar Palomar (BSC) |

## Chapter Change Log

| Version | Description of Change |
|---|---|
| **V 1.0** | Initial draft for internal review |
| **V 1.1** | Adding content to the Executive Summary and the Introduction.<br>Minor format changes |
| **V 1.2** | Internal reviewers feedback |
| **V 1.3** | Version after internal review |
| **V 1.4** | Rearranging the structure of the document |

# Chapter 3: Performance Modelling with Coyote Simulator: cores, NoC and memories

## Chapter Information

| | |
|---|---|
| **Author** | Borja Pérez (BSC) |
| **Contributors** | Alexander Fell (BSC), Mariano Benito (BSC), Rahul Shrivastava (BSC) |
| **Reviewers** | John Davis (BSC), Alireza Monemi (BSC), Peter Wilson (BSC), Teresa Cervero (BSC) |

## Chapter Change Log

| Version | Description of Change |
|---|---|
| **V 1.0** | Initial draft for internal review |
| **V 1.1** | Adding content according to the internal reviewers comments |
| **V 1.2** | Final document after internal review |

## COPYRIGHT

© Copyright by the MEEP consortium, 2020

This document contains material, which is the copyright of MEEP Consortium members and the European Commission, and may not be reproduced or copied without permission, except as mandated by the European Commission Grant Agreement no. 946002 for reviewing and dissemination purposes.

ACKNOWLEDGEMENTS

The MEEP project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 946002. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Croatia, Turkey.

The partners in the project are BARCELONA SUPERCOMPUTING CENTER - CENTRO NACIONAL DE SUPERCOMPUTACION (BSC), FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING, UNIVERSITY OF ZAGREB (UNIZG-FER), & THE SCIENTIFIC AND TECHNOLOGICAL RESEARCH COUNCIL OF TURKEY, INFORMATICS AND INFORMATION SECURITY RESEARCH CENTER (TÜBİTAK BILGEM).

The content of this document is the result of extensive discussions within the MEEP © Consortium as a whole.

## DISCLAIMER

The content of the publication herein is the sole responsibility of the publishers and it does not necessarily represent the views expressed by the European Commission or its services. The information contained in this document is provided by the copyright holders "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the members of the MEEP collaboration, including the copyright holders, or the European Commission be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of the information contained in this document, even if advised of the possibility of such damage.

# Table of Contents

# 1. Executive Summary

This document is part of a collection of deliverables that describes the MareNostrum Experimental Exascale Platform (MEEP) Project. This project is organized into six work packages (WP), where only three of them are in charge of the technical development: WP4, WP5 and WP6. In greater detail, this layered structure and the relationship between WPs are depicted in Figure 1.

- WP4 describes the Accelerated Compute and Memory Engine (ACME) accelerator architecture and its related RTL, verification and performance modeling simulations.
- WP5 covers software, impacting all the layers of the software stack.
- WP6 is where the other WPs come together, by running SW applications in a miniaturized version of the ACME architecture on the emulation platform.

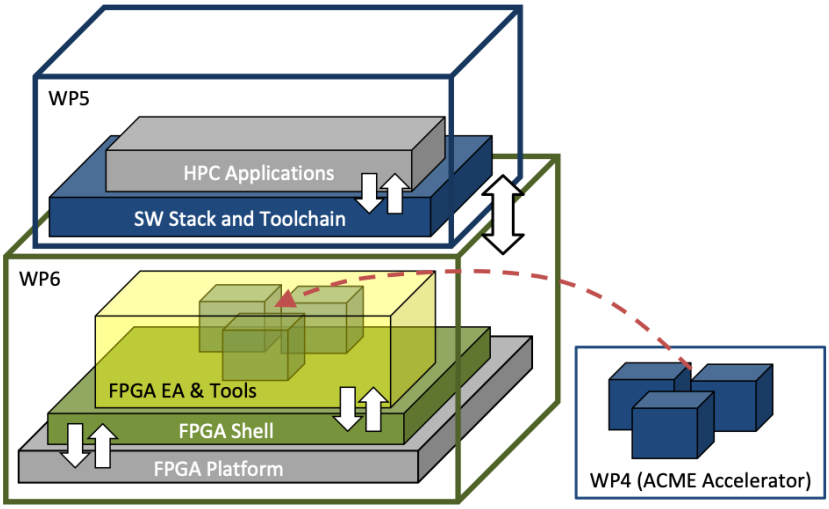



*Figure 1. The relationship of the different work packages, software (WP5), RTL and Architecture (WP4), and FPGA emulator (WP6)*

Based on the information presented in the previous deliverable, *D4.1 System architecture and emulation platform specification*, this deliverable goes deeper into implementation details. For simplicity, the information has been split into three specialized chapters, according to the nature and scope of their reported activities: one for the RTL team (Chapter 1), one for the design verification (DV) team (Chapter 2), and one for the Performance Modelling team (Chapter 3).

**D4.2: *FPGA RTL revision 1 release. Complete verification environment* (M18)** After the completion of the first deliverable, D4.1 document, the computational components that constitute the proposed accelerator architecture are being developed. These designs are aligned with the requirements specified in D4.1. For clarity, this deliverable has been separated into three different documents:

**D4.2 *Chapter 1 - FPGA RTL revision 1 release***. An initial release of the developed accelerator RTL design is explained in this document, which will serve as a mid-term checkpoint for the RTL design process. The content of this document includes the design of the basic computational core,

namely Vector And Systolic Tile core, together with its components like the scalar core and its computational coprocessors, the vector processing unit (VPU), and the systolic arrays (SA).

**D4.2** ***Chapter 2 - Verification Strategy***. A full verification strategy and plan, which is in accordance with the ACME requirements and specifications, are explained. The strategy includes the verification environment, methodology and tools used to guarantee the high quality of the released verification components.

**D4.2** ***Chapter 3 - Performance Modelling with Coyote simulator: core, NoC and memories***. A detailed view of the current status of the Coyote simulator is provided. We also show preliminary simulation results showing that the model can distinguish expected behaviours such as effects of NoC latency.

# CHAPTER 1: FPGA RTL revision 1 release

## 2. Summary

The scope of this chapter is constrained to the D4.2 Part A - *FPGA RTL revision 1 release*. Due to the research nature of the ACME accelerator, the IPs will be refined over time based on verification and performance simulation results, architectural decisions and further design exploration as well as results from the FPGA emulator. The primary purpose of this document is to describe the implementation of the computational unit that builds the vector and systolic array (VAS) accelerator as part of the computational engine of the ACME accelerator. More in detail, the scope of this document is circumscribed to the VAS Tile core unit, which is composed of a scalar core, and three loosely coupled co-processors (a vector processor unit and two specialized systolic arrays). Thus, the information in the document has been structured into four more sections (Section 2 - Section 5); Section 2 starts by contextualizing the RTL work as part of the MEEP project. This section also presents the current development status of the different components that build the proposed accelerator (ACME), and enables this document to stand on its own.

Section 3 is focused on the milestone for month 18 (M18); which is the *MS1: FPGA RTL revision 1 complete*. It provides a detailed bottom-up description of the different modules that compose the processor core in the ACME VAS Tile.

In parallel with the activities described in the previous section, the RTL team has been running other activities to progress upon the scaled-down version of the ACME accelerator suitable for emulating in the MEEP FPGA framework. Section 4 presents all the stretch goals obtained at this point, and also the description of some relevant tasks closely related to the next release of the VAS Tile core.

A summary of the outcomes produced by the RTL team, together with a plan for developing future activities, are provided in Section 5.

## 3. Introduction

The design team, as part of WP4, is concentrating on designing and developing the RTL code that fulfills the requirements and specifications for the MEEP targeted accelerator (ACME). The high-level view of the ACME accelerator is shown in Figure A.2. The expected achievements for WP4 are being achieved by following an incremental bottom-up approach, starting by working on the minimal computational components of the accelerator (block level), and then integrating them to build up more complex structures in the architectural hierarchy (system level, and then SoC level).

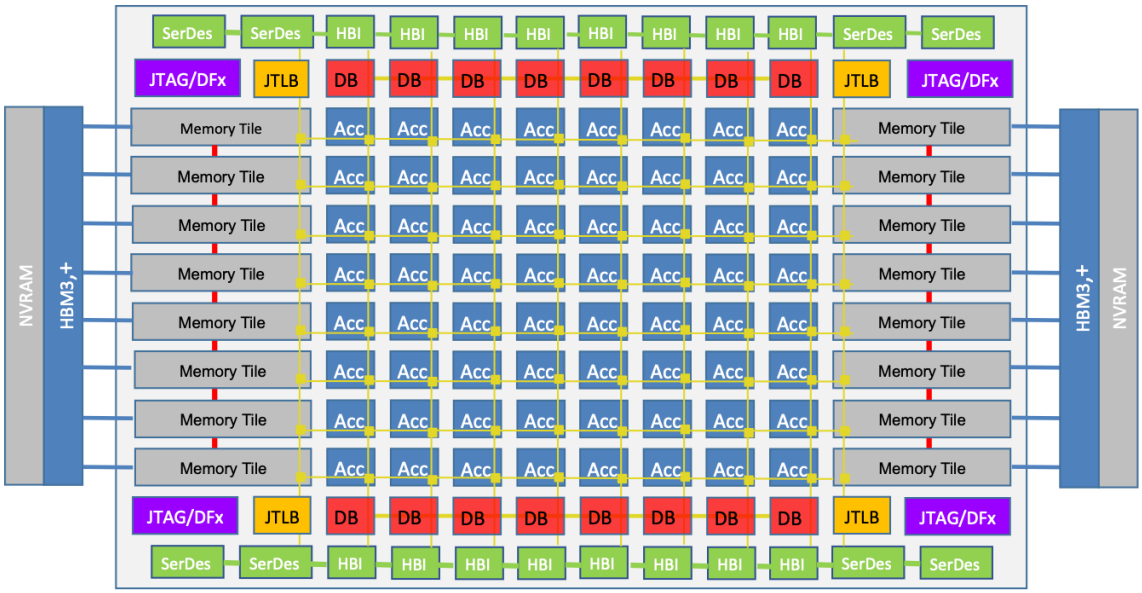



*Figure 2. A high-level view of the ACME accelerator architecture*

As Figure 2 shows, the ACME accelerator is composed of multiple instances of two main components: (1) the VAS Tile (VAS in Figure A.2), and (2) the Memory Tile. These components are interconnected using networks on chip (NoC). More information about the overall ACME accelerator is available in the previous deliverable, *D4.1 System architecture and emulated platform specifications*.

It is important to highlight the fact that the design team is working on a scaled-down version of ACME - small enough to fit in the FPGA, but big enough to test and validate the expected performance improvement of the ACME accelerator when it executes different kinds of HPC applications with dense and sparse workloads. This scaled-down version will include some number of VAS Tiles (interconnected among using a mesh topology NoC), and Memory Tiles. The number of final instances of each of these components will be dependent on two main factors: (1) the size of each developed component in terms of resources, and (2) the reserved resources on the FPGA-based emulation platform for the emulated accelerator (EA).

According to the MEEP project proposal, WP4 should achieve its first milestone (MS1) consisting of *FPGA RTL revision 1 complete, with a complete verification environment* at M18. The verification environment is out of the scope of this document, and the following sections are only related to the RTL activities. More in detail, following a bottom-up approach, like the one depicted in Figure 3, the scope of this document is constrained to the VAS Tile core component, which implies working on the RISC-V scalar core, the VPU, and the SAs.

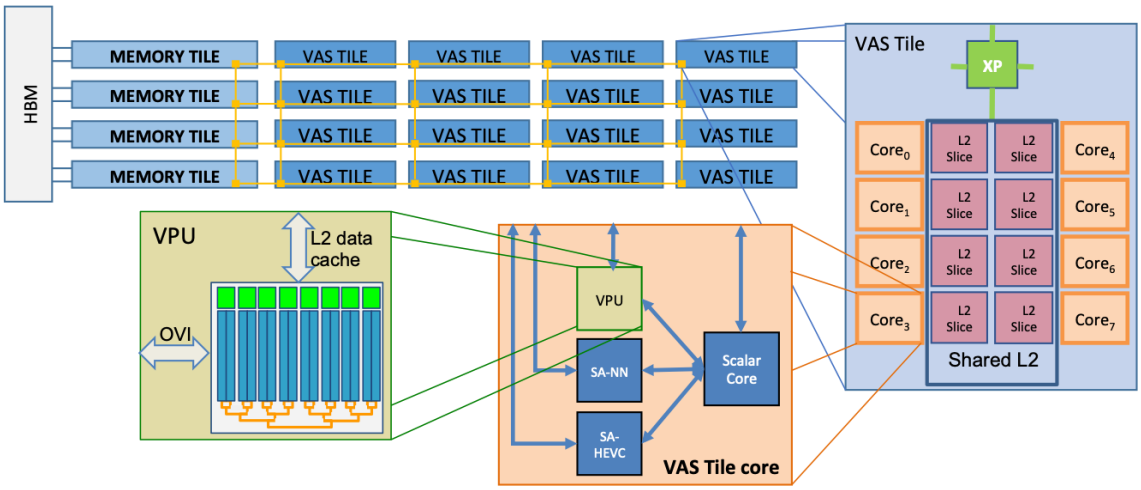



*Figure 3. Top-down hierarchical view of the ACME VAS Tile component*

Additionally to the initial release of the VAS Tile core (Section 3), this document also describes other stretch activities done along these previous months (Section 4). All these will contribute to extending the current capabilities of the VAS Tile core.

## 3.1. Definitions and notations

Table 1 describes the frequently-used ACME concepts mentioned in this document.

| Concept | Definition |
|---|---|
| ACME | Accelerated Compute and Memory Engine accelerator, composed of a grid of Tiles. It is a self-hosted and disaggregated architecture. |
| Tile | An element with a specific function inside the ACME architecture. There are two different types of tiles in the targeted accelerator: VAS Tile (highly used for performing compute-bound operations), and Memory Tile (highly used for performing memory-bound operations). |
| VAS Tile | Vector and Systolic Array Tile defined as a multi-core system, with shared L2 data cache memory and NoC interfaces to interconnect with other VAS Tiles and Memory Tiles.<br>Within a VAS Tile there are up to 8 VAS Tile cores. |
| VAS Tile core | It is an IP composed of a RISC-V scalar core with three loosely coupled co-processors: one VPU and two specialized SAs. As part of the ACME disaggregated architecture, the VAS Tile core is where scalar and arithmetic vector instructions are executed. |
| Lagarto | BSC in order RISC-V processor. |
| VPU | Vector Processing Unit suitable for executing RISC-V ISA vector extension instructions. It has two kinds of interfaces: one with the scalar core, and another with the L2 data cache/scratchpad. |
| SA | Systolic Array accelerator. ACME includes two specialized designs; one for processing neural networks (SA-NN), and another for HEVC video processing (SA-HEVC). These accelerators have the same interfaces to other blocks by using a SA-Shell. |
| SA-Shell | It is a wrapper interface template to ease the integration of similar kinds of SAs in ACME. It defines two types of interfaces, one for control, and another for memory. |
| Dvino | Originally, it is a BSC SoC composed of the integration of a Lagarto core, a 2-lanes VPU, and an L1 and an L2 cache memory. |

| | |
|---|---|
| | The 2-lanes VPU comes from the EPI project, and the memory hierarchy (except the L1 data cache) comes from a lowRISC project. |
| Memory Tile | A cluster of functionality including the Memory Controller CPU (MCPU), Shared L3/Row Buffer, L2 Instruction Cache, TLB, and NoC interfaces for performing memory instructions. |
| MCPU | Memory Controller CPU. As part of the ACME disaggregated architecture, the MCPU is a processor core responsible for the execution of all the memory and atomic instructions. |
| NoC | Network on Chip. Communication mechanism used for interconnecting different Tiles (VAS Tiles to VAS Tiles, and Memory Tiles to VAS Tiles. |
| ME | A microengine is a DMA controller mechanism to handle the data movements between the scratchpad and the vector register file in the VPU. |
| L2 cache / Scratchpad | L2 data cache that might be fully or partially configurable to be used as a private space of memory. |
| SoC | Acronym for system on chip is an IC that integrates all the components into a single chip. |
| OVI | Open Vector Interface. It is an interface and bus protocol to interconnect a Vector Processor Unit with a CPU. |

*Table 1. Definition of ACME concepts*

## 3.2. Source code repositories

The code that completes this deliverable is organized in several projects in a Gitlab *MEEP RTL designs* repository as follows:

**Section 3 (M18 deliverables):**

- VPU v1.1: https://gitlab.bsc.es/meep/rtl_designs/meep-vpu/-/tree/v1.1.1
- SA-HEVC: https://git.hpc.fer.hr/ldragic/sa-hevc-acc-ext
- SA-NN: https://gitlab.bsc.es/meep/rtl_designs/meep-TBTK/meep_tbtk_nn
- Scalar core: https://gitlab.bsc.es/meep/rtl_designs/drac-inorder/-/tree/feature/meep-SAs-ovi
- VAS Tile core:https://gitlab.bsc.es/meep/rtl_designs/meep_dvino/-/tree/v.1.0.0

**Section 4 (Stretch goals):**

- VPU v1.2: https://gitlab.bsc.es/meep/rtl_designs/meep-vpu/-/tree/v1.2
- ME: https://gitlab.bsc.es/meep/rtl_designs/meep-uengines/-/tree/dev

# 4. M18 goals: FPGA RTL initial release

The RTL Team is following an incremental bottom-up implementation approach, starting at the block level and moving up to higher hierarchical levels. At this point in time, the M18 outcomes include developments at block and system level up to the VAS Tile core, and more details about them are provided in the current section.

## 4.1. Block level IPs

Block level development includes three key components, as it is shown in Figure A.4, (1) the VPU, (2) two different specialized SAs and (3) the scalar core. These blocks have been implemented in parallel with no dependencies among them, except on the design and definition of their interfaces, only in those cases in which a future interaction is required. As it will be described later on in this document, the integration of these components builds the minimal computational unit of the ACME accelerator, the VAS Tile core.

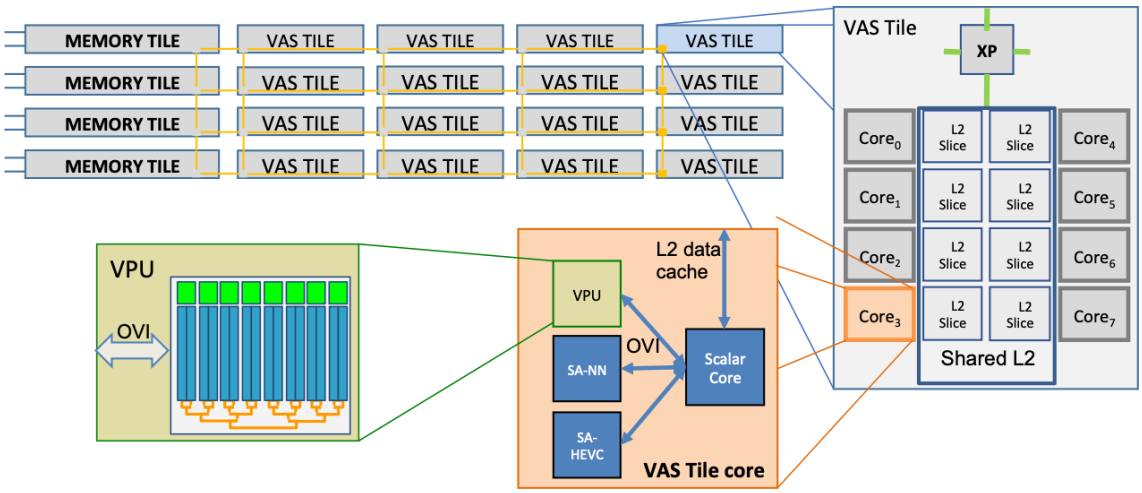



*Figure 4. Representation of the available IPs at M18 (colored ones)*

This section goes deeper in detail about the colored blocks represented in Figure 4.

### 4.1.1. Vector Processing Unit (VPU)

The ACME accelerator is a collection of coprocessors that accelerate various applications connected to a scalar core that does all the management and orchestration of the accelerators. When the Vector Accelerator is present, each scalar processor has a Vector Processing Unit (VPU) which implements the RISC-V Vector Extension specification version v0.7.1, with extensions.

Our starting point is the vector processor that has been designed in EPI. BSC and UNIZG-FER are the main contributors to that original VPU. This is connected to a scalar processor via the Open Vector Interface [A1]. Instructions are forwarded, along with cache lines, over this interface to the VPU. This interface is sufficient for applications with unit stride access patterns. For applications with data that is not laid out in memory as stride one, this interface becomes wasteful and energy inefficient, transporting useless data from the DRAM, through the cache hierarchy to the VPU to be discarded, although still stored on-chip. Thus, we extend the VPU to also have its own memory interface to the shared L2 scratchpad, where data is stored in the scratchpad as unit stride vectors, pre-processed by the Memory Controller CPU (MCPU), containing a dense representation of the data. This way, all data from the scratchpad is useful to coprocessors. In the case of the VPU, this is similar to a Level 2 vector register file (The Level 1 register is the physical register file in the VPU.). The combination of scalar core, VPU, and a portion of the L2/scratchpad is a *processing subsystem*. We modify the VPU number of lanes to balance logic utilization. Furthermore, we group the lanes into groups of two lanes (lane-pair) that can be mapped to threads. This provides an easy partitioning of the vector register file and intra-lane communication. For dense workloads, all lane-pairs can be fused and utilized and mapped to one

thread. For sparse workloads, we can reduce the computational resources to a single lane-pair per thread and if applicable map multiple threads within a single core. This exposes many outstanding memory requests to the memory controller. Furthermore, because the MCPU knows about the virtual vector length, the MCPU can see into the future memory requests, exposing memory efficiency across many threads mapped to the lanes. This simple architecture exposes additional compute or memory resources when needed and provides a simple scaling mechanism to do so.

#### 4.1.1.1. Features overview

The Vector Processing Unit has initiated a transition from its original EPI based design to incrementally integrate those features detailed for the accelerator, according to the ACME specifications included in the previous deliverable D4.1. This schedule of releases aims to periodically offer viable releases that demonstrate new functionalities so both design and verification can progress in parallel. A general overview of VPU development status is shown in Table 2 as follows:

<table>
<tr><th colspan="3">General VPU architecture features</th></tr>
<tr><th>Reference design (EPI)</th><th>MEEP VPU v1.1</th><th>MEEP VPU v1.2 Beta</th></tr>
<tr><td>8 lanes</td><td colspan="2">4 lanes (2 lane-pairs) per VPU by default, and configurable up to 8 lane-pairs</td></tr>
<tr><td>Vlength: 256 elements x 64 bits</td><td colspan="2">Maximum vector length: 16 elements x 64 bits per lane-pair of Fused Vector Lanes (FVL)</td></tr>
<tr><td colspan="3">1 Fused Multiply Accumulate (FMA) unit per lane (2 DP FLOP/cycle)</td></tr>
<tr><td colspan="3">Support for 64 and 32 bit FP operation</td></tr>
<tr><td colspan="3">Support for 64, 32, 16, and 8 bit integer operations, signed and unsigned</td></tr>
<tr><td colspan="3">Single vector lane-to-memory data transfer width: 64 bits per cycle</td></tr>
<tr><td>Number of physical vector registers per core: 40</td><td colspan="2">A configurable number of physical vector registers per core: 32, 40 (by default), 48, 64...</td></tr>
<tr><td colspan="2">Vector Register File (VRF) Single-Port limited access</td><td>VRF Dual-Port access to allow for continuous L2 data streaming<br><br>Modified VRF number of banks to improve logic utilization<br><br>Redesign of lane control logic to leverage VRF concurrent read/write accesses</td></tr>
<tr><td colspan="3">Limited out-of-order capability in vector memory operations:<ul><li>Load/Store, and arithmetic operation can be executed in parallel</li><li>Two loads in-flight</li></ul></td></tr>
</table>

*Table 2. VPU features (M18 VPU v1.1 release; next one VPU v1.2)*

The objectives of this VPU v1.1 release are: (1) reduce logic utilization of the VPU, so that is possible to instantiate the multiple modules that compose ACME at the emulation platform; (2) reduce the total memory size of the classic hardware registers internal to the VPU that are used to store short vectors; (3) start to encapsulate and organize the logic of modules planned to be modified in next development stages.

The VPU is redesigned with several lanes with the intention to minimize its logic utilization, provided that is still possible to demonstrate all of its functionalities of interest. In the design roadmap it is envisioned that lanes will be grouped in pairs so each of them can run independent instruction threads, being possible to scale up or down based on the computation intensity of a thread. This allows executing one thread using four lanes or two threads using two lanes each. Furthermore, this configuration is enough to demonstrate the activity of the new and improved inter-lane communication logic that substitutes the original ring lane interconnect. By downsizing the VPU, example designs can be composed of a higher number of submodules despite the limited resources available in our FPGAs. The emulation capability of inter-modules interaction is hence strengthened. We are aware of the fact that decrementing the number of lanes (from 16 to 4) will impact on the final throughput. However, this first approach reduces the level of the design complexity and serves as a valid Proof-of-Concept to get initial insights about the strengths and limitations of the proposed architecture. Once the scaled-down version of the VPU fulfills the expected requirements, several simulations might be run by exploring how the number of available pairs of lanes impact on the system (area, performance, and power consumption).

The evolution of the structure of the VPU is visible by comparing the original EPI VPU (Figure 5.a) with the current MEEP VPU (Figure 5.b). The block diagrams show how the VPU is being moved from having one vector lane as a basic processing element, in the case of the EPI VPU towards one pair-lanes as a basic processing unit in MEEP.

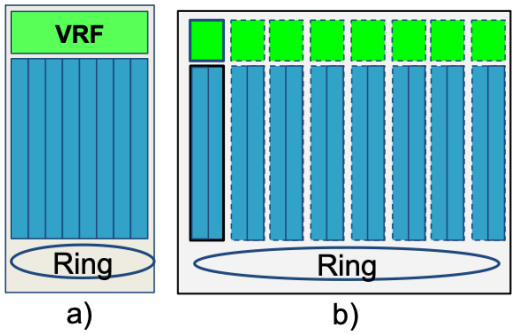



*Figure 5. A simplified view of the VPU evolution. (a)EPI VPU; (b)MEEP VPU with fused-lane pairs*

VPU arithmetic modules operate on data stored in the VRF. The VRF is divided into several slices equal to the VPU number of lanes. Likewise, each of these slices is partitioned into 5 banks. In total, the original EPI VRF stored the data of 40 physical registers, each of them with a maximum vector length of 256 elements (VELEN) for a SEW of 64 bits. Of those registers, 32 are directly addressable by RISC-V vector instructions while the remaining 8 are employed to increase performance through registers renaming. In conclusion, each VRF bank held 256 elements x 64 bits (2 KB), for a single VRF slice 5 x 2 KB = 10 KB, and in an example VPU with 8 lanes 8 x 10 = 80 KB of memory.

ACME wants to support both short and long vectors. MEEP proposal is to reduce the number of vector elements on each of those physical registers, hence VPU internal VRF memory usage is diminished but allows to support short vectors. For long vectors, the VRF is virtually extended to the L2 scratchpad through DMA access and this will be part of future releases.

The initial VRF register size could be further decreased to support 8 elements if simulations show that is beneficial. An important element to consider in VRF modification is the future support of multi-threading in the scope of a vector lane-pair. In that scenario, each thread (or a pair of fused vector lanes) should have access to an entire memory space composed of a minimum of 32 independently addressable physical registers, each of them built from 16 vector elements of a maximum SEW of 64 bits. This translates to a total memory utilization of (VPU_LANES/2) x 16 x 64 bits x N_REGS where N_REGS is within the [32.inf) range. In an example VPU with 4 lanes and 40 physical registers, total memory use is (4/2) x 16 x 64 x 40 or 10 KB.

The number of VRF banks in a VRF slice is 5 to compensate for the bandwidth limitations derived from the SRAMs used for the VRF implementation being Single-Port. This was a design decision based on the much higher area cost of Dual-Port SRAM cells for the original ASIC target technology. Vector elements of each register are shared across all the VRF slices for computation of an instruction to occur in parallel. To obtain the products of contiguous vector elements at the same cycle, each VRF slice stores VELEN/VPU_LANES values (VELEN_LANE) with a stride of VPU_LANES. VRF mapping is that each lane LANE_ID, stores elements V[LANE_ID+5xN][LANE_ID+8xM], where LANE_ID $\epsilon$ [VPU_LANES-1..0], N $\epsilon$ [N_REGS..0], and M $\epsilon$ [VELEN_LANE-1..0]. When SEW is other than the memory block data width (64 bits), each vector element is subdivided so that a 64-bit word W64.0 is composed of {W8.7, W8.6, …, W8.0}. The combination of an even number of vector elements and VPU lanes, and an odd number of VRF banks translates into a physical register mapping following a "*barber's pole*" [A2] pattern. In order to read 5 consecutive elements of a selected physical register, 2 addresses need to be generated because of the misalignment introduced by the mapping pattern. An example of the aforementioned VRF data distribution is depicted in Figure 6.

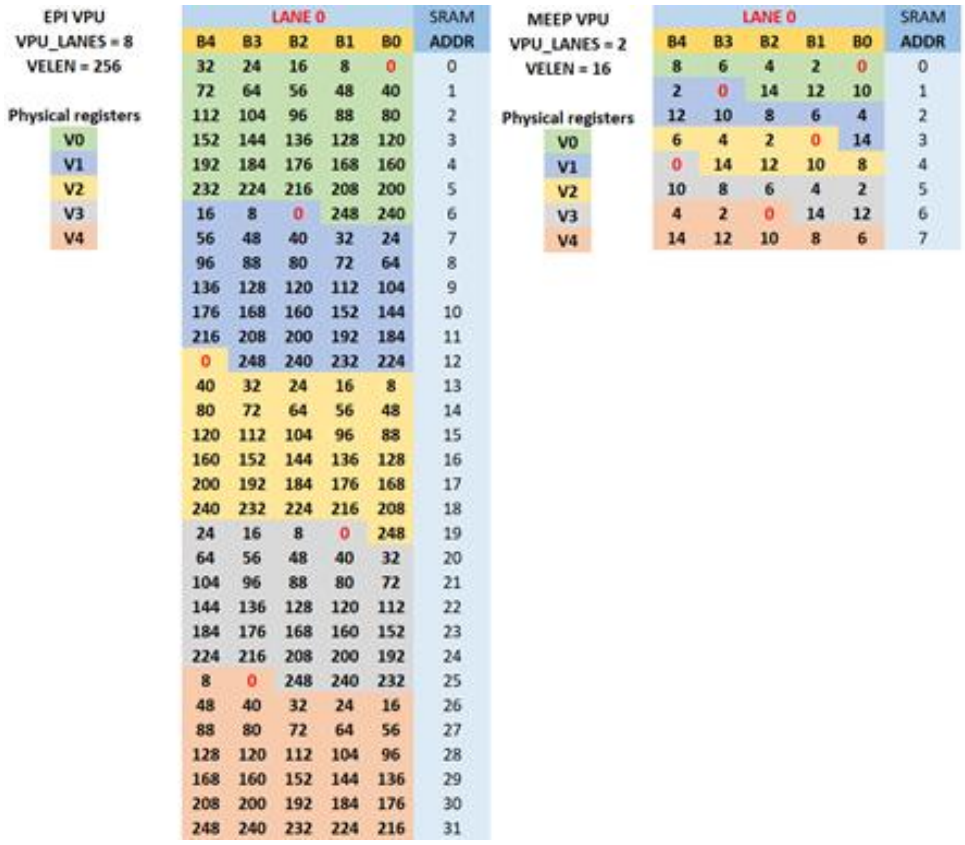



*Figure 6. VRF physical registers mapping in EPI and MEEP v.1.1 VPUs*

RISC-V instructions specify which physical registers have to be accessed to complete the issued command. In general, vector start is set to the zero position of each of those registers. As illustrated in Figure A.6, the zero index position on a particular VRF bank for each of the physical registers varies according to the VPU number of lanes and vector elements (VELEN). In the example provided, EPI VPU case has {V0[0]:B0, V1[0]:B2, V2[0]:B4, V3[0]:B1, V4[0]:B3}, whereas for MEEP VPU {V0[0]:B0, V1[0]:B3, V2[0]:B1, V3[0]:B4, V4[0]:B2}. This inconsistency between the physical register elements addresses and their VRF bank mapping occurs because of the necessity of modifying the SRAM address pointer generator logic accordingly. The crossbars and shufflers are available at different parts of the design used to correctly allocate data to be read or written into the VRF need also to be adjusted.

### 4.1.2. Systolic Arrays

The Vas Tile Core includes two different Systolic Arrays (SAs) accelerators as part of its design: (1) one SA for neural networks (SA-NN), and (2) one SA for video processing (SA-HEVC). These two SAs co-processors are designed to outperform the execution of specific applications. As it might be seen in Figure A.7, both SAs use a wrapper (*SA-Shell*) to create common and homogeneous interfaces with the scalar core (*OVI*) and the L2 data cache when it is fully or partially conFigured as a scratchpad (*Memory interface*).

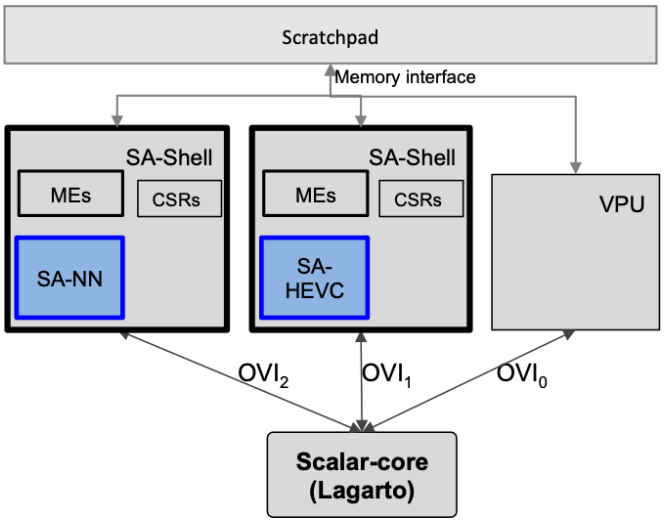



*Figure 7. Block diagram of all the co-processors in a VAS Tile core*

The SA-Shell will be introduced later on in this document, and consequently this section is exclusively dedicated to explain the implementations of the SAs (blue boxes in Figure 7).

#### 4.1.2.1. SA accelerator for video processing

The SA-HEVC is a systolic array accelerator specialized for image and video processing based on the HEVC/H.265 video coding standard. The accelerator consists of four processing cores (Direct Cosine Transform-DCT, Quantization - Q, Dequantization - $Q^{-1}$, and Inverse DCT) and a control module as shown in Figure 8.

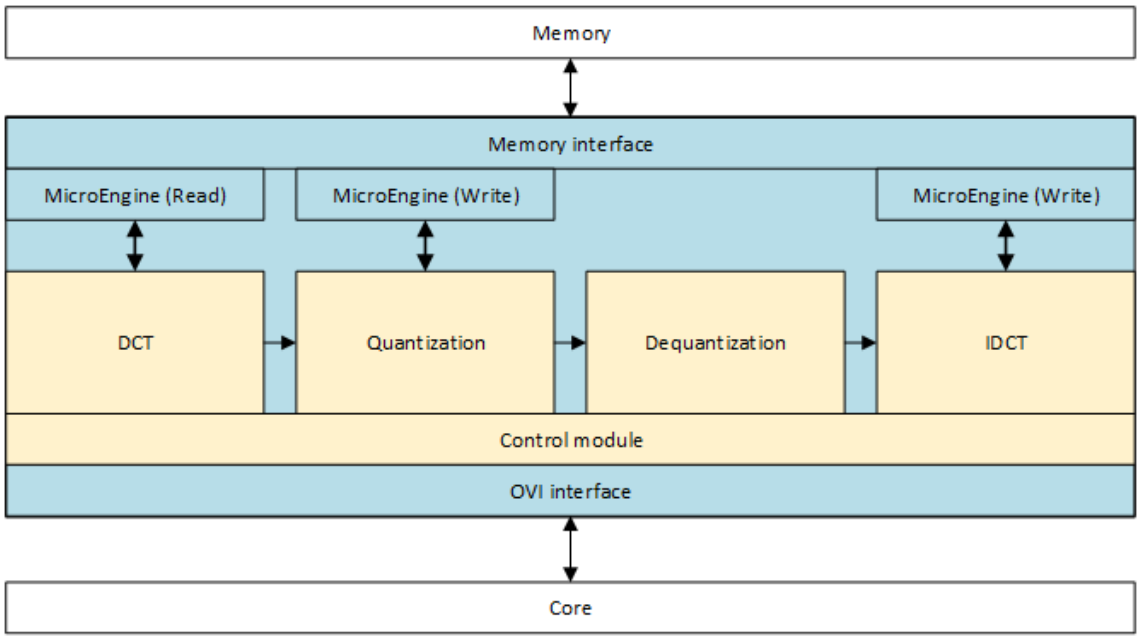



*Figure 8. The main structure of the SA-HEVC*

The accelerator can be parameterized depending on the internal datapath width. The ideal scenario allows a maximum throughput of 512-bit internal datapath width, which is consistent with ACME infrastructure. The accelerator is fully pipelined, which means that it can process 512-bit of input data per cycle.

The SA-HEVC accelerator has one input register and produces two output results. The input data to the accelerator consists of blocks of residual data. Each input sample is 16-bit wide and the total block size can vary depending on the input image size. The first output of the accelerator is a block of 16-bit levels while the second output is a 16-bit inverse-transformed residual.

The accelerator needs to be configured before a specific task is executed. Configuration Register (CSR) is used to store configuration information. The details of the register are shown in Figure 9.

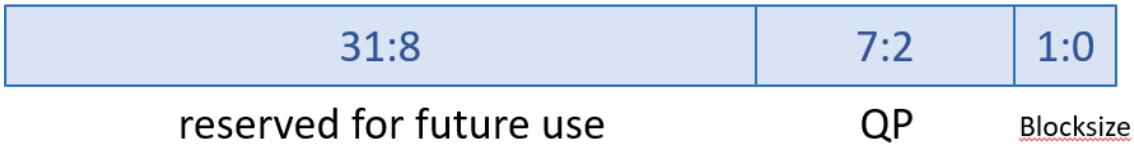



*Figure 9. A CSR configuration*

The operation of the SA-HEVC accelerator is consistent with the operation in the SA-Shell, which is described in the *Periodic Technical Report, M1-M18 Part B*. The details about the internal architecture of the accelerator were described in D4.1.

#### 4.1.2.2. SA accelerator for Neural Network

The SA-NN is one of the systolic array based accelerators in the MEEP project. It is a generalized neural network inference accelerator that can be used to map any pre-trained neural network. SA-NN can be parameterized as follows:

- Floating-point format (default: bfloat16)
- Number of pipelines in Fused Multiply-Accumulate (MAC) Unit
- SA Shell Interface data size (default: 512 bits)
- Number of Primary Elements (default: 256 (16x16))
- Number of SA-NNs (default: 1)

The SA-NN has three types of inputs; input data, bias data and weight data. The accelerator in Figure A.10 which will be used in ACME is the size of 16x16. The systolic array Network block consists of weight registers and MAC units. For a single accelerator we have:

- 16x2=32 bytes input data register,
- 16x2=32 bytes bias data register,
- 16x16x2=512 bytes weight data register,
- 16x2=32 bytes output register

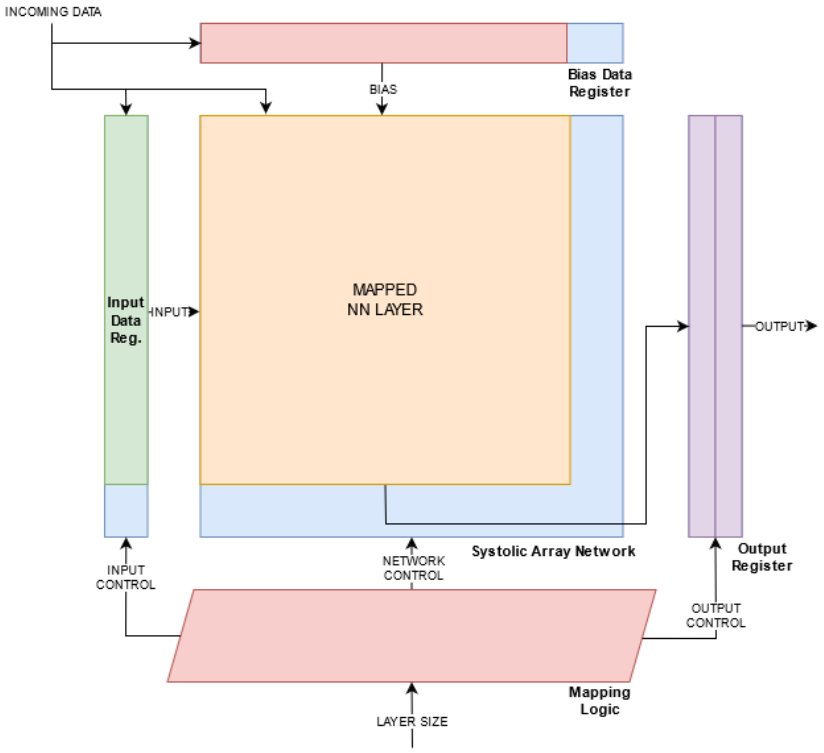



*Figure 10. Block diagram of SA-NN*

The accelerator takes the inputs from the scratchpad memory, performs up to 256 floating-point fused multiply-add operations and generates a result for up to 16 nodes. The general structure of a generalized neural network is shown in Figure 11. The output of each layer is used as the 'input data' of the next layer. So, SA-NN treats every layer as a separate neural network.

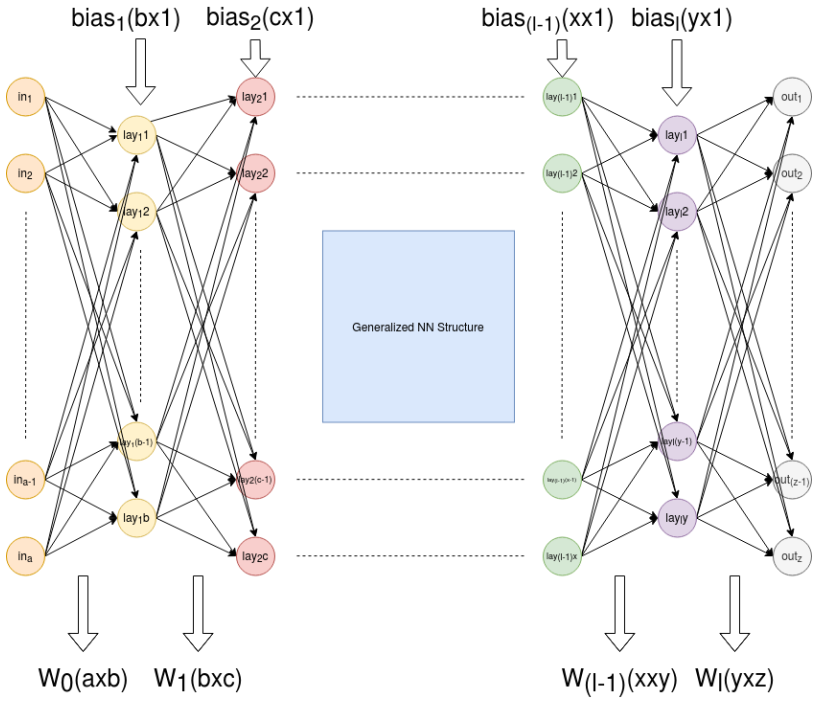



*Figure 11. Generalized Neural Network Structure*

For a particular layer, the node equation is shown in Eq.1 where N, W, I and B stands for node vector, weight matrix, input vector and bias vector respectively. The output vector is calculated as O = f(N) where N is the node vector of the layer and f is the activation function.

$$\begin{bmatrix} N_1 \\ N_2 \\ \vdots \\ N_{(n-1)} \\ N_n \end{bmatrix} = \begin{bmatrix} W_{11} & \cdots & W_{1m} \\ W_{21} & \cdots & W_{2m} \\ \vdots & \ddots & \vdots \\ W_{(n-1)1} & \cdots & W_{(n-1)m} \\ W_{n1} & \cdots & W_{nm} \end{bmatrix} * \begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_{(n-1)} \\ I_n \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_{(n-1)} \\ B_n \end{bmatrix}$$

Eq.1. Generalized single layer of a neural network

We partition a single layer into pieces of 16x16 MAC operations so that there is a maximum of 16 nodes and 16 'input data'. If there are more than 16 'input data', the result of the first iteration will be the bias of the second iteration until the weighted sum of all the input pieces is calculated. Eq.2 and Eq.3 show the first two iterations of the first and second 16 node partitions. If a particular layer has more than 16 nodes, this process is repeated for each partition until the whole layer is calculated.

$$\begin{bmatrix} N'_{1p} \\ N'_{2p} \\ \vdots \\ N'_{15p} \\ N'_{16p} \end{bmatrix} = \begin{bmatrix} W_{11} & \cdots & W_{116} \\ W_{21} & \cdots & W_{216} \\ \vdots & \ddots & \vdots \\ W_{151} & \cdots & W_{1516} \\ W_{161} & \cdots & W_{1616} \end{bmatrix} * \begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_{15} \\ I_{16} \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_{15} \\ B_{16} \end{bmatrix}$$

*Eq.2. Generalized first partition of a single neural network layer*

$$\begin{bmatrix} N''_{1p} \\ N''_{2p} \\ \vdots \\ N''_{15p} \\ N''_{16p} \end{bmatrix} = \begin{bmatrix} W_{117} & \cdots & W_{132} \\ W_{217} & \cdots & W_{232} \\ \vdots & \ddots & \vdots \\ W_{1517} & \cdots & W_{1532} \\ W_{1617} & \cdots & W_{1632} \end{bmatrix} * \begin{bmatrix} I_{17} \\ I_{18} \\ \vdots \\ I_{31} \\ I_{32} \end{bmatrix} + \begin{bmatrix} N'_{1p} \\ N'_{2p} \\ \vdots \\ N'_{15p} \\ N'_{16p} \end{bmatrix}$$

*Eq.3. Generalized second partition of a single neural network layer*

In short, to execute a generalized neural network in the SA-NN accelerator, the neural network must be partitioned as:

- Map each layer separately
- Partition a single layer into pieces of 16x16 MAC operations so that there is a maximum of 16 nodes and 16 input data.
- Load these 16 input data, 16 bias data and 256 weight data into their specific registers of the accelerator
- Accumulate the partial results to calculate the result of the first 16x16 MAC operations.
- Then repeat this process until all the nodes in that particular layer are calculated.

Load operation to input registers and store operation from output registers will be done by the micro-engines inside the SA Shell. SA-NN also has configuration registers which indicate how

much of the 16x16 SA is utilized, and a neural network saturation register that indicates that the floating-point result is saturated and operation should be repeated using normalized inputs.

### 4.1.3. Scalar core

The scalar core plays an important role in the ACME architecture since it is where the disaggregation takes place. Apart from this functional specification, the accelerator also imposes some requirements over the architecture of the cores, such as being a 64 bits architecture, compliant with the RISC-V ISA, having support for vector and atomic instructions (V and A extensions respectively).

Considering the academia and the industry, there exists a considerable number of electable RISC-V IP cores. Many of them have been developed for academia for research and educational purposes, others are proprietary licensed. Many of them are not maintained, there is not a community around them or there is no support at all. Then, the list of potential candidates was reduced to seven different IPs: Avispado, Ariane, BlackParrot, Noel-V, Lagarto, RSD and RVSoC. These have been characterized in Tables 3 and 4.

| IP core | Arch. | ISA extensions | V-inst support | Exec. mode | Linux |
|---|---|---|---|---|---|
| Avispado | 64 bits | I, M, A, C, FD | Yes | In-Order | Yes |
| Ariane | 64 bits | I, M, A, C, (F) | Yes | In-Order | Yes |
| BlackParrot | 32/64 bits | I, M, A, C, FD | No | In-Order | Yes |
| Noel-V | 64 bits | I, M, A | No | In-Order | Yes |
| Lagarto | 64 bits | I, M, A | Yes | In-Order | Yes |
| RSD | 32 bits | I, M | No | Out-of-Order | No |
| RVSoC | 32 bits | I, M, A, C | No | In-Order | Yes |
| **I**: Based Integer ISA<br>**M**: Integer Multiplication and Division<br>**A**: Atomics - LR/SC & fetch-and-op<br>**C**: Compressed instructions (16-bit)<br>**F**: Single-Precision Floating Point (32-bit)<br>**D**: Double-Precision Floating Point (64-bit) | | | | | |

*Table 3. Characterization of several RISC-V cores (I)*

| IP core | Interfaces | FPGA friendly | Freq (MHz) | Language |
|---|---|---|---|---|
| Ariane | AXI | Yes | 100 | SystemVerilog |
| Avispado | OVI, AXI, CHI | Yes | 50 | Encrypted |
| BlackParrot | BedRock NoC, Mem NoC, I/O NoC | Yes | 100 | SystemVerilog |
| Lagarto | OVI, AXI | Yes | 100 | SystemVerilog |
| Noel-V | Amba (AHB) | Yes | 80 | VHDL |
| RSD | AXI or AHB | Yes | 100 | SystemVerilog |
| RVSoC | N/A | Yes | 104 | Verilog |

*Table 4. Characterization of several RISC-V core (II)*

Within this set of processors Avispado, Ariane, BlackParrot and Lagarto were the ones that could fit better in ACME. Apart from the technical reasons that might be extracted from TablesA. 3 and A.4, complementary characteristics like quick and direct support, good documentation, updated releases and clean code sustained the final decision. In the end, Lagarto was selected. This is an RV64IMA 5-stage single-issue in-order processor that supports the integer ISA version 2.1 and privileged ISA version 1.11.

In addition to the previous reasons, Lagarto is an in-house RISC-V processor that can get benefits of improvement/enhancement by the MEEP project. Related to this, it is a well-known processor in BSC and it is possible to get direct support from other teams in charge of its evolution. From then, we can get updates, bug fixes and improvements. Moreover, this processor already integrates the EPI VPU using the OVI interface.

#### 4.1.3.1. Support for multiple coprocessors

To support three different co-processors in a VAS Tile core, the scalar core has to be able to issue instructions to each of them directly. Each of these co-processors will support the same interface protocol, OVI. In addition, each of these co-processors executes different kinds of instructions. As it is depicted in Figure 12, these things have a direct impact on Lagarto's pipeline.

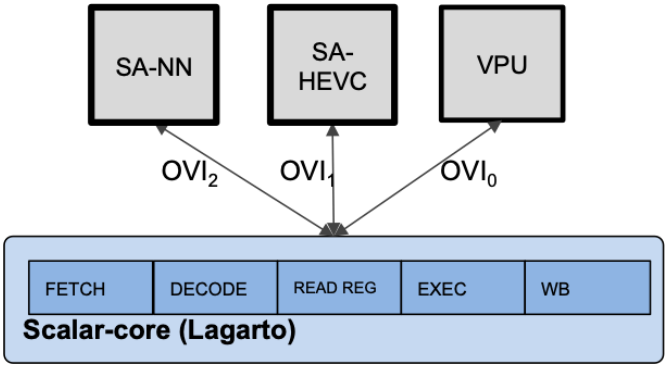



*Figure 12. Lagarto with three loosely coupled co-processors through OVI interface*

The Open Vector Interface (OVI) was defined in the European Processor Initiative (EPI) to connect the scalar core with the VPU. Here, the idea is to do the same for each of the SAs, starting with one of them, and then repeat the process for the next one. That means adding a second OVI ($OVI_1$) [A3] for the SA-HEVC, and a third one ($OVI_2$) for the SA-NN.

The scalar core has to decide which co-processor is in charge of executing each instruction. This decision is not arbitrary, and it relies on the nature of the instruction and its format. In particular, the SAs MEEP project is using a combination of custom and CSR instructions, some set of them specific for the SA-NN, and some for the SA-HEVC. Whereas these custom instructions are formalized, it is possible to make progress on the OVI enumeration by working with the CSR instructions.

As an initial step, we are using CSR instructions to conFigure the Systolic Array. The current implementation of Lagarto core already supports CSR instructions. Within the address space defined for User CSRs, available in the privileged mode of the ISA, we are using the *non-standard* address space, as is highlighted in Table 5. That decision gives us the capability of using two ranges of addresses: (1) from 0x800 to 0x8FF, and (2) from 0x0C00 to 0xCFF.

| CSR Address | | | Hex | Use and Accessibility |
|---|---|---|---|---|
| [11:10] | [9:8] | [7:6] | | |
| User CSRs | | | | |
| 00 | 00 | XX | 0x000-0x0FF | Standard read/write |
| 01 | 00 | XX | 0x400-0x4FF | Standard read/write |
| 10 | 00 | XX | 0x800-0x8FF | Non-standard read/write |
| 11 | 00 | 00-10 | 0xC00-0xCBF | Standard read-only |
| 11 | 00 | 11 | 0xC00-0xCFF | Non-standard read-only |

*Table 5. CSR address space specified in RISC-V privileged architecture*

In particular, the CSR address space from 0x800 to 0x83F has been reserved for SA-HEVC, and from 0x840 to 0x87F for SA-NN, which means a total of 64 reads and write registers for each SA. An example of how these CSRs can be used is explained below.

```
csrrs rd, 0x800, x0 # Read register from SA-HEVC (OVI1)
csrrs x0, 0x800, rs # Write rs into register SA-HEVC (OVI1)
csrrs rd, 0x840, x0 # Read register from SA-NN (OVI2)
csrrs x0, 0x840, rs # Write rs into register SA-NN (OVI2)
```

At the Decode stage, the core recognizes a *SA instruction* by checking the CSR address. The whole instruction is passed through the remaining stages of the pipeline until the Writeback stage, where the instruction is issued in a non-speculative manner through the corresponding OVI interface.

In the case of not having credits available for issuing vector instructions, the pipeline is stalled at the Writeback stage until the SA sets the issue.credit signal, indicating that another SA instruction can be accepted.

When issuing a SA instruction, a new Scoreboard ID is sent along with the instruction to track when it is completed. This creates a mechanism in the core to keep track of all the SA instructions inflight.

The scalar operand required by the instruction is sent through the OVI bus issue.scalar_opnd, containing the value of a scalar register. In the same manner, when completing a SA instruction, a scalar operand may be required to be written, which means that it is possible to have dependencies between scalar and SA instructions.

The signal issue.valid indicates that the contents of the issue group are valid.

Since it is expected that the SA instructions issued to the SA do not generate exceptions, the dispatch group of signals is not necessary for this integration.

The completed group of signals is used by the SA for indicating the completion of a SA instruction. These signals allow the core to commit a SA instruction and write the scalar operand to the scalar register file in case of being required.

## 4.2. System level IPs

ACME architecture might be decomposed into smaller pieces. Its microarchitecture distinguishes between two main ones: (1) the VAS Tile core, and (2) the Memory Tile. As a disaggregated architecture, ACME distributes the execution of the instructions among these two components. All the VAS Tiles execute scalar and arithmetic and logic vector instructions. On the other hand, Memory Tiles are responsible for executing memory and atomic operations. As Figure 13 depicts, each of the cores within a VAS Tile is composed of the combination of all the block-level IPs that have been presented in the previous sections of this document: VPU, SAs and scalar core.

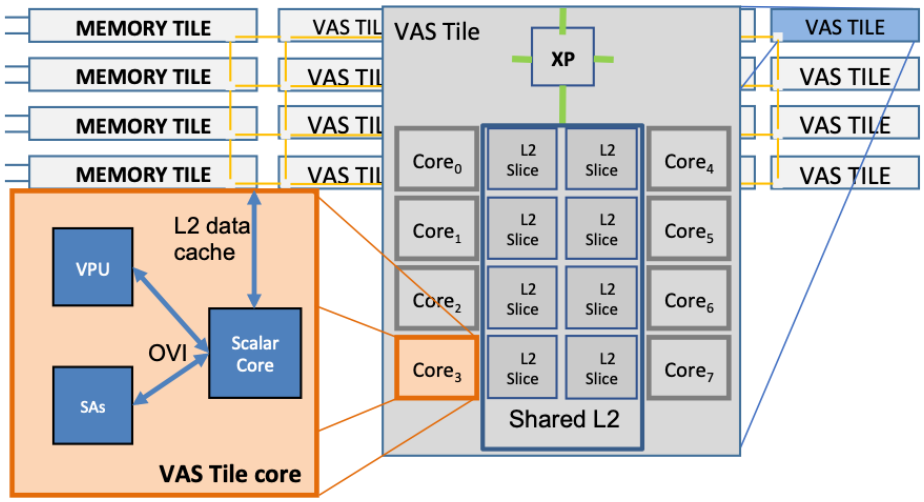



*Figure 13. System-level view of a VAS Tile core in ACME*

### 4.2.1. VAS Tile Core

The design team has been working on the development of a VAS Tile core component, as an integration of the aforementioned block level IPs. This component will evolve, and it will include the improved version of each of its internal components.

At this point, the integration of the SAs is not possible because the SA-Shell design is still under development. More details about it might be found in the *Periodic Technical Report M1-M18, Part B.*

#### 4.2.1.1. VAS Tile core Reference design

BSC has designed the second generation of a RISC-V general processor [4], and the official SoC is named *DVINO*. It is composed of the integration of a Lagarto core with a 2-lanes VPU, a memory hierarchy up to L2 cache and several peripherals, such as a UART and a Performance Monitoring Unit (PMU). The similarities of this SoC with the structure of a VAS Tile core in ACME, make it suitable as a baseline reference design approach. Thus, after modifications in the design to adapt it to the ACME requirements, this design becomes a placeholder for integrating all the block level modules that compose the VAS Tile core.

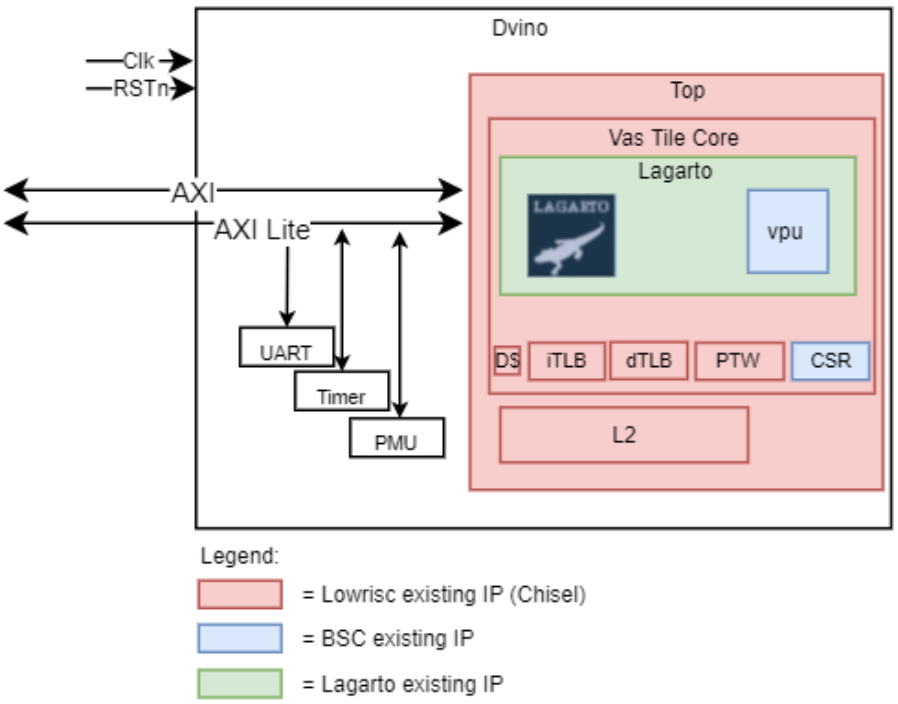



*Figure 14. Diagram of DVINO as an integrated design platform*

DVINO is a System-on-Chip (SoC) with a single core processor (Figure 14) and a 2-lane VPU. It also includes an L1 Instruction Cache developed in the BSC, this is a 4-way 16KB cache with 2-cycle access of latency.

The cache memory hierarchy used is from LowRISC. More in detail, DVINO adapts the cache design developed in the Untethered lowRISC SoC version 0.2 [A1]. The L1 Data Cache is a 4-way 16KB cache with 2-cycle access of latency and the L2 cache is an 8-way 64KB with 3 cycles of access latency.

The colors in Figure A.14 represent the ownership of each of the blocks. According to that color code, the RTL team is focused on the blue (VPU) and the green (Lagarto)parts. In the future, the SA co-processors will be integrated within the green box. The red blocks are borrowed from the open source lowRISC project, and they help to isolate the behavior of the VAS Tile core from the rest of the system, only for testbench purposes. As an example, the L2 red block might be seen as the L2 slice corresponding to one VAS Tile core.

DVINO includes some peripherals such as a UART, a Timer and a Performance Monitor Unit for providing higher visibility to the designer about the behavior of the system.

The peripherals and main memory are connected using the AMBA AXI protocol. AXI4-Lite is used for peripherals and AXI4 for memory interconnection.

#### 4.2.1.1.1. DVINO Software support

DVINO provides a set of scripts for controlling the operating system activities such as generating the bootROM, the bootloader, and THE Linux kernel.

The bootloader used is OpenSBI (RISC-V Open Source Supervisor Binary Interface). OpenSBI is an open-source implementation of the RISC-V Supervisor Binary Interface (SBI) specifications

aimed at providing runtime services in M-mode. It is used as a bootloader, where we concatenated the Linux Kernel in the payload.

The Linux kernel supported is version 5.8.

It is also provided with baremetal unit tests to check the correct functionality of DVINO. These tests can be loaded directly into main memory without the intervention of the bootloader.

#### 4.2.1.2. DVINO as a CI/CD Integration design

Using DVINO as a reference design for integrating block level designs offers a flavor of the VAS Tile core, bringing the possibility of testing and analyzing its features, performance and behavior easily. More specifically, this design might be seen as a hardware project vehicle for creating a continuous integration and continuous design workflow (CI/CD), from which the RTL design creates an iterative and incremental procedure to evolve the VAS Tile Core component. New mechanisms have been enabled to provide higher visibility about the system behavior, which provide useful information to the designers for improving and debugging the integration process, before the design is sent to the verification team.

Right now, DVINO is used for integrating the current versions of its different components, such as the VPU v1.1, and the new version of Lagarto with support for multiple co-processors. Any new version of the block level modules will be integrated into DVINO as soon as it is available (VPU, SA-Shell, SAs and/or Lagarto).

Regarding the CI/CD mechanisms, DVINO provides a controlled simulation environment for exercising the VAS Tile Core using commercial CAD tools[1], the console, or the graphical user interface (GUI). There it is possible to simulate the RTL code when it executes unitary tests that exercise the Scalar Core (Lagarto) and its attached co-processors under different scenarios, helping in the development process of the VAS Tile Core.

Once the VAS Tile core design has been developed and verified, it is implemented onto the FPGA platform. DVINO, as an integration platform, is also used for hardware validation purposes on the Xilinx platform Alveo U280. For this, DVINO includes interfaces to facilitate the connection to the FPGA shell. This allows it to easily synthesize and program the Alveo 280 and run software unit tests to validate the system. More details about this use are provided in the deliverable D6.2 Section 3.1.1.

DVINO v1.0.0 is the release for M18 which contains the MEEP VPU v1.1 described in the previous section 3.1.1 and a minimal set of peripherals as a UART, Timer and a Performance Monitor Unit. The diagram shows the interface, using AMBA AXI4 and AXI4-lite as the main connection with the FPGA Shell. AMBA AXI4, AXI4-Lite and AXI4-Stream have been adopted by Xilinx as main communication buses in their products.

[1]The RTL Team is using the Questa Prime simulator tool from Mentor Graphics (Siemens), acquired through Europractice.

# 5. Stretch goals

Further than the outcomes proposed for the first milestone (MS1), the design team has achieved some stretch goals that are described in this section.

The completed activities have an impact at the VAS Tile core level, since the designs are constrained to the co-processors. On one hand, there is the next version of the VPU (v1.2). Although the design has been completed, it is not delivered yet, since it is still under verification. Another module, closely related to the VPU, is the microengine (ME), which will provide the mechanism for moving data from/to the scratchpad to/from to the vector register file in the VPU.

## 5.1. VPU v1.2

In EPI base design, the reads and writes to each of the 5 banks that compose a VRF slice need to occur at separate cycles due to the Single-Port nature of the SRAM cells used for their implementation. Access to the VRF is controlled by a Finite State Machine (FSM). The FSM sequentially transitions through five different states: FILL_A/FILL_B/FILL_C that fill the source buffers with data to be consumed by the execution unit; WB that allows for data produced by the execution unit or inter-lane ring to be written-back to the VRF; and MEM_BUF where load or store memory operations are allowed. One of the reasons for this was the fact of having a one-port VRF. The evolution of the MEEP VPU from the v1.1 to 1.2 tackles this feature, and has effects on different parts of a vector lane, as depicted in Figure 15. The number of banks at a VRF slice, as well as the memory blocks that compose each of the source, load, store, and write-back buffers are reduced from 5 to 4. Derived from this change, address pointers generation to read and write from the SRAMs is simpler. On top of this, the former requirement of crossbar logic that shuffles written and read VRF data words to cope with the misalignment between physical registers mapping and VRF banks is now eliminated. Performance can also benefit from the new dual-port access to the VRF by modifying the FSM so only FILL_A/FILL_B/FILL_C and MEM_BUF states exist. Now FILL states can provide source buffers with data, while writes into the VRF can happen simultaneously in the shape of loads if a memory instruction were inflight, or write-backs from the execution unit. Likewise, in MEM_BUF state reads from VRF for store instructions can coexist with write-backs.

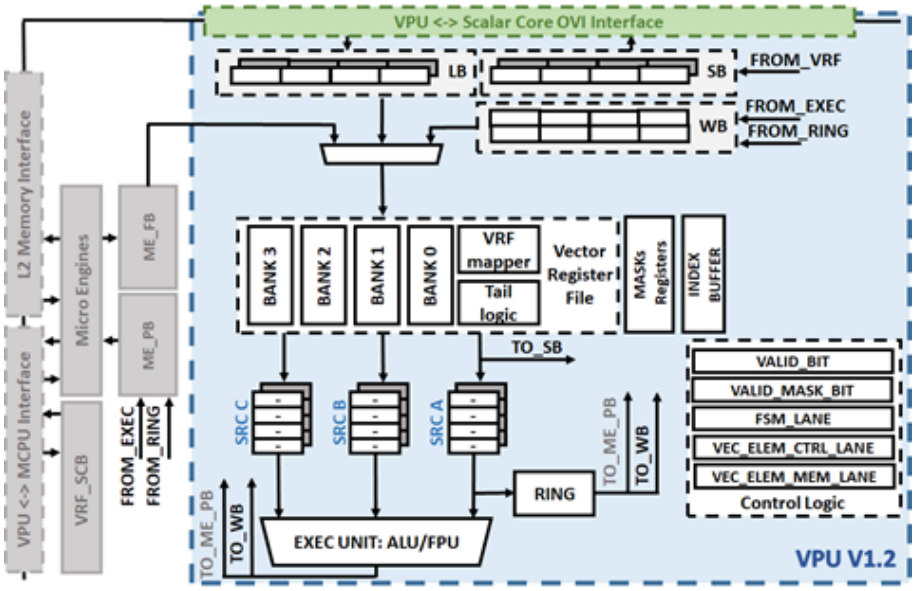



*Figure 15. General block diagram of a Vector Lane in VPU v1.2*

## 5.2. MicroEngines

The ME's role is to retrieve vectors from Scratchpad (SP) and store their data in the VRF's load buffers and to store computation results from VPU's buffer acting as a WB buffer. To perform its role the ME needs three types of interfaces as depicted in Figure 16.

The VPU sets registers through an Advanced Peripheral Bus (APB) interface to indicate the sources' address, the size, the destination address and the destination size. The mechanism to set these values is out of the scope of this deliverable, and it will be described in the future, when the Memory Tile is in place.

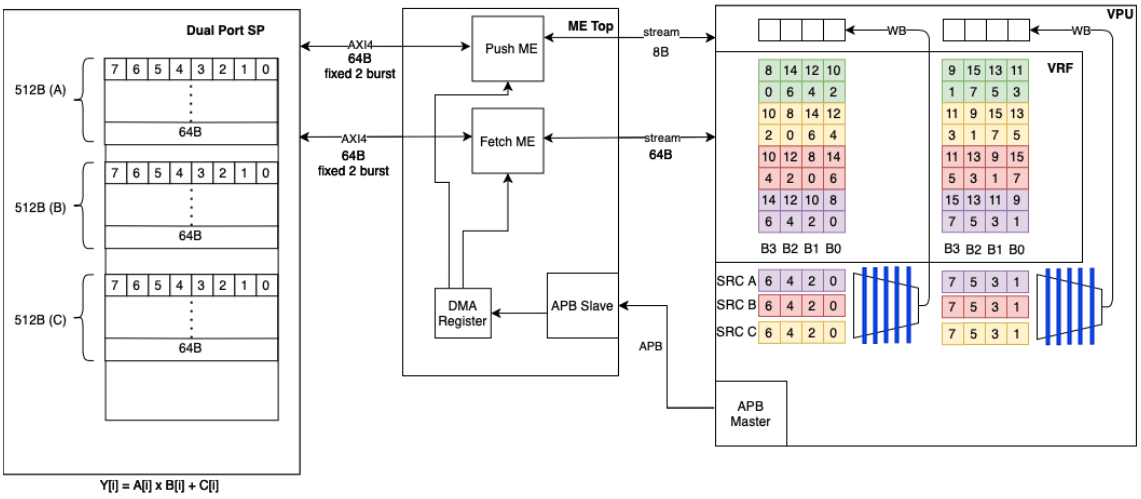



*Figure 16. 1+1 ME Block Diagram*

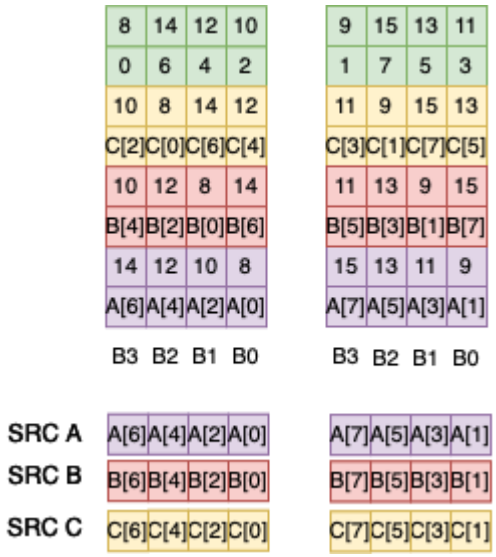



*Figure 17. VRF Vector mapping*

In Figure 16 an explanation of MEs' operation with a VPU coprocessor is presented. This architecture interfaces to a modified version of EPI's VPU. The modifications include fewer bank numbers (5 to 4) and hence 4 source buffers (serving the ALU), the VRF holds 16 elements of a vector per pair of fused lanes instead of 256 elements per 8 lanes. VRF banks are dual ported (i.e., RW). The description is based on a pair of fused lanes. To support multithreading with more lanes, the ME design should be replicated.

The ME contains two separate operation modes. Fetch ME block which retrieves data from Scratchpad to transfer to the VRF and Push ME block which transfers data, when made available by the VPU, from the WB buffers to be stored in the Scratchpad.

The operation of Fetch ME would be as follows. On the VPU interface (i.e., stream: tready, tvalid, tdata) 8 elements per clock cycle (i.e a throughput of 64B per cycle) can be written to the 4 SRAM banks at the same time. The first 8 elements would be mapped in locations with indices 0 and 1 (Figure 17). These elements would represent the first 8 elements of vector A. In the next clock cycle the first 8 elements of vector B are written and subsequently vector C. Now that corresponding elements of the MAC operation are ready in VRF they would fill source A (1 clock cycle), then source B (1 clock cycle) then source C (1 clock cycle). After that, a write back or wait cycle occurs and at that same time the operands start sliding through the ALU one by one. It should be noted that ALU takes 5 clock cycles to multiply and accumulate. Also, at the same time, since source buffers are duplicated, it is expected that source A then B then C are filled with new elements from VRF. In this way the operation is pipelined. The latency would be 4 clock cycles, but the throughput is still 64B per cycle. Since there is another instance of ME for store to SP (i.e., ME Push), as soon as an element is written into WB buffers it will be consumed by ME Push to be stored in the scratchpad once a total of 8 elements or 16 elements (Burst 2) are buffered.

Since the ME would eventually interact with an arbiter from the scratchpad side and vectors are memory mapped, standard AXI4 protocol will be used. AXI data width could be 512 bits (1 cache line), reducing the port pressure and the logic of having to sequence the data (into 64 bit slices) from the scratchpad to the ME.

**Stream Interface (VPU-ME)**: The stream interface is an AXI stream interface. In its initial implementation it would contain tready, tvalid and tdata signals. For a more general

implementation for an ME that works for VPU as well as for SA, tlast could be added depending on its necessity. This interface data width would be 64B for Fetch ME and 8B for Push ME.

**VPU-L2 Interface**: For simulation purposes this interface is an AXI-SRAM bridge. The implementation contains a downsizer that could be necessary to process data needed in a specific bit-width. Eventually, an arbiter would be incorporated to select which coprocessor will be served by the ME for future implementation. In its initial version, the bridge is duplicated to support an independent Fetch and Push MEs to operate in parallel. In subsequent implementations, we expect to have a single dual-ported SRAM attached to an arbiter.

Here, we describe how the ME's configuration for VPU in Figure A.16 could match the bandwidth requirements. The bandwidth requirement is 16 FMAs / cycle / 8 pairs of lanes. This translates to one FMA / cycle / lane.

By exposing the pipeline as shown in Figure 18, we can see that every clock cycle we can obtain a WB or (push to scratchpad), which satisfies the bandwidth requirement. For that, the bandwidth on the stream interface should be 192B (24 elements) / 4 cycles / pair of lanes.

The AXI4 interface should match that bandwidth. By showing an example in Figure 19, after 2 cycles latency, we can see that (from AXI to stream) we are able to fetch 192B every 4 cycles.

This implementation ensures that the VRF access strategy for both L1D$ and L2D$ data path is reused and it reduces port pressure on the scratchpad. However, it limits the flexibility on the choice of AXI data bit width (512-bit only to match the bandwidth) and requires a high number of wires per ME. In our estimation the number of wires would be 5 channels AXI, 32-bit addr x 2 (read and write addr) + 512 x 2 (Read and Write) = 1,088 (ignoring response channel).

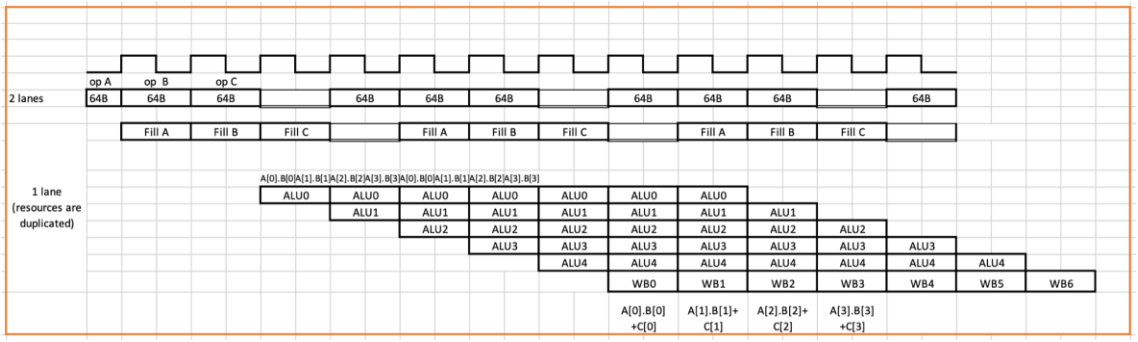



*Figure 18. VPU exposed pipeline with AXI-stream interface for 1+1 MEs*

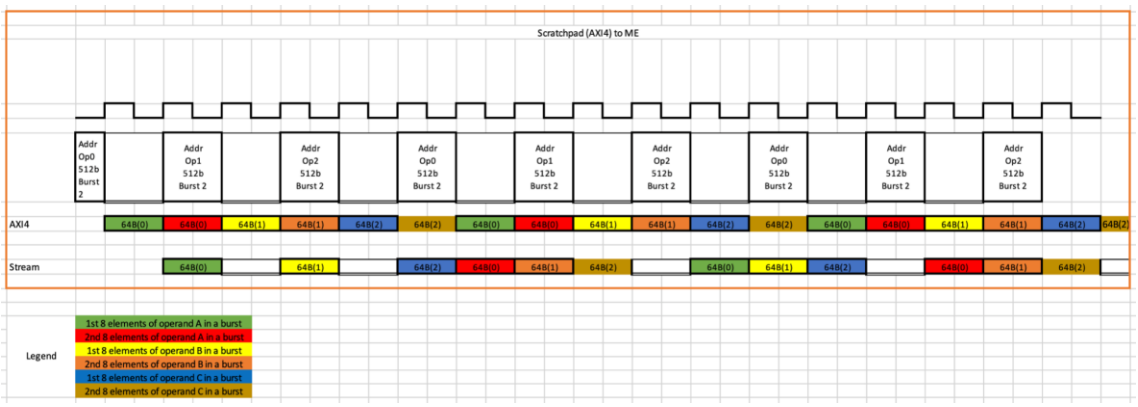



*Figure 19. AXI-Full to AXI-stream data transfer for 1+1 MEs*

# 6. Conclusions and Next steps

To alleviate the complexity of designing the targeted accelerator architecture in MEEP (ACME), WP4 has been structured as a collection of five different main tasks (T4.1-T4.5), each one focused on a different set of functional blocks. This deliverable shows the progression and the achievements obtained at this point of the project by the RTL team.

The RTL team has followed a bottom-up design approach, starting by delivering stable releases of the following computational blocks: the VPU, the SAs and the scalar core. By the integration of these blocks, a first release of the VAS Tile core has been built. In addition, the RTL team has enabled DVINO as an experimental integration project to test the future improvements of the different blocks that compose the VAS Tile core, before the IP is delivered to the DV team for verification or the FPGA team for its implementation in the emulation platform.

Further than the expected milestone at this time for WP4 (MS1), the RTL team has made significant progress in other tasks that will contribute to extend the current functionalities of some of the computational block components, or preparing them for future improvements; and also to build up system level components of the ACME accelerator. As a summary of the achievements presented in this chapter, several components of the ACME accelerator are delivered, starting from those ones with finer granularity (block level modules), such as the scalar core, and the three co-processors (VPU, SA-NN and SA-HEVC). A first release of the VAS Tile core is also delivered, which is built up as an integration of the aforementioned block level modules.

The code for each of these IPs has its own project in the Gitlab *MEEP RTL designs* repository. A summary of all these is collected in Table 6.

| **M18 DELIVERABLES** | |
|---|---|
| **IP** | **URL** |
| VPU | https://gitlab.bsc.es/meep/rtl_designs/meep-vpu/-/tree/v1.1.1 |
| SA-HEVC | https://git.hpc.fer.hr/ldragic/sa-hevc-acc-ext |
| SA-NN | https://gitlab.bsc.es/meep/rtl_designs/meep-TBTK/meep_tbtk_nn |

| | |
|---|---|
| scalar core | https://gitlab.bsc.es/meep/rtl_designs/drac-inorder/-/tree/feature/meep-SAs-ovi |
| VAS Tile core | https://gitlab.bsc.es/meep/rtl_designs/meep_dvino/-/tree/v.1.0.0 |
| **STRETCH GOALS** | |
| VPU | https://gitlab.bsc.es/meep/rtl_designs/meep-vpu/-/tree/v1.2 |
| ME | https://gitlab.bsc.es/meep/rtl_designs/meep-uengines/-/tree/dev |

*Table 6. Gitlab RTL design repository*

## 6.1. Next Steps

All the next activities will be focused on having a minimal viable ACME accelerator in which its main components might be integrated, and most of the architectural features of this accelerator might be stressed and tested. The RTL team will continue with a bottom-up implementation approach, but the prioritization of one component over another will take into account several factors, such as the design complexity, the impact of that component on the accelerator behaviour and/or performance, the need for it in a minimally viable design version (is it required for getting better knowledge and idea about how the ACME behave? or is it a component that would add advanced features?). As part of a co-design project, the RTL team is influenced by other teams' work. Thus, some answers to these questions will be provided by the Performance Modelling team, and others by the software requirements of the overall project (WP5).

However, some activities need to be done, even if their low level details are not fully clear. The following activities will move in the direction of evolving the VPU to include the memory interface with the scratchpad, with all its implications; having the first version of an ACME VAS Tile, and start the development of the Memory Tile from its core module: the MCPU.

# CHAPTER 2: Verification Strategy

## 7. Summary

The scope of this chapter is the D4.2 *Part B Verification Strategy* document. Due to the research nature of the ACME accelerator, the strategy will be refined over time based on performance simulation results, architectural decisions and further design exploration as well as results from the FPGA emulator. The primary purpose of this document is to describe the verification methodology, strategy, technical principles and verification plan/intent to be followed by the design verification team members to scale from a module level verification environment for an ACME component (VPU) up to a system on chip level ACME verification. It also specifies the detailed steps/verification flow taken to functionally verify the design and also outlines the advantages of using verification flow like UVM methodology, assertions, functional coverage, code coverage, and random test cases. The verification flow process, used in the MEEP project, starting from understanding the design specifications, coming up with the Test/Verification plan and construction of the testbench and associated testbench components to meet functional verification goals is discussed in detail in this document. The document outlines the relationship between block/module level verification environment and system on chip ACME verification environment and how they interact with each other.

Since a bottom-up approach is used to verify ACME, the document begins with the submodule level verification, followed by the block-level of the VPU and moves on to describe the VAS Tile Core verification strategy {a scalar core processor with a VPU and AXI interconnects}; next upper level works on the VAS Tile and Memory Tile system verification and eventually describes the strategy used to functionally verify the ACME System Level design. The document includes the development of the verification infrastructure to support verification metrics like functional coverage, code coverage, assertions and the usage of random test cases to exercise corner case scenarios. A roadmap for the system level verification has been provided like introduction of logic in testbench to support multiple cores, techniques to integrate block level environments (VPU, Memory Tile, NoC), the use of riscv-dv tool to generate random test cases for corner cases scenarios (refer to section 1.1 of Technical report).

This chapter is structured into seven sections. Section 2 outlines all the definitions and notations frequently used in the document to provide context to all readers. Section 3 introduces the ACME design and outlines all the key verification activities and framework at a high level to functionally verify the ACME design. Section 4 presents the verification methodology and all the verification techniques including formal verification in the form of assertions, use of verification metrics (code coverage, functional coverage) used in the MEEP project. Section 5 gets into details of UVM methodology, which begins with the description of a UVM Testbench template in the subsection 5.1. The purpose of this template repository is to create a UVM environment pattern/code, independent of the Device Under Test (DUT), creating a common infrastructure to verify IPs/blocks and system level infrastructure. These templates are heavily used in that of the MEEP project. The subsection 5.2 explains the rationale behind the usage of coverage metrics (functional and code coverage) and outlines their utility with regards to functional verification. The type of test cases used in the MEEP project are described in subsection 5.3. Verification plan /test intent and the verification workflow are described in subsections 5.4 and 5.5 respectively.

The workflow is described in detail to illustrate how the verification team interacts with design teams (RTL and FPGA).

Section 6 presents the bottom-up hierarchical testbench approach used in the MEEP project; starting at module-level and going up to the SoC-level testbench. This section presents details related to each of the different developed testbenches (VPU, VAS Tile Core, VAS Tile, ACME System Level test Bench) in order to generate stimulus, functionally verify and guarantee the correctness of the accelerator specifications proposed in MEEP (ACME). This also ensures that all the design components are verified at the block level as well as that of the system level thereby ensuring tape out confidence. The details of the test bench architecture of the ACME System level test bench followed by the logical sequence of steps to integrate block level verification environments into system level infrastructure are provided in this section.

Section 7 presents the infrastructure, in terms of computational resources and tools used to perform all the verification activities. It has pointers to all the source code repositories in gitlab. This section also outlines the details of the CI/CD regressions used by the Verification team once there are major code changes to design and test bench infrastructure and code merges in various gitlab branches. Finally, this chapter (section 8) concludes with a summary of the main outputs of the Verification team and explains the list of next steps to be done.

The Verification Strategy remains the same along the lifecycle of the MEEP project, but low-level details are susceptible to being added and/or modified in line with changes to design and architectural specifications. The steps mentioned in the Integration strategy can be used in other BSC projects. The ACME accelerator is an architecture in research, which obligates the Design Verification (DV) team to adapt, improve and/or refine different pieces of the verification environment.

# 8. Definitions and notations

Table 7 provides a definition for the most frequently-used ACME concepts mentioned in this document.

| Concept | Definition |
|---|---|
| ACME | Accelerated Compute and Memory Engine accelerator, composed of a grid of Tiles. It is a self-hosted and disaggregated architecture. |
| Tile | An element with a specific function inside the ACME architecture. There are two different types of tiles in the targeted accelerator: VAS Tile (highly used for performing compute-bound operations), and Memory Tile (highly used for performing memory-bound operations). |
| VAS Tile | Vector and Systolic Array Tile defined as a multi-core system, with shared L2 data cache memory and NoC interfaces to interconnect with other VAS Tiles and Memory Tiles.<br>Within a VAS Tile there are up to 8 VAS Tile cores. |
| VAS Tile core | It is an IP composed of a RISC-V scalar core with three loosely coupled co-processors: one VPU and two specialized SAs. As part of the ACME disaggregated architecture, the VAS Tile core is where scalar and arithmetic vector instructions are executed. |
| Lagarto | BSC single-issued in-order RISC-V processor. |

| | |
|---|---|
| VPU | Vector Processing Unit suitable for executing RISC-V ISA vector extension instructions. It has two kinds of interfaces: one with the scalar core, and another with the L2 data cache/scratchpad. |
| SA | Systolic Array accelerator. ACME includes two specialized designs; one for processing neural networks (SA-NN), and another for HEVC video processing (SA-HEVC). These accelerators have the same interfaces to other blocks by using a SA-Shell. |
| DVINO | Originally, it is a BSC SoC composed of the integration of a Lagarto core, a 2-lanes VPU, and a L1 and L2 cache memory.<br>The 2-lanes VPU is derived from EPI project; and the memory hierarchy (except the L1 data cache) comes from a lowRISC [1] project. |
| Memory Tile | Cluster of functionality including the Memory Controller CPU (MCPU), Shared L3/Row Buffer, L2 Instruction Cache, TLB, and NoC interfaces for outperform memory instructions. |
| MCPU | Memory Controller CPU. As part of the ACME disaggregated architecture, the MCPU is a processor core responsible for the execution of all the memory and atomic instructions. |
| NoC | Network on Chip. Communication mechanism used for interconnecting different Tiles (VAS Tiles to VAS Tiles, and Memory Tiles to VAS Tiles. |
| ME | A microengine is a DMA controller mechanism to handle the data movements between the scratchpad and the vector register file in the VPU. |
| L2 cache / Scratchpad | L2 data cache that might be fully or partially configurable to be used as a private space of memory. |
| SoC | Acronym for system on chip is an IC which integrates all the components into a single chip. |
| OVI | Open Vector Interface. It is an interface and bus protocol to interconnect a Vector Processor Unit with a CPU. |
| UVM | Universal Verification Methodology is a standardized methodology for verifying integrated circuit designs. The UVM standard improves interoperability and reduces the cost of rewriting IPs for each new project or electronic design automation tool. It also makes it easier to reuse verification components. |
| UVC | Universal Verification Component is a part of the UVM. It is a set of software and related artifacts providing a self-contained, plug&play verification solution for a particular protocol interface or logic function, compatible with the UVM. |
| DUT | It is a developed IP undergoing testing, either at first manufacture or later during its life cycle as part of ongoing functional testing. |

*Table 7. Definition of ACME concepts*

# 9. Introduction

Verification is a complex activity, which involves understanding the design specification and coming up with a verification plan/strategy which comprehensively verifies the functionality of the design. The ramp up process of Verification further involves careful planning of development of new verification components, reusing existing components and ensuring that the verification strategy is scalable and configurable. The goal of this document is to introduce an overall high level and complete strategy for the Design Verification of the targeting the ACME accelerator as it is shown in Figure 20. Basic understanding and previous knowledge of the MEEP objectives and the ACME architecture is assumed for the reader. More information regarding the ACME accelerator can be found in the document *D4.1 System Architecture and emulation platform specification*.

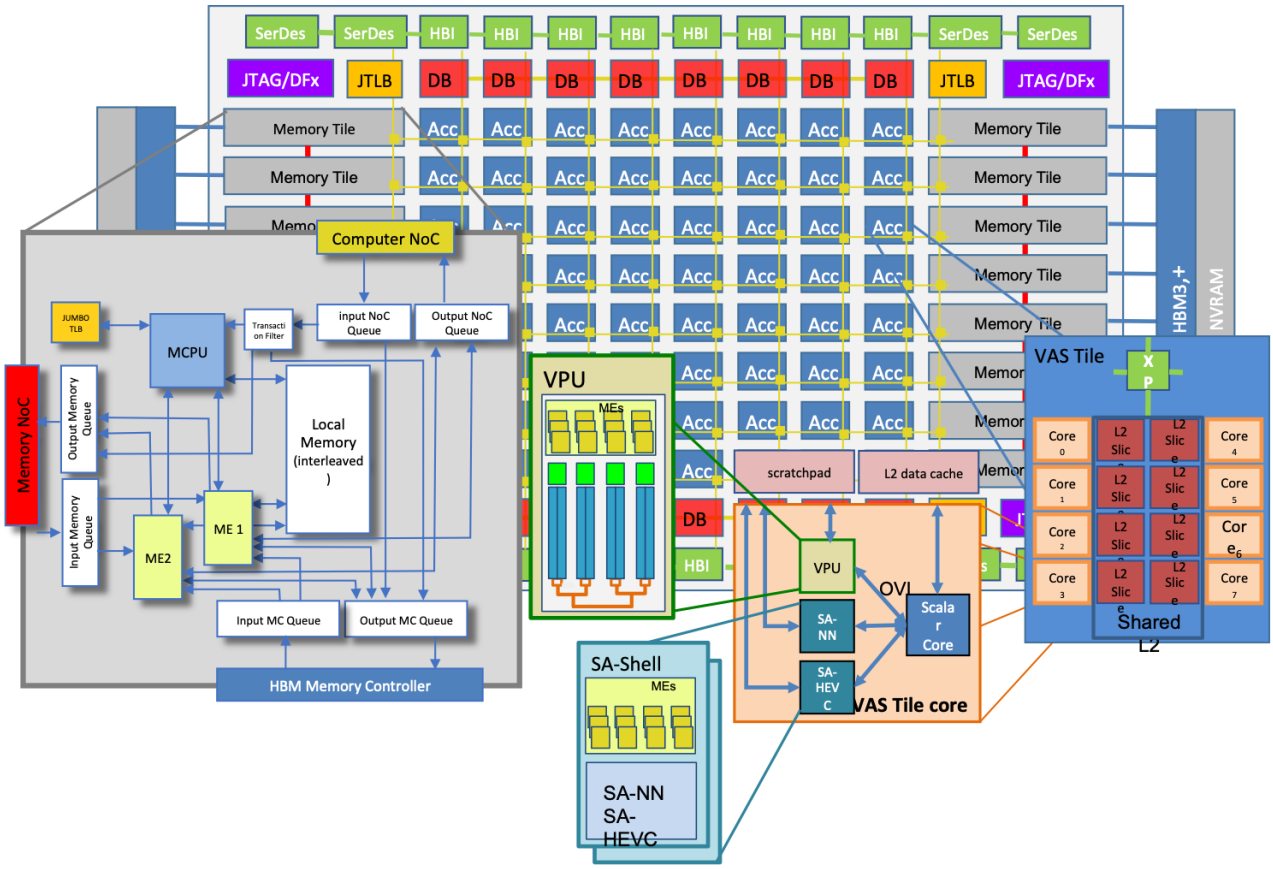



*Figure 20. Hierarchical view of the ACME accelerator*

The DV team, in conjunction with the RTL team, has set several goals for verification with the aim of being compliant with the ACME specifications and MEEP requirements:

- Define the verification framework, which includes:
    - Determining the Verification methodology to work with,
    - The internal workflow, and also the interactions with other hardware teams, such as RTL, and resulting implementation target, which in MEEP is a collection of FPGAs.
- Identify and select a set of tools and the baseline infrastructure required to work on the verification activities.
- Design a bottom-up hierarchical and incremental verification strategy. This point will be covered in detail in Section 6 *testbenches*[2].
    - Test plans at the module or block level verification.
    - Test plans at system-level (System on Chip - SoC) verification.

[2] The DV plan is a combination of workflow and the components in the workflow with the overall goal of producing high quality RTL. All the Test Plans at the IP/Block Level are based on the current understanding of the design and the architecture and are not comprehensive. Only there are a subset of tests. Post M18, more tests will be added as a part of the test plan.

# 10. Verification Methodology

The final system (ACME) is a hierarchical collection of subsystems, and the verification activities must verify all of them. Each such subsystem is referred to as the Design Under Test (DUT) in the verification of that subsystem. In order to maximize the quality and comprehensively verify the functionality of every design, the DV team is implementing the following mechanisms to stimulate, stress and debug the Device Under Test (DUT): UVM model, assertions and coverage. The main objective of using UVM Methodology is to produce reusable, highly configurable, and scalable code which will eventually enable the project to verify multi-core designs. In addition to these formal verification techniques such as assertions are used in testbench to improve the quality of verification and support the design team in their development activities. The main intent behind usage of code and functional coverage is to determine if all the verification objectives are met and to track the status of verification objectives by all key stakeholders in the MEEP project. These techniques are outlined in detail in the section below

- **Universal Verification Methodology (UVM) model**: UVM will be used to develop verification of components, comparing the simulated behavior of the DUT (the corresponding RTL design), against a golden or reference model, by using constrained random, metric driven and self-chicking verification environments. The behavior of the DUT is checked against a golden reference model thereby verifying the functionality of the design. The DV plan is to design a configurable UVM testbench model (environment along with tests, with class-based templates for the different objects and components), and then drive both directed tests and constrained random tests for the specific characteristics of the DUT. With this strategy, the DV team wants to achieve two main goals: first comprehensively verify the design with directed, corner case scenarios and allow reusability and scalability across testbench models through class based verification approach. In order to accomplish the above goals the document also talks about the UVM templates to build up a reusable verification model. The developed templates illustrate examples of interactions between UVM testbench components (driver to sequencer), examples of using uvm_config_db where the DV engineer can pass virtual interfaces through the testbench environment. The templates finally illustrate the concepts of virtual sequences and virtual sequencers which are going to be extensively used in the ACME system level verification.

- **Assertions**: They are SystemVerilog properties that validate the correct behavior and state of the design. Some of the assertions are coded in the RTL by the design team. The DV team will play a complementary role where will add assertions in test bench for checking design properties / functionality checks, which have not been added by the design team. These assertions are enabled by the UVM Testbench. Assertions provide better observability of design prototypes , detect bugs much faster by embedding checks in the form of properties that verify certain design scenarios / events which happen over the entire simulation cycle. All assertions have an associated cover statement to detect that the condition has been triggered. An assertion-based feature is covered when the condition is true, at least once during simulation, and the assertion is not violated.

- **Coverage**: It is a metric used to measure how much the design is exercised, defined as the percentage of verification objectives that have been met written in SystemVerilog Assertions (SVA) language. Here two types of coverage metrics will be used:

- Functional coverage: User-specified measure to tie the verification environment to the design intent or functionality. This metric will be obtained by using a combination of data-oriented coverage and control-oriented coverage. The former will check the occurrence of combinations of data values by writing *covergroups*. In this case, a covergroup is considered covered when all its defined coverpoints are higher than zero. The latter, the control-oriented coverage will check the occurrence or sequences of behaviors. A Functional Coverage plan is provided by the DV team, treating the design as a black box and this is reviewed by both the DV & RTL design team to ensure that all the coverpoints and cross coverages needed to cover the functionality of design are chalked out and subsequently implemented by the DV team. The goal of functional coverage modelling is to achieve a 100% functional coverage with any exclusions/exceptions being specified by the design team (RTL team). This is an iterative process and may involve doing causal analysis as to why the functional coverage goals are not achieved. Subsequent iterations would involve coding up more test cases to meet functional coverage goals. The risc-dv framework from Google will help to automate the generation of directed and random test cases which will help us to meet the functional coverage goals.

- Code coverage: It measures how much of the design and test bench code has been exercised. This would be used to supplement aspects like Verification plan coverage and functional coverage. The verification team is looking to achieve a 100% code coverage. However, trying to reach a high ratio of coverage might turn out to be costly, while not necessarily producing enough benefit. Consequently, the minimal coverage will be 90%, and from there, particular analysis will be done to keep a trade-off between coverage and effort (time and complexity). The code coverage will also include aspects like toggle coverage, branch coverage and coverage of any finite state machines (FSM) specified by the design team in their specifications. This can be facilitated by simulator specific switches.

In addition to the above mentioned techniques, several complementary pre-silicon verification techniques are combined together to deal with the complexity of the targeted accelerator and maximize the effectiveness and efficiency of the verification and validation activities. In addition to this pure Verilog test benches will be used to run sanity tests/ sample simulations on VAS Tile multi core structures using Open Piton framework . Verification activities for MEEP project will include the following:

- Formal verification at subsystems/blocks and system level.
- Connectivity checking and interconnection of design blocks with each other
- Reuse Verification IP's such as AXI, APB to functionally to verify interactions between various design blocks
- Random, constrained-random, and torture testing.
- Design simulation and analyze the results of FPGA emulation.

# 11. Verification Strategy

This section provides a plan of action/strategy used from a Verification perspective to ensure that all key design components, used in the MEEP project, are functionally verified thoroughly. This also includes verification at block level (VPU, Microengines, MCPU, VAS Tile Core) and verification at system level (ACME). This section also talks briefly about reference models and their role in verification along with the use of directed and random test cases. It also outlines the use of Verification plan which is a key activity which needs to be formulated for verification at block and system level.

## 11.1. UVM Testbench template

With the objective of creating scalable and reusable testbench code in MEEP, the decision was made to use System Verilog with UVM Methodology embedded in testbench code. UVM templates provide code which can be used in the MEEP project as well as other BSC projects for verification purposes. Some of the concepts and the verification flow described in individual testbenches {VPU, VAS Tile Core, ACME System Level} heavily use the concepts outlined in these UVM Templates. The Templates illustrate the mechanism of communication between various testbench components and also how the uvm sequence operates on transaction items. It also illustrates advanced testbench concepts like virtual sequences and virtual sequencers which are used in ACME System level testbenches. Also this framework enables new verification team members to quickly adapt to the UVM framework and apply the same towards the verification goals outlined for the MEEP project.

In this template repository (MEEP Design Verification Gitlab repository; project *Uvm Template Repo {dev branch}:*

https://gitlab.bsc.es/meep/meep-design-verification/uvm_template_repo/-/tree/dev)

There are four different repositories as shown in Figure 21. As a practical use case, this repository uses as an example IP a RAM_16x8 design as a DUT/design. The main use case of these repositories is to do the following:

- Illustrate the communication mechanisms between verification components (driver & sequencer).
- Setup a testbench with use of Virtual sequencers (V_SEQs).
- Show how a UVM config_DB can be used to pass virtual interfaces to various verification components.
- Capture code coverage in a UVM Testbench.

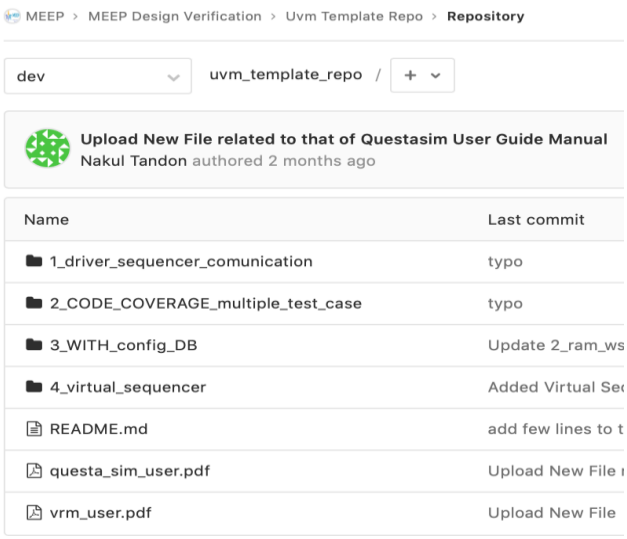



*Figure 21. Main folder structure of the UVM Template repository*

The intent of this repository is: (1) providing reusable code for each of the UVM components and objects required for building any UVM testbench; and (2) offering a usable example to work on to get familiar with the UVM methodology. Table 8 shows all the UVM related terminology used in the UVM Template repository along with their short descriptions.

| Serial number | UVM terminology | Description |
|---|---|---|
| 1 | Agent | An Agent is a combination of a UVM Driver, a UVM Sequencer and a UVM Monitor. It can be of two types: Active or Passive. An Active agent has all the driver, monitor and sequencer present; whereas a Passive agent only has the monitor present in it.<br>A UVM driver's port is connected to a Sequencer export in the connect phase. |
| 2 | Driver | It is a UVM component which drives transactions on the pins of the DUT, responsible for pin-len wiggling. |
| 3 | Sequencer | It is a UVM component which talks to the driver on one end and is associated with the sequence on the other end. A single Sequencer can be associated with multiple sequences. |
| 4 | Monitor | It is a UVM component which monitors design signals (input as well as output ports) and transforms this design activity to transaction items. |
| 5 | Scoreboard | It is a UVM component which is connected to that of the monitor in the connect phase and it keeps track whether the design's actual transactions/outputs match with that of the expected output (which could be a reference model). |
| 6 | Interface | It is a placeholder for that of design signals (input and output). This can also contain properti statements for assertions. |
| 7 | Virtual Interface | It is a placeholder which points to that of the actual interface and is used because in the UVM methodology the design which is in the form of module talks to that of the class based verification components and objects. |
| 8 | Virtual sequencer | It is a handle to that of the actual sequencer/sequencers. This is used in complex verification environments where there are multiple sequencers, transaction items and they are associated with a virtual sequence |
| 9 | Virtual sequence | It is a handle to one to many UVM sequences, Virtual sequencers are kicked off on a particular Virtual sequencer and is used in complex verification environments. |

*Table 8. Terminology used in UVM Template and UVM Test Bench*

### 11.1.1. UVM Templates for a Driver-Sequencer Interaction.

Since this repository (*1_driver_sequencer_communication*) is dedicated for the Driver_sequencer_Communication in the UVM environment (ENV), the following classes are included in the following path *uvm_template_repo/1_driver_sequencer_comunication/RAM_Verification_In_UVM/2_ENV*. These templates have detailed code outlined for the following UVM components namely driver, sequencer, monitor, agent which is encapsulated in a UVM environment. Tests in this template call sequences (W/RSR) and the driver port (W/RD) is connected to a sequencer export. A monitor (W/RM) is connected to a scoreboard which is shown in Figure 22.

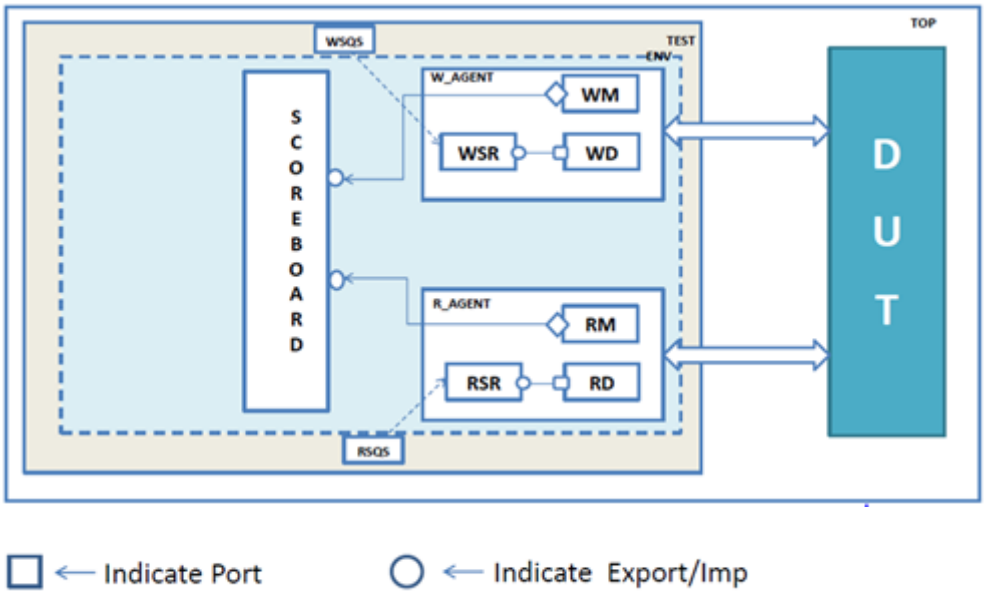



*Figure 22. UVM Testbench Architecture diagram of Driver-Sequencer Communication*

### 11.1.2. UVM Templates for the Code Coverage

This template repository (*2_CODE_COVERAGE_multiple_test_case*) has sample code to achieve 100% code coverage using UVM testbench code. This repository also has code where inline constraints are written in transaction items in that of UVM sequences. These UVM sequences are derived from that of base sequences. Figure 23 illustrates a code template where code coverage is maximized.

```
task body();
  repeat(50) begin //generate 50 times data of read trans
    rtrans_h = ram_rtrans::type_id::create("rtrans_h");
    start_item(rtrans_h);
    assert(rtrans_h.randomize() with {rd_addr<50;});
    finish_item(rtrans_h);
  end
endtask
```

*Figure 23. Code illustrating inline constraints used in Transaction items to maximize code coverage*

### 11.1.3. UVM Templates for UVM Configure DB

This repository (*3_WITH_Config_DB*) is dedicated to use of a UVM configuration database which can be used to pass transaction objects and virtual interfaces to the lower level UVM components. This repository illustrates code examples of configurable UVM environments, interfaces as shown in Figure 24, 25 and 26 respectively.

```
class ram_config extends uvm_object;
   `uvm_object_utils(ram_config)

   function new(string name = "ram_config");
     super.new(name);
   endfunction

   uvm_active_passive_enum p1 = UVM_ACTIVE;
   //Agents are by default active (monitor,driver,sequencer
   // are created only if active..) (if passive then only monitor
   // should be created..)
   bit has_sb = 1;
   // Scoreboard created only if has_sb=1 otherwise not..
endclass
```

*Figure 24. Code illustrating configurable UVM environment*

```
initial begin
  uvm_config_db #(virtual ram_if)::set(null,"*","vif",inf);
  //interface set here * for all components below top module
  //here 1st argument is null bcs this is module not a class
  //if class then null replace by this keyword..
  //though this feature no need to connect interface between
  //top & agent through test file..directly this will take care of that..
  run_test();
end
```

*Figure 25. UVM Template Code above showing usage of UVM conFigureDB to store a Virtual Interface*

```
initial begin
  uvm_config_db #(virtual ram_if)::set(null,"*","vif",inf);
  //interface set here * for all components below top module
  //here 1st argument is null bcs this is module not a class
  //if class then null replace by this keyword..
  //though this feature no need to connect interface between
  //top & agent through test file..directly this will take care of that..
  run_test();
end
```

*Figure 26. UVM Template Code above showing method of referencing Virtual Interface in Lower Level UVM components*

### 11.1.4. UVM Templates for Virtual Sequencer

UVM code templates have been provided for virtual sequencer and virtual sequences, which are used in complex testbenches consisting of multiple transaction items, and interfaces. This is used to verify complex designs consisting of multiple design protocols, such as AXI, APB, PCIe, ethernet, and UART among others. The verification flow for a UVM testbench with virtual sequencers is shown in Figure B.8. There are two main components used here namely the virtual sequencers (V_SEQR) with

virtual sequences (V_SEQS). The virtual sequencer is encapsulated in the environment and has the pointer to one or more actual sequencers. The connection of virtual sequencer to the actual sequencer is done in the connect phase of the UVM framework.

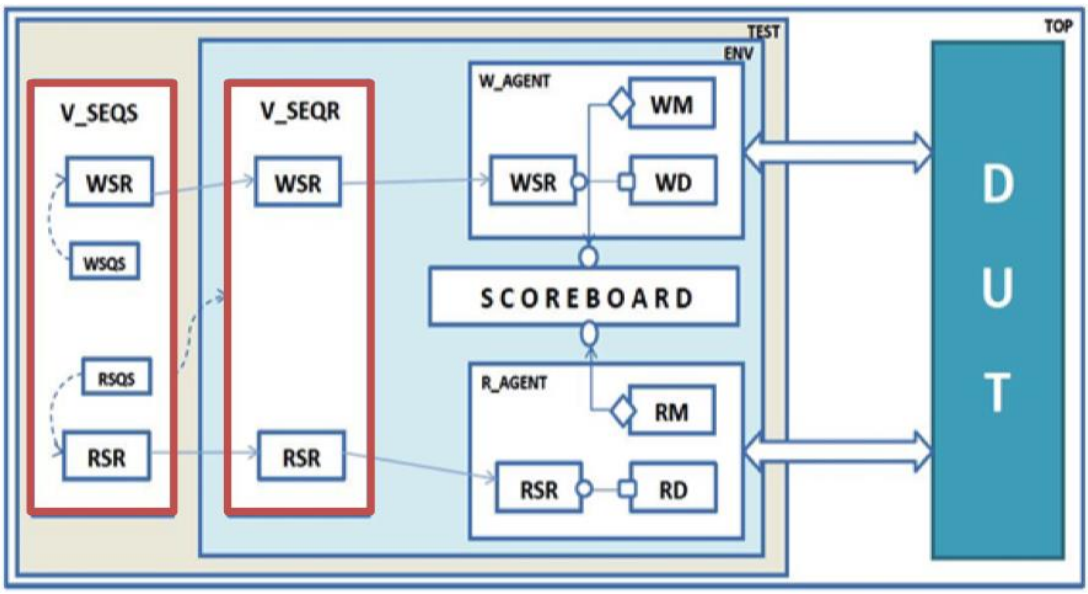



*Figure 27. UVM TestBench Architecture for Virtual sequencer and sequences Code Template*

### 11.1.5. Reference Models Used for Processor Verification

For the purposes of checking correctness of executed instructions, ***Spike***, the open source RISC-V ISA Instruction Set Simulator (ISS), is used as a reference model. Spike has been modified to be aligned with VPU (vector specification v0.7.1 compliant) and to retrieve vector state after executing an instruction. It is pre-compiled to a shared object (spike.so) which is used in the UVM Testbench to execute instructions and provide reference results to be compared in co-simulation with the VPU.

Spike implements a functional model for a RISC-V hart. For the MEEP project, which includes the VAS Tile core and the VPU standalone, we have used EPI's version as a baseline. In it, the original repository was forked and a Spike wrapper was created: spike-dpi.cc. This wrapper implements several SystemVerilog Direct Programming Interface functions (DPI-C) which represent interfaces for the UVM communication with this C reference model. Spike is useful because writing a reference model from scratch for a complex multi-processor design like ACME is an activity which can take years to develop. The document also talks about the steps used to integrate Spike in the Verification environment. To guarantee the highest quality RTL, we are also evaluating industry tools like MDEV flow from Imperas to compare the design outputs against golden reference models. This has been kept as a future task.

For MEEP, the Spike wrapper has been extended to get more information from the Spike simulator when an instruction is executed. This information is then used in the scoreboard for comparison against the DUT status result.

The DPI-C functions available are:

- setup: loads testprogram binary into spike.
- stop/start_execution

- step: executes next instruction in spike.
- get_spike_commit_info: It returns spike state after an instruction is committed.
- run_until_vector_ins: the function executes scalar instructions that are of interest only in the scalar core. Therefore, these are executed only by Spike and results are not used for further comparison. Vector instructions are also executed by Spike and results stored. The same instruction is then pushed by the corresponding Agent and executed by the DUT.
- feed_reduction_result
- get_memory_data

The plan is to extend these functions to be able to interact with Spike using Interrupts.

The Scoreboard plays a key role in working with Spike. It will read instruction results from the destination register and compare them against Spike results. This co-simulation is done in a *step & compare* fashion following an instruction-by-instruction principle. Therefore, simulation can be stopped as soon as the first mismatch has been observed.Some debug test programs are running to check the correct functionality of the testbench. There are three possible outputs from a simulation execution: *testprogram* returns an error code, missmatching between the DUT and the reference model, or successful execution.

In this case the test program was returning an error code, this one is reported by the testbench as follows:

```
# UVM_INFO @ 736596: uvm_test_top.host_env.host.m_monitor [TOHOST] tohost call
# UVM_ERROR @ 736596: uvm_test_top.host_env.scoreboard [TOHOST] Fail finish call with code 0000000000000001
```

In case Spike execution does not match with RTL results, the issue is reported by the testbench as follows:

```
# UVM_WARNING @ 736581: uvm_test_top.host_env.scoreboard [SIGNATURE] RTL PC (0000000080000060) and Spike PC(000000008000004c) do not match
# UVM_WARNING @ 736581: uvm_test_top.host_env.scoreboard [SIGNATURE] RTL (zzzzzzzz00001f17) and Spike (0000a183) instruction do not match
# UVM_WARNING @ 736581: uvm_test_top.host_env.scoreboard [SIGNATURE] RTL (zzzzzzzzzzzzzz2e) and Spike (0000000000000003) register destination do not match
# UVM_WARNING @ 736581: uvm_test_top.host_env.scoreboard [SIGNATURE] RTL (0000000080001060) and Spike (000000000000e060) data do not match
# UVM_ERROR /  @ 736581: uvm_test_top.host_env.scoreboard [SIGNATURE] RTL and Spike signature do not match, see previous warnings. RTL PC=0000000080000060 Spike PC=000000008000004c
```

The total number of errors is reported at the end of the testbench:

```
# ** Report counts by severity
# UVM_INFO :   68
# UVM_WARNING :   21
# UVM_ERROR :    8
# UVM_FATAL :    0
```

Both the signatures from the scalar core and the VPU are dumped into a file so that it can be checked after the simulation finishes.

Scalar core signature snapshot:

```
3 0x0000000080000414 (0x04000c13) x24 0x0000000000000040
3 0x0000000080000418 (0x00cc7fd7) x31 0x0000000000000040
3 0x000000008000041c (0x00000013)
3 0x0000000080000420 (0x00100093) x 1 0x0000000000000001
3 0x0000000080000424 (0x00200113) x 2 0x0000000000000002
3 0x0000000080000428 (0x00300193) x 3 0x0000000000000003
3 0x000000008000042c (0x00400213) x 4 0x0000000000000004
3 0x0000000080000430 (0x00500293) x 5 0x0000000000000005
3 0x0000000080000434 (0x5e004057) (VPU)
```

CSR prv lvl  Instruction  Register data(if applicable)

Program counter  Destination register(if applicable)

### 11.1.6. Coverage (Code Coverage & Functional Coverage)

Coverage is defined as the percentage of verification objectives that have been met. It is used as a metric for evaluating the progress of a verification project. The coverage percentage allows the user to tell how well a design has been tested and if all the scenarios of interest have been exercised.

There are two types of coverage metrics: code and functional coverage.

- **Code Coverage:** it measures how much of the code has been executed which includes design and test bench code. Code, path, expression, FSM coverage are types of code coverage. In order to accomplish 100% code coverage, each and every possible type of data has to be tested on the design code. However, a 100% code coverage doesn't ensure the design is fully verified, just that the written code is fully exercised.
    - As an example, for the bit [1:0] Example signal, each of the four possible values {00, 01, 10, 11} need to be driven to achieve a 100% Code Coverage.
    - **Functional Coverage:** it is a user-specified measurement to tie the verification environment to the design intent or functionality. SystemVerilog provides functional coverage constructs. In functional coverage the design is considered as a black box. Please refer to details in section 4 on this.

### 11.1.7. Test generation

These scenarios are applied to any UVM testbench, System Verilog test benches and pure Verilog testbenches written to do verification at both block level and system level. These tests include assembly test cases, C based test cases and the following tests:

- RISC-V ISA Tests (https://github.com/riscv/riscv-tests).
- Sanity tests: Simplest job running only one test in order to confirm that testbench and RTL Design code are running as intended from a functional perspective.
- Regression tests using CI/CD.
- Smoke tests.
- Random RISCV Tests using riscv-dv Utility from Google.
- Vector tests, Virtual Memory tests, Compression tests.
- Tests exercising Interrupts.

The following test suites have been declared for sign off criteria:

**ISA_TESTS:** 79 tests to check that the implemented instructions are compliant with the RISC-V "V" Vector Extension specification Version 0.7.1

**C_APPS Test Suite:** 5 tests known as *5J's Benchmark* (matmul, fftw, axpy, somier, spmv) written in C and testing vector instructions used by most of these kernels.

## 11.2. FPGA emulation

The last activity of the verification team is to validate the design, which means to guarantee the design is functionally compliant with the specifications when it is tested on a device. Here the verification team will analyze the results obtained by that of the FPGA team and check if those simulation scenarios can be used at their end and if their results are consistent with the FPGA team. If there are any failures reported on device, the verification team will try to replicate those scenarios at their end and do a causal analysis of the failures.

## 11.3. Verification plan

The verification test plan is a document created by the Verification team together and is a combination of directed and constrained random test scenarios to comprehensively verify the design. This document once prepared is reviewed together with the Design and the Architecture teams to ensure that all possible functional scenarios to verify the design are included.

It contains a design description (main functionalities and features being verified), objectives of the plan, testing strategies, including the list of tests that should be run and defines a completion (sign off) criteria, the scope of the plan, testing resources, a schedule and the deliverables.

A test plan is a dynamic resource, which is subject to change when there are design and architecture changes proposed which are different from original design specifications. Figure 28 shows a list of ISA Tests in that of the Verification plan which can be executed at block level {VPU, VAS Tile core, VAS Tile multi-core and that of ACME System level}, and includes atomic instructions, integer multiplication, division, user-level and integer tests.

| | | | |
|---|---|---|---|
| rv64mi | | ma_addr.S | ASM program to test misaligned ld/st trap |
| rv64mi | | ma_fetch.S | commented out |
| rv64mi | | mcsr.S | ASM program to test various M-mode CSRs |
| rv64mi | | sbreak.S | commented out |
| rv64mi | | scall.S | commented out |
| rv64si | | csr.S | ASM program to test CSRRx and CSRRcI instructions |
| rv64si | | dirty.S | ASM program to test VM referenced and dirty bits |
| rv64si | | icache-alias.S | ASM program to test that instruction memory appears to be physically addressed, i.e., that disagreements in the low-order VPN and PPN bits don't cause the wrong instruction to be fetched. It also tests that changing a page mapping takes effect without executing FENCE.I. |
| rv64si | | ma_fetch.S | ASM program to test misaligned fetch trap |
| rv64si | | sbreak.S | ASM program to test syscall trap |
| rv64si | | scall.S | ASM program to test syscall trap |
| rv64si | | wfi.S | ASM program to test wait-for-interrupt instruction |
| rv64ua | | amoadd_d.S | ASM program to test amoadd.d instruction |
| rv64ua | | amoadd_w.S | ASM program to test amoadd.w instruction |
| rv64ua | | amoand_d.S | ASM program to test amoand.d instruction |
| rv64ua | | amoand_w.S | ASM program to test amoand.w instruction |

*Figure 28. ACME Verification Plan view [ISA Tests].*

Some of the proposed Verification test items shown below are included in our test plan to verify the ACME Design as well as subsets of ACME/MEEP at VPU level, VAS Tile core, VAS Tile levels are shown below:

- Try all the Privileged Instruction set [Version 1.11 of RISC-V]
- Illegal Instruction sets and traps to be executed [Example: Unaligned memory addresses]
- 5J's kernels: Series of tests (matmul, fftw, axpy, somier, spmv).
- SMD-TESTS: These come from SemiDynamic. Some of these tests have critical instructions' combinations.
- DIRECTED-TESTS: These are used to test specific features (for example sequences of vext when vl changes every X instructions,etc.). These are only used from time to time.
- Try execution of Indexed Load and Store Operations in Assembly.
- Randomly Generated ASM Tests from that of the riscv-dv Utility.
- Execute Assembly in which Interrupts and exceptions are realized.
- Execute assembly tests which exercise LMUL = 1 and LMUL > 1.
- Execute assembly tests which exercise Vector Instruction formats like Vector masking.
- Execute assembly tests which involve configuration setting Instructions [vsetvli and vsetvl].
- Execute assembly test cases with Vector strided instructions.
- Execute Vector Load/Store segment instructions [zvlsseq].
- Execute Vector Floating Point Instructions [Check with the Design & Architecture team when the design would support this].

- Execute assembly instructions with Vector widening and narrowing instructions.
- Assembly tests which have a number of branch instructions.
- Try different programs for loading RAM [bitwise shift test, gather_scatter].
- Assertions to be enabled on OVI Interface [between VPU & Scalar core] + assertions to be enabled on the other OVI Interfaces to Systolic Array [Awaiting release of the design specifications].
- Run AXI VIP Slave read tests with random delays and randomized slave response signals.
- Run tests which verify the Microengine and the VRF in VPU. This will also include the interlane communication in VPUI based on the bitonic sort. [These include load and store tests]. {Finer details to be dug in once MCPU specs are clear].
- In the future, try a subset of instructions in RISC-V Vector extension set v0.10. This will include segmented sub vector operations like sum, max-min.

In addition, we define the strategies used to integrate a block level testbench environment into a system level testbench. Verification plans have been outlined for the Design to be verified. The Table 9 shown below illustrates some of the Verification methodologies, verification metrics, nature of tests used in the different verification test benches to verify the MEEP design.

| | UVM | Coverage | | Assertions | Random tests | FPGA emulation | Verilog/SV testbench (No UVM) |
|---|---|---|---|---|---|---|---|
| | | Funct | Code | | | | |
| **Block or sub-module Level** | | | | | | | |
| **Microengine** | YES | YES | YES | NO | YES | NO | NO |
| **inter-lane communication** | YES | YES | YES | NO | YES | NO | NO |
| **Module Level** | | | | | | | |
| **Scalar-core** | YES | YES | YES | YES | YES | YES | NO |
| **VPU** | YES | YES | YES | YES | YES | YES | NO |
| **SA** | YES | YES | YES | YES | YES | YES | NO |
| **SA-Shell** | YES | YES | YES | NO | YES | YES | NO |
| **L2/scratchpad** | YES | YES | YES | NO | YES | NO | NO |
| **MCPU** | YES | YES | YES | YES | YES | YES | NO |
| **NoC** | YES | YES | YES | YES | YES | YES | NO |
| **System-level** | | | | | | | |
| **VAS Tile core** | YES | YES | YES | YES | YES | YES | NO |
| **VAS Tile** | NO | NO | YES | NO | YES | YES | YES |
| **Memory Tile** | YES | YES | YES | YES | YES | YES | NO |
| **ACME** | YES | YES | YES | YES | YES | YES | NO |

*Table 9. Verification Methodologies used in Testbenches*

MEEP servers are used to run simulations and execute tests. Most of the verification activities rely on the simulations of different tests on the devices under test (DUT), debugging and analysis of results. The DV plan is a layered approach that targets verification of a DUT at a variety of levels from the module, to the VAS Tile (multi-core tile) and up to the ACME accelerator SoC. The overall DV strategy is organized as a set of nested frameworks. For the core and the VPU, we use the DV scaffolding shown in Figure 29.

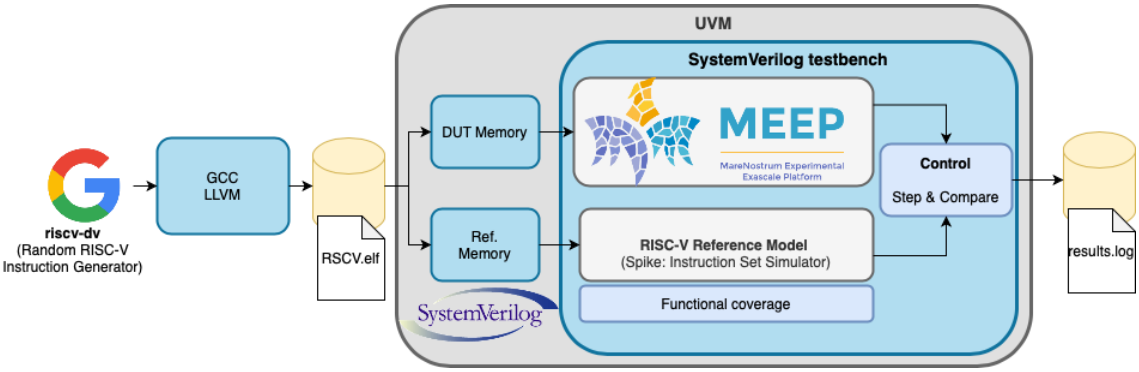



*Figure 29. Reference Verification Flow*

The MEEP DV team exploits the previous work done by other BSC projects, more specifically European Processor Initiative (EPI) [5] for one of the sub-blocks namely the VPU. The former was focused on the Vector Processor Unit (VPU) verification. Because MEEP starts with this IP, it makes sense to reuse the same infrastructure as a starting point. The aim is to adapt and enrich those according to the ACME accelerator needs: e.g., Multi-core support, NoC support, advanced memory hierarchies, HBM support, Memory Controller CPU support and OS CPU support. Overall, MEEP's goal is to build reusable and flexible infrastructure that can easily be adapted for future projects by promoting an open, flexible, modular, and reusable design and set of policies across all of the submodules, modules and up to the full SoC.

## 11.4. Workflow

Verification is an intermediate stage in a digital design lifecycle. It starts as soon as the first draft of architecture specifications are released and gains momentum once the RTL design team delivers an IP. This means the design has passed some sanity/directed tests to ensure the expected functionalities under well-known test scenarios. Then, the DV team will exercise the IP to verify that the design is functionally compliant with its specifications by testing the IP under different simulation scenarios. If the IP satisfies all the tests, it will be worth dropping it into the FPGA then optimized in accordance with the available resources in the device. In the last stage of the lifecycle, the DV team also plays an important role. The DV team will be responsible for validating the implemented design, by exercising and testing its functionalities and behavior, and check whether the IP code continues being in accordance with the specifications and requirements of the initial design. The main phases of the workflow among hardware teams are depicted in Figure 30.

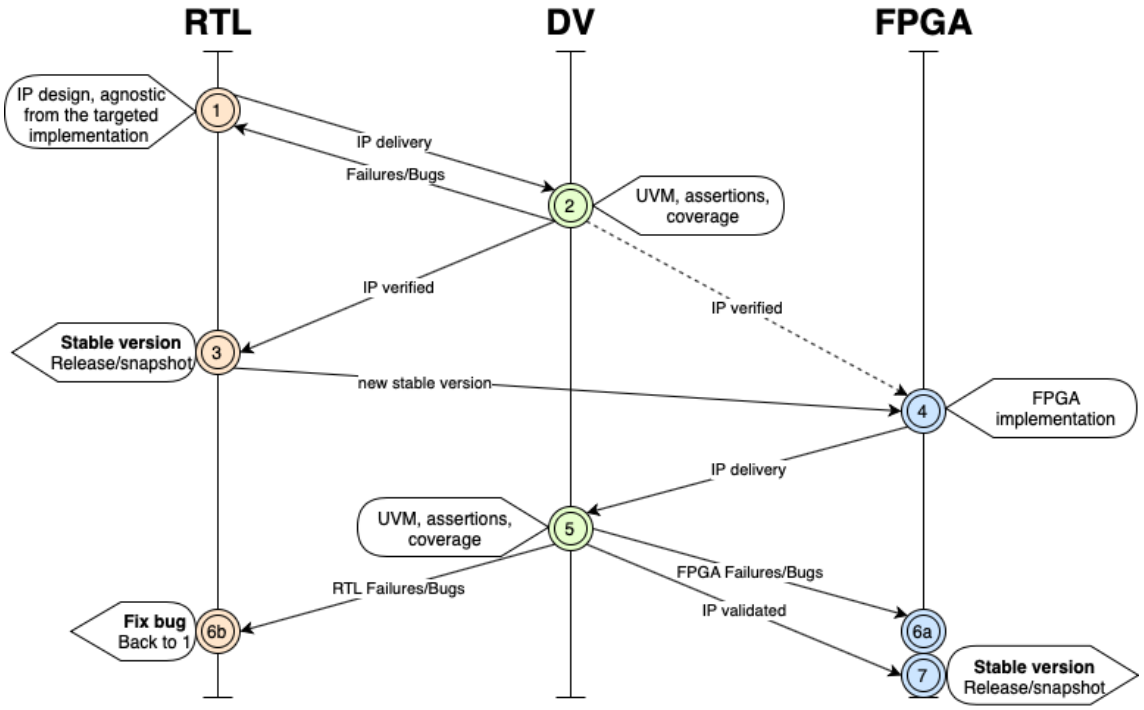



*Figure 30. Workflow among hardware teams*

The workflow implies a fluid and well-known communication procedure among teams. The workflow includes some (semi)automatic mechanisms relying on the features offered by the code repository that the team is using. However, there are also face-to-face meetings to discuss technical details between all the different teams involved in the workflow, and also weekly follow-up internal meetings only for the DV team members.

According to the information shown in Figure B.2, the main steps of the defined workflow are the following ones:

**Step #1:** Design team releases the RTL code to the Verification team.

**Step #2:** Verification team runs directed and random tests with the latest Design code and reports design bugs if any to the design team. Design team makes incremental changes to code to address these bugs.

**Step #3:** Code delivery is done to the FPGA Team to check for any failures after all prior design bugs and testbench bugs are fixed. In parallel verification efforts are ongoing to stress design with more test, corner case scenarios to detect design bugs. Also assertions are put up in the testbench to verify design features.

**Steps #4 and 5:** Any resulting bugs arising from the FPGA team are notified to the Verification team for checking the test scenario; also feedback is given to the RTL team.

**Step #6:** Any resultant design bugs are fixed by the design team and verified by Verification. This process is repeated iteratively from the beginning until the design passes all the tests executed by the Verification team.

# 12. Bottom-up hierarchical structure of the ACME testbenches

The ACME components are verified following a bottom-up approach, like the one shown in Figure B.14; starting at block level and then moving up into a hierarchical and incremental system composition, up to the system top level; which in this case is the ACME accelerator. With this structure, the ACME System Level testbench integrates all the block level verification components like the VPU, agents developed to verify the VAS Tile core design like that of host behavior agent and that of the scalar and VPU dump agents. Also new agents will be developed for the NoC, which would be sitting between the Memory Tile and that of the VAS Tile. Currently the Microengines are being developed by that of the design team for which the VPU standalone testbench will need to be integrated with that of the BFM (Bus Functional Model) developed for verification of that of MCPU and that of the Microengines. Verification of that of the MCPU along with that of the Microengines is being planned for post M18, initially at the block level, and then be integrated with that of the system level testbench. The plan is to ensure that all the block level verification environments for key components in MEEP like VPU, VAS Tile core, MCPU, scratchpad can be reused in the system level verification environment. Further these block level verification environments will be imported as System Verilog based packages in the ACME System Level test bench. Also this block level verification infrastructure would be appropriately port mapped to their respective design hierarchies via usage of assign statements in top level test bench.

A single reference model is used in all the testbenches used in the MEEP project. At this time, we have started by establishing a flow with the well-known RISC-V open source ISS, which is Spike. In parallel, an industry ISS is being evaluated; more specifically, Imperas. The objective is to study the pros and cons of using one or another as a reference model into our verification flow. Also as a future scope, the team is looking at usage of Spike and Imperas for that of multi-core and for that of asynchronous events like interrupts.

Currently, the testbenches developed by the verification team are the ones shown in Figure 31, and each one has its corresponding project in *MEEP Design Verification* Gitlab repository. However, new testbenches will be developed in sync with the RTL team in order to cover each of the ACME components, such as the MCPU at block level, and the Memory Tile at system level, among others.

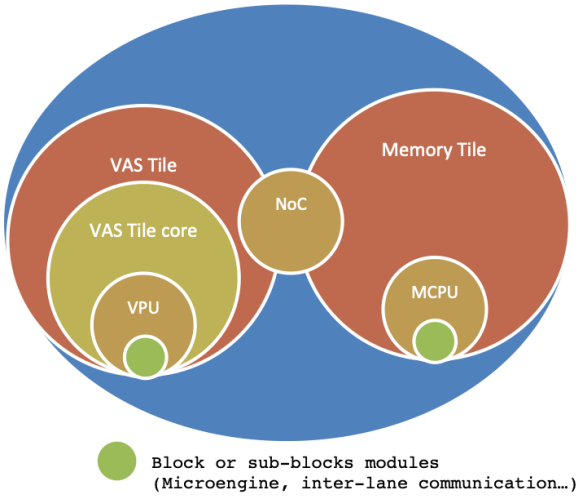



*Figure 31. Bottom-up hierarchical DV approach*

This section is subject to enhancements and it will be revisited in the future, once all the low-level designs and architecture specifications are outlined and there is a design and architecture freeze. In addition, individual Verification Plan documents will be released for that of VPU Standalone, VAS Tile core, and VAS Tile Tests. 64-bit RISC-V ISA tests which are included as a part of the ACME System level verification plan. These include the 64-bit base integer instruction set. The letter "M" stands for standard extensions for integer multiplication and division. The letter "U" stands for that of the user-mode and "A" stands for atomic instructions. Figure 32 shows all possible DUT configurations which can be used with the concept of virtual sequencers encapsulated in an environment. This DUT/design could be the following starting from lowest level:

- VPU standalone
- VPU with In order Core {VAS Tile Core}
- VAS Tile Multicore
- MEEP {subset of ACME}
- ACME

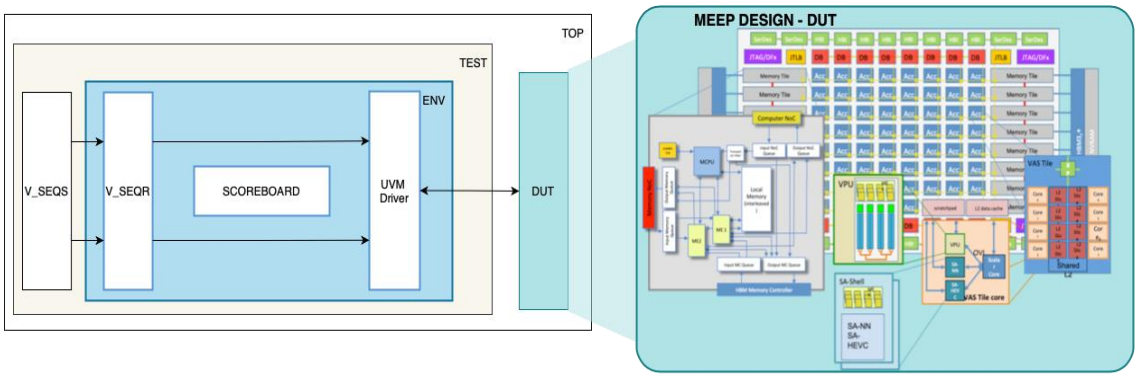



Figure 32. A hierarchical bottom-up view of the possible DUT related to the ACME Accelerator

## 12.1. Module / Sub-module level verification

The following blocks have been identified for Block/Module/Sub Module Verification, the following ones:

- VPU
- VPU + Microengines
- Inter-Lane communication module
- Memory Tile
- SA-shell
- L2/Scratchpad.
- NOC

The approach being used to do module/sub-module verification is a judicious combination of directed test cases and constraint random test cases. These constrained random test cases can be generated in our case through an appropriate configuration of that of the riscv-dv classes or through UVM test cases. Also, SystemVerilog supports randomization of transaction items. Module/Sub-Module level verification techniques used by RTL teams are for sanity checks on the design with the aim of ensuring connectivity checks between design modules/blocks, and basic functionalities. These techniques are usually relying on basic Verilog/VHDL testbenches which are not reusable and, consequently they can be used only for directed tests. The DV team plays a complementary role in creating more random test scenarios to exhaustively test the design as they rely on SystemVerilog which supports constraints and in UVM where the test environment is isolated from the Verification environment.

### 12.1.1. VPU Block-level Standalone Testbench

Being part of the core, together with the Scalar Core and the Systolic Array, the Vector Processing Unit (VPU) is one of the most complex components of MEEP Design. It was originally developed and verified by the EPI project team. MEEP reused this design and modified it in accordance with ACME Specification. Therefore, the MEEP DV team also reused the VPU UVM testbench from the EPI project, modifying it for verification of ACME VPU.

RTL Design team plans releases of the VPU component as per the specification. In accordance with the latest design release, the DV team plans incremental verification approaches of design changes. For each release, both teams have to agree on sign off criteria. The VPU Standalone Test Plan {including ISA tests, c_app tests, sanity tests}, described and shown below in Figure 34, contains detailed information for first ACME.VPU v1.1 release verification sign off, and draft plan for future releases. More information about the ACME VPU v1.1 design is available in the document first chapter of the document. VPU Standalone testbench is a standard UVM testbench shown in Figure 33.

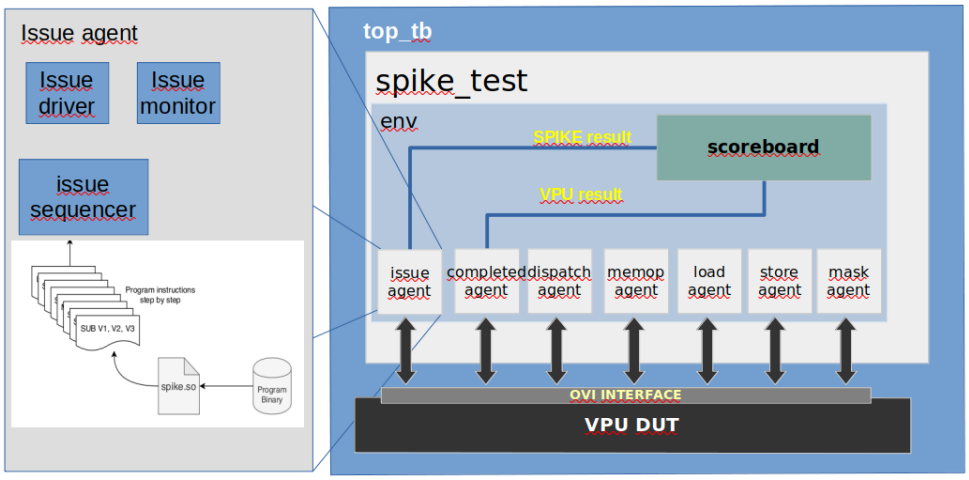



*Figure 33. VPU testbench Layout with UVM agents connected to OVI interface, and the detail of the Issue Agent (on the left).*

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 | FEATURE | TARGET | VPU TAG | DESCRIPTION | STATUS | ISA_TESTS | C_APPS |
| 2 | Fused-Vector Lanes (FVL) and Vector Register File (VRF) | | | | | | |
| 3 | FVL v1.0 - Logic encapsulation | M18 | V1.1.1 | Logic encapsulation, initial modules organization with no functional implications. | PASS | 98.68% | 100,00 % |
| 4 | FVL v2.0 - L2D$ direct access | TBA | | | | | |
| 5 | FVL v3.0 - Dual-Port VRF | TBA | | | | | |
| 6 | FVL v4.0 - Enabled lanes on-demand | TBA | | | | | |
| 7 | FVL v5.0 - Multi-threaded lanes | TBA | | | | | |
| 8 | Micro-engines (ME) design for MEEP | | | | | | |
| 9 | Load operation | TBA | | | | | |
| 10 | Store operation | TBA | | | | | |

*Figure 34. VPU Test Plan in addition to ISA tests + C_app tests*

The test instantiates an environment that contains different agents, scoreboard and memory model that will be described in the following subsections. The agents are connected to the VPU over the Open Vector Interface (OVI) and all together represent the UVC - VPU interface model based on "AVISPADO - VPU Interface version 1.0" specification by SemiDynamics Technology Services SL [B8].

#### 12.1.1.1. UVM Agents

Seven UVM agents control the OVI interface shown on Figure 33 from the UVM testbench side. Each of these agents correspond to one of the OVI subinterfaces needed to stimulate the VPU. More in detail, four of them are bi-directional (having both blue and green arrows in Figure 35 - ISSUE, MEMOP, STORE, MASK_IDX). DISPATCH and LOAD are one directional in direction from the UVC to the VPU and COMPLETED is one directional from the VPU to the UVC direction. Each UVM agent contains a driver, monitor and sequencer, with more details provided below.

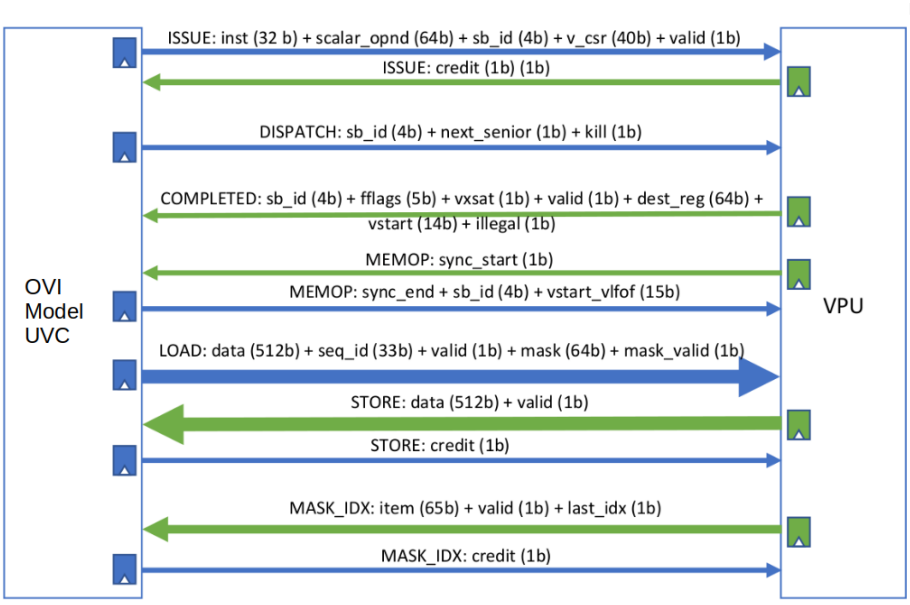



*Figure 35. Scalar core Model - VPU interface*

**ISSUE Agent:**

Issue agent controls the ISSUE subinterface. This agent issues instructions from UVM Verification Component (UVC) towards the VPU on the execution stage. Another important function of Issue Agent's monitor is sending the result of the reference model (Spike) to the Scoreboard. This will be described in more detail in the Scoreboard section.

The VPU co-simulation with Spike is implemented in spike_test UVM test, which calls the spike_seq UVM sequence. This one is an extension of the issue_seq (issue sequencer in Figure below), a sequence executed by Issue Agent as shown in Figure B.8.

**COMPLETED Agent:**

Completed agent monitors the COMPLETED subinterface. Over this interface, the VPU informs the UVC that an instruction has been completed. This valid signal also triggers result comparison versus reference model results. After finishing each vector instruction execution, the VPU asserts the completed.valid signal triggering result comparison in the Scoreboard.

**DISPATCH Agent:**

Dispatch Agent controls the DISPATCH subinterface. This agent plays a role in instruction speculation, setting the "seniority" or age of the instruction and/or deciding to kill some instruction. UVC kills instruction with some probability, which is a parameter that can be controlled in the testbench configuration.

**MEMOP Agent:**

Memop agent controls the MEMOP subinterface used for the synchronization of memory operations: LOAD and STORE. VPU controls sync_start signal that informs UVC if the VPU is ready to send or receive new memory instruction. UVC controls sync_end signal where the UVC informs the VPU that it has successfully finished sending or receiving data.

**LOAD and STORE Agents:**

Load and Store agent controls the LOAD and STORE subinterface of OVI interface used for fetching data from memory and storing data into memory.

**MASK_IDX Agent:**

In the case of masked Load/Store instruction, masks are transmitted on this subinterface from VPU to UVC. In case of indexed Loads/Stores, the same bus is used for sending indices from VPU to UVC.

#### 12.1.1.2. Reference model and scoreboard

Spike has been modified to be aligned with VPU (vector specification v0.7.1 compliant) and to retrieve vector state after executing an instruction. It is pre-compiled to a shared binary object (spike.so) which is used in the UVM Testbench to execute instructions and provide reference results to be compared in co-simulation with the VPU. The instructions are read from the program binary file. Several SystemVerilog DPI-C functions from Spike represent interfaces for the UVM communication with this C reference model.

Spike executes all kinds of instructions (scalar and vector instructions). In the case of the scalar ones, these are executed only by Spike and results are not used for further comparison. Vector instructions are also executed by Spike and results are stored. The same instruction is then pushed by the Issue Agent on Issue subinterface and executed by VPU Device Under Test (DUT). The Scoreboard will read instruction results from the destination register and compare them against Spike results. This co-simulation is done on instruction-by-instruction principle. Therefore, simulation can be stopped as soon as the first mismatch has been observed.

#### 12.1.1.3. Assertions

There are more than 60 enabled assertions, developed by EPI project DV team and reused for ACME.VPU, checking correct behavior on the OVI interface.

#### 12.1.1.4. Verification Flow

The Git repository of VPU UVM Standalone Testbench is hosted on gitlab.bsc.es server at: https://gitlab.bsc.es/meep/meep-design-verification/vpu-standalone-verification.git

UVM testbench instantiation and test execution is done using compile_and_run.py script. MEEP uses Mentor Graphics ModelSim or QuestaSim. The script can execute one test or multiple tests at once - a test suite, which is used for the regression runs.

The most common run test is the **spike_test**, which reads different program binaries with the instruction. The binaries are generated randomly using Google open source platform *riscv-dv* [8] or written manually to target specific instructions and sequences of instructions (**directed tests**).

**sanity-test:** Simplest job running only one test in order to confirm testbench and RTL Design sanity. This sanity job ensures the pipeline failure in case of major code breaks. The Figure 36 shows a snippet of C_APP tests being run to verify VPU with CI/CD.

```
$ echo "This job runs c_apps regression."
This job runs c_apps regression.
$ python3 compile_and_sim.py --target meep -t spike_test -s -nsu --save-results -r c_apps
2021-05-31 20:46:12,918 :: INFO     :: Executing test 001_axpy
2021-05-31 20:56:00,318 :: INFO     ::     - 0
2021-05-31 20:56:00,335 :: INFO     :: Executing test 004_somier
2021-05-31 21:29:41,696 :: INFO     ::     - 0
2021-05-31 21:29:41,711 :: INFO     :: Executing test 005_fftw
2021-05-31 21:30:35,915 :: INFO     ::     - 0
2021-05-31 21:30:35,932 :: INFO     :: Executing test 002_matmul_JJ
2021-05-31 21:30:40,039 :: INFO     ::     - 0
2021-05-31 21:30:40,055 :: INFO     :: Executing test 003_spmv
2021-05-31 21:30:54,413 :: INFO     ::     - 0
2021-05-31 21:30:54,579 :: INFO     :: ************************************************************
2021-05-31 21:30:54,579 :: INFO     :: Passing rate: 100.00
2021-05-31 21:30:54,579 :: INFO     :: ************************************************************
```

*Figure 36. CI/CD regression run example*

### 12.1.2. SA-Shell

Each of the SAs will be wrapped by the SA-Shell, which has the same control interfaces (with the scalar core) and memory interfaces (with the L2/scratchpad) as the VPU block. The block level verification environment for that of the SA-shell verification is planned to be done by reusing the UVM infrastructure of VAS Tile core and having all the design files compiled related to that of the SA-Shell. Directed tests and random riscv-dv tests would also be used to verify the SA-Shell.

### 12.1.3. Systolic Arrays

These blocks are designed and developed by MEEP Partners as shown in Figure 37, and they will have to provide the specifications of their designs in order to adapt the UVM environment, and the corresponding agents, to their requirements and functionalities. It is important to highlight that each of these SAs will interact with the SA-Shell. Block level VAS Tile core UVM environment will be re-used to verify the systolic arrays. System Verilog assertions will be used to verify the OVI Interface connecting scalar core to the SA.

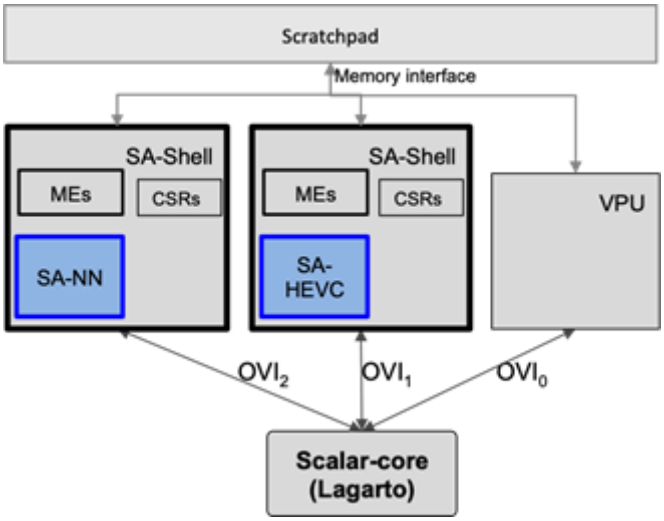



*Figure 37. Block Diagram of SA and its associated shell*

### 12.1.4. L2 data cache and scratchpad

Functional verification of L2 Data cache and scratchpad will be done in conjunction with that of the VPU test bench and for this the VPU test bench will be re-used for this purpose. This would also involve the integration of AXI Lite VIP in VPU test bench and running AXI read and write test cases. Please refer to section 6.2.5.1.

### 12.1.5. RISC-V Processor (Scalar core and MCPU)

The overall verification strategy to validate the correctness of a RISC-V core relies on the experience of previous RISCV projects. In this sense, any RISC-V processor will follow the Reference Verification Flow proposed by the OpenHW Group. Actually, the same sets of tests might be applied. This is applicable for verifying that the design is compliant with the RISC-V ISA.

In addition, the DV team needs to work on adding new features to the Scalar core agents in order to reflect the details of the ACME processor, in terms of functionality and behavior; which impacts on the interfaces. The UVM frameworks for the core is shown in Figure 38.

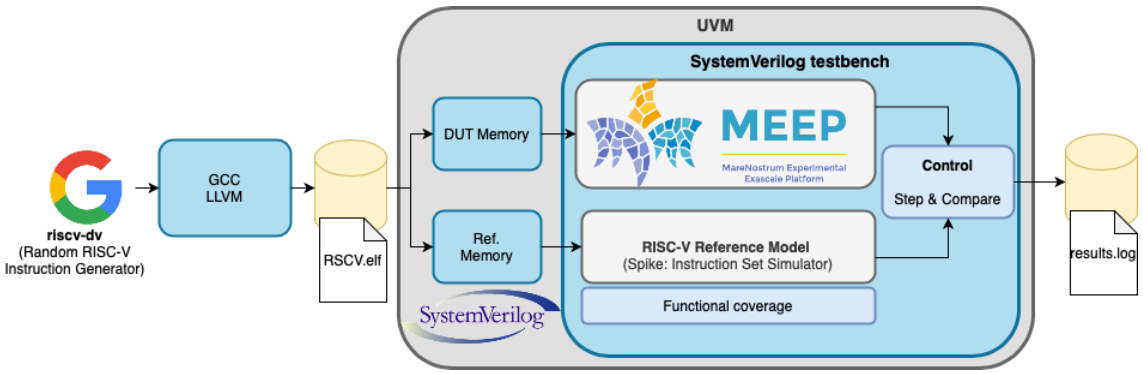



*Figure 38. Reference Verification Flow for RISC-V cores*

#### 12.1.5.1. MCPU:

Functional verification of the MCPU will start off with block level activity to see its interactions with that of the VPU. The steps shown below illustrate the approach to functionally verify the MCPU:

- **Step #1**: Initially verify the MCPU IP at the Block Level. [Test Interface between MCPU and that of VPU through the AXI4 Interface Diagram shown in Figure B.20]
  - **Step #1.1**: Integrate the AXI4-lite VIP in VPU testbench.
  - **Step #1.2**: Add Design references to that of MCPU & Microengine RTL along with Interconnect.
  - **Step #1.3**: Add all AXI address maps in testbench related to that of microengines banks.
  - **Step #1.4:** Program the microengines to perform load/store operations.
  - **Step #1.5**: Perform all the AXI operations related to read and write address and data

operations as shown in Figure B.20 on left bottom.

- **Step #2**: UVC agent developed here at Block Level can be imported as a Package at the ACME System Level including the Verification environment.

- **Step #3**: All test and sequence packages declared here can be imported to the Top Level ACME.

Figure 39 shows below the MCPU to the VPU Interface:

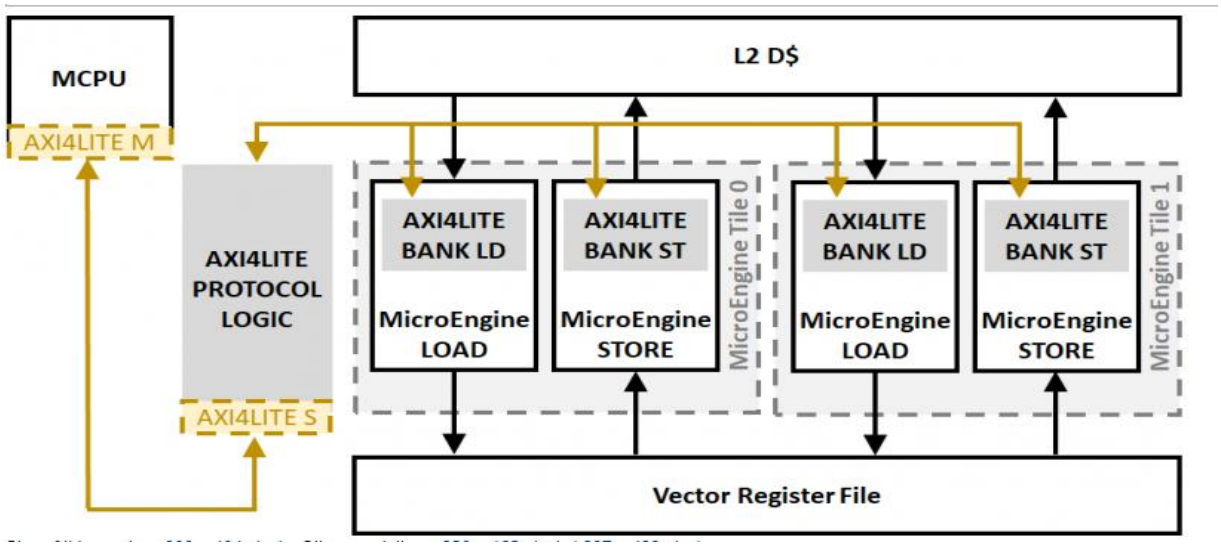



*Figure 39. Block Diagram showing the Interface of the MCPU to the VPU*

The Verification tests which can be executed at the MCPU to that of VPU Interface are the following:

- Perform Load and Store operations using a particular Microengine bank.

- Target Load and Store Operations using all the possible Microengine banks and all feasible Microengine address maps provided by the Design team.

- Perform all the AXI4 Transaction combinations which exercise read, write data and address channels as well as the write response channels. [Figure 40 shows the combination of AXI tests which can be exercised.]

| Signal | Size (bits) | Group | Direction | Description |
|---|---|---|---|---|
| ACLK | 1 | Global Signal | M->S | Clock source. |
| ARESETn | 1 | Global Signal | M->S | Global reset source, active low. |
| AWVALID | 1 | Write Address Channel | M->S | Write address valid |
| AWREADY | 1 | Write Address Channel | M<-S | Write address ready |
| AWADDR | 32 | Write Address Channel | M->S | Write address |
| AWPROT | 3 | Write Address Channel | M->S | Protection type (unused) |
| ARVALID | 1 | Read Address Channel | M->S | Read address valid |
| ARREADY | 1 | Read Address Channel | M<-S | Read address ready |
| ARADDR | 32 | Read Address Channel | M->S | Read address |
| ARPROT | 3 | Read Address Channel | M->S | Protection type (unused) |
| WVALID | 1 | Write Data Channel | M->S | Write valid |
| WREADY | 1 | Write Data Channel | M->S | Write ready |
| WDATA | 32/64 | Write Data Channel | M->S | Write Data |
| WSTRB | 8-Apr | Write Data Channel | M->S | Write Strobe |
| RVALID | 1 | Read Data Channel | M<-S | Read Valid |
| RREADY | 1 | Read Data Channel | M->S | Read ready |
| RDATA | 32/64 | Read Data Channel | M<-S | Read Data |
| RRESP | 2 | Read Data Channel | M<-S | Read Response |
| BVALID | 1 | Write Response Channel | M<-S | Write response valid |
| BREADY | 1 | Write Response Channel | M<-S | Write response ready |
| BRESP | 2 | Write Response Channel | M->S | Write response |

*Figure 40. List of AXI4 Tests which can be done to test the MCPU interface to that of VPU*

### 12.1.6. Network on Chip (NoC)

- Test to facilitate transaction from one VAS Tile to another VAS Tile using Address NoC Agent [Remote L2 Request].
- Test to facilitate transaction from one VAS Tile to another VAS Tile using Data Transfer NoC Agent [L2 Data ACK to another Tile].
- Enable VAS Tile traffic to the Memory Tile (MCPU) [Memory Write via Data Transfer NoC].
- Enable VAS Tile traffic to the Memory Tile (MCPU) [Memory Load request via Address NoC].
- Enable VAS Tile traffic to the Memory Tile (MCPU) [transactions via Control NoC. This is like a L2 scratchpad ACK command.
- Enable VAS Tile traffic to the Memory Tile (MCPU) [Scratchpad Data Reply via Data Transfer NoC].
- Enable Memory Tile (MCPU and Memory Controller) traffic to the VAS Tile transactions via Data Transfer NoC.

- Enable VAS to MCPU transaction and from that of MCPU to that of VAS via Address only NoC [Example: Core issues *vsetvl* instruction to the core].
- MCPU to VAS Tile communication using a Data Transfer NoC Agent.
- VAS Tile to MCPU communication using a Control only NoC Agent [Scenario where the scratchpad command is completed].
- Memory Tile to another Memory Tile (MCPU) [It services two types of requests: (1) Memory operations (DRAM reads and writes), (2) Vector transactions (These get converted into Memory ops. One can come for a VAS Tile core or Memory Tile; or it goes to/from the VAS Tile core/MCPU pairs. The memory operation returns the result to the original memory requestor: VAS Tile core or Memory Tile.

## 12.2. System-level verification

The system-level verification in the ACME architecture includes: the VAS Tile core, the VAS Tile and the Memory Tile.

### 12.2.1. VAS Tile core

A VAS Tile core in ACME represents the computational module which will be instantiated several times in order to compose a more complex accelerator system (a VAS Tile). This component includes the modules depicted in Figure 41: a RISC-V scalar core, the L2 data cache, a scratchpad, and the specialized coprocessors; the VPU and the Systolic Arrays, one dedicated to process neural networks, and another for image processing.

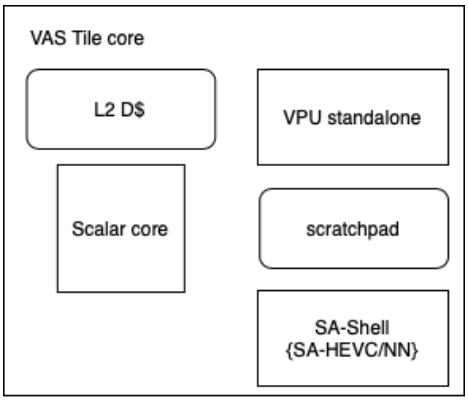



Figure 41. Main components of a VAS Tile core in ACME accelerator

As a first approach to an ACME VAS Tile core, the RTL team is modifying an IP, also developed in BSC. This IP is known as *DVINO*, and it contains the main scalar core pipeline pieces needed by the VAS Tile core role, with several additions in the memory hierarchy and coprocessors. This design will be used as an integration design to test and verify the functionality of its inner modules as they evolve during the MEEP project lifetime.

The main goal is to develop a reusable UVM testbench, able to be used to stress the whole ACME accelerator. More specifically, we aim to reuse the AXI interface test, and the scalar core scoreboard together with the UVM components needed by it.

The UVM testbench is developed from scratch, and has the following agents like host behavioral, signature dump, and has logic for comparison against Spike reference model.

For the AXI interface to allow communication between the VAS Tile core and the rest of the system, a commercial Verification IP (VIP) will be used, more specifically, the Questa VIP (QVIP) [B9].

The project can be found in gitlab, within the *MEEP Design Verification* repository (https://gitlab.bsc.es/meep/meep-design-verification/axi-block-level-verification/-/tree/dev), and its structure is described in Figure 42.

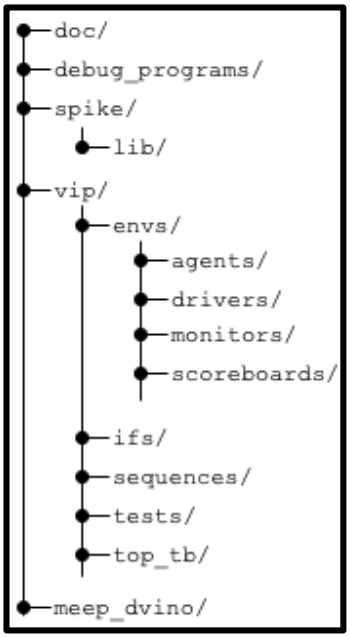



Figure 42. Structure of the VAS Tile core repository

- doc: documentation; including a specific verification plan for the current status of the IP.
- debug_programs: test programs intended to debut new features of the testbench.
- spike: compiled spike version used by testbench.
  - lib inside is needed by spike binaries.
- vip: testbench components.
  - envs: UVM environments and components subfolders.
  - ifs: SystemVerilog interfaces.
  - sequences: sequences and sequence items.
  - tests: UVM tests.
  - top: SystemVerilog testbench modules and parameter packages.
- meep_dvino: Submodule containing rtl.

#### 12.2.1.1. Verification Strategy

The block diagram of the VAS Tile core testbench, depicted in Figure B.24, is developed in SystemVerilog using UVM. The verification flow starts with the execution of a test program loaded in the memory model (Memory block in Figure B.24), which provides the stimuli to the VAS Tile core.

The test program may be manually written in assembly or C language, or may be randomly generated. More specifically, or the random generated tests the riscv-dv environment [10] will be

used. Then, it is compiled to an.hex format, and then placed into a memory model, which in this case is just an AXI4 QVIP slave agent and acts as the DRAM of the system. The QVIP AXI library is an industrial Verification IP developed by one of the design automation companies (Mentor Graphics - Siemens) and it helps to improve the quality of the RTL, and also to speed up development by providing AXI models and AXI assertions.

The test starts when the clock starts ticking and the reset is released. The core is monitored when an instruction is committed. Then, the testbench will compare its state against the reference model. During the execution, both the AXI4 interface, used for the memory transactions, and the internal AXI4-Lite interface, used for block interconnection, are monitored. The test goes to the end when the scalar core executes the "tohost" routine, which shall be called from the test program.

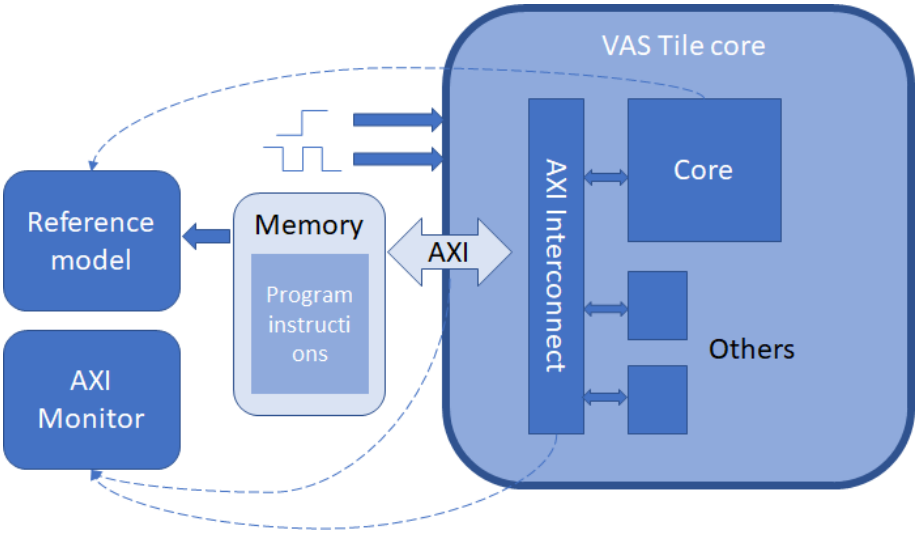



Figure 43. VAS Tile core UVM testbench

The scheme of the VAS Tile core as a DUT is shown in Figure 44. Actually, the *dvino_asic_top* in that Figure 43 represents the VAS Tile core design playing the role of a DUT in the UVM environment. Note the interfaces between the VAS Tile core and the rest of the system is through AXI and AXI Lite interfaces. At this point in time the scalar core is connected to one coprocessor, the VPU. However, in the future, we are targeting the verification of that of the Systolic array and multiple OVI Interfaces; one OVI Interface from that of scalar core to that of VPU and other two to the Systolic arrays (one per SA).

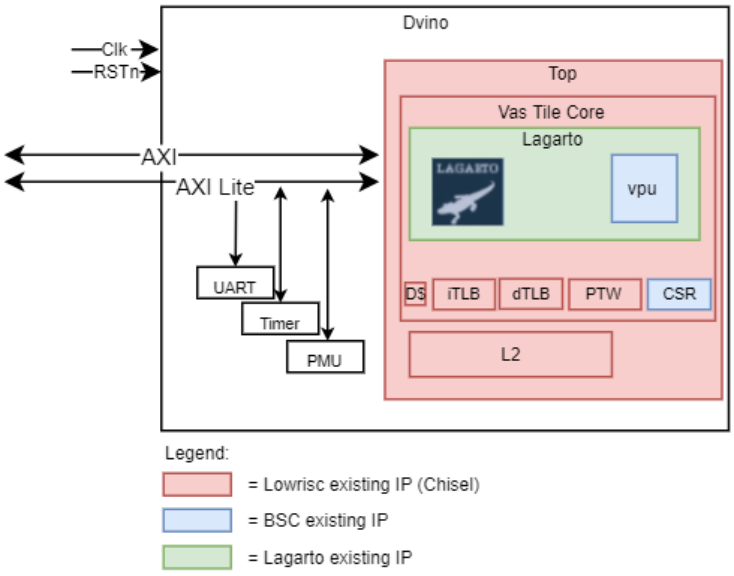



*Figure 44. Design Block Diagram of the VAS Tile core as a DUT*

Every executed instruction by the VAS Tile core will be compared against a reference model to determine if the result is as expected. In addition, the AXI interfaces both external (to the memory system) and internal (for intra block communication, such as UART, timer, or PMU) will be monitored. The final goal of this verification flow is to exercise as much of the RTL design as possible in order to ensure its correct functionality and behaviour. The test plan also stresses the VAS Tile core by randomly triggering asynchronous interrupts.

#### 12.2.1.1.1. Reference model

The plan is to extend the current DPI-C functions to be able to interact with Spike also using interruptions. For MEEP the Spike wrapper has been extended to get more information from the Spike simulator when an instruction is executed. This information is then used in the scoreboard for comparison against the RTL's scalar core.

### 12.2.1.2. Test plan

The Test plan can be found in the Gitlab project (Table 12) under the *doc/* folder (*AXI4_vplan*). A snippet of it is shown in Table 10.

Bear in mind that directed tests in the table below refer to the ISA and vector tests.

| **Test Title** | **Description of Tests** | **Checked by assertion directive or simulation** |
|---|---|---|
| axi4_vip_assert | Enable AXI4 Questa VIP embedded assertions for all tests. | Assertion |

| | | |
|---|---|---|
| signature_check | Check that processor signature matches spike signature:<br>Monitors to observe scalar core state and vpu state.<br>Scoreboard to step and compare with spike for signature<br>This will verify that the program is properly running on the processor which implies that the AXI transactions were correct. | Checker |
| timing_randomise | Randomise slave timing response within spec limits<br>This will stress the AXI under no ideal situations. | Random_item |
| mem_instruction_randomise | Randomise instructions that imply memory transactions (lw, sw, etc ...).<br>This will generate programs that make more use of the AXI4 interface. | Random_item |
| base_test | Test that will serve as a base to be extended.<br>Will instantiate AXI4 VIP including assertions, load program in memory, and check processor signature. | Test |
| random_axi4_seq | Will randomise AXI4 slave sequence by using timing_randomise and mem_instruction_randomise.<br>1. First randomise program using mem_instruction_randomise.<br>2. Then load program into memory using backdoor access to AXI QVIP.<br>3. For every slave transaction use timing_randomise.<br>4. Test will finish when the program reaches the end. | Sequence |
| random_test | Extension of base_test, will use random_axi4_seq. | Test |
| directed_test | Extension of base test will use programs manually written, either in C or in assembly code.<br>This test may still make use of timing_randomise. | Test |
| peripheral_test | Connect and exercise the different peripherals that may be connected in the FPGA: PCIe, Aurora, Ethernet, DDR4.<br>Each peripheral will be modelled using an AXI4 and AXI4-lite QVIP agent.<br>They will be exercised by the program running in the DUT. | Test |
| axi4_io_monitor | Inside the VAS tile the AXI4-lite bus will be used for a different set of io peripherals, such as the UART.<br>There will be a monitor instantiated that will check all the AXI4 and AXI4-lite transactions. Since this is an internal bus, the stimuli will be applied indirectly by the program running in the CPU. | Monitor |

*Table 10. Test plan*

### 12.2.2. VAS Tile

A VAS Tile in ACME is a multi-core system, in which each core is an instance of a VAS Tile Core. All the cores have a shared L2 data cache, and, as an initial approach, each core will have a private scratchpad. The schematic of this component is shown in Figure 45. More information

about the RTL implementation details of the VAS Tile approach can be found in the *Periodic Technical Report, M1-M18 Part B.*

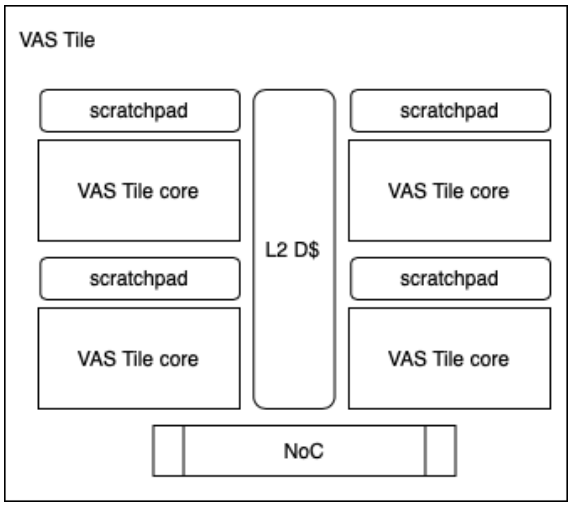



*Figure 45. A VAS Tile schematic view*

The first VAS Tile approach relies on the OpenPiton project, which provides a scalable, configurable, and open-source NoC-based infrastructure. This setup was modified to include that of the BSC core used in conjunction with that of NoC.

Based on this decision, the current activities are focused on analyzing that project in order to understand how to reuse, adapt and improve the system for the ACME requirements and specifications, mainly focusing on the NoC implementation as a baseline.

The project can be found in gitlab, within the *MEEP Design Verification* repository (*https://gitlab.bsc.es/meep/meep-design-verification/lagaro_modified/-/tree/dev*), and the structure of the most important directories is as shown in Figure 46.

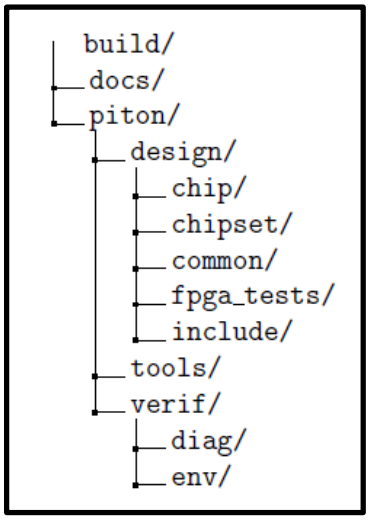



*Figure 46. Openpiton Directory structure*

At the top file level are the build/, docs/, and piton/ directories.

- The build/ directory is a working directory for. It acts as a scratch directory for files generated when building simulation models, compiling tests, running simulations, etc. All logs files are created inside the build directory.

- The design/ directory contains all synthesizable Verilog for OpenPiton.
- All scripts and tools used in the OpenPiton infrastructure reside in the tools/ directory.
- The verif/ directory contains all verification files. This includes assembly and C tests, or diags, unit tests, and simulation models.
    - Within verif/, the diag/ directory contains all assembly and C tests. In addition, it also contains diaglists, which define parameters for certain tests and defines groups of tests, or regressions, and common assembly and C test infrastructure.
    - The env/ directory contains non-synthesizable Verilog files (testbenches) needed to build simulation models.
    - For unit testing simulation models, the tests are located within the env/ directory as opposed to the diag/ directory.
    - In general, the manycore simulation model will run assembly and C tests in the diag/ directory, and all other simulation models will run based on unit tests in env/. Infrastructure for unit testing is provided in the env/test_infrastruct/ directory along with a script to quickly and easily generate a new simulation model, env/create env.py.

### 12.2.3. Memory Tile

The verification of Memory Tile will initially be at the block level and once that is accomplished, it will be expanded to be included in the system level verification environment where it would interact with that of NoC and the VAS Tile. Block level verification is planned with the objective of functionally verifying the interactions between Memory Tile, and the different components of a VAS Tile Core {scalar core, micro-engines and that of the Vector Processing Unit}. Once this is implemented, the next step would be to integrate this environment with that of NoC and VAS Tile and ensure that functionality is not broken between Memory Tile and that of the VAS Tile. The following Figure 47 shows a tentative block diagram of the Memory Tile in ACME, which are still evolving as design and simulation progress.

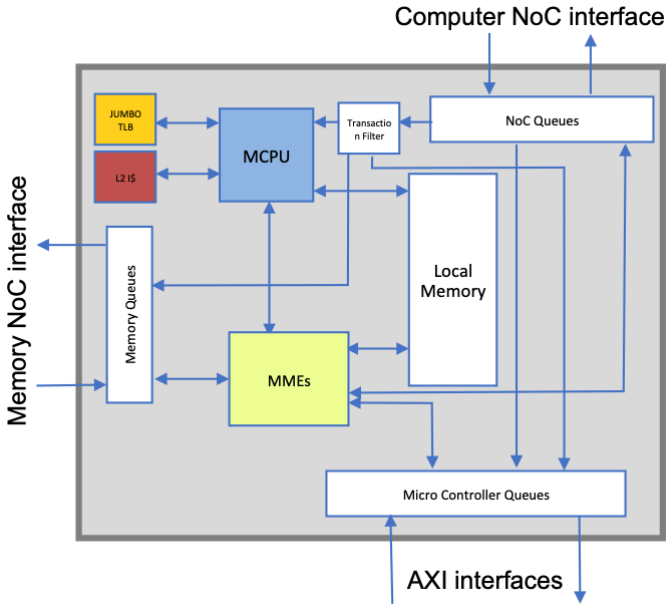



*Figure 47. Block diagram of a Memory Tile in ACME*

## 12.3. SoC level verification: ACME

The main objective outlined in this section is to get into the structure and organisation of the ACME Soc level testbench and the strategy used to integrate all the block-level verification components for system level design. Building up a Verification testbench to verify complex ACME accelerator architecture at the system level involves putting up a structure which is scalable to verify not only single core but also multi core processor design. Figure 48 depicts a schematic view of the ACME accelerator, in which all the main components are integrated: a grid of VAS Tiles, Memory Tiles, the NoC, and the Memory Controllers (MC). Actually, the HBM is not part of the accelerator itself, but it highlights the need to understand the full ACME accelerator and all its internal and external interfaces; which in the case of the emulation platform these interfaces will connect with the MEEP FPGA Shell.

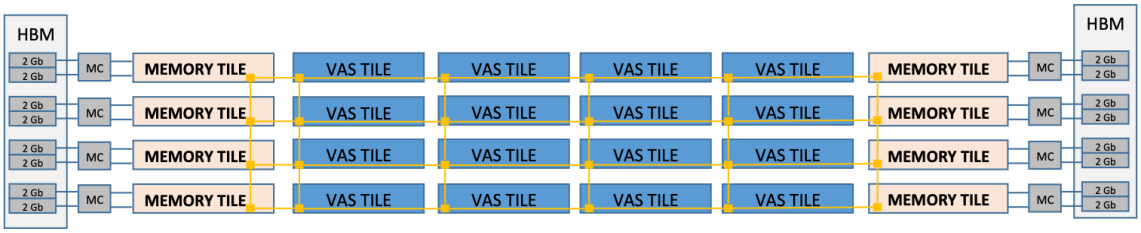



*Figure 48. ACME schematic view*

Re-usability of all the testbench components as well as keeping the structure configurable are given a high priority to verify various design configurations. The testbench is architectured in a manner that the earlier work, done at the ACME VPU block level and VAS Tile core respectively, with AXI4 QVIP integrated can be re-used and integrated along with other NoC Agents which are being planned. The integration strategy has also been done for that of the Memory Tile attached to the VAS Tile. The goal of the ACME system level testbench is to ensure that high quality and functional RTL is delivered to the FPGA team. FPGA team is involved in testing

aspects of the host system and behavior of the PCIe system which is used as a hard IP on the FPGA board along with that of high bandwidth memory. Verification done by the DV team complements the efforts of the FPGA team and ensures that the RTL code works well with components of the MEEP FPGA Shell.

### 1.1.1. ACME Verification Strategy

The ACME System Level testbench is being developed in SystemVerilog with the UVM Methodology. The verification intent of this infrastructure is to stress test design starting from a single core to multi-cores with directed tests as well as random test cases generated by that of the riscv-dv engine. The starting point of stimulus are programs running on a single/multi core VAS Tiles. The verification strategy used in this project entails running through all the RISC-V ISA sets starting from that of v0.7.1 Vector extension set to that of v0.10. The stimulus also provides a reference model like Spike/Imperas along with that of ACME Design and this is compared to design transactions. The plan in future from verification involves extensive verification of the NoC which sends transactions among Memory Tiles and VAS Tiles, and also among VAS Tiles. Also the intent of the Verification here is to ensure that the ACME design is able to handle asynchronous events like Interrupts and check the response of the design to illegal instructions. Assertions are also being planned on the OVI interface to check the design functionality; and functional and code coverage statistics are planned to get confidence from a DV perspective. Once the MCPU and Microengines are tested at that of the level of the VPU [Block Level] the plan is to integrate them into the ACME system testbench.

The plan to build a system level testbench entails integrating all the block level verification components like the current VPU, VAS Tile core, future MCPU and Memory Tile testbenches with AXI4 QVIP integrated, along with all of the VPU and NoC Agents. This integration process is depicted in Figure 49. The NoC agents need to be constructed from the scratch based on the interfaces and signal level details of the NoC specifications in the design.

A full view of the verification strategy along with the design is depicted in Figure 50. The top left Figure represents a simplified view of a generic UVM environment. Then, the Figure on the top right side represents the different RTL designs you might have as a DUT (VPU, VAS Tile core, VAS Tile...) {Modules and submodules have not been included for simplicity}. And finally, the Figure at the bottom represents the main elements that need to be present as part of the UVM environment to verify the DUT. All the agents surrounded by a dot line are the ones that need to be coded up from scratch. The VPU Agent is borrowed from the EPI project.

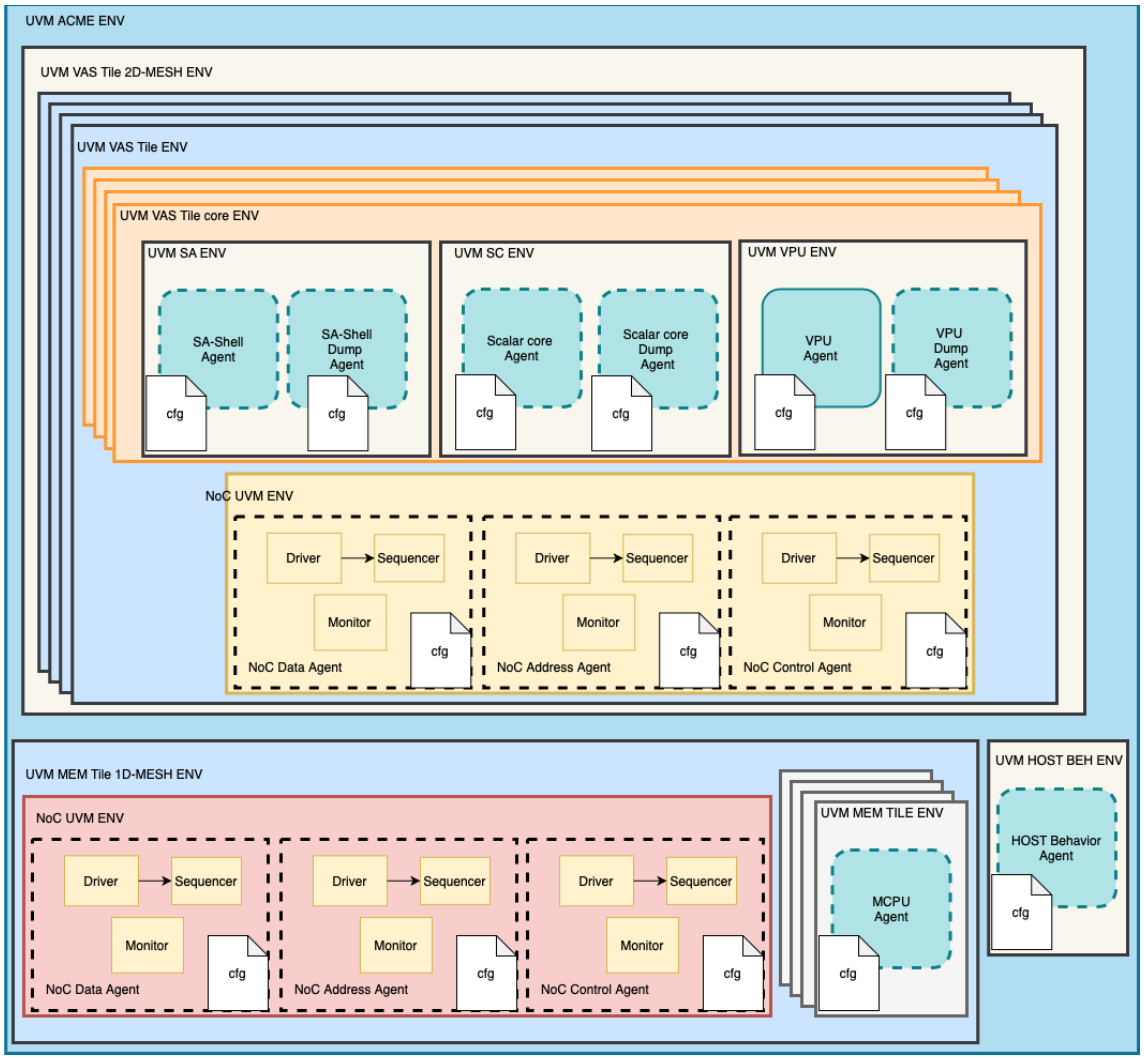



*Figure 49. Diagram illustrating all the Verification components to be integrated in system level testbench [VPU + VAS Tile core testbench + NoC UVC Agents + RISC V-DV Google Utility].*

The system environment is encapsulated in that of test structure. Each of the components has its individual tests, which are imported as a part of the system test package in the top level testbench. The Top level ACME testbench instantiates the top design, and has references to all the virtual interfaces used in the testbench. All the agents used in the testbench can be conFigured to be either as active or passive with a default configuration being provided in the base test. The individual tests which extend the base test can override the default configuration of that of the agent. Stimulus is provided via riscv-dv structure and this stimulus is applied to both the design as well as the reference model. Scoreboard does the comparison of the DUT and the reference model transactions and has the logic to tell whether the given test scenario passed or failed.

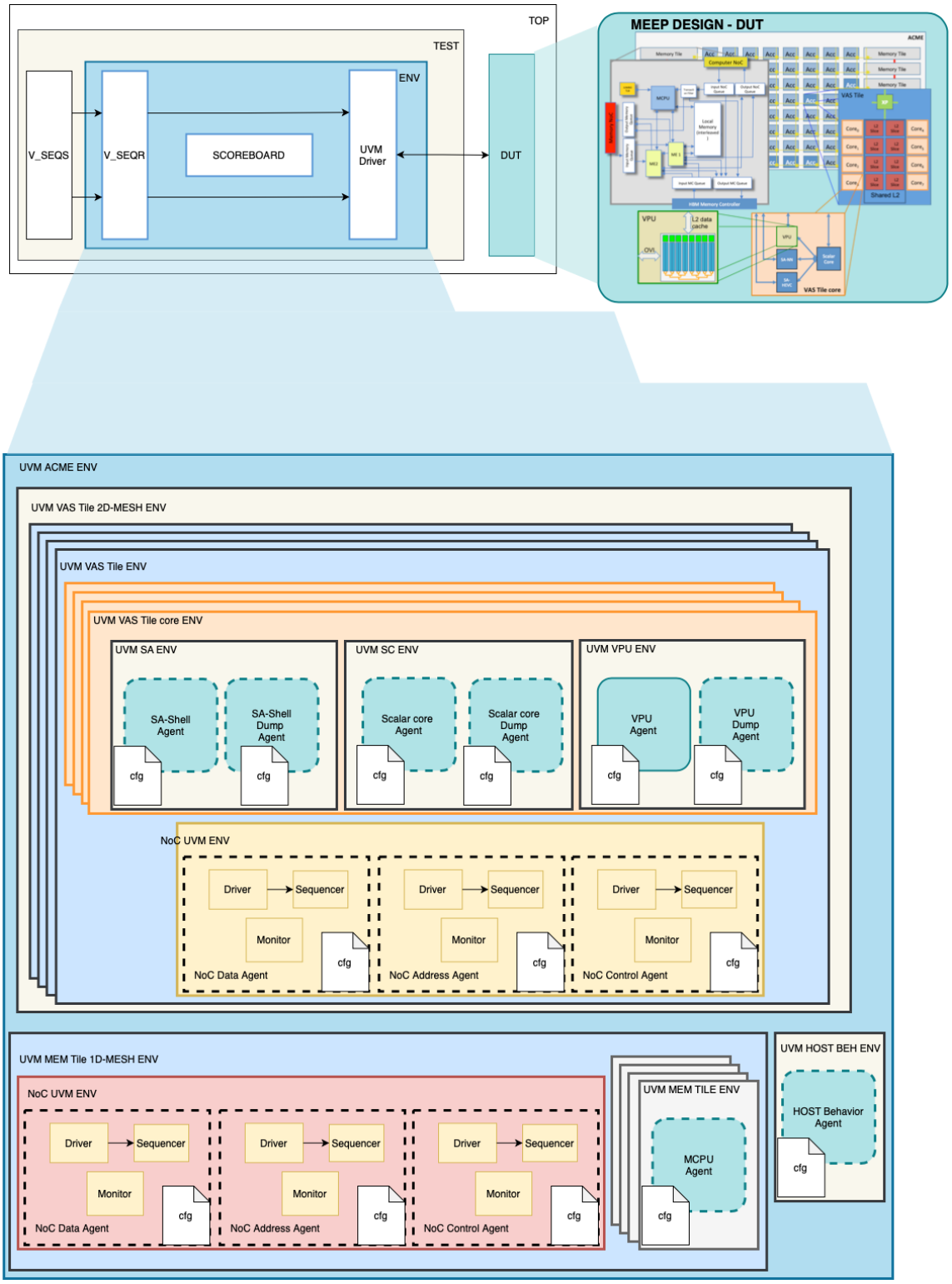



*Figure 50. ACME Test bench with all UVM components adapted for Multi-Core*

There are three agents for that of NoC which includes the NoC Data agent, the NoC Address agent and the NoC Control agent; each one of them having their individual environments and they are encapsulated in a NoC UVC Package. There are other agents shown pertaining to that of host behavior agent, scalar dump and that of the VPU dump agent and also the agents pertaining to that of the VPU Block level agents. All these individual agents with their respective environments

are encapsulated in that of the UVM ACME ENV which has its virtual sequencer, which has references to individual sequencers.

The integration process involves several steps, which will be done in the following order:

- **Step #1**: Create 3 NoC agents namely Data Transfer NoC, Address NoC, and Control NoC. Each agent is a combination of a driver, sequencer and monitor; all of them will be placed in the MEEP Design Verification Gitlab repository (https://gitlab.bsc.es/meep/meep-design-verification/dvino_system_level_verif) within the *uvc_agent* folder. This step is executed once all the design interface signals, transaction items are specified. NoC Interfaces need to be coded up.
- **Step #2:** Create a NoC UVM Environment which encapsulates all the 3 NoC agents.
- **Step #3**: Import a NoC UVC Package, which includes all the 3 NoC agents along with that of their environment. [UVC Stands for a Verification component].
- **Step #4**: NoC Sequences and tests to be coded up. All the NoC tests will be included as a part of the system_level_dvino_test_pkg which will be imported at the Top level testbench.
- **Step #5**: Add all the VPU tests, DVINO block level tests in the dvino_system_level_test_pkg which will be imported at the top level environment.
- **Step #6**: Import all the Block Level NoC Agents along with that of the NoC environment as a NoC_uvc_pkg which will be imported in the system level UVM environment.
- **Step #7**: Import all the Block Level VPU Agents along with that of the VPU environment as a vpu_uvc_pkg which will be imported in the system level UVM environment.
- **Step #8**: Instantiate all the block level environments like VPU, NoC, VAS Tile core in the system level environment.
- **Step #9**: Add a System Level Virtual sequencer, which will have the pointer to that of the individual VPU, DVINO and NoC sequencers. The connection of virtual sequencers to that of the actual sequencers will be done in the connect phase of the environment.
- **Step #10**: Import all the AXI VIP Packages as a part of the top level testbench.
- **Step #11**: ConFigure all that of the Interfaces of the VPU, VAS Tile core testbench {host behaviour and signature dump scalar and VPU}, NoC in the top level configuration.
- **Step #12**: Ensure that all the Block level testbench configurations have a reference to their individual interfaces.
- **Step #13**: Block level testbench configurations will have their configurations set in a system level base test.
- **Step #14**: Code up a System level Base test from which all the tests are extended. Default configurations are done in base tests, which can be extended in that of the test case.
- **Step #1**5: Add up a system level base virtual sequence from which all the other virtual sequences extend for.

- **Step #1**6: Integrate a scoreboard in the environment. This scoreboard will be connected to the monitor in the UVM connection phase. This also entails the integration of the reference model (Spike or Imperas) in the environment.
- **Step #1**7: Add System Level *defines* in the testbench, which can be passed to the rest of the testbench files. These *defines* will conFigure the number of lanes in VPU, refer to the design hierarchy and specify the widths of the address and data in that of transaction items and the number of cores to be used in the design.
- **Step #18**: Port map block level verification infrastructure to that of design signals in top level design.
- **Step #19**: Logic to be added in the environment to instantiate all the necessary number of agents based on the number of cores.
- **Step #20**: Understand the riscv-dv code from OpenHW and integrate the same in the testbench as a folder.
- **Step #21**: Integrate AXI4-lite setup in the testbench to test UART and peripherals.
- **Step #22**: Create MCPU agents, which will talk to other ACME design components through that of the NoC design block. This will need to be integrated as that of a UVC Package. [First level of Verification would be done at Block Level with MCPU talking to that of the VPU through that of the AXI4 Interface, section below has the details.] These integration steps are briefly outlined in Figure 51 as shown below:

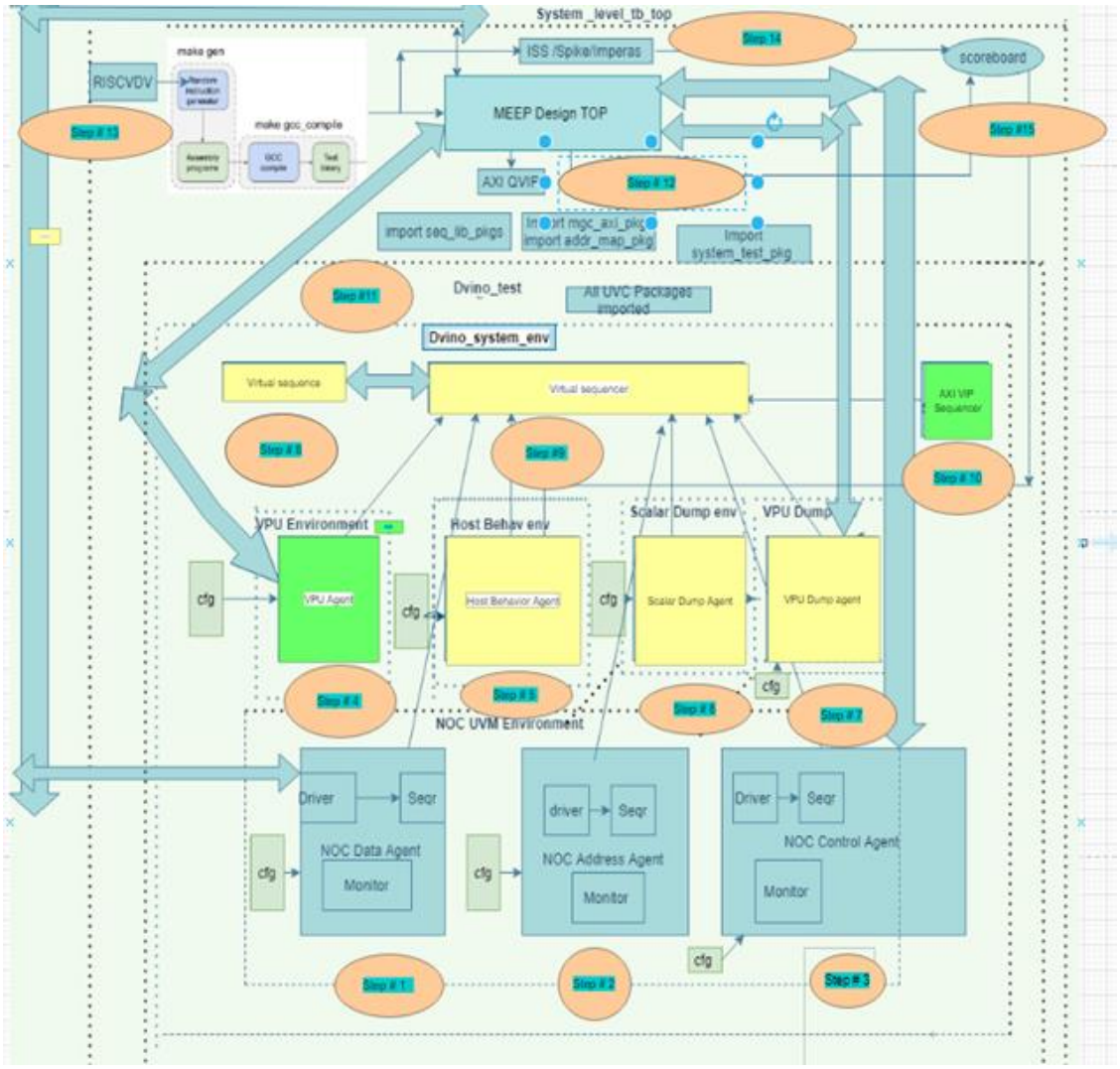



*Figure 51. Integration steps in ACME System Level Test Bench*

**Developed code is available at the following area on gitlab {dev branch}**

https://gitlab.bsc.es/meep/meep-design-verification/dvino_system_level_verif

The following is the folder structure of the System Level MEEP testbench:

It consists of the following folders with their explanations: Figure 52 illustrates the folder structure of the ACME System Level Testbench:

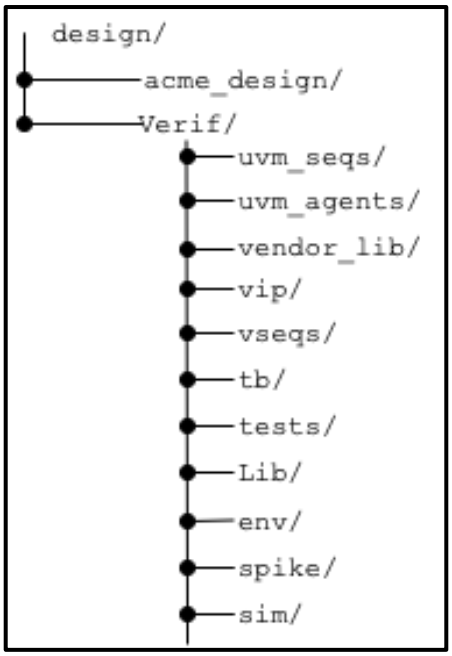



*Figure 52. Folder Structure of MEEP System Level testbench*

- **acme_design**: This folder contains all the RTL files for that of ACME.
- **Verif:** This folder contains all the UVM components and transactions.
- **tb:** This has the Testbench Top file where the Design is instantiated along with the System level virtual sequencer, top level configuration file and all the local parameters and defines.
- **uvm_agents:** This folder has subfolders for the various agents. This is the placeholder where all the NoC agents, vpu agents, host behavior agents, scalar and VPU dump agents are placed along with their accompanying logic. This folder has code for the driver, monitor, sequencer along with that of the transaction items and the environment. As of now the agents that we have planned to integrate are the following: NoC {3 agents: data, address and control}, VPU agents, host behavior agent, scalar and VPU dump agent. In case the design team adds any new blocks /sub-blocks with major functionality then the DV team will plan to add an agent to verify the block.
- **uvc_seqs:** All the UVM based sequences would be placed in this folder. This includes all the NoC sequences {in future}, host behavior sequences and that of AXI read sequences. We also plan to put up Spike /Imperas sequences which will be called through that of DPI Imports.
- **meep:** Pointer to all the *Meep Design RTL* files.
- **vseqs:** This has the pointer to the System level base virtual sequences. All the other virtual sequences will extend the base virtual sequence {NoC, AXI}.
- **tests:** Pointer to all the ACME System Level tests. This folder will be organised into several sub folders containing that of axi tests, NoC tests, spike and vpu tests. This folder will also have reference to that of the meep system level base test case from which all the

other test cases are derived. These test cases are encapsulated into a test package which will be imported to that of the testbench top.

- **vendor_lib:** Pointer to that of the Google riscv-dv utility with its base instruction and derived classes for generating randomly generated assembly code to exercise arithmetic instructions, vector loads and stores, interrupts. This in future will support v0.10 RISC-V Vector instruction set.

- **env:** This folder will have testbench logic for the System Level environment where all the individual block level environments will be instantiated and all the block level UVC packages will be instantiated.

- **spike:** Pointer to that of the Spike compiled binaries.

- **lib:** This folder will have any libraries related to Spike /Imperas. Plan is also to use this folder to add a testbench scoreboard which would be instantiated in that environment.

### 12.3.1. System Level Test Bench Next steps

Some of the test bench enhancements for ACME test bench which need to be done are the following

- Add the concept of Virtual sequencers.
- Replace bind constructs with hierarchical assignments in the testbench top.
- Add testbench debug and display messages via uvm_report_info.
- Add transaction printing utility.
- Add defines {`ifndef and `define statements] to all UVM classes [Objects and UVM components].
- The number of agents to be instantiated to be made a function of the number of design cores.
- Add a functional coverage model for the testbench via addition of cover groups and cover points in the monitor.
- Add assertions at the OVI Interface connecting the Systolic Arrays.
- Move run tasks from the Interface to that of the uvm_components.
- Set of all Virtual Interfaces in the testbench at the testbench top.
- Code up System Level Defines file.
- Code up tests, which extend base tests.
- Code up virtual sequences which extend base virtual sequence.
- Makefile updates.
- Resolving testbench compile errors.

# 13. Infrastructure

The Verification team will adapt the activities to the level of maturity of the team's developments, and also to the progress status of the ACME accelerator. The run of the different sets of tests can be executed in different places. Currently, the MEEP project has four servers to work with: satu, nanu, picu and femu. All of them have the same characteristics (Table 11), and each one has an AlveoU280 FPGA. These are single-thread performance boxes that overclock the CPU to do fast

FPGA and simulation runs. One of these machines has been tagged for *Production*, whereas the other three for *Development*. These machines are well-suited for executing light-weight workloads within the DV cycles. However, regression tests introduce a significant computational workload, and other BSC servers could be used to distribute the computation; such as nodes in Nord 3 (Table 12), and MareNostrum 4 (Table 13).

| Component | Characteristics |
|---|---|
| **MEEP servers (Phase 1)** | |
| CPU | AMD Ryzen 7 3800X 3.9 GHz AM4<br>(8 cores/16 threads) |
| Operating System | Ubuntu 20.04.1 LTS |
| Mainboard | Gigabyte X570 AORUS ELITE |
| Memory | DDR4 4*32GiB 3200 MHz 64 bits (Kingston HyperX Fury) |
| VGA | NVidia GT1030-SL-2G-BRK (Asus) |
| HDD | 500GB SSD M.2 PCIe NVME Western Digital SN750 & 4TB Toshiba 3.5” SATA |
| **MEEP servers (Phase 2)** | |
| System | 2 Racks, with 6 servers each one [B2] |
| CPU | AMD EPYC™ 7002 series processor family<br>Dual processors<br>(Up to 64-core, 128 threads per processor) |
| Operating System | Ubuntu 20.04.1 LTS |
| Mainboard | MZ52 - G41 |
| Memory | 32 x DIMM slots<br>DDR4 memory supported only<br>8-Channel memory per processor architecture<br>RDIMM modules up to 128GB supported<br>LRDIMM modules up to 128GB supported<br>Memory speed: Up to 3200*/ 2933 MHz |
| VGA | Integrated in Aspeed® AST2500<br>2D Video Graphic Adapter with PCIe bus interface<br>1920x1200@60Hz 32bpp, DDR4 SDRAM |
| HDD | 2 x 2.5" NVMe, 8 x 2.5" SATA hot-swappable HDD/SSD bays (non-supported SAS devices) |

*Table 11. Computational resources characterization available for DV activities (MEEP servers)*

| Nord3 |
|---|
| Nord III [B3] is a supercomputer based on Intel SandyBridge processors, iDataPlex Compute Racks, a Linux Operating System and an Infiniband interconnection. The current Peak Performance is 251,6 Teraflops. The total number of processors is 12,096 Intel<br>SandyBridge-EP E5–2670 cores at 2.6 GHz (756 compute nodes) with at least 24.2 TB of main memory.<br>● 9 iDataPlex compute racks. Each one composed of:<br>○ 84 IBM dx360 M4 compute nodes. Each of these nodes contains:<br>■ 2x E5–2670 SandyBridge-EP 2.6GHz cache 20MB 8-core<br>■ 500GB 7200 rpm SATA II local HDD<br>■ One of the following RAM configurations:<br>● Default nodes: 32 GB/node<br>● Medium memory nodes: 64 GB/node<br>● High memory nodes: 128 GB/node<br>○ 4 Mellanox 36-port Managed FDR10 IB Switches<br>○ 2 BNT RackSwitch G8052F (Management Network)<br>○ 2 BNT RackSwitch G8052F (GPFS Network)<br>○ 4 Power Distribution Units |

*Table 12. Computational resources characterization available for DV activities (Nord 3)*

| MareNostrum4 |
|---|
| MareNostrum4 is a supercomputer based on Intel Xeon Platinum processors from the Skylake generation [B4]. It is a Lenovo system composed of SD530 Compute Racks, an Intel Omni-Path high performance network interconnect and running SuSE Linux Enterprise Server as operating system. Its current Linpack Rmax Performance is 6.2272 Petaflops.<br>This general-purpose block consists of 48 racks housing 3456 nodes with a grand total of 165,888 processor cores and 390 Terabytes of main memory. Compute nodes are equipped with:<br><br>● 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each @ 2.10GHz for a total of 48 cores per node<br>● L1d 32K; L1i cache 32K; L2 cache 1024K; L3 cache 33792K<br>● 96 GB of main memory 1.880 GB/core, 12x 8GB 2667Mhz DIMM (216 nodes high memory, 10368 cores with 7.928 GB/core)<br>● 100 Gbit/s Intel Omni-Path HFI Silicon 100 Series PCI-E adapter<br>● 10 Gbit Ethernet<br>● 200 GB local SSD available as temporary storage during jobs<br>● The processors support well-known vectorization instructions such as SSE, AVX up to AVX–512 |

*Table 13. Computational resources characterization available for DV activities (MareNostrum 4)*

In addition, these servers have installed all the software tools the different MEEP teams need to develop their activities (Table 14).

| Activity | Tool | Vendor Provider |
|---|---|---|
| Simulation and Debug | Questa Core Prime 2019.4 | Mentor Graphics (Siemens) |
| Industry Verification IP | Questa VIP AMBA Family App SW | |
| Risc-v compilation | risc-v toolchain | RISC-V Open Source |
| Instruction Set Simulator (Reference model) | Spike | |
| | M*DEV | Imperas |
| RISC-V instruction generator | riscv-dv | Open Source tool developed by Google |
| FPGA synthesis | Vivado 2020.1 | Xilinx |

*Table 14. List of tools used for verification activities shown below*

## 13.1. Continuous Integration and Continuous Design (CI/CD)

By following standard industry practices, another feature to be exploited from the Gitlab repository is the continuous integration and continuous design/deployment (CI/CD) with different types of automated testing. The DV team will use continuous methodologies with the aim of facilitating the development, testing/verification of the RTL design and will use this to provide feedback on the health of testbench and design. The CI/CD will also allow reducing human error injection along the different stages of the verification cycle, and contributing to the reliability and repeatability of the whole verification process.

The first implementation of the CI/CD relies on the current MEEP infrastructure, depicted in Figure B.34. The CI/CD flow will mature and evolve with the MEEP project with the integration of Jenkins [B6] to control and manage the execution of parallel jobs; including utilizing BSC supercomputer resources by distributing the regression and other types of runs onto different nodes in MareNostrum4.

All team members get access to the BSC infrastructures, software applications, and servers through a VPN, and their own credentials. The Gitlab repository is hosted in BSC (*gitlab.bsc.es*), where the source code and triggers are monitored, but the execution of all the simulations will be run in BSC/MEEP servers. An example of one of the implemented CI/CD flows is shown in Figure 53.

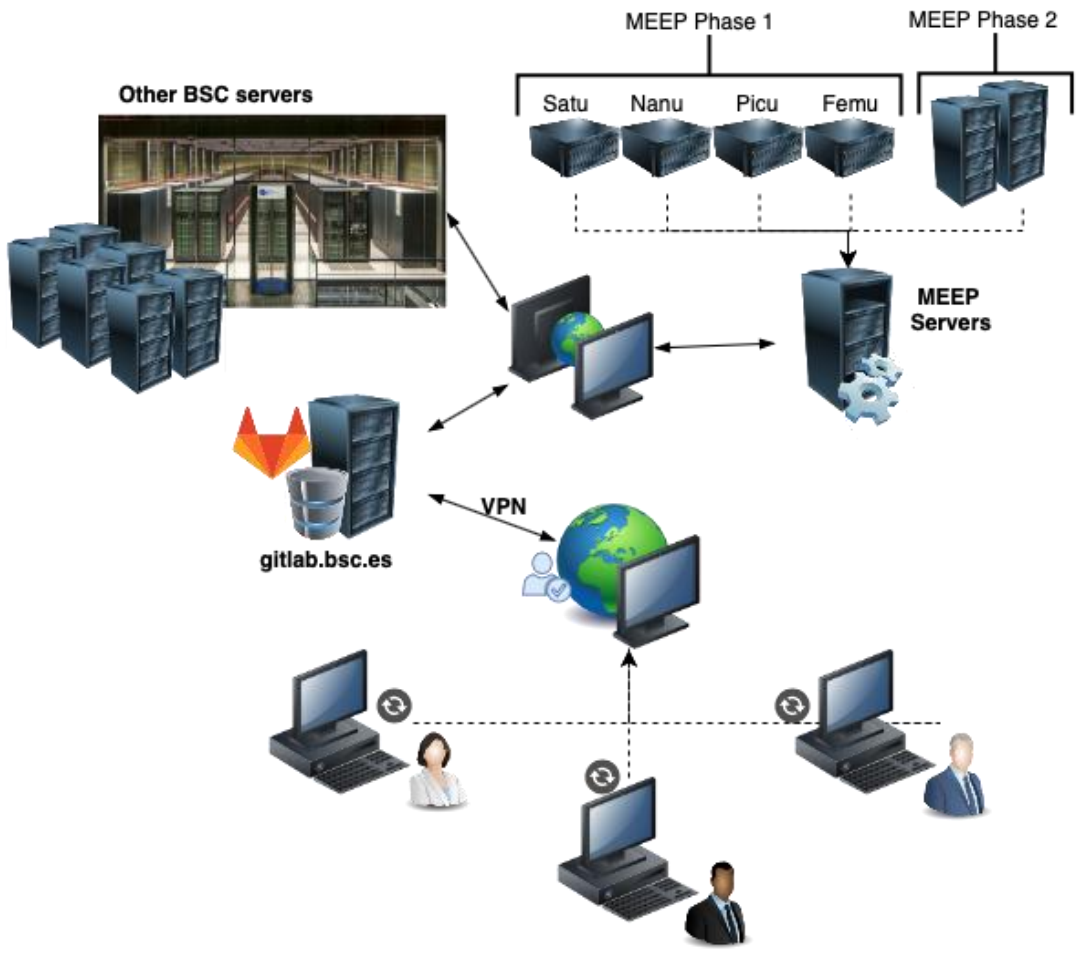



*Figure 53. Schematic of MEEP infrastructure*

The intent of CI/CD regressions is to launch the regression tests on various design blocks both at the system level and the IP level [VPU, VAS Tile core, VAS Tile, and up to ACME System Level Verification]. This mechanism is exercised when there are major releases to that of design code or testbench code or both of them. The intent of CI/CD is to test these code release scenarios and to determine the health of the delivered verification and design code. The CI/CD infrastructure is also conceived to run smoke/sanity tests whenever there are changes to testbench code from a verification perspective. Figure 54 shows the three possible status of CI/CD pipeline; cancelled, failure and passed. The failure obtained in CI/CD happens when not all the proposed test cases have been executed successfully, which implies that some of the tests have issues either in testbench code or some of them are genuine design bugs. CI/CD is used when code merges are done from feature specific branches, development branches to that of the master branch. This system is used to kick off tests to ensure that code merges do not result in design and testbench compile errors.

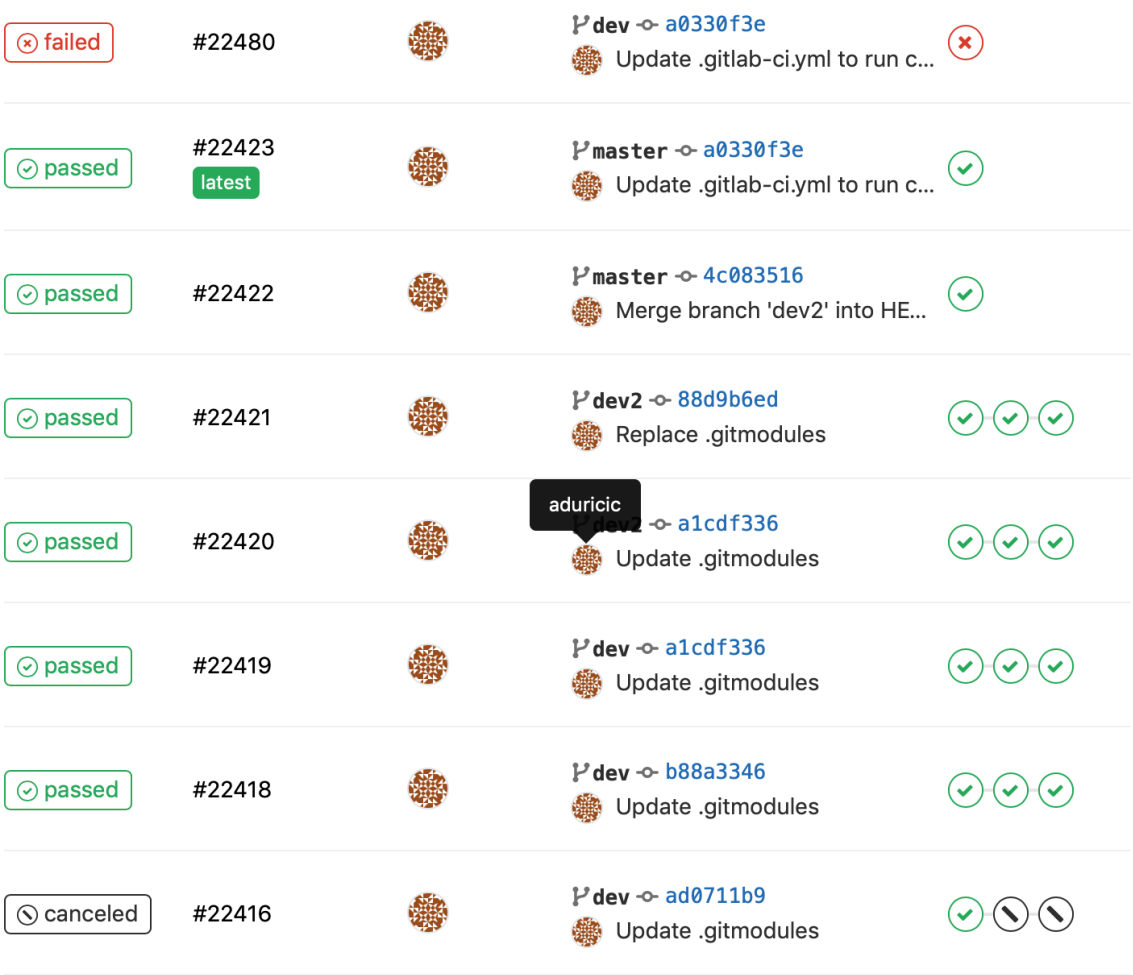



*Figure 54. Results of different runs by gitlab CI/CD feature*

## 13.2. Source code repository

All the source code is stored in a common repository. The MEEP project is using Gitlab as a collaborative source code version control tool. Thus, all team members can contribute to different projects and also keep the code updated and available for others.

The MEEP repository is organized into groups, and the specific working group for Verification purposes is *MEEP Design Verification*. Within that group, each kind of the current set of DUT's has its own verification project. More projects will be added along the time, in order to cover all the design hierarchy levels up to the SoC level. Figure 55 shows the five current source-code projects on-going within the DV team:

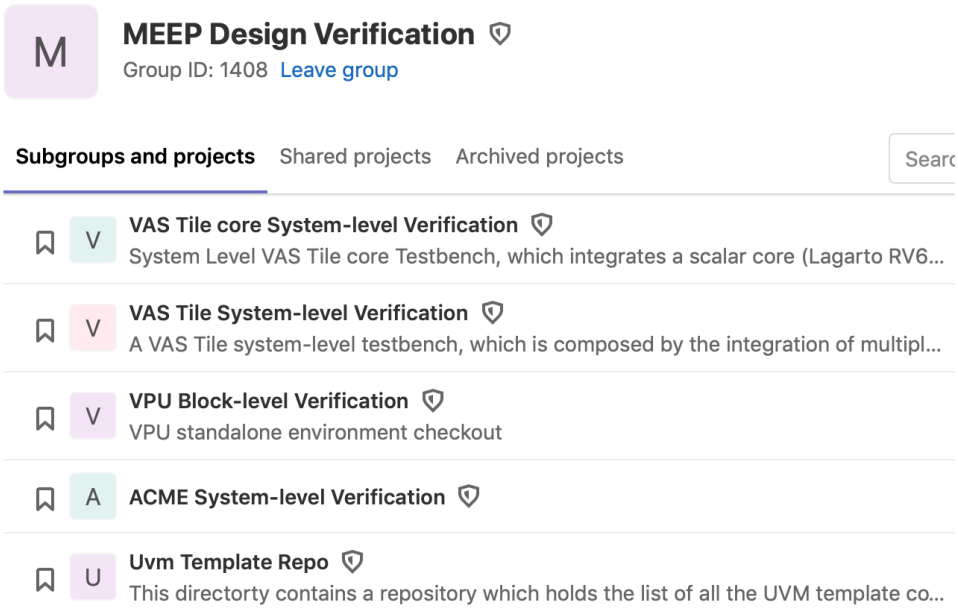



Figure 55. Current list of projects in the Verification source code repository

- VPU Block-level Verification: it is in charge of the VPU verification.
- VAS Tile Core System-level Verification: it is in charge of the VAS Tile core verification.
- VAS Tile System-level Verification: This project refers to the first approach to the VAS Tile with the OpenPiton System in which an in-order processor is used as a core. Later on, the core will evolve towards a VAS Tile Core, with a scalar core, and three co-processors (one vector processing unit, and two specialized systolic arrays).
- UVM testbench patterns: *UVM template Repo* project includes the available templates for developing a UVM environment. The DV team will continue working on this to add new components to cover the whole spectrum of possibilities required by the ACME accelerator architecture.
- ACME System-level Verification: It is focused on the interactions between the ACME accelerator and the MEEP Shell through AXI4 and AXI Lite protocols. This is the system level implementation of DVINO-ACME, which also includes NoC agents once these specifications are outlined. This will also include other sub-modules like Microengines, VPU, MCPU and also the Systolic array implementations which in due course will also include Systolic arrays for image processing and that of neural nets. This system level verification will focus on a multi-core setup.
    - VAS Tile core repository with DVINO: "*dvino_system_level_verif*" - https://gitlab.bsc.es/meep/meep-design-verification/dvino_system_level_verif

| M18 DELIVERABLES | |
|---|---|
| **IP** | **URL** |
| VPU | https://gitlab.bsc.es/meep/meep-design-verification/vpu-standalone-verification |
| VAS Tile core | AXI interfaces: "*Axi block level verification*": https://gitlab.bsc.es/meep/meep-design-verification/axi-block-level-verification/-/tree/dev |
| VAS Tile | https://gitlab.bsc.es/meep/meep-design-verification/lagaro_modified/-/tree/dev |
| ACME | https://gitlab.bsc.es/meep/meep-design-verification/dvino_system_level_verif |
| UVM model | Driver sequencer communication: https://gitlab.bsc.es/meep/meep-design-verification/uvm_template_repo/-/tree/dev/1_driver_sequencer_comunication%2FRAM_Verification_In_UVM |
| | Code_Coverage_Multiple_testcases: https://gitlab.bsc.es/meep/meep-design-verification/uvm_template_repo/-/tree/dev/2_CODE_COVERAGE_multiple_test_case%2FRAM_Verification_In_UVM2 |
| | Added_Configurable_database (Config_DB): https://gitlab.bsc.es/meep/meep-design-verification/uvm_template_repo/-/tree/dev/3_WITH_config_DB%2FRAM_Verification_In_UVM2 |
| | With_Virtual_Sequencer: https://gitlab.bsc.es/meep/meep-design-verification/uvm_template_repo/-/tree/dev/4_virtual_sequencer%2Fvirtual_sequencer%2FRAM_Verification_In_UVM2 |

*Table 15. Gitlab Design Verification repository*

# 14. Conclusions

Verification team have accomplished the following items so far:

- Established a methodology to verify standalone VPU releases from the design team.
- UVM testbench in place to verify VAS Tile Core Designs [Core + VPU + AXI Interconnect] along with Spike integrations.
- Generated RISC-V Tests to test the VAS Tile Multi-Core design.
- Defined a Verification strategy and integration plan for that of the ACME System Level Verification.
- Designed an ACME testbench architecture along with coding in progress to meet System Level Verification goals.

# CHAPTER 3: Performance Modelling with Coyote Simulator: cores, NoC and memories

## 15. Summary

This chapter, as part of D4.2, refers to D4.2. *Part C Performance Modelling with Coyote simulator: core, NoC and memories*. Due to the research nature of the ACME accelerator, all results provided by Coyote contribute to the substantiation of architectural decisions, the refinement of the design, and also the exploration of the impact of applying different policies, strategies, etc, on the performance of the whole accelerator. The primary purpose of this document is to describe the details and features offered by Coyote at this point (M18). In addition, some experiments have been conducted in Coyote by modeling the primary components of the ACME accelerator, and the results have been analyzed to provide some insights of the behavior of the accelerator.

The rest of the document is structured into five sections, starting with Section 2 by contextualizing Coyote as an execution-driven simulator for RISC-V architectures within the MEEP project. This section also presents how Coyote affects the development of the proposed accelerator (ACME), and enables this document to stand on its own.

Then, Section 3 focuses the content on Coyote itself, by providing a detailed characterization of the simulator. The section explains the additional features that have been added to the simulator by taking the status of M9 as a reference. Thus, it is possible to have an unobstructed view of all the progress done along these nine months.

Section 4 focuses on the analysis of the results of simulations, which have been executed in order to refine models for the ACME accelerator and also to estimate the impact of experimental features implemented in the Coyote simulator. The purpose of these experiments is threefold: (1) exploiting the new capabilities of the simulator, (2) supporting architectural decisions, and (3) collecting insights about the architectural behavior of ACME that might sustain and guide the RTL development tasks. The experiments are focused on the three dominant groups of components of the architecture of the accelerator: the computational engine, the memory engine, and the communication among both kinds of engines.

Finally, as a conclusion, Section 5 summarizes the outcomes and achievements at mid-term of the project (M18). It also provides some ideas about follow-up activities.

## 16. Introduction

As a pre-silicon validation platform of IPs, MEEP provides a set of tools to explore, test and refine new ideas at different development stages of a new hardware architecture. Earlier stages require faster and lower fidelity tools that enable design analysis and also comparisons between unique design aspects within a reasonable time. At this point is where Coyote comes into play.

As introduced in D4.1 section 4.3.3, Coyote is a new execution-driven, event-based simulator for RISC-V, built using two open source tools: an instruction set simulator (Spike), and a modeling framework (Sparta). The former offers simulation capabilities compliant with RISC-V (support for multi-core systems and vector extensions). The latter provides all modeling capabilities for supporting enough flexibility and adaptability to support the rapid evolution and needs for current and future systems, and being able to explore their design space. Coyote's aim is to reuse and exploit the strengths of these two tools, and at the same time to overcome all their weaknesses, bridging the gap between flexibility and high-throughput simulations in modeling HPC-targeted architectures. In this sense, Coyote needs to capture the common needs related to HPC architectures, such as high core counts and complex memory hierarchies, and the way to get this is by following a data movement approach. This balances accuracy and simulation throughput, enabling the simulation of large-scale Exascale systems.

## 16.1. Coyote simulator in WP4

Coyote is the core of the performance modeling activities within WP4. For simplicity, from now on *Coyote* will be used for referring to the simulator, and also the performance modeling activities. As it is shown in Figure C.2, Coyote is tightly related to other two activities: ACME architecture, and its design and implementation. The third activity (*Design Verification*) is out of the scope of this document. More information can be found in the *D4.2 Part B Verification Strategy* chapter.

The ACME architecture provides the high-level specifications of the accelerator, whereas the RTL design is responsible for designing and implementing the low-level details of the accelerator, being sure that the final design complies with the architectural specifications.

Before Coyote can start working, it needs to receive the specifications of the ACME accelerator, since they are used to model its underlying architecture. Once Coyote produces results, these are disseminated to both teams (Architecture and RTL). On the one hand, results obtained using Coyote might help to substantiate or reconsider architectural decisions. In any case, the information permits enhancing the architecture in early stages of the digital design cycle. On the other hand, the analysis of the design space exploration of ACME at simulation level provides clues and insights about the behavior of the architecture to the RTL design.

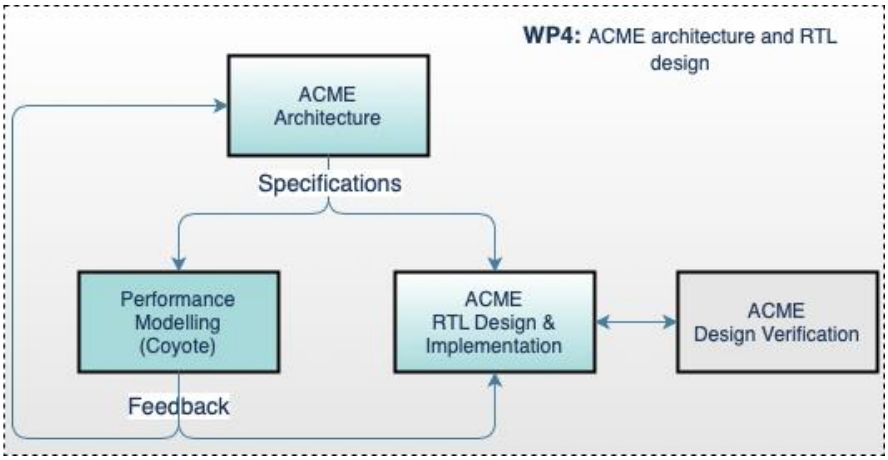



*Figure 56. Relationship between Coyote and other tasks developed in WP4*

Although the basic flow depicted in Figure 56 is correct, it requires Coyote to be already developed and in a stable and mature stage. That is not the case, since that is precisely one of the expected outcomes for the MEEP project. As a result, a more accurate workflow has to consider the development status of the Coyote simulator itself. It is important to be aware that the extension of the features of Coyote, as of any simulator, is an endless process, which feeds off new architectural requirements and its simulation results. This is considered in Figure 57.

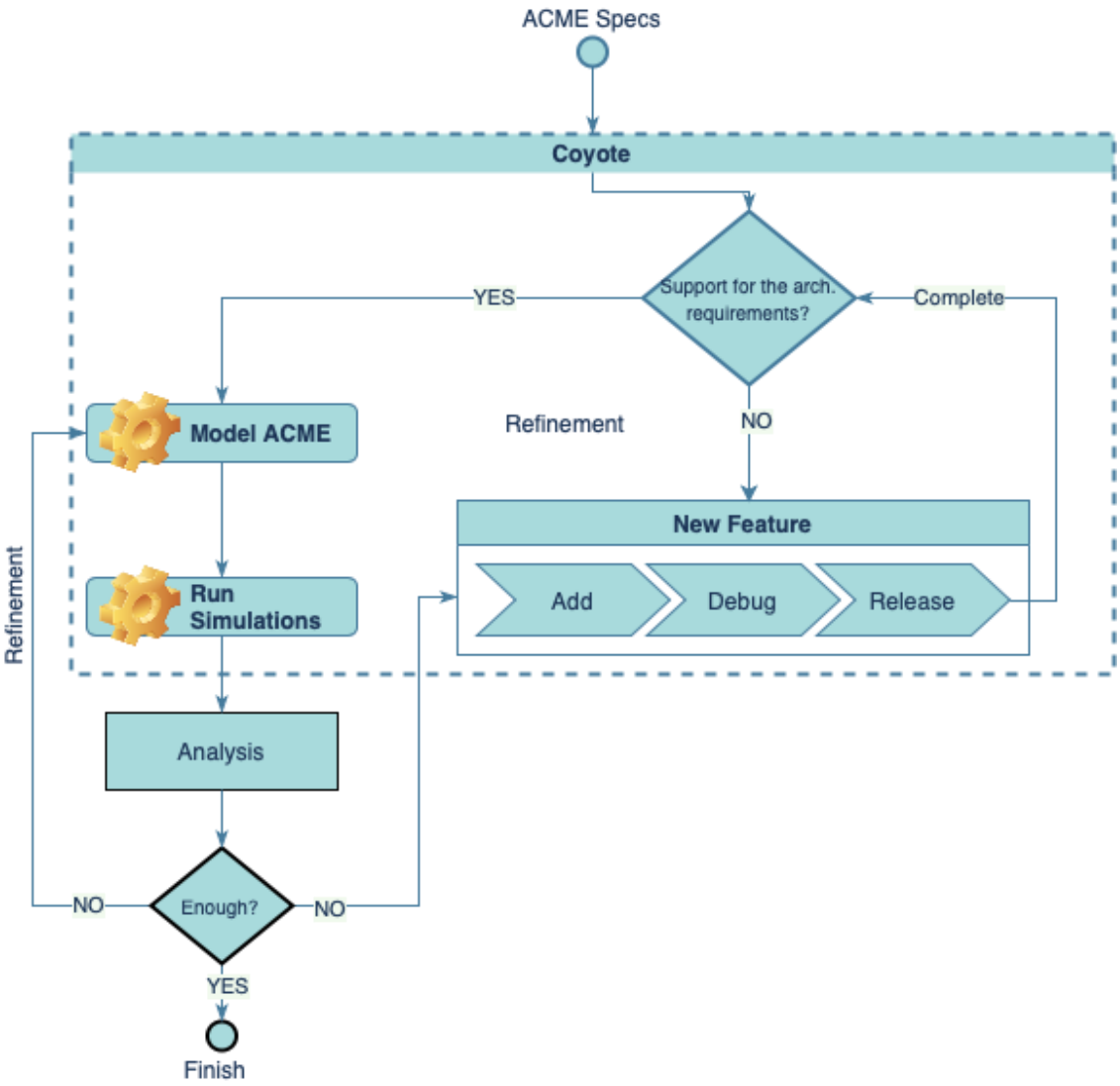



*Figure 57. Basic Coyote's development workflow*

The workflow shown in Figure 57 combines the flow for analyzing an architectural model (left side of the Figure) with the internal development process of Coyote (*New Feature*). This is a more accurate flow for Coyote, considering its level of maturity and evolutionary process. This implies that to support many ACME specifications, Coyote needs to extend its capabilities. Once a required feature is included into the catalogue of Coyote, a more refined model of ACME can be implemented. After that, in accordance with the interests of the architects and designers, several simulations enable data analyses to uncover valuable information for all the parties involved, influencing the workflow. The analysis of the simulation results contributes to refine the architectural model, and to debug, improve, and extend Coyote.

## 16.2. Coyote release at mid-term project (M18)

According to the MEEP project proposal, at M18 the WP4 should have achieved its first milestone (MS1) consisting of *FPGA RTL revision 1 complete, with complete verification environment*. At time performance modeling activities were not considered, however first progress about it was shown in the previous deliverable D4.1. In this sense, this deliverable is focused on adding the required features to Coyote in order to be able to model the main components of the targeted architecture. As part of that goal, the performance modeling team has centered its effort on adding the required capabilities to Coyote to be able to model a network on chip (NoC). This is because of the relevance of this in ACME as a communication component, and its expected impact on the performance of the accelerator. Moreover, adding new features to Coyote helps to get more visibility on how the targeted ACME architecture is suitable for improving the efficiency of compute and memory bound applications. An initial benchmark composed of 5 different well-known kernels in HPC, has been selected to run the simulations. Some of these kernels are characterized as dense, others as sparse from the data memory access pattern perspective, which will contribute to check the performance improvements associated with the ACME architecture.

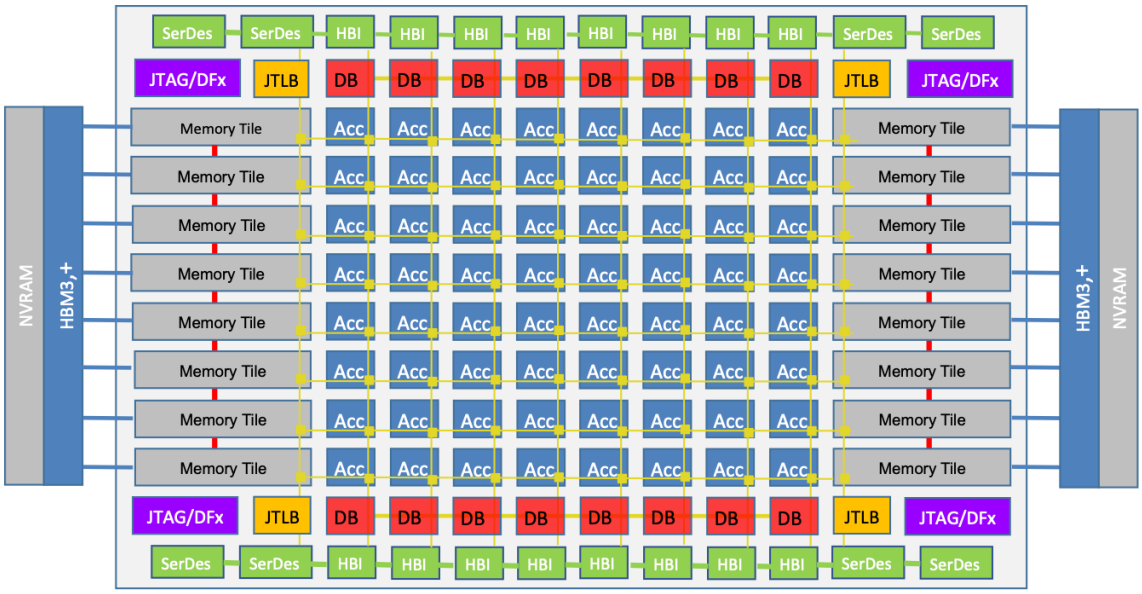



*Figure 58. High-level view of the ACME accelerator architecture*

From a top-level view, the ACME architecture comprises multiple instantiations of three key components: VAS Tile (VAS in Figure 59), Memory Tile and communication among them (green and red lines across the accelerator). The grid of VAS Tiles composes the computational engine of the accelerator, whereas all the Memory Tiles together build the memory engine. The mechanism to establish the communication among the different components is a Network on Chip. All these pieces play an important role in ACME, since depending on the characteristics of the HPC applications to be executed on the accelerator, the focus is on different components.

Such a coarse level of simulation is feasible, but reliable reasoning regarding ACME requires greater detail. For this reason, Coyote works one level deeper, considering the functionality of some of the internal pieces of both VAS Tiles (Figure 59), Memory Tiles (Figure 60), for simulation based on the movement of data throughout each of the components.

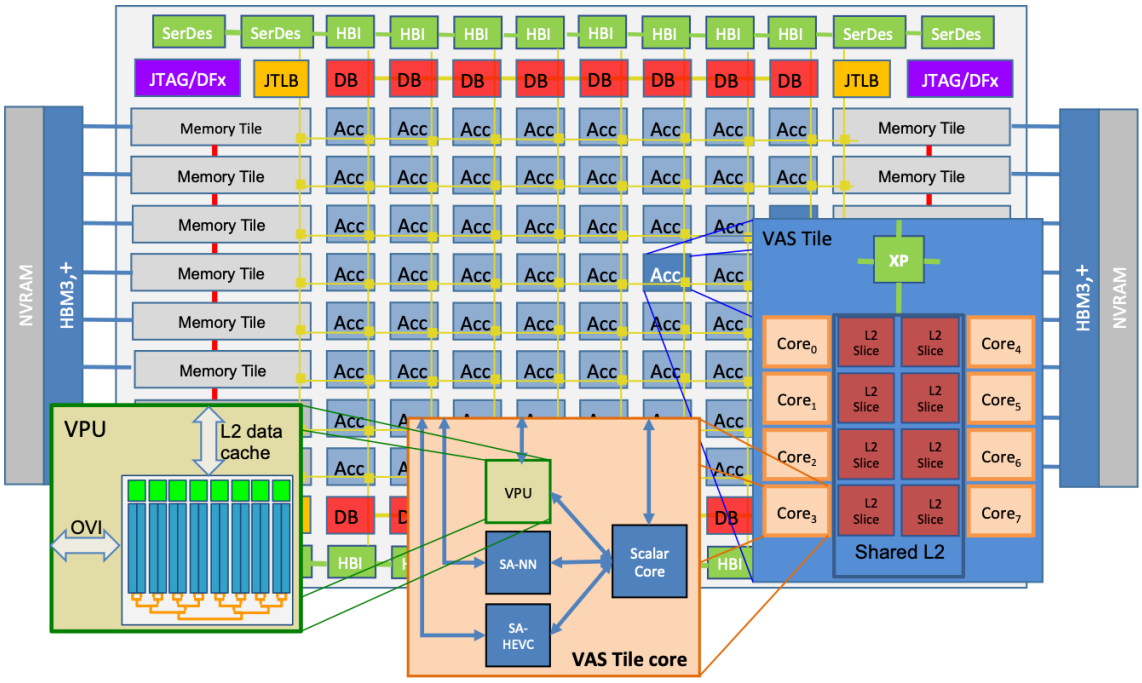



*Figure 59. Top-down view of VAS Tile component, VAS Tile core and the VPU*

As presented in Figure 59, a VAS Tile component in ACME is a multi-core system with a shared banked L2 data cache (red box in the Figure). Each of the cores is composed of a scalar core and three specialized co-processors (orange box in the Figure): one vector processor unit (VPU) to execute vector instructions (yellow box in the Figure), and two systolic arrays (SAs), one for neural networks (SA-NN) and one for video processing (SA-HEVC). Accurately modeling the internal behavior of each and every of these individual components would defeat the purpose of Coyote itself: fast simulation of exascale systems. For this reason, some elements are only functionally modeled. For example, Coyote does not implement a detailed model for the VPU, but it is able to execute vector instructions at VAS Tile level.

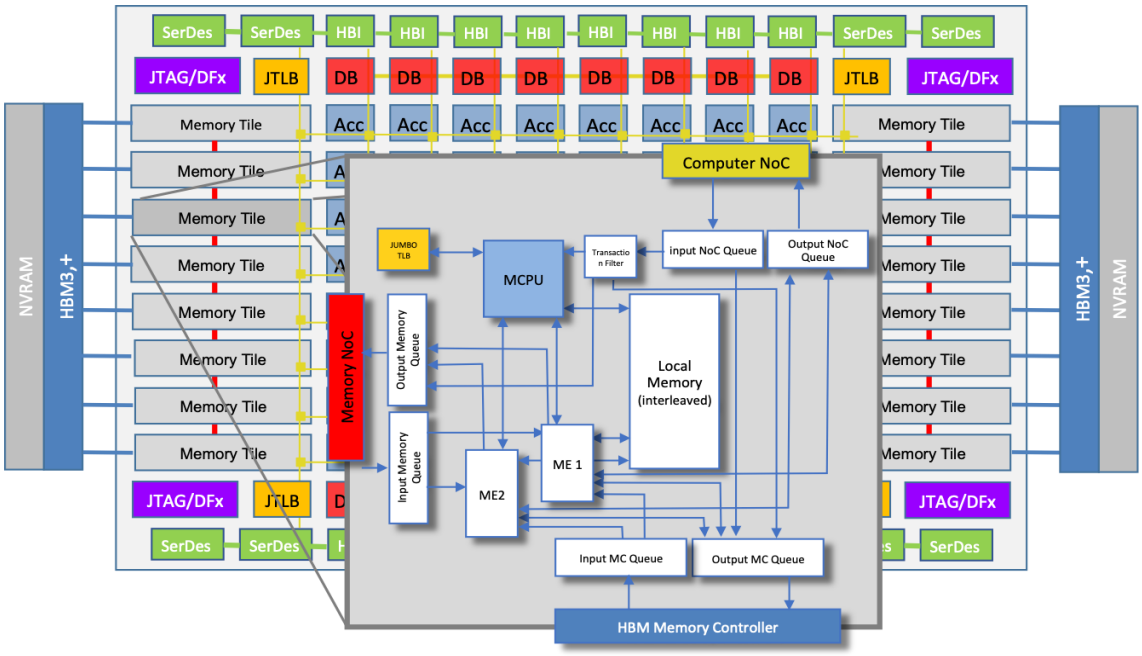



*Figure 60. Schematic of an internal structure of a Memory Tile*

Another example is the Memory Tile. It is a complex component built by several modules. The memories, the MCPU and the NoCs' routers will be modeled per Memory Tile. In Figure 60, the *Compute NoC* represents the router used by the NoC to communicate a VAS Tile with a MCPU. The *Memory NoC* represents the connection between different Memory Tiles. At this point in time, the Memory NoC is not modelled in Coyote, and consequently all kind of communications between VAS Tiles and/or Memory Tiles is done throughout the Compute NoC.

The way Coyote has evolved along these months is summarized in Table 16, using M9 as a starting point with the release v0.1. From that time new features have been incorporated to the simulator, until getting the current stable release at M18 v0.4.

| Feature | Release 0.1 | Release 0.2 | Release 0.3 | Release 0.4 |
|---|---|---|---|---|
| Multicore | Supported | Supported | Supported | Supported |
| Multithreading | | | | Supported |
| Vector extension | Supported | Supported | Supported | Supported |
| EPI Intrinsics | Supported | Supported | Supported | Supported |
| RAW Dependencies | Supported | Supported | Supported | Supported |
| Banked L2 | Supported | Supported | Supported | Supported |
| L2 Data Mapping Policies (P2B, SI) | Supported | Supported | Supported | Supported |
| Tracing | Supported | Supported | Supported | Supported |
| Tiles | | Supported | Supported | Supported |
| Notion of Memory Controllers | | Supported | Supported | Supported |
| Functional NoC | | Supported | Supported | Supported |
| Detailed Memory Controllers | | | Supported | Supported |
| Simple NoC | | | | Supported |
| Detailed NoC | | | | Supported |
| MCPUs | | | | Expected |
| Instruction specific latency | | | | Supported |

*Table 16. Coyote modeling capabilities*

At M9 Coyote could model an architecture with a private L1 cache (data and instruction) and a shared banked L2 cache. The simulator allowed to conFigure the size and the associativity of the L2 cache, and also the number of in-flight misses and the hit/miss latencies. Regarding the data mapping policies, Coyote supported page to bank (P2B) and set interleaving (SI). This status corresponds to the release version 0.1 in Table 16.

The rest of the document provides more details about the latest release v0.4 and the simulation experiments run by exploiting those characteristics to model ACME accelerator.

All the code of Coyote can be found in the MEEP Performance Modelling Gitlab repository: https://gitlab.bsc.es/meep/meep-performance-modelling/coyote-tool. This repo is structured into four subprojects, as depicted in Figure 61.

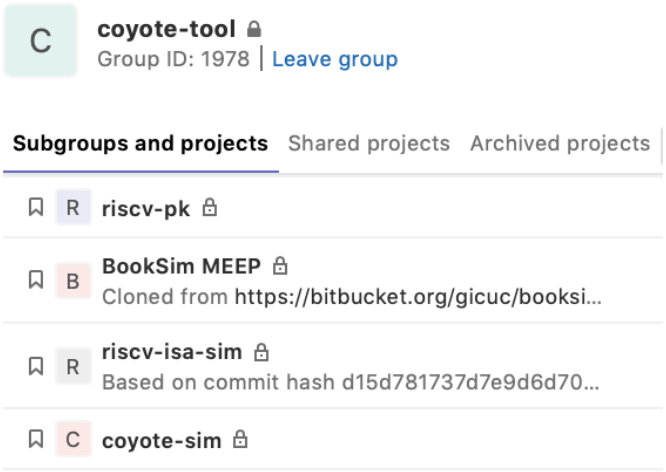



*Figure 61. Subprojects as part of Coyote's simulator*

Each of the subprojects serve a different purpose within Coyote, which is described below:

- **riscv-pk:** A development mirror of the official riscv proxy kernel repo [C5] to adapt it to the needs of Coyote. This is related to the task presented in Section 3.2.
- **BookSim MEEP**: An extension of the Booksim implementation available in [C14]. This subproject provides support for the detailed NoC model presented in Section 3.1.3.1.3.
- **riscv-isa-sim**: An extension for the Spike simulator [C15] to add the functional simulation features required by Coyote, such as its overall interaction with Sparta, the tracking of RAW dependencies or the latencies for instructions. The new features for M18 related to this subproject are introduced in Section 3.1.1.
- **coyote-sim**: The overall project for the Coyote simulator, which includes the Sparta based model itself and puts together all the other repos as submodules. This is involved in one way or another in all the new features of Section 3 and also with the visualizations in Section 4.3.

## 16.3. Definitions and notations

Table 17 provides a definition for the most used concepts about ACME handled over the document.

| Concept | Definition |
|---|---|
| ACME | Accelerated Compute and Memory Engine accelerator, composed of VAS and Memory Tiles interconnected by an NoC. It is a self-hosted and disaggregated architecture. |
| Tile | A key component with a specific function inside the ACME architecture. There are two different types in the targeted accelerator: VAS Tile (highly utilized for performing compute-bound operations), and Memory Tile (highly utilized for performing memory-bound operations). |
| VAS Tile | Vector and Systolic Array Tile defined as a multi-core system, with shared L2 data cache memory and NoC interfaces to interconnect with other VAS Tiles and Memory Tiles.<br>Within a VAS Tile there are up to 8 VAS Tile cores. |
| VAS Tile core | It is an IP comprising a RISC-V scalar core with three decoupled co-processors: one VPU |

| | |
|---|---|
| | and two specialized SAs. As part of the ACME disaggregated architecture, the VAS Tile core executes scalar and arithmetic vector instructions. |
| VPU | Vector Processing Unit suitable for executing RISC-V ISA vector extension instructions. It has two kinds of interfaces: to the scalar core and to the L2 data cache/scratchpad. |
| SA | Systolic Array accelerator. ACME includes two specialized designs; one for processing neural networks (SA-NN), and another for HEVC video processing (SA-HEVC). These accelerators have the same interfaces as the VPU. |
| Memory Tile | A key component including the Memory Controller CPU (MCPU), Shared L3/Row Buffer, L2 Instruction Cache, TLB, and NoC interfaces to accelerate memory operations. |
| MCPU | Memory Controller CPU. As part of the ACME disaggregated architecture, the MCPU executes the memory and atomic operations. |
| Hart | Hardware thread. |
| NoC | Network on Chip. Network inside the accelerator for interconnecting different Tiles. |
| Flit | A flow control unit, also known as flow control digit [C11], is the atomic piece of information recognized by the flow control mechanism. |
| RAW | Read After Write dependency. One of the input registers used by an instruction is not yet ready. The instruction may not be executed until the instruction producing this register has been completed. |

*Table 17. Definition of some concepts related to ACME*

# 17. Coyote status at M18 (release v0.4)

Deliverable 4.1 presented the initial features and supported applications in Coyote as of M9. This section covers the additional features that have been implemented to more closely model ACME.

## 17.1. New features in Coyote

The release of Coyote introduced in M9, v0.1 in Table 16, featured the modeling of simple tiled systems, including their L2s, implementing different sharing modes and data mapping policies, scalar cores and VPUs (only one VAS Tile - VAS in Figure 62).

Figures 62 and 63 graphically highlight the status regarding the features supported in M9 and M18 respectively. In both Figures, the shadowed boxes represent the unsupported architectural components.

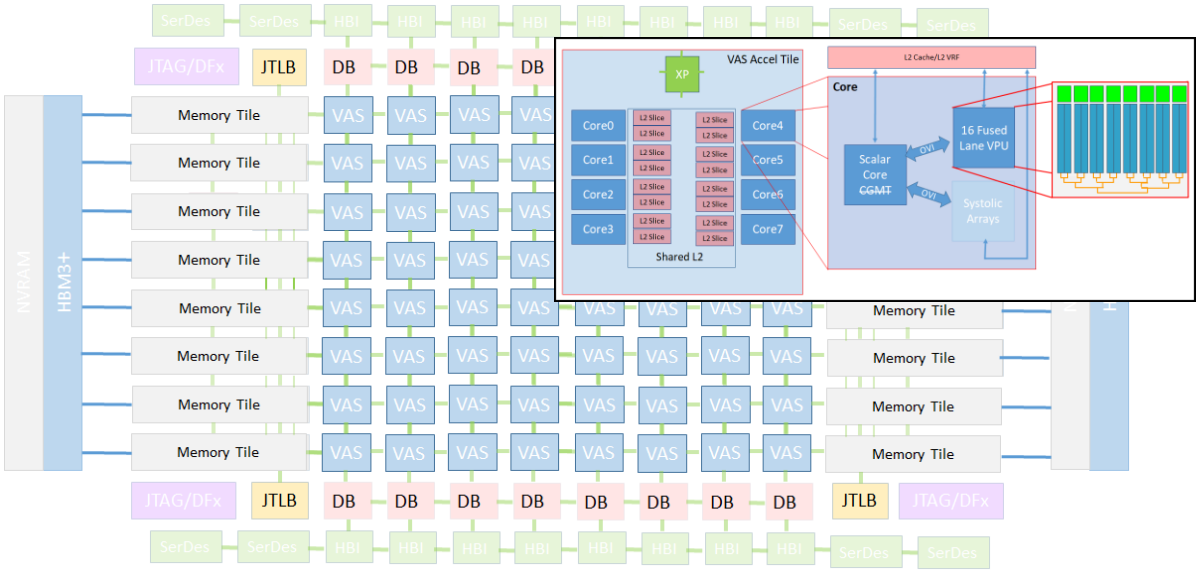



*Figure 62. Representation of the features supported in M9 (Greyed out for unsupported)*


*Figure 63. Representation of the features supported in M18 (Greyed out for unsupported)*

A comparison between Figures 62 and 63 shows that in fact several additional features have been added to enable a more accurate modeling of exascale systems as part of this M18. These can be grouped into 3 key areas, which match the most relevant components in ACME: the VAS Tile, the Memory Tile and the Network-On-Chip interconnecting them.

### 17.1.1. VAS Tile

VAS Tiles represent the main computing element in ACME, integrating scalar cores, and computational capabilities such as VPUs or systolic arrays and also the lower levels of the memory hierarchy. Spike provides the functional simulation of the RISC-V architecture in these models, while data movement and the passage of time is modeled using Sparta. The following new features have been implemented via an interplay between these two elements.

#### 17.1.1.1. Coarse Grain Multithreading

Multithreading is a technology in which several hardware threads can run concurrently within the same core to increase the utilization of the resources by using thread level parallelism. In general, resources like execution units and L1 caches are shared, while the registers are private to each thread. In coarse grain multithreading, the thread in a core does a context switch when it encounters a cache miss or some other delaying event or a fixed number of execution cycles have elapsed.

The scalar core in the VAS tile is a coarse grain multithreaded (CGMT) core. It is important to simulate this technique in Coyote, as it can provide important insights for both memory bound applications (in which case more threads run in a core) and compute bound applications (in which fewer threads exist).

Coyote implements the CGMT policy where a thread in a core keeps running until it encounters either a cache miss, a barrier, or a RAW dependency. The next active thread is scheduled to run based on a round-robin policy after some fixed number of cycles are elapsed to account for thread switching overhead.

The set of applications that Coyote is targeting uses barriers for synchronization among all the participating threads. Implementing the barrier uses a lock (which protects the variable named **cnt** that stores the count of the threads that have reached the barrier) and a spin loop. The thread that reaches the barrier early atomically increments the **cnt** and spin-waits for the slowest thread to reach the barrier.

Since there is no way for Coyote to differentiate between the spin-loop and the loops which are part of the algorithm that the application implements, Coyote never realizes that the execution of the barrier has started. As a result, an unmodified Coyote keeps executing the spin loop infinitely and does not schedule the next active thread. In order to solve this issue, the barrier implementation has been replaced with a new instruction named **coyote_barrier,** which acts like a barrier marker for Coyote. Once this instruction is encountered, Coyote either stalls the executing thread (if it was one of the fast threads that reach the barrier early), or marks all the threads as active (if it was the last thread to reach the barrier). This effectively mimics the functional behavior of a barrier but, for the moment, disregards its impact on performance. Accurately modeling is left for future work, after coherence support is added to Coyote.

To introduce the **coyote_barrier** instruction in Coyote, the base version of the Spike is changed to add a new encoding for the **coyote_barrier** instruction. A function that encapsulates this instruction is also in a header, so applications can easily invoke it.

#### 17.1.1.2. Instruction latencies

Since the base version of Spike is a functional simulator, it does not model the latencies of instructions. It assumes that every instruction executes in one cycle. However, in order to model timings accurately, Coyote needs to consider the latencies of different instructions.

We have taken the latencies from e6500 core [13] (a RISC based 2 issue Power PC architecture) reference manual as a baseline and improved upon it to match the characteristics of ACME and populated the instruction latency map, which maps the instruction name to its latency. The latencies in the table will be modified as necessary to track the progress of the ACME design

As part of instruction execution, Coyote looks up the instruction latency for an instruction in the latency table, and the instruction's destination register is marked as being available only after the passage of that amount of time.

If the current instruction reads the data from the register, which is not yet available, the core executing the current instruction stalls, and an instruction-latency event is generated, which gets triggered once the corresponding register becomes available. As part of the trigger, the core which was stalled earlier is notified. The core is made active, if all the registers (Integer, Float, Vector) needed for the stalled instruction are available.

Since the latencies of memory operations are unknown beforehand, it is assumed that the data accessed as a result of executing the memory instruction is a hit and the instruction-latency map is populated appropriately. At the time of the memory instruction execution, if the load causes a miss in the L1 cache, the latency of that memory instruction is set to infinity (since when the data will be available is not known). Once the data is available in the cache, the availability of the destination register is reset appropriately.

#### 17.1.1.3. Improved RAW Dependency tracking

Since the original version of Spike is a single-cycle functional simulator, it does not need to track Read After Write (RAW) dependencies. But to model timings accurately, Coyote needs this functionality. In the release of Coyote presented in M9, given the base Spike infrastructure, RAW dependencies of any instruction are calculated while the instruction is executing. But, since this is not the way register scoreboarding algorithms function, Coyote needed to perform some bookkeeping tasks to make the simulation behave correctly. For example, one of the bookkeeping tasks is to set the destination register of the current executing instruction (which has a RAW dependency on a previous load instruction) as not available, storing the availability status of the destination register in a data structure, and marking it as available when the previous load instruction has completed execution.

To make the design cleaner, in the current deliverable, Coyote changes the way RAW dependencies are tracked to more closely match the register scoreboarding algorithm. Coyote maintains a list of registers used by each instruction. Before starting the execution of an instruction, the availability of the registers used in that instruction is checked. Only if all the registers used in that instruction are available, the instruction is executed, and the program counter is incremented to the next instruction.

This change obviates the need to perform the book-keeping tasks mentioned earlier, produces easy-to-understand code, and drastically reduces the simulator execution time.

#### 17.1.1.4. MCPU supporting features

The Memory Controller CPU (MCPU) is part of the Memory Tile. When it is in operation, all vector memory operations encountered by the scalar core must be forwarded to an MCPU, which then reads/writes the data from/to the main memory in a way such that only the required data elements are communicated, while the data elements that are not required, are discarded. This is the case for non-unit stride memory operations, in which every n-th vector element is needed or indexed vector memory operations.

To introduce MCPU support in Coyote, the base version of Spike has been changed to communicate the Virtual Vector Length (VVL), and Application Vector Length (AVL) to the MCPU as part of vsetvl instruction execution. The implementation of different vector load/store instructions is also changed to encapsulate the details about the instruction in an object and communicate this to the MCPU so that the MCPU can intelligently optimize the memory operations. The encapsulated object contains information such as the addresses of the data elements, the type of memory operation (load/store) and the data access pattern (indexed, strided, unit-strided). Also, the changes related to L2-bypassing for vector memory operations have been introduced in the deliverable.

According to the MCPU design, it manages data movement and uses part of the L2 cache memory as a scratchpad, constituting an extension of the vector register file of the VPUs. The L2 already available in Coyote has been extended to support a basic scratchpad. It operates on a command basis, which allows the MCPU to allocate/deallocate scratchpad ways by disabling L2 ways and to read/write values into the scratchpad. In the future, more operations will be modified to accommodate the design requirements of the MCPU.

### 17.1.2. Memory Tile

The memory tiles in ACME encompass the elements of the architecture related to the management of data in the higher levels of the memory hierarchy. This includes the memory controllers, the Memory CPUs and the L3/RowBuffers that they interact with. These elements were absent or modeled simplistically in M9. The following features are the advancements regarding the modeling of memory tiles in Coyote, which have been implemented using Sparta.

The detailed design of the Memory Tile and its components is still a work in progress, but the necessary ability to provide detailed models of DRAM and of a memory controller have been added to Coyote.

#### 17.1.2.1. Memory Controller

A memory controller is the interface to access data (and instructions) in main memory. It receives requests that are then converted into a sequence of commands that exercise the correct channels, banks, rows, etc. of the memory it handles, to produce the desired effect. The actual mapping of a request to the geometry of the memory, which depends on the technology of the memory, is encoded in the request's address itself. Different mapping policies have very different behaviors, which may impact bank parallelism or even the average access latency to the memory (e.g., in HBM only one row per memory bank can be open at a time, which needs to be closed if it does not contain the data for the current request).

For the reasons above, memory controllers need to be modeled in a manner that allows to account for these minute effects and that offers sufficient flexibility to evaluate different policies. A memory controller can perform a single operation (at most) per cycle and several decisions need to be made:

- If a bank is idle and several requests that target it are available, which one is selected?
- Out of the available commands, which one is scheduled next?

To account for this flexibility, memory controllers in Coyote are implemented as a class that handles several memory banks, which have a configurable number of rows, columns, element size and latencies. Banks implement a state machine in which event changes are triggered by the latencies and commands from the memory controller. For the latter, the most common commands have been considered (open, close, read and write), each one with its own latency.

Memory controllers also hold a command scheduler and a request scheduler to answer the questions posed above. They have been implemented using inheritance to allow for easy extensibility, in case different policies need to be evaluated. The command scheduler currently performs a Round-Robin selection across the banks that have pending commands. The request scheduler also operates in Round-Robin across the banks that have pending requests, but within a bank, fetches are prioritized overloads, which in turn have higher priority than stores. This is to account for stores being "fire and forget", loads having a certain likelihood of stalling the core if not handled swiftly, and fetches being completely blocked. The memory controllers also support different, easy to extend, address mapping policies. Coyote currently supports the well-known open-page and close-page policies.

#### 17.1.2.2. Memory CPU (MCPU)

The implementation of the MCPU in the Memory Tile is embedded in a wrapper. This wrapper accepts the classes and therefore the communication sent to the MCPU. It translates the data into a representation that is independent of Sparta and Spike. This allows us to execute the MCPU code on a generic processor within an operating system or even on a bare-metal processor given that the wrapper is adopted according to the scenario.

The basic version of the MCPU includes the following features:

- Handling of transactions containing vector memory operations from the VAS Tile.
- Translations of virtual addresses into physical ones by Translation Look-Up Buffers (TLB) in the Vector Address Generator. Due to the length of the vector, a memory operation may be composed of multiple memory requests which can be forwarded to the physical memory. Therefore, a translation needs to be completed for every request to compute its physical address.
- The removal of parasitic bytes to coalesce elements into a dense vector for indexed and non-unit stride memory loads. In this case, the parasitic bytes are undesired, neighboring elements.
- A bypass for the MCPU for L2 cache misses caused by scalar memory operations executed by the Scalar Core in the VAS Tile or page look-ups from the TLB in the VAS Tiles.
- Communication of memory requests to remote Memory Tiles. Each Memory Tile handles only a portion of the overall available memory. If a physical address is not serviced by the local memory address range, it needs to be forwarded to the Memory Tile, whose address range includes the address of the request.
- A local LLC following write back caching policies. It caches only data from the local memory, avoiding coherency protocols.

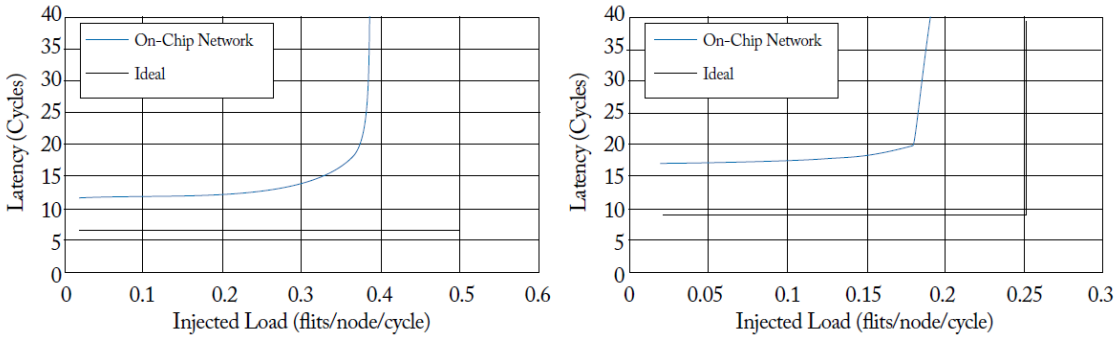
The implementation of the MCPU and the required components, such as Vector Address Generation Unit, caches and micro engines, is an iterative process starting from a basic version offering only a minimum set of functionalities. Many decisions for improvements and added functionalities depend on the results of the simulation. For instance, how and in what situation the MCPU will optimize the memory operations intelligently (refer to section 3.1.1.4) is still an open question.

### 17.1.3. Network on Chip (NoC)

A scalable low-latency and high-throughput communication fabric to connect all tiles of the ACME accelerator is critical. Buses and crossbars are the dominant fabrics in chips with a low number of cores. Due to the limitations in bandwidth on the former and in scalability on the latter, networks-on-chip (NoCs) are the perfect replacement for those typical fabrics in many-core chips like the ACME accelerator. As introduced in the first deliverable D4.1 in Section 4.6, ACME uses a mesh topology illustrated in Figure 58.

The NoC comprises routers, represented as yellow boxes inside tiles in Figure 58, links, represented as yellow lines between tiles in the same Figure, and processing elements. In each tile there are multiple traffic generators that are concentrated, from the point of view of the NoC, by the crossbar internal to the VAS Tile and defined as the processing element (PE) of that tile. The router in the tile connects with the others in its cardinal points and to the PE. Between the routers there are multiple links with different widths. These multiple networks prevent the protocol deadlock problem [C3] and isolate the load of these different networks. This also could help to reduce the power consumption of the system because a small message, like a request for a cache line, is sent through a narrow physical link whereas a large message, like a cache line with 64 bytes of data, uses a wider link.

The ACME specifications, according to what was introduced in the deliverable D4.1, define the NoC as a grid-like rectangular 2-dimensional interconnection fabric between the processing, memory, and I/O tiles. The theoretical behavior of the mesh topology is well-known and can be found in the literature, such as [C10, C11]. Figure 64 plots the average packet latency for an offered load sweep using two synthetic traffic patterns: random uniform and bit-complement. The Figures also plot the ideal latency and throughput, considering the ideal latency equal to the average hop count, for each traffic, plus one cycle to cross the link between the final router and the destination node, and calculating the ideal throughput as the maximum channel load at the bisection links. The performance obtained differs from the ideal due to inefficiencies in routing and arbitration, and different router micro-architecture designs. State-of-the-art virtual channel routers deliver about 80% throughput of the ideal for both traffic patterns, whereas wormhole flow control routers without virtual channels saturate much earlier than the curves shown below [C10].




a. Random uniform traffic (Avg. hops = 5.33)

b. Bit-complement traffic (Avg. hops = 8)

*Figure 64. Average packet latency versus offered load for two synthetic traffic patterns using an 8 × 8 mesh. (Source: [C10])*

The NoC in ACME is split into three physical networks: Data-transfer, Address-only and Control. Conceptually, the NoC is proposed as a simple, brute-force NoC that avoids the use of virtual-channels, priorities and any other intricate feature. The NoCs maintain point-to-point ordering between any pair of nodes, without bypassing. As explained in the following section, three different models are proposed to represent the NoC into Coyote.

### 17.1.3.1. NoC Models in Coyote

From the simulation point of view, the NoC can be seen as a black box with several ports connected that manages certain message types. Its behavior is determined by certain configuration parameters, and it generates several outputs to be analyzed. This abstract view can be seen in Figure 65.

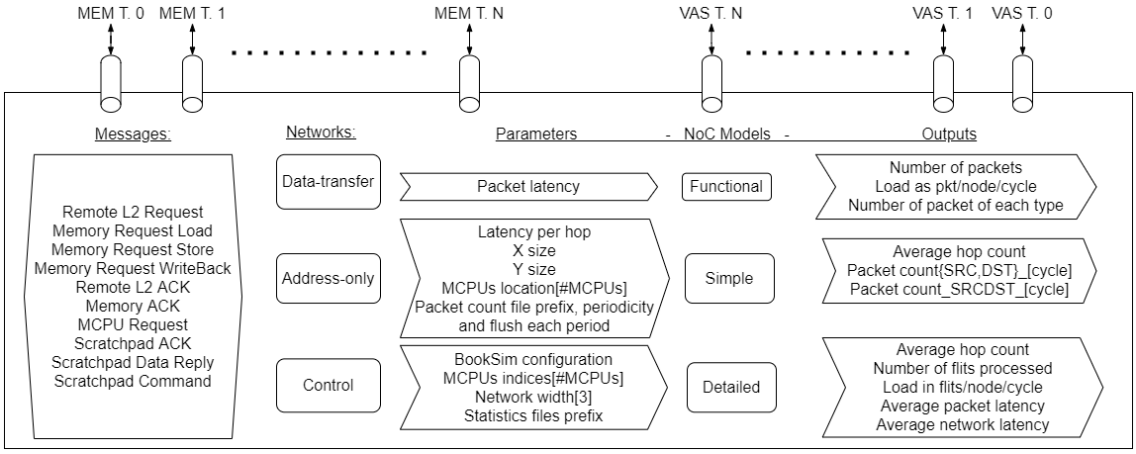



*Figure 65. Abstract representation of NoC in Coyote*

The way a NoC is modeled depends on the required NoC level of detail and its intent. Ideally, a NoC is an element that allows the interconnection between different elements with a certain delay. This can be refined in detail to the point where it is possible to simulate, cycle-by-cycle, all the steps in the pipeline of the NoC routers. Depending on the flexibility and the level of detail of the model, different metrics can be obtained. Accordingly, in order to trade off the level of detail and the performance of the simulator, three different models have been developed as part of Coyote:

Functional, Simple, and Detailed. These NoC models are agnostic of the level of detail of the traffic generated by the memory subsystem modeled in Coyote.

#### 17.1.3.1.1. Functional model

The functional model allows fast simulation and exemplifies an ideal NoC with a fixed delay in its transactions. This delay, as a first stage, has been determined based on the current literature. However, in a later stage, it will be obtained from the insight provided by the detailed model.

This model intends to satisfy the need for NoC simplicity, equal to a low simulation penalty when different aspects of the architecture, not directly related to the NoC, are tested. This allows a latency sweep to analyze the impact of NoC latency over the execution time of the benchmarks launched with Coyote, as you can see in Section 4.2.1.

#### 17.1.3.1.2. Simple model

The simple model increases the simulation penalty to be able to analyze the impact of the architectural placement of tiles and to extract source and destination heat-maps.

This model mimics the behavior of a non-fault-tolerant mesh and returns the average hop count of the traffic managed by the NoC. In fixed intervals, it updates a matrix indicating the number of packets sent and received for statistical purposes. In the same way, this model obtains the number of packets interchanged between each source and destination pair on the NoC. Furthermore, the model calculates the number of hops between the source and destination of the packet and forwards the packet with a delay that multiplies this hop count by a parametrized 'latency per hop' value. Again, congestion and internal queueing effects are not modeled.

#### 17.1.3.1.3. Detailed model

The detailed model integrates the BookSim 2 interconnection network simulator [C4] into Coyote. This model generates accurate metrics and simulates a detailed model of the NoC, although at an increased simulation overhead. This model allows us to test different topologies, routing policies, and router microarchitectures, to analyze the impact of contention and congestion. For example, it provides an accurate average packet latency that can be used in the functional model explained above.

The NoC modeled as a baseline using this model is based on the ACME specifications. It defines three different physical networks: Data-transfer, Address-only and Control, each one has a different channel width, and routers with five ports, one input buffer per port, a round-robin arbitration policy and a pipeline of three stages: routing computation, switch allocation and switch traversal. Apart from the cycles represented by these stages, the injection and link traversal times are also modeled. A block diagram detailing the NoC in a tile can be seen in Figure 66.

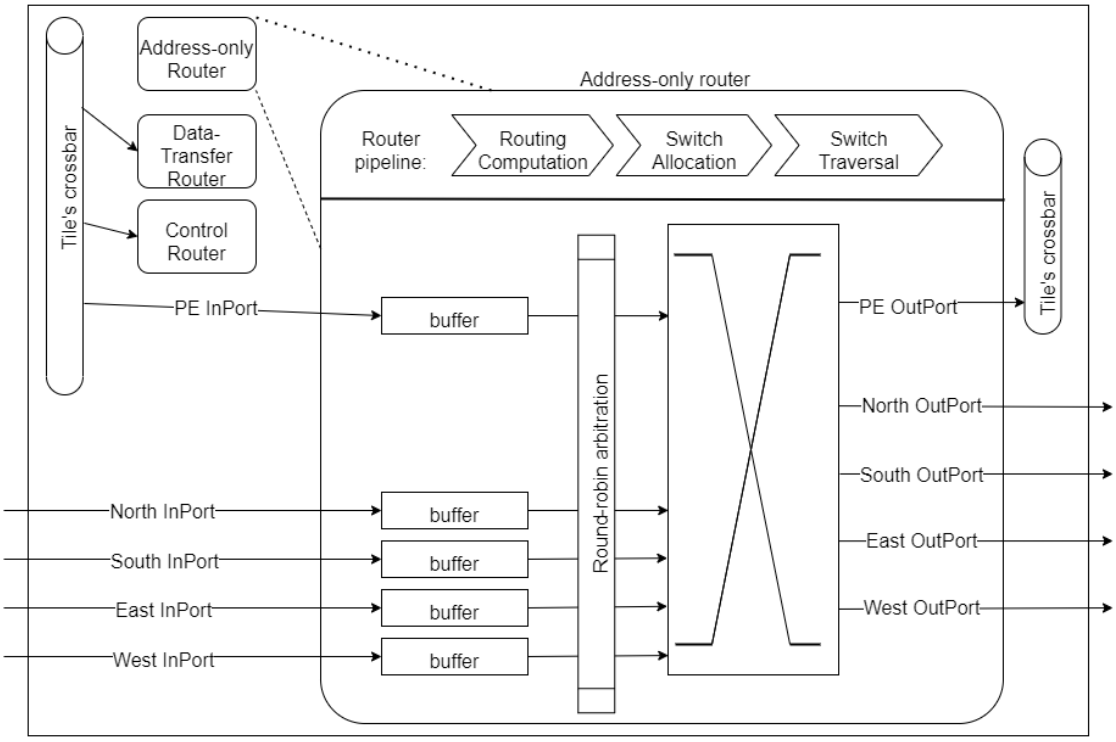



*Figure 66. Block diagram of NoC into ACME tile*

## 17.2. Application support

The initial target of Coyote is baremetal applications, which may be compiled using the EPI LLVM compiler to leverage the EPI VPU vector intrinsics. This requires minor editing of the source for applications in order to do without OS support, which includes:

- Avoiding syscalls. As a result, file I/O and dynamic allocation of memory are not available. The data of the application needs to be static.
- Hand-parallelizing the application by specifying entry points for each simulated thread.

By following these guidelines, four out of the five kernels used for verification in EPI were adapted to Coyote execution. All of them are vectorized using the EPI intrinsics and leverage multiple cores. These are:

- Dgemm
- SpMV
- Axpy
- Somier

However, a caveat was encountered when trying to adapt the fifth kernel, FFT: it uses the fftw library. This makes its adaptation to baremetal complex, because the library internally uses syscalls.

### 17.2.1. Next steps

As a consequence of the difficulty of adapting general applications requiring syscalls to the simulator, a currently ongoing task is to extend Coyote to support a subset of syscalls. To do so, the riscv proxy kernel (riscv-pk) [5] is being used as a basis, which is an open source lightweight

application execution environment that can host statically linked RISC-V ELF binaries and includes the Berkeley Boot Loader (bbl) as a supervisor execution environment for tethered RISC-V systems. By proxying a subset of syscalls to the host computer, the riscv-pk enables the simulation of more standard applications in Spike, but it only has support for single-threaded applications.

To leverage its capabilities in Coyote, the RISC-V proxy kernel is being extended to support multi-threaded applications in a shared memory space. The multi-step plan for the extension of riscv-pk goes as follows:

**Step #1:** Initial proof of concept: use the original Spike simulator and modify pk to launch a simple multithreaded program without any syscalls. Verify that all harts and their state are correctly initialized (e.g. stack, CSRs, handling routines...). All harts will be provided with the same entry point, the *main* function. Note that the purpose is not to have higher level parallel paradigms as OpenMP or pthreads in Coyote, but to enable a subset of basic syscalls, which will not initially include, for instance, *fork/clone*, which usually requires extensive OS support.

**Step #2:** Have all harts executing "simple" syscalls, such as the ones required to print messages or *getcpu*, to enable each core to identify itself.

**Step #3:** Integration in Coyote.

**Step #4:** Have harts execute more complex syscalls, such as the ones associated with I/O or dynamic memory allocation, on all cores. These syscalls typically need to be thread safe. This might be implemented by extending Coyote with extra instructions to mimic the behavior (not the performance) of critical regions, similarly to the support for barriers introduced in section 3.1.1.1.

By reaching step 4, Coyote will run applications that use common syscalls, but some modifications will still be necessary for the hand-parallelization of the sequential code. All harts will run the same code, they will need to identify themselves and partition the workload. This is an enormous improvement that will significantly ease the future adaptation of other applications to run in Coyote. The further extension of Coyote and pk to support more common parallelization approaches, such as pthreads or OpenMP will be evaluated considering the rest of the project, as it is likely to require a significant implementation effort that might be better devoted to other tasks.

# 18. ACME Simulation results with Coyote

Although Coyote still lacks certain capabilities, it now allows us to model many aspects of ACME. As part of our verification of Coyote, we have implemented a number of studies. These are simulations looking at various aspects of the current ACME design.

The results must be read with caution, since the primary goal is to exercise the models in useful situations.

The following sections present the results of these studies. They may be classified in 3 different groups:

- An evaluation of some tile sizing specs presented in deliverable 4.1.

- An analysis of the NoC using the models included in Coyote to identify the requirements of the network on chip.
- A paraver-based analysis to extract low level information from traces.

The experiments have been run using the supported applications shown in Section 3.2 using input sizes at least twice the total size of the L2 cache in each of the experiments, to ensure a considerable memory traffic generation.

## 18.1. Tile sizing analysis

Deliverable 4.1 introduced the ACME architecture. This section presents a set of experiments intended to confirm the sizing decisions regarding ACME that were made in M9, in order to identify potential opportunities for improvement, by avoiding bottlenecks or unnecessary over-provisions.

The following experiments have been performed using a scaled-down version of ACME. The baseline is a system with a single tile and a single memory controller resembling an ACME tile, with the configuration shown in Table 18. For each of the experiments, all the parameters are fixed but one that is iterated over to analyze its impact on performance.

| Parameter | Value |
|---|---|
| Number of tiles | 1 |
| Number of memory controllers | 1 |
| Number of cores | 8 |
| Frequency | 1 GHz |
| Num vector lanes per core | 8 |
| Maximum vector length | 8 |
| Number of L2 banks | 16 |
| L2 bank size | 256 Kilobytes |
| L2 Line size | 64 Bytes |
| L2 Associativity | 16 |
| Cache data placement policy | set interleaving |
| NoC Model | Functional |
| NoC Latency | 30 |
| Number of memory banks | 8 |
| Address mapping policy | close-page |

*Table 18. Baseline for the tile sizing experiments*

### 18.1.1. Core count

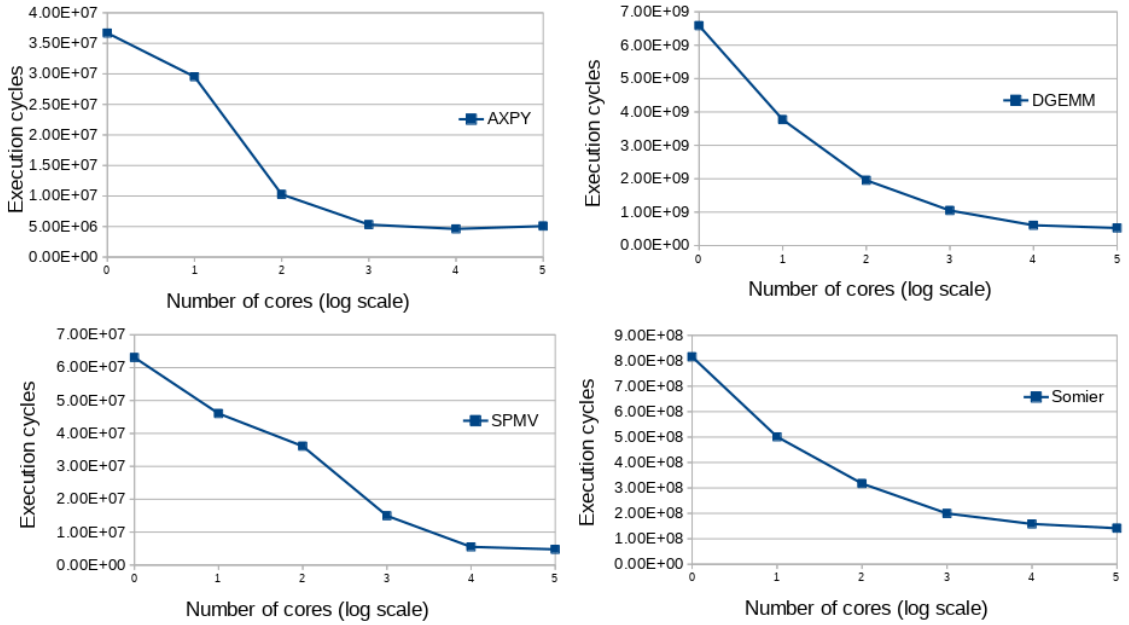



*Figure 67. Effects of number of cores on execution time*

Figure 67 plots the effects of increasing the number of cores (shown in x-axis on logarithmic scale in powers of 2) on the execution time (shown in y-axis as number of cycles) of the applications.

As is evident from the Figures, the execution time decreases linearly till the core count reaches 8. Increasing the number of cores beyond 16 does not result in significant reduction in execution time. This is because increasing the number of cores results in more memory requests getting generated, and the benefits obtained by increasing the parallelism are outweighed by the increase in stalls in the caches due to the larger number of outstanding memory requests. Therefore, the design decision to keep the number of cores to 8 per tile seems apt based on the experiments.

### 18.1.2. L2 Bank size

For this experiment the number of banks is kept constant, while their size is increased, effectively increasing the overall size of the L2. The number of cores in the tile is fixed to 8.

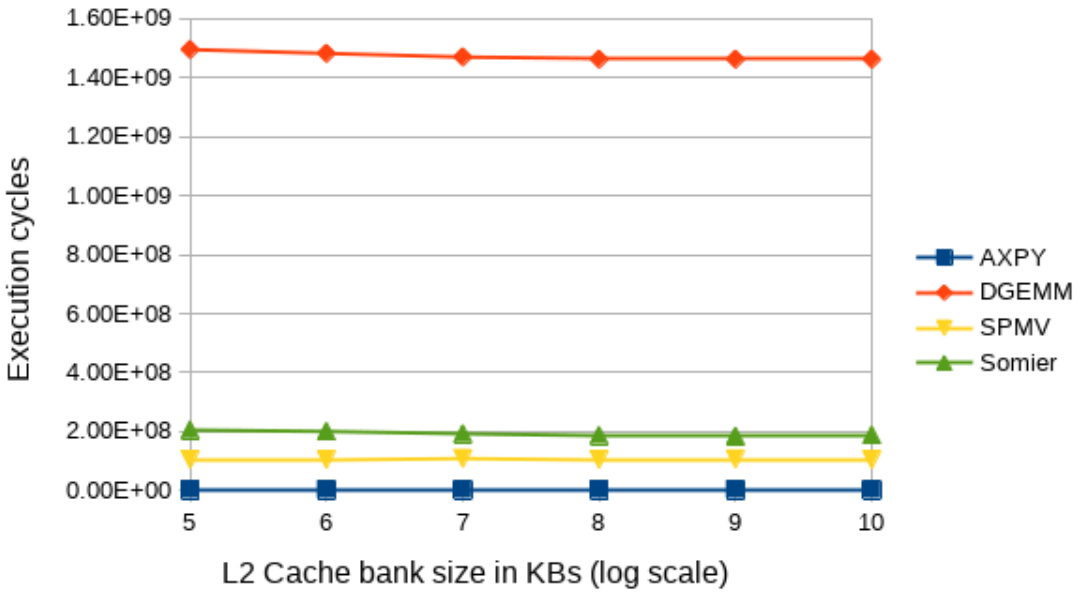



*Figure 68. Effects of L2 cache bank size on execution time*

Figure 68 plots the effects of increasing the L2 bank size (shown in x-axis as logarithmic scale in powers of 2) on the execution time (shown in y-axis as number of cycles) of the applications.

As shown in the Figure, the increase in L2 cache size has minimal impact on execution time. Figure 69 shows the miss ratio for the same experiment. As expected, axpy, a streaming application with no data reuse in the L2, sees no improvement at all as the cache size increases. However, the improvement in the other applications is also negligible. The conclusion is that capacity misses, which are the ones avoided by incrementing the size of the cache, are not an issue for the evaluated applications. A paraver visualization to analyze the miss type breakdown is currently under development.

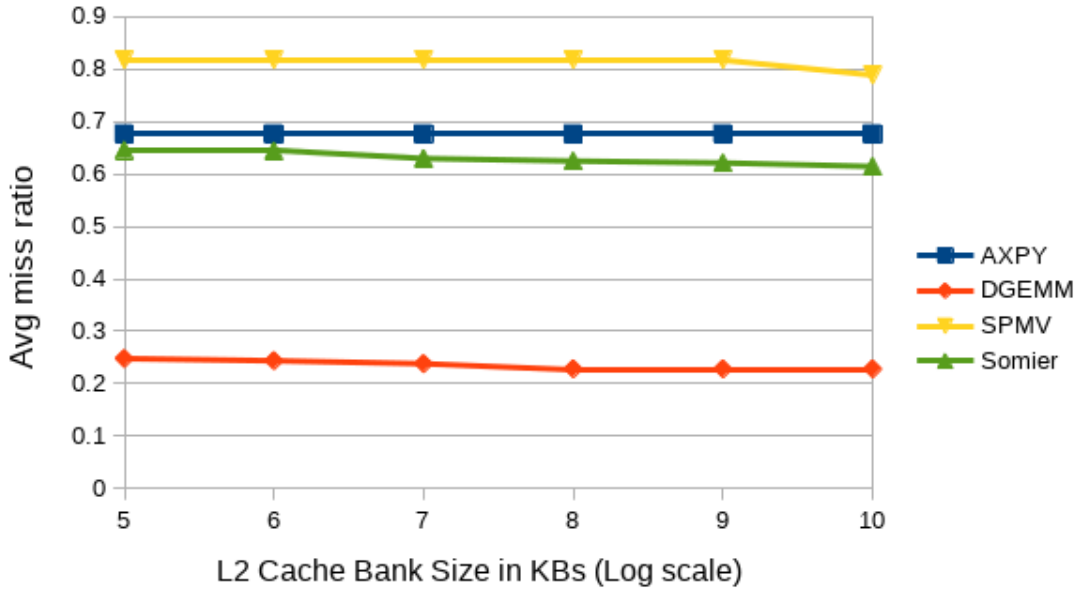



*Figure 69. Effects of L2 cache bank policy on the miss ratio*

### 18.1.3. L2 cache bank data mapping policy

Coyote supports two data mapping policies for the L2 cache, in which different bits are used to identify the L2 bank that corresponds to the requested address:

- **Set interleaving** - The least significant of the set bits are used to identify the bank. The result is that consecutive sets are mapped to consecutive banks, effectively interleaving sets across the L2 cache.
- **Page-to-Bank** - The most significant of the set bits are used to identify the bank. Several consecutive sets are assigned to the same bank.

Figure 69 shows the impact of L2 cache data mapping policies on the execution time (shown as speedup of page-to-bank over set-interleaving policy) for an L2 cache size of 4 MB and 8 cores per VAS Tile. The purpose of this study is to demonstrate the depth of the experiments that can be performed with the help of Paraver.

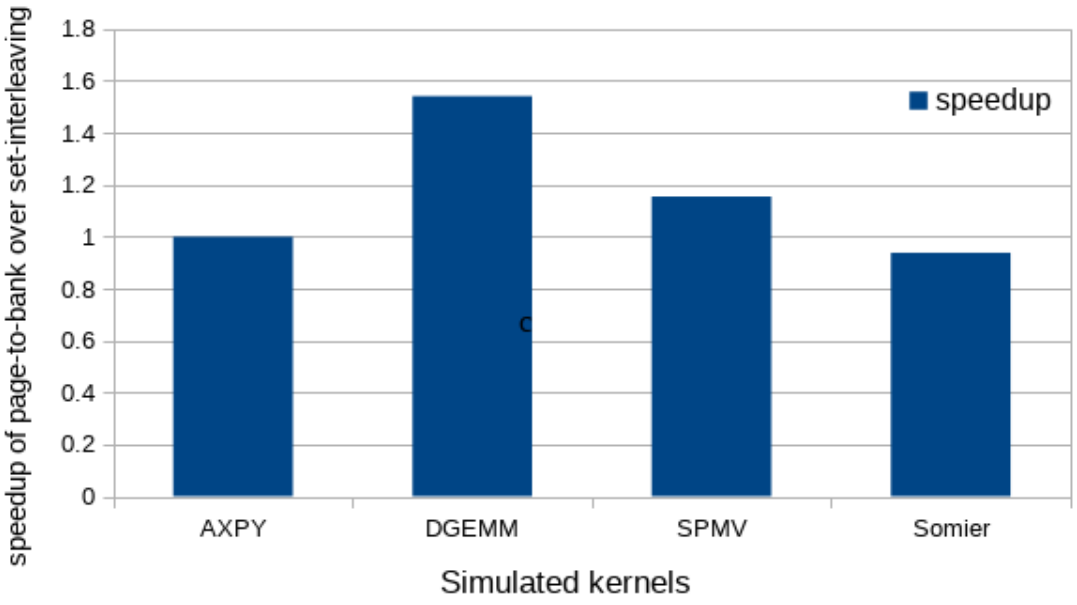



*Figure 70. Effect on execution time of page-to-bank mapping policy over set-interleaving*

As shown in Figure 70, the effect of L2 cache data mapping policy is quite significant for DGEMM, while smaller differences can be observed in the other applications. A Paraver analysis of trace for DGEMM is shown in section 4.3.3 that explains this behavior.

## 18.2. NoC analysis

As noted above, at this stage we are evaluating our model, not the ACME design itself. There are a number of effects we will wish to study in a more complete and accurate model; these include

- The effect of transaction latencies on performance
- The existence and nature of transaction hot spots
- The effect of transaction widths - it is possible to trade payload width for an increased number of transaction packets, for example

Even with the current state of the model, we are able to distinguish these various facets of performance, as detailed in the following sections. After modeling the NoC as explained in Section 3.1.3 and its subsections, the first and early experiments carried out to analyze the NoC are presented in this section. First, Section 4.2.1 evaluates the bounds for the average packet latency that have no noticeable impact on the performance of the executed kernels. Second, an evaluation of the NoC traffic distribution is performed in Section 4.2.2. Then, an evaluation of the impact of the channel width using packets from 1 up to 7 flits is detailed in Section 4.2.3. Finally, Section 4.2.4 presents an evaluation of the impact of different NoC designs and a theoretical analysis of the OpenPiton's NoC and how to model it in Coyote is explained in Section 4.2.5.

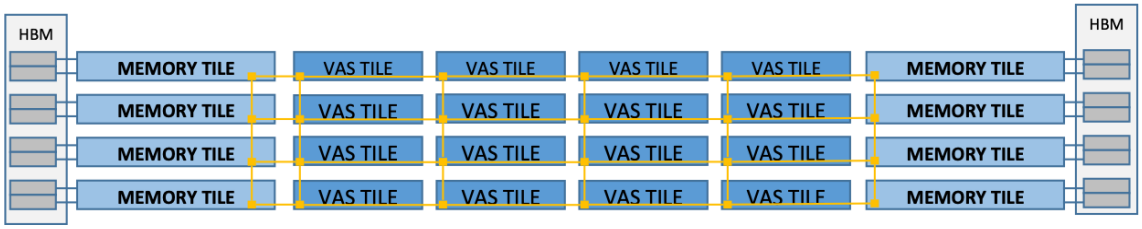



*Figure 71. Block diagram of scaled-down version of ACME highlighting in yellow the interconnection fabric between the tiles*

The system and NoC design evaluated in this section is a scaled-down version of ACME, which you can see in Figure C.17, to reduce simulation times. It consists of 128 cores, which have 1 thread each, in groups of 8 per tile (16 VAS tiles) and 8 MCPUs, resulting in 24 tiles. The tiles are connected by three 6 × 4 mesh NoCs offering an aggregated bisection bandwidth[3] of 608 GB/s, considering 64B, 64b and 32b as the network width for Data-transfer, Address-only and Control NoC, respectively. This result should be defined more accurately once the headers for each network or message type and the use of single or multi-flit (flow control unit) packets are established.

### 18.2.1. NoC latency bounds

This experiment uses the functional model to perform an NoC latency sweep to evaluate its impact on the execution time of the executed kernels. The tiles of the modeled system are interconnected by an ideal fabric with a parametrized average packet latency.

This evaluation sweeps the average packet latency parameter in a non-continuous range of 1 to 2000 cycles and compares two L2 cache sharing policies: 'tile private' and '*fully shared*'.

The following Figures plot the NoC average packet latency on the X-axis and the kernel execution time on the Y-axis, both in nanoseconds, for each executed kernel using two different L2 cache sharing policies.

[3] The bisection bandwidth (BBW) of a network is the total bandwidth available between the two partitions resulting from a bisection of the network into two partitions in such a way that the bandwidth between them is minimum. This gives the true bandwidth available in the system.

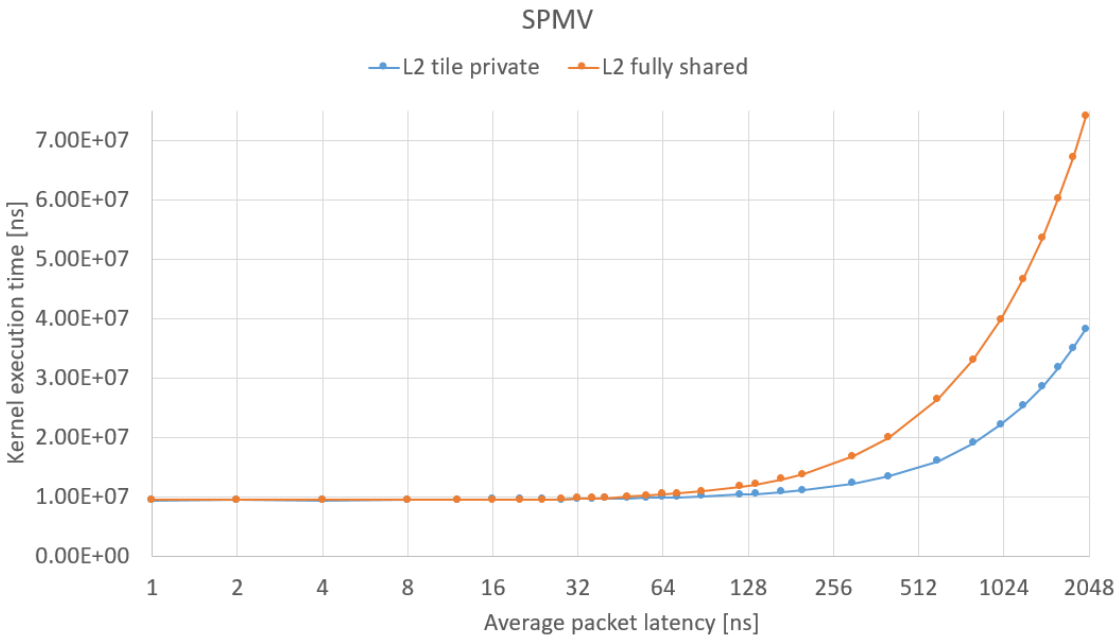



*Figure 72. NoC average packet latency impact on SPMV kernel execution time*

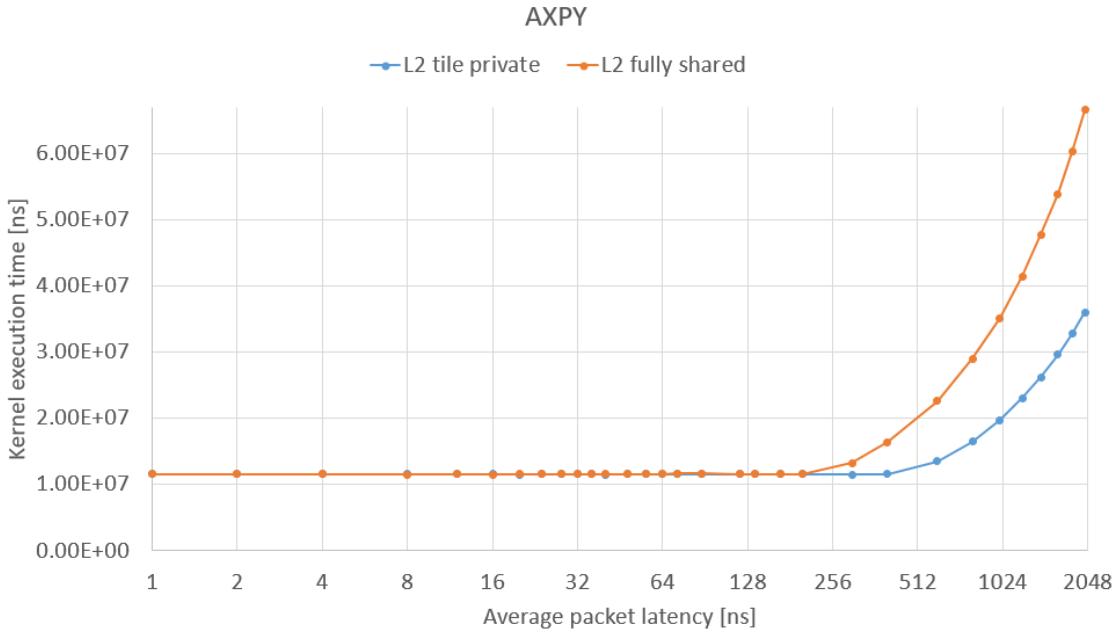



*Figure 73. NoC average packet latency impact on AXPY kernel execution time*

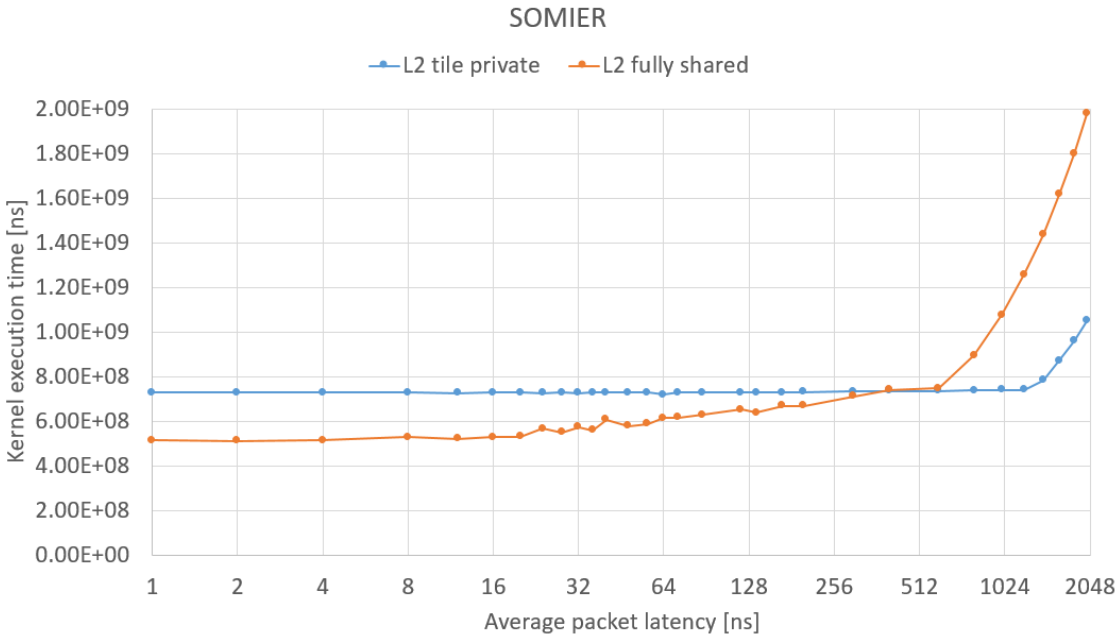



*Figure 74. NoC average packet latency impact on SOMIER kernel execution time*

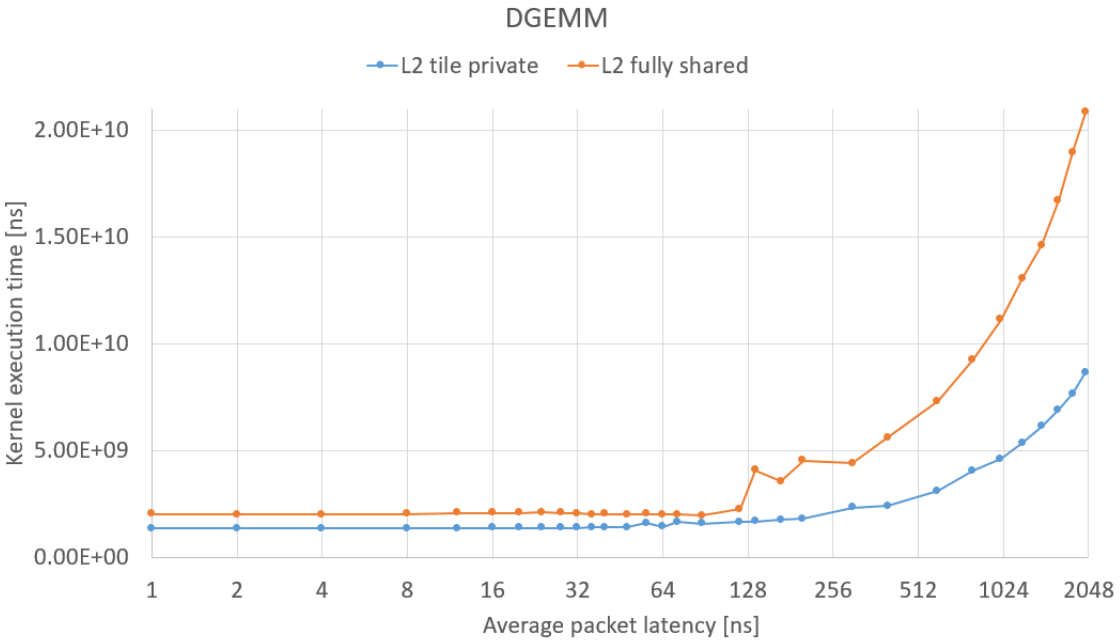



*Figure 75. NoC average packet latency impact on DGEMM kernel execution time*

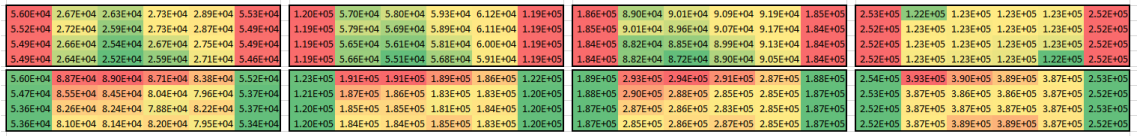
These results suggest a relative insensitivity to NoC average packet latency. The execution time does not increase, even using a '*fully shared*' L2 cache sharing policy that is more sensitive to NoC latency, up to going beyond a hundred cycles. As an early conclusion, a simple mesh fabric with a router pipeline of 3 stages would prevent the NoC from becoming a bottleneck for ACME.

### 1.1.1. Traffic distribution

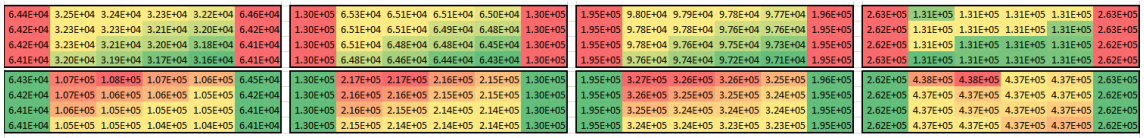
This experiment uses the simple model to perform a NoC traffic distribution analysis. The tiles of the modeled system are interconnected by an ideal fabric, which follows the memory and VAS tile placement shown in Figure 71, with a parametrized hop latency of 4 cycles. This parameter

results in average packet latencies of approximately 20 cycles using '*tile private*' second-level cache sharing policy and approximately 16 cycles for the 'fully shared' policy due to a lower average hop count, which is equal to 4.77 and 3.79 hops, respectively.

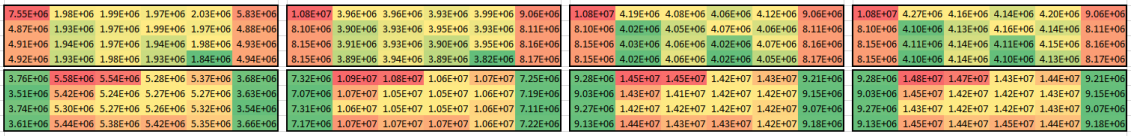
Figure 76 shows the aggregate packet destination heatmap of the three NoC networks for 4 points in time using the L2 cache sharing policies and kernel detailed in Section 3.2. The periodicity to extract the results is selected for each one of the executed kernels to generate four evenly spaced points across the execution of the kernels, the latest one being very close to the end of the execution, which allows the representation of the results at the end of the execution instead of the values at the latest captured period. Three colors are used to indicate the number of packets received by each tile, which go from green (lower) to red (higher).



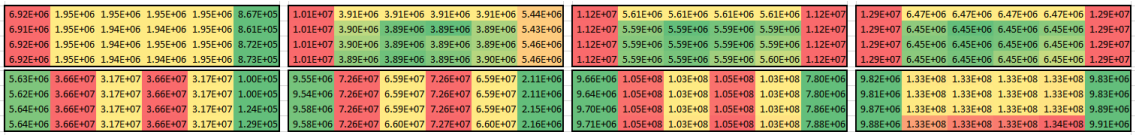
a) SPMV



b) AXPY



c) SOMIER



d) DGEMM

*Figure 76. Heat-maps of traffic destinations, aggregating the three NoCs. The data was captured at four points during the kernels' execution, shown from left to right. 'Tile private' policy is represented in the top row and 'fully shared' in the bottom row*

A quick look at the results shows that the '*fully shared*' policy concentrates most of the traffic in the VAS tiles, as opposed to '*tile private'*, which puts more stress on the memory tiles. In addition, the Figures also suggest that traffic is more or less uniformly distributed spatially and temporarily for SPMV, AXPY and SOMIER, when employing either of the cache policies. Note that in this configuration (see Figure 71), twice as many VAS Tiles exist compared to Memory Tiles. So for correctly balanced programs with even access patterns and a tile private policy, MEM tiles being involved in twice as many packets as VAS tiles is the expected result. Furthermore, it is clear that the DGEMM kernel generates a NoC traffic that is not uniformly distributed in all periods between memory tiles, regardless of the L2 cache sharing policy employed, and neither between

VAS tiles when using the '*fully shared*' policy. These notions are analyzed in greater depth in the following paragraphs.

The traffic on the NoC executing the SPMV kernel is uniformly distributed spatially and temporarily. As it was above-mentioned, the traffic received by MEM tiles is bigger than the traffic received by VAS tiles when '*tile private*' policy is used. The differences between the number of messages in the VAS tiles are negligible using both L2 cache sharing policies. Following the same trends, executing AXPY the traffic on the NoC is well distributed. When using the '*fully shared*' policy, a stair shaped pattern can be observed in the plots. However, when analyzing the numbers themselves, the differences are not significant. The overall highest and lowest packet counts are so close that similar values get assigned different colors.

The NoC traffic for SOMIER shows a hotspot in MEM tiles 0 and 1, which receive 53% and 18% more traffic than the average of the other six Memory Tiles, respectively. The traffic on the other six tiles is uniformly distributed. After analysis, the number of the packets received by Memory Tiles during the shown periods, it can be concluded that the results of SOMIER should be taken carefully because the values remain constant from the second period onwards using '*tile private*' and from the third one using '*fully shared*'. This result will differ when a proper modelling of contention accessing the NoC and coherence will be supported by Coyote.

The NoC traffic captured during the execution of the DGEMM kernel shows that the memory tiles on the left receive more messages than those on the right during the first period, regardless of the L2 cache sharing policy employed. Specifically, using 'tile private', for the first period the average number of packets for the memory tiles on the left is 697% higher than that for the ones on the right. This difference decreases over the second period to only 86%. In the same line, when using 'fully shared' the difference sky-rockets up to 5000% for both the first and second periods, but it starts levelling during the third period, until becoming uniformly distributed at the end of the execution. The following paragraph goes in depth with respect to this traffic imbalance. Regarding the VAS tiles, using the '*tile private*' policy, there are a few more packets on tiles 0 to 3, but reviewing the numbers, the differences are not big. The results with '*fully shared*' show that the traffic on the VAS tiles is not uniformly distributed for a significant amount of execution time, but it is evenly distributed at the end of the execution.

The traffic received by the MEM tiles at the first period of DGEMM kernel comes from Data-transfer and Address-only NoCs by approximately 5% and 95%, respectively. Indeed, that traffic from Data-transfer, which are store requests and write-backs, is well-balanced between all memory tiles. Then, the traffic source heat-map for Address-only network at the first period and '*tile private*' policy is shown in Figure 77. It shows that the memory request traffic is uniformly generated by all VAS tiles, so, it can be concluded that the main reason that causes the imbalance of requests between memory tiles are the different data mapping policies used, and the memory blocks requested by the application. The distribution of memory requests by VAS tiles together with the destination imbalance can be seen more clearly in Figure 78 that shows the communication pairs for the Address-only network during the first period of DGEMM using *'tile private*' policy.

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1927037 | 1929536 | 1929029 | 1930890 | 0 |
| 0 | 1924427 | 1921092 | 1920026 | 1923714 | 0 |
| 0 | 1922643 | 1920756 | 1921405 | 1923243 | 0 |
| 0 | 1922395 | 1919175 | 1919356 | 1921846 | 0 |

*Figure 77. Heat-map representing the number of packets generated by each tile in the Address-only network during the first period of DGEMM using the 'tile private' policy*

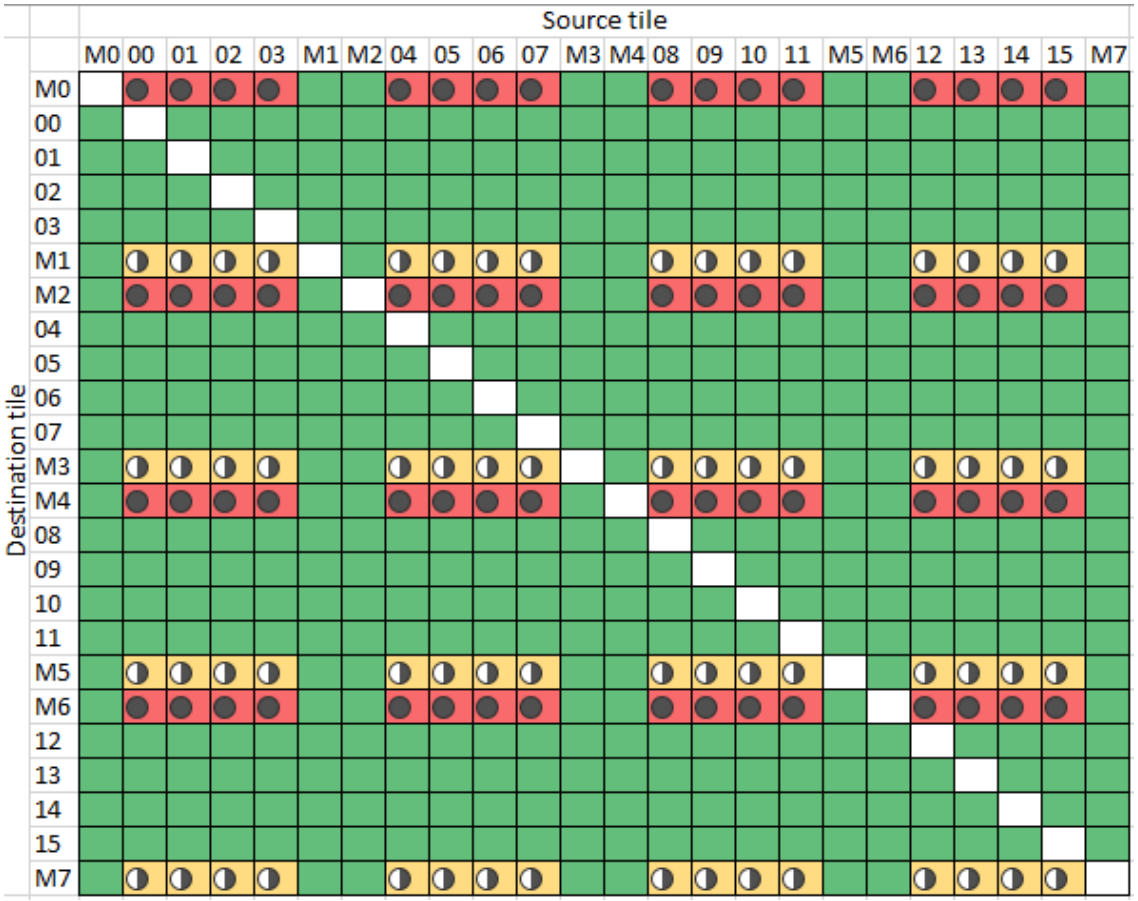
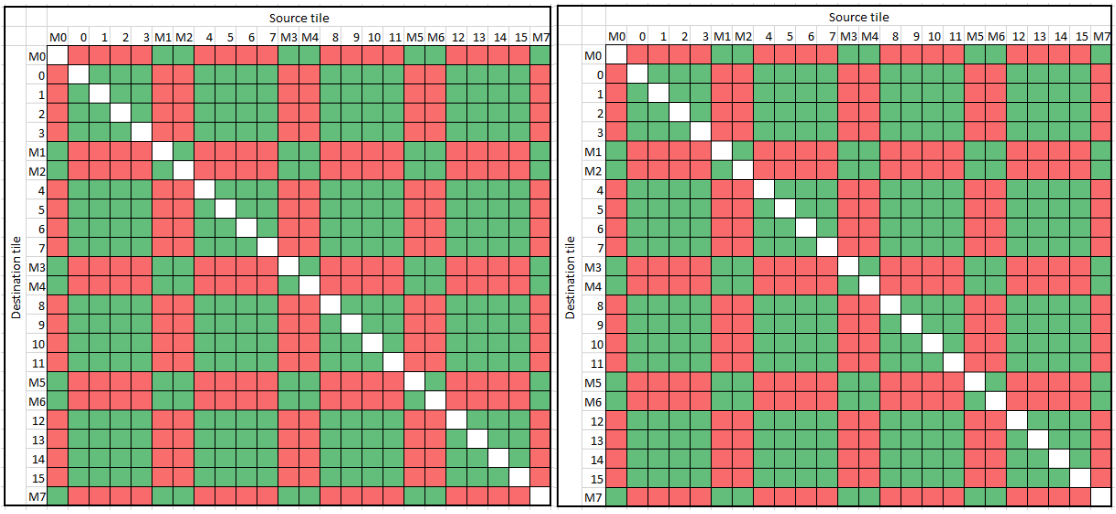


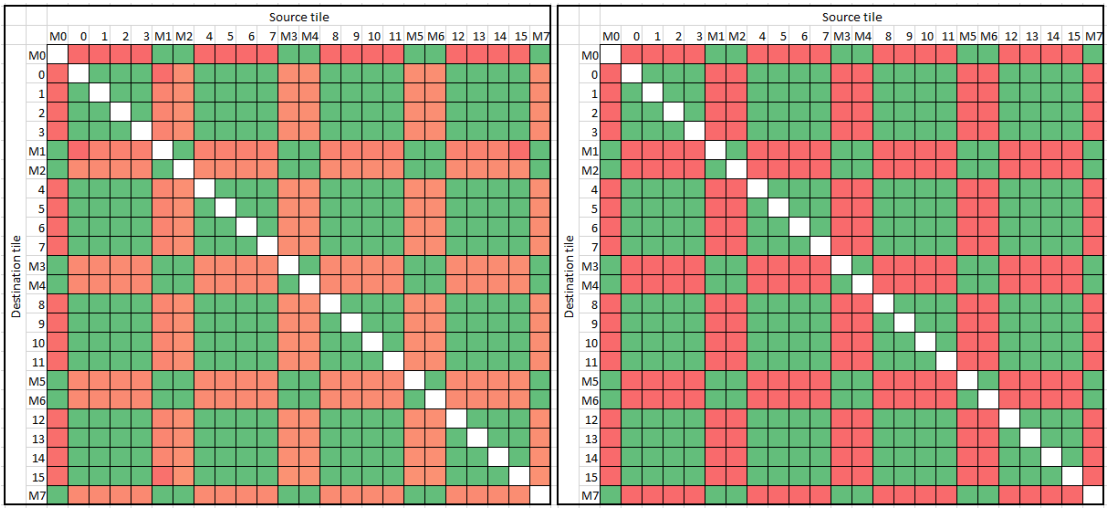

*Figure 78. Heat-map representing the number of packets exchanged between each pair of tiles in the Address-only network during the first period of DGEMM, using the 'tile private' policy. Green implies no packets, half bullet ~ 50 k and bullet ~ 430 k packets. VAS and MEM tiles are represented, by a number and by M with a number, respectively*

Going in depth regarding the communication pairs, Figures 79 and 80 illustrate the communication pairs aggregating the three NoC networks at the end of the execution of each kernel, for both L2 cache sharing policies.



a) SPMV

b) AXPY

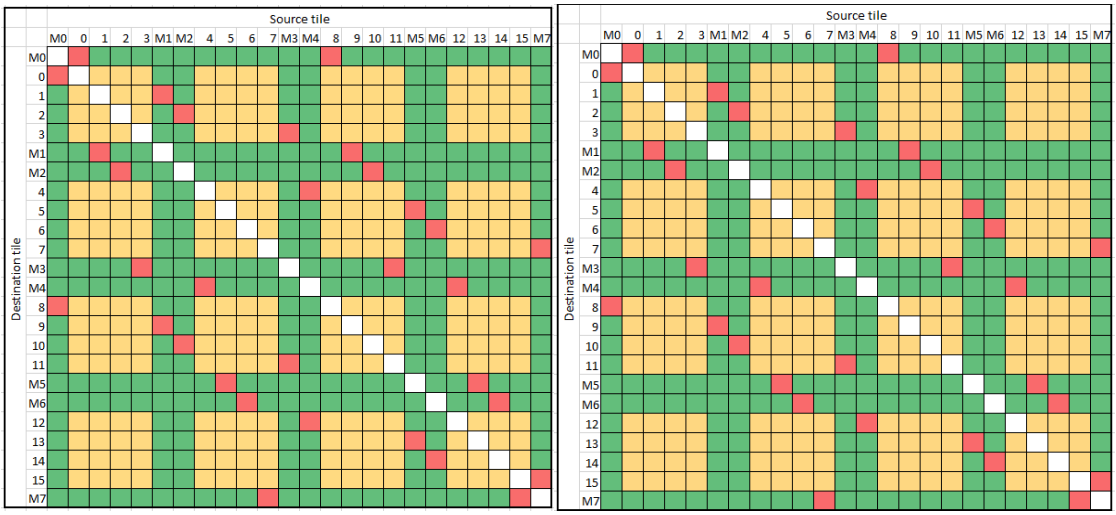



c) SOMIER

d) DGEMM

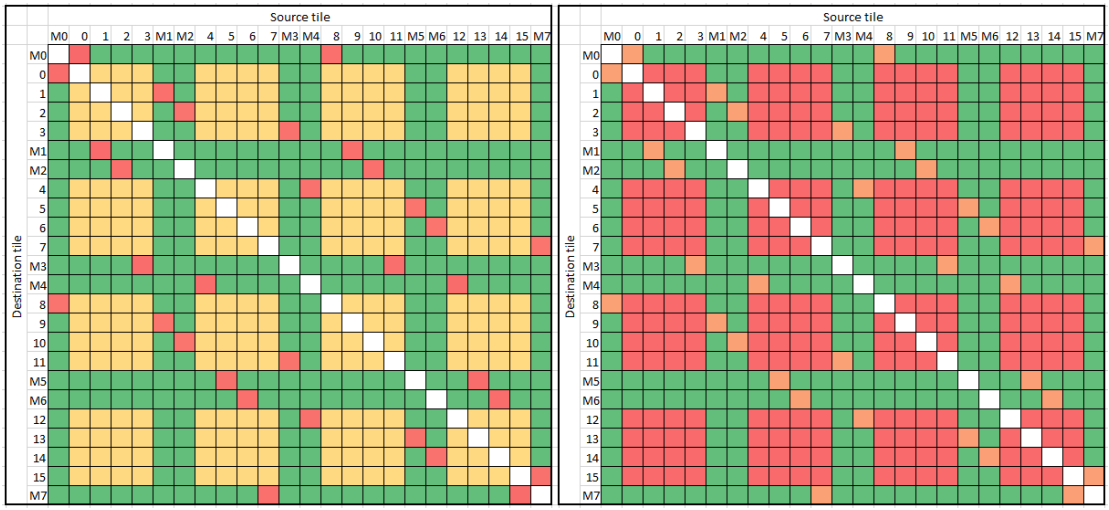
*Figure 79. Communication pairs heat-map aggregating the three NoC networks at the end of the kernel execution, using the 'tile private' L2 cache sharing policy*

The results from Figure 79 explain that the NoC traffic is uniformly generated from VAS to MEM tiles for all executed kernels except for SOMIER, which has two hotspots on memory tiles 0 and 1 (bigger on 0), as was highlighted in the analysis of Figure 76.c.




a) SPMV

b) AXPY




c) SOMIER d) DGEMM

*Figure 80. Communication pairs heat-map aggregating the three NoC networks at the end of the kernel executions using the 'fully shared' L2 cache sharing policy*

Figure 80 suggests that the behavior of the NoC traffic for SPMV, AXPY and SOMIER kernels is identical. However, it is also clear that DGEMM leverages the cache to reduce the pressure over memory tiles. These plots show that the data-mapping policies employed results in communications from VAS to MEM tiles from 0 to 0, 1 to 1, …, 7 to 7, 8 to 0, 9 to 1, …, and 15 to 7. The issue with that resulting data mapping is that these communicated pairs of tiles are far apart.

These early results regarding the traffic distribution should be re-evaluated using different data-mapping policies and must include an improved analysis of the temporal distribution, by capturing a bigger amount of smaller chunks of execution time, to be able to represent the communication patterns along the time at a finer grain.

### 18.2.2. Flit Size

This experiment uses the detailed model to perform a NoC flit-size sweep to evaluate its impact on the NoC. The tiles are interconnected by a 6 x 4 mesh using three physical networks, the routers are conFigured using the parameters shown in Table 19 and router stages and link traversals have a delay of 1 cycle.

| Parameter | Value |
|---|---|
| Number of ports | 5 (4 cardinalities + PE) |
| Injection queues | 1 |
| Routing | Dimension-ordered X-Y |
| Arbitration policy | Round-robin |
| Channel width (Data-transfer, Address-only, Control) | 603, 99, 63 bits |
| Headers size | 27 bits |
| Pipeline depth | 3 stages |

| | |
|---|---|
| Frequency | 1 GHz |
| Switching mechanism | Wormhole |
| Queue size (4 cardinalities ports) | 16 flits |

*Table 19. Router configuration parameters*

This experiment is performed using two L2 cache sharing policies: 'tile private' and 'fully shared' and sweeps the flit-size of the Data-transfer network in a non-continuous range from 512 to 64 bits plus headers that equals to packets of 1, 2, 4 and 7 flits and bisection bandwidths of 603, 315, 171 and 99 GB/s. SOMIER is not used in this experiment because it stresses the NoC injection queues due to the fact that the tile's crossbar is not properly modeled yet in Coyote and the NoC is not able to stall the PEs when the injection queue is full. This implies that the injection queues in BookSim are quite big and cores are able to inject an 'infinite' number of messages to NoC that are enqueued into it, consequently generating unrealistically big values for the average packet latency and the execution time results.

Figure 81 shows the NoC load (flits/node/cycle) represented as percentage for each kernel and packet size evaluated. As shown, the pressure over the NoC increases as the number of flits per packet is increased. Moreover, as expected, the 'fully shared' cache policy increases the NoC load compared to 'tile private'.

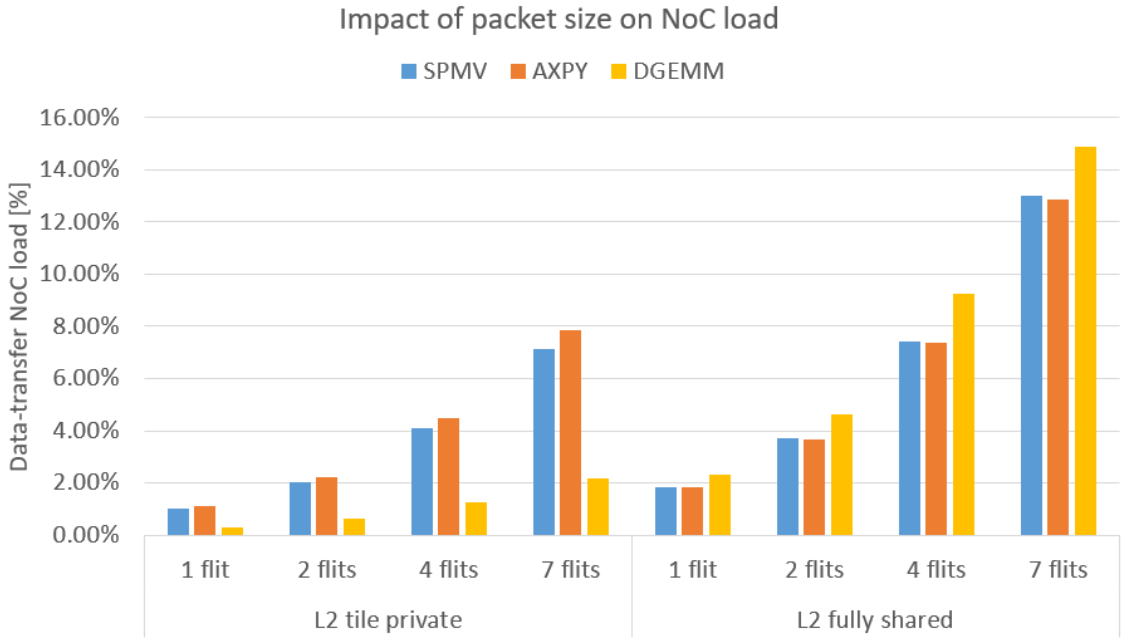



*Figure 81. Data-transfer NoC load for different packet sizes*

The average packet latency obtained for each kernel and packet size evaluated is shown in Figure 82. Due to the serialization latency, the more flits per packet, the more average packet latency. The increment is as expected for almost all the kernels except for DGEMM, which suffers a big impact when the number of flits is increased to 7 using the 'fully shared' policy. It is not plotted, but increasing the number of flits in DGEMM also has a big impact on the L2 cache miss ratio that is ~ 0.11 for packets of 1, 2 and 4 flits and raises to 0.24 for packets of 7 flits. This impact was not seen in the results for the other kernels.

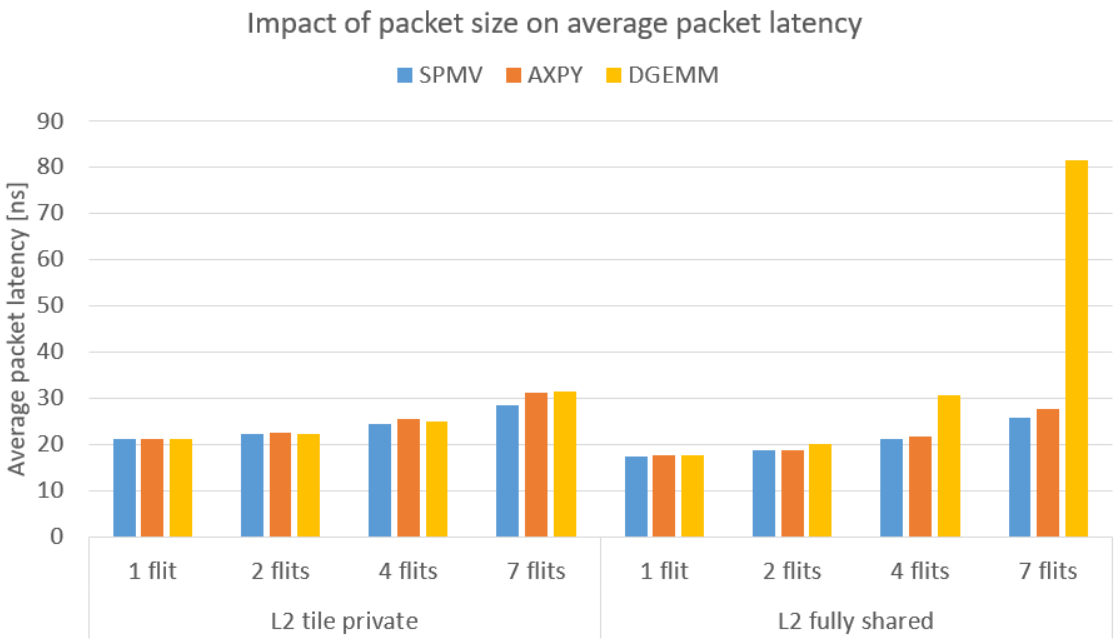



*Figure 82. Data-transfer NoC average packet latency for different packet sizes*

Figure 83 shows the speedup, compared to using a full network width, for the different evaluated flit-sizes (represented as the number of flits per packet) and each executed kernel. Using the '*tile private*' cache policy, the impact of using narrower links is negligible. However, using 7 flits per packet under the '*fully shared*' policy, SPMV and especially DGEMM, suffer an impact on their performance. Furthermore, on the latter, the slump on its performance can be noticeable also using 4 flits per packet.

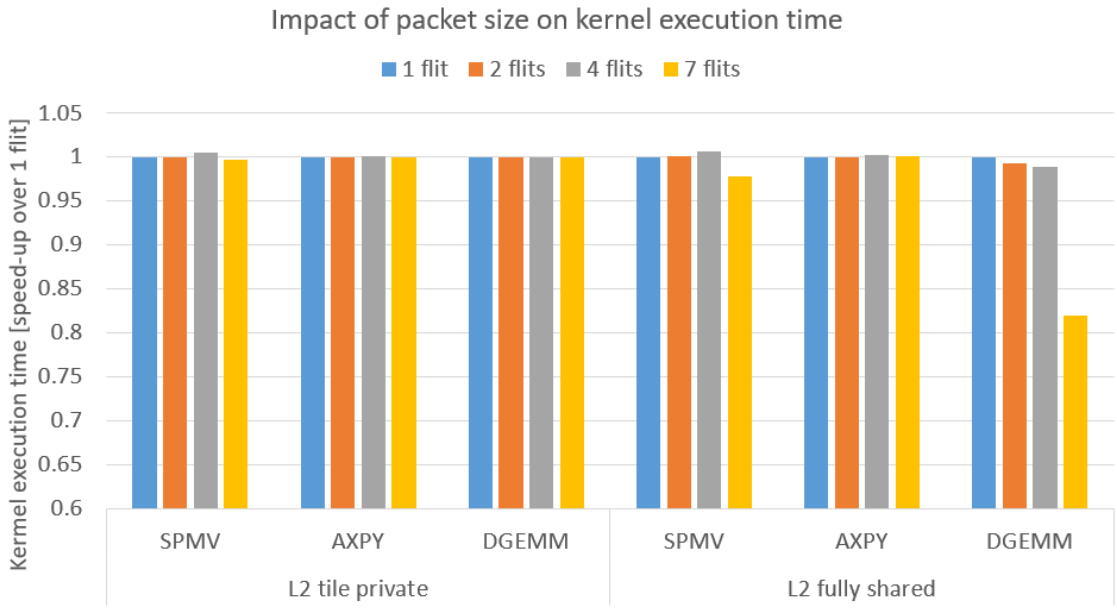



*Figure 83. Kernel execution time comparison for different packet sizes (higher is better). Note that the y-axis scale is reduced to improve its understandability*

The previous results suggest that a narrower channel, which implies that, for example, a cache line transaction packet must be split into multiple flits, could be tolerated. As expected, the network load increases in narrower networks by serializing the data into multiple flits but, even

using 7 flits per packet and the fully shared L2 cache policy, the NoC load is low. However, the high impact on average packet latency and performance of using small flits and the 'fully shared' on DGEMM needs to be considered. Furthermore, a relevant reason to take this result into consideration is the fact that the baseline implementation approach to the ACME VAS Tile component is using OpenPiton, as described in the *MEEP Periodic Technical Report M1-M18, Part B*, which uses packets of up to 11 flits [C8]. This conclusion should be re-evaluated once a comparison of area, energy consumption and critical path delay difference has been performed regarding the use of multi-flit packets. New features added to Coyote, such as coherence or a more accurate modeling of contention will also imply revisiting this study.

### 18.2.3. Design

This experiment uses the detailed model to evaluate different NoC designs regarding physical or virtual networks, the number of injection queues and the use of priorities. The tiles are connected by a 6 × 4 mesh using different designs, using the parameters for the router shown in the previous section. The number of injection queues and the arbiter type parameters are modified from the reference values in Table C.4 for the evaluation of each of the designs covered in this section.

This experiment is performed using two L2 cache sharing policies: 'tile private' and 'fully shared' and evaluates the following router micro-architecture designs:

- ACME: a design with three physical networks, one injection queue per port and a round-robin arbitration policy (see Figure C.12).
- Virtual networks: a design that changes the physical networks to virtual ones (see Figure 84). The router pipeline still has 3 stages (leveraging look-ahead routing) to keep it coherent with the rest of the simulations.
- Priority in transit: a design derived from ACME that uses a round-robin with priorities arbitration policy to evaluate the in-transit prioritization of the traffic (see Figure 85).
- Multiple IQs: a design derived from ACME that adds multiple injection queues (see Figure 86).
- Priority at injection: a design derived from the 'Multiple IQs' that adds the priority field to the arbitration (see Figure 87).

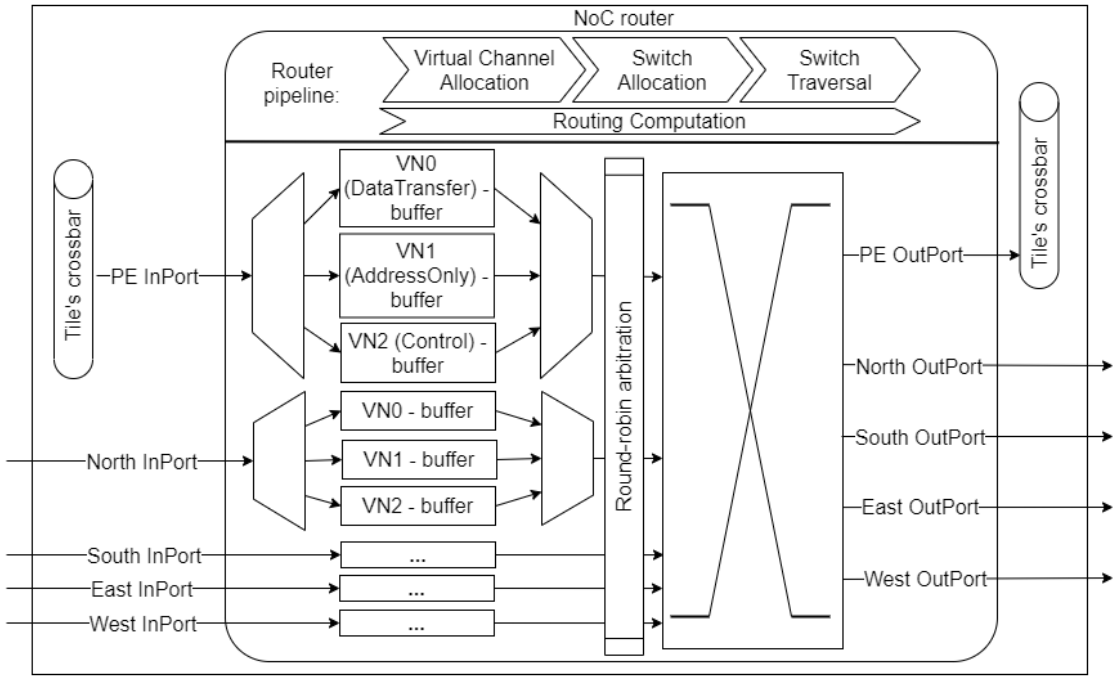



*Figure 84. Design with 3 virtual networks and a round-robin arbitration policy*

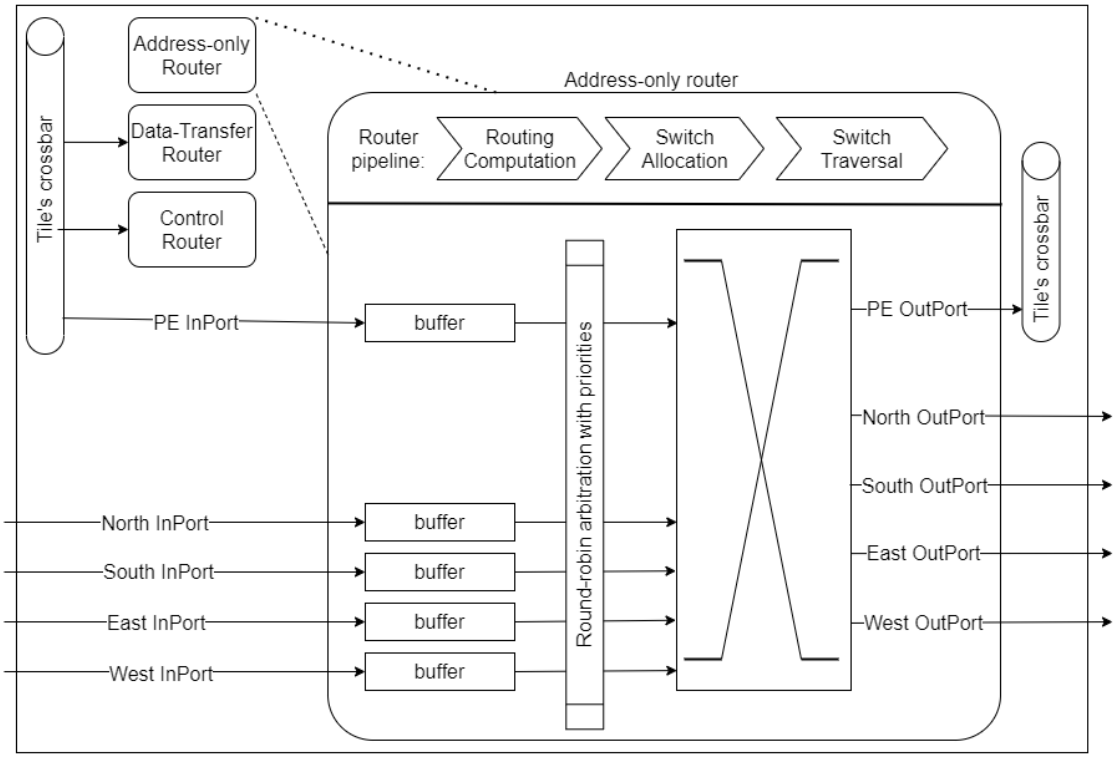



*Figure 85. Design with 3 physical networks and a round-robin with priorities arbiter*

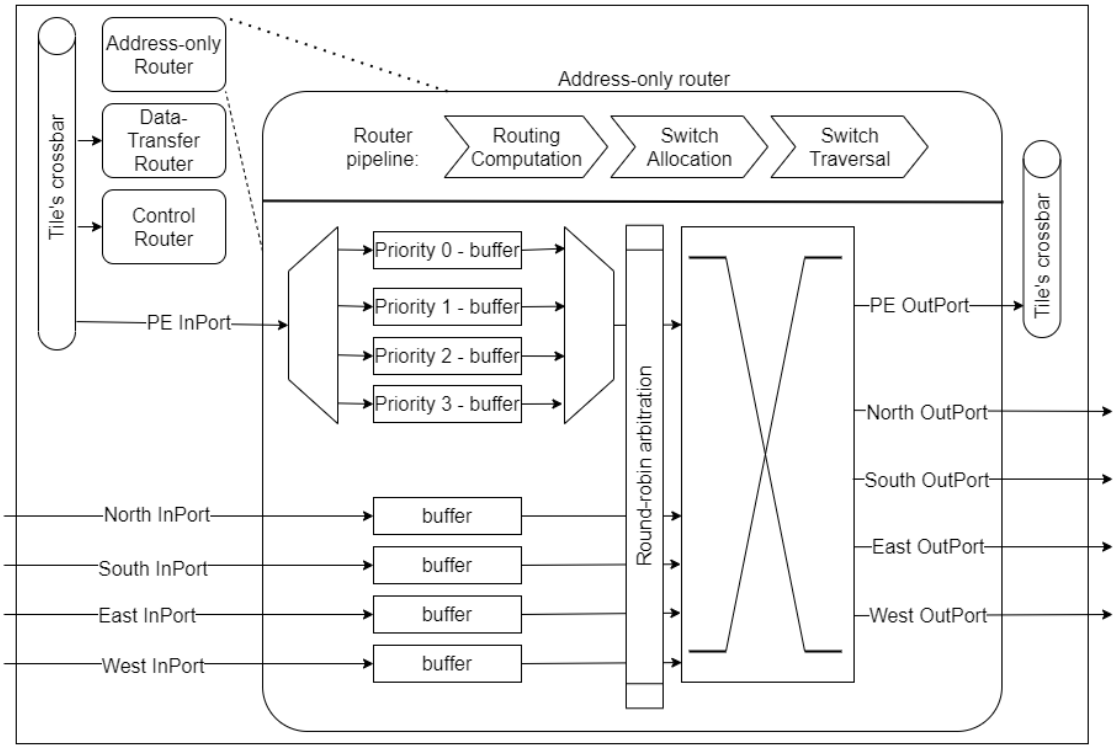



*Figure 86. Design with 3 physical networks and 4 injection queues that are mapped to packets with different priority values. Note that this field is not used during arbitration*

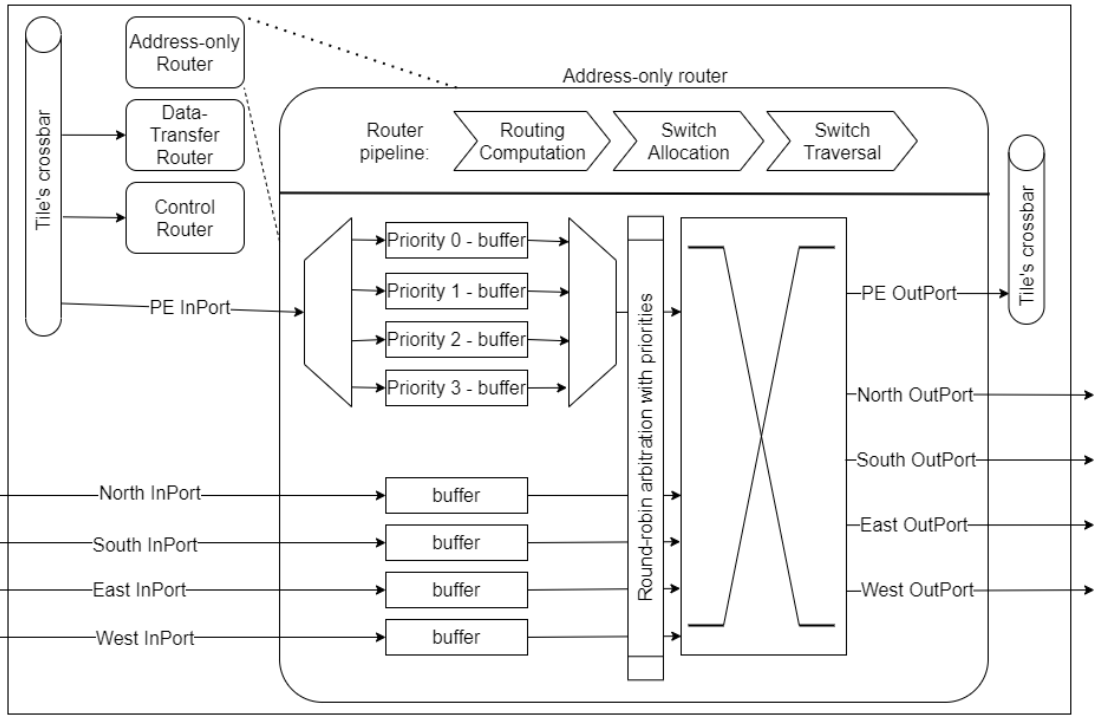



*Figure 87. Design with 3 physical networks, 4 injection queues and a round-robin with priorities arbitration policy*

In the same way as in the previous experiment, the SOMIER kernel is not evaluated due to the excessive stress it puts on injection queues. The NoC load (flits/node/cycle), represented as percentage, obtained for each executed micro-kernel can be seen in Figure 88. The Data-transfer NoC load under the '*tile private*' cache policy is the same for all the designs evaluated, except for

the option that uses virtual networks that approximately doubles it. The results of NoC load under '*fully shared*' L2 cache sharing policy are not equal for any of the evaluated micro-architecture, but the differences are imperceptible. Moreover, the load in the NoC is very low for all the kernels and designs evaluated, even using the '*fully shared*' cache policy that approximately doubles the pressure on the NoC compared to the '*tile private*' policy.

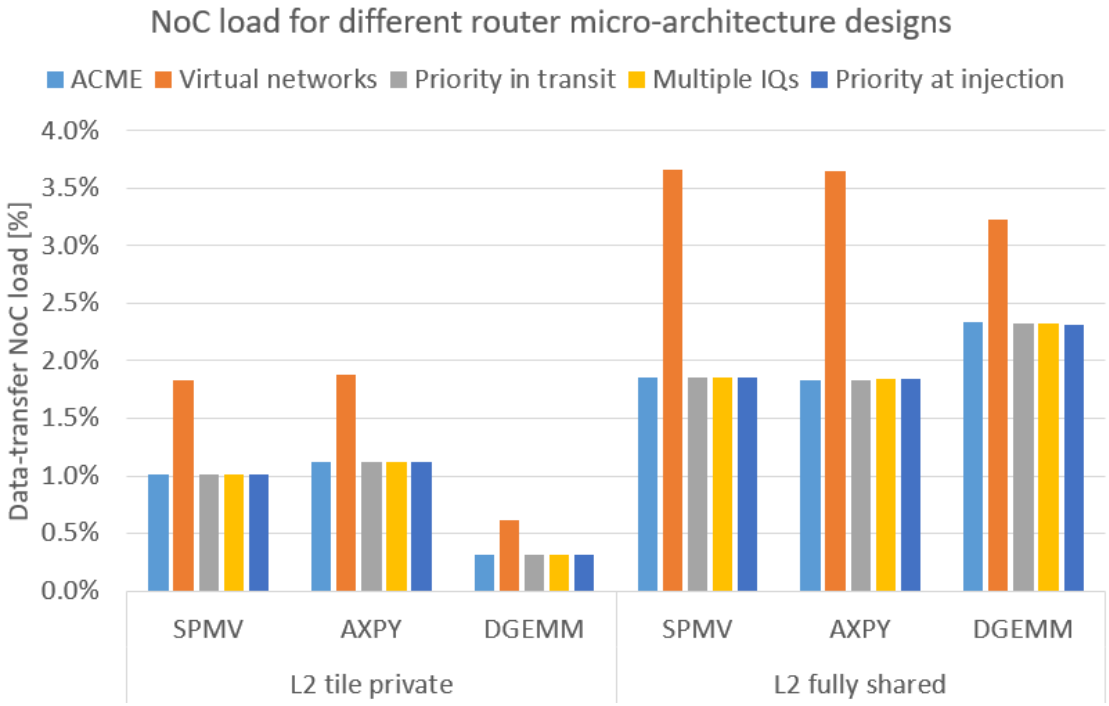



*Figure 88. Data-transfer NoC load for different router microarchitecture designs*

Figure 89 shows the kernel execution time (speedup compared to ACME) obtained for each executed kernel. The differences between obtained results are negligible under '*tile private*' policy. Similarly, those under '*tile private*' are quite imperceptible except for DGEMM kernel, which suffers a big impact on its execution time due to the added contention by the multiplexing of physical networks into virtual ones. This is not seen in NoC load (see Figure C.34) because both variables in the equation increase (packets and cycles).

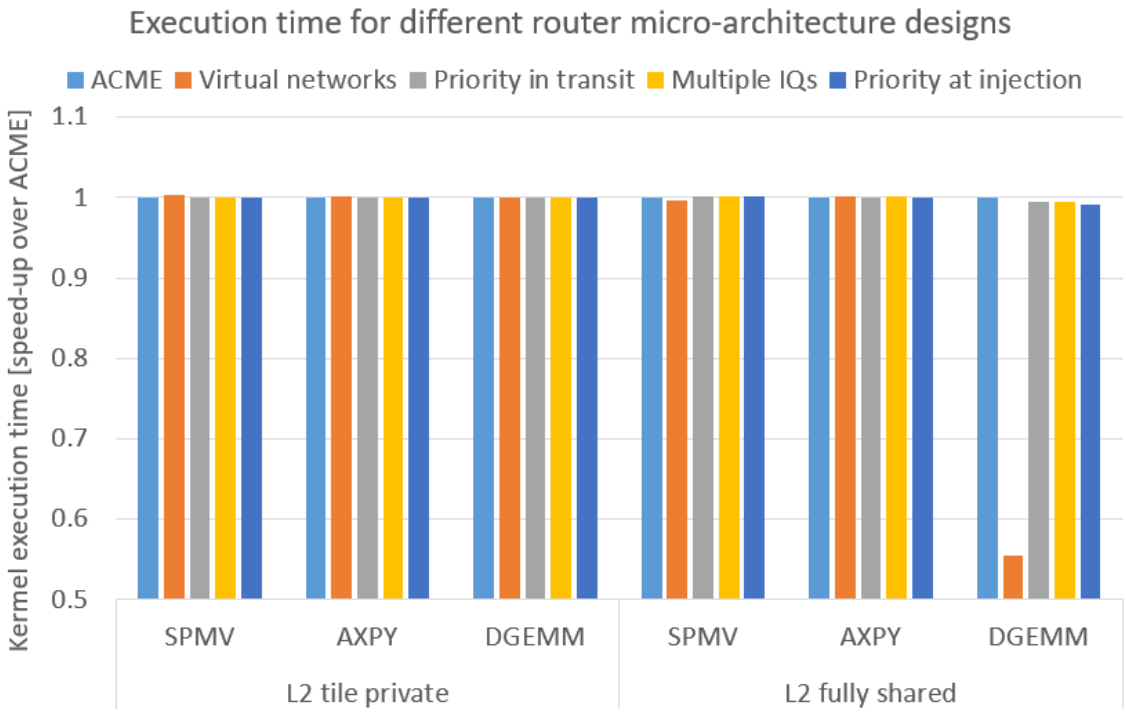



*Figure 89. Execution time comparison for different router microarchitecture designs (higher is better). Note that the y-axis scale is reduced to improve its understandability*

In summary, the results under '*tile private*' policy are the same for all the designs evaluated, except for the design with virtual networks that approximately doubles the load while it maintains the same kernel execution time for almost all kernels. The differences under '*fully shared*' are imperceptible except for DGEMM that suffers a big impact on its execution time. The result of this kernel using virtual networks is currently under analysis by the performance modeling team because it seems that this particular NoC design triggers the poor performance but none of the performance metrics of the NoC itself seem to be affected, whereas some architectural metrics, mainly regarding the cache and memory controllers, present big differences. The most plausible cause for this behavior is an effect on the ordering in which requests are delivered to the different elements of the architecture.

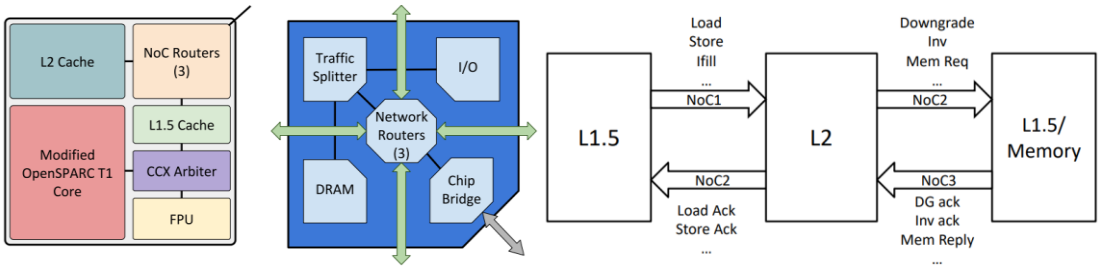
This analysis could be improved by adding a certain level of randomization to the simulator, either at the memory subsystem or at the NoC evaluation driven by BookSim, via different simulation seeds and is future work. The whole experiment must be re-evaluated using multi-flit packets and if the pressure over the NoC increases, using, for example, several threads per core, because the behavior of the different designs could change noticeably when a certain level of congestion and contention appears.

### 18.2.4. OpenPiton NoC

A theoretical analysis of the OpenPiton NoC is performed with the intent of modeling its NoC in Coyote and comparing the performance of this model, using the above-mentioned 5 micro-kernels, with the NoC designs evaluated in Section 4.2.4.

The OpenPiton processor from Princeton [CZ] is a tiled-manycore research framework that provides a memory subsystem interconnected via three NoCs, by default, in a 2D mesh topology.

The architecture of the tile and chipset, the three NoCs and a portion of the messages that travel through them are shown in Figure 90.




a) Architecture of tile and chipset [C8] b) Three NoCs and messages [C9]

*Figure 90. High level overview of architecture and the three NoCs of OpenPiton*

The messages of the memory subsystem are transported across the three networks based on: NoC1 transports requests from L1.5 to the L2, NoC2 transports messages initiated by the L2 to L1.5 or memory controller and NoC3 transports responses from the memory and write-backs from the L1.5 to the L2. To ensure deadlock-free operation across the message classes, all memory subsystem elements give different priorities to each NoC channel; the priority order is set so NoC3 has the highest, followed by NoC2 and then NoC1 with the lowest priority. This ensures that NoC3 is never blocked, so response packets are always drained. The NoCs also maintain point-to-point ordering between a single source and destination.

The three NoCs are physical networks with two 64-bit unidirectional links that use credit-based flow control. To cover the cost of the link-level flow-control mechanism, there are 4 flits deep FIFO buffers in each wormhole router. Packets are routed using dimension-ordered X-Y wormhole routing, without any virtual channel, to avoid routing and protocol deadlocks. Each router takes 1 cycle routing the packets along the same dimension, and 2 cycles when a packet must make a turn. There are no references to either input or output buffers on the OpenPiton NoC routers.

Derived from the use of 64-bit channels, some messages, like a write-back or a store request, require 11 flits, while others, like simple coherence messages, require 1 or 3 flits. Packet header formats for all message types can be found in the OpenPiton micro-architecture specification [C8].

The detailed model of NoC can be used to model the OpenPiton NoC in Coyote. The three modeled networks, adjusting their width to model the number of flits used by OpenPiton for each message type, can transport the messages modeled in Coyote similarly to OpenPiton. The dimension-ordered X-Y routing and wormhole switching mechanism are already employed. The prioritization over different NoCs can be easily modeled in Coyote as part of a future development task regarding the arbitration of the crossbar in the tiles. However, a router pipeline of one, or at maximum two, cycles seems challenging to model as part of Coyote. The transit buffer depth of 4 flits can be straightforwardly modeled. The 5 flits buffer modeled at the output of the L2 cache in OpenPiton are similar to setting this number of flits as an input buffer to injection ports. However, this cannot be set right now because of the aforementioned injection queue issue that will be fixed in the future.

### 18.2.5. Conclusions about the NoC

The above sections detail the early experiments about the NoC. To summarize those sections, the following points can be distinguished:

- The results of the latency bounds evaluation bring to light a relative insensitivity of the performance to the average packet latency. The theoretical latency of a mesh, considering the size required by ACME accelerator, can be tolerated without an impact on the performance of the system.
- The distribution of the traffic network traffic observed is good, even though there are some points to study regarding the data-mapping policies employed. Additionally, a temporal distribution analysis with more detail should be performed.
- The flit-size evaluation suggests that a narrower link than the one proposed for the Data-transfer NoC in ACME could be employed without a significant impact on the performance of the accelerator. However, an evaluation of area, energy consumption and critical path delay difference could be interesting regarding the use of multi-flit packets. Furthermore, architectural decisions regarding this topic have to consider multiple variables: the target device and technology, the scale of the final design and the working context.
- The results regarding the NoC design are not totally conclusive, and the whole experiment needs to be re-evaluated using multi-flit packets and if the pressure over the NoC increases because the behavior of the different designs could change noticeably when a certain level of congestion and contention appears.

To wrap up the NoC analysis, it seems that, right now, the NoC is not a bottleneck for the performance of ACME. However, this must be re-evaluated once the modeling of the interface between the memory subsystem and the NoC routers is improved in Coyote. In addition, the impact of the NoC on performance is expected to increase when coherence is modeled. Furthermore, there are plans to add new statistics to Coyote and to perform further experiments to improve the understandability of the behavior of the executed micro-kernels.

## 18.3. Paraver analysis

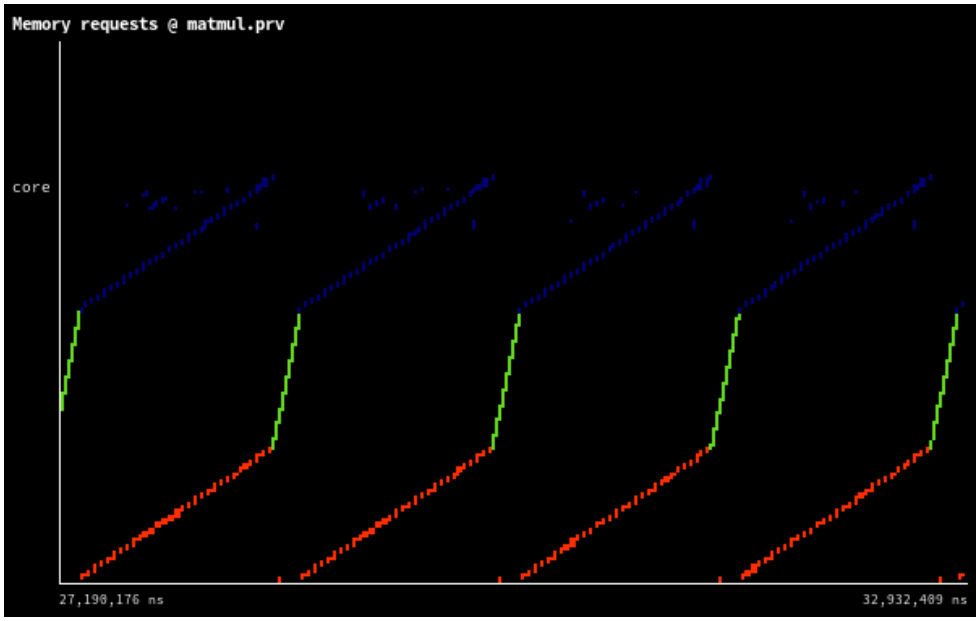
Coyote itself does not provide experimental analysis tools. In order to enable a detailed study of the low level details that in the end determine the behavior of applications, Coyote can optionally generate a trace of selected events of interest for later analysis. This trace can then be interactively explored using the Paraver visualization tool developed in BSC [C6]. The following subsections introduce different kinds of analysis to showcase the kind of visualization capabilities featured in Coyote and the insight they provide. Many of the examples will be for Coyote simulations for systems with a single core. This is for the sake of the explanation, but equivalent analysis can be performed on simulations that model an arbitrary number of cores.

### 18.3.1. Memory access patterns

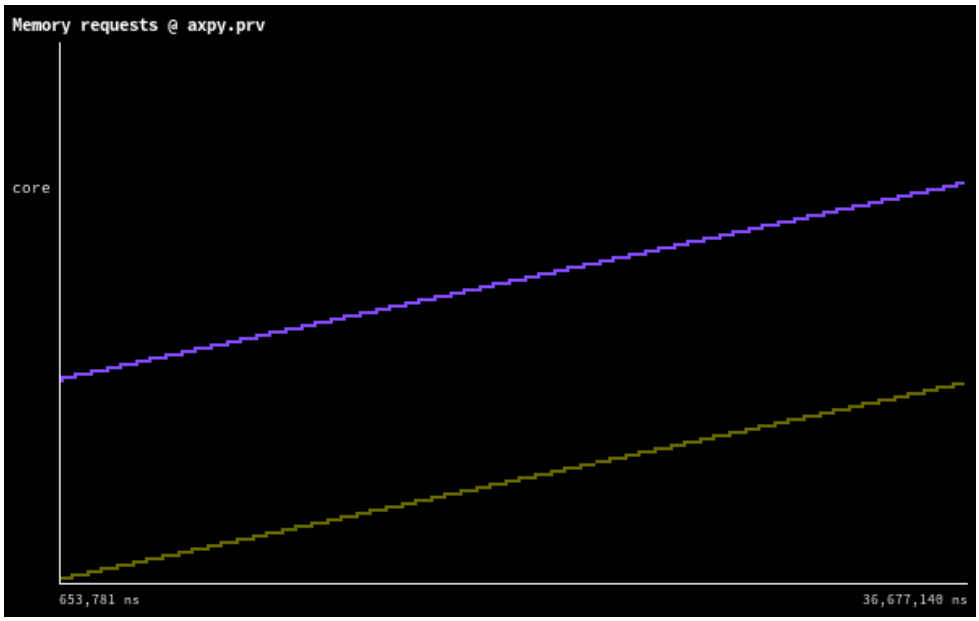
How the memory is accessed determines in the end how the application behaves. The intent of ACME is to reinvent this by means of intelligent algorithms performed on the MCPUs.

Devising the MCPU policies and evaluating their effectiveness requires mechanisms to understand how memory is accessed. To do so, Coyote has been extended to generate a trace of the memory requests. Considering that the applications that Coyote currently runs are baremetal with static data, the trace is extended with the information of the data structure associated with each memory request. This is possible because the address range for each data structure is known at compile time. However, once syscalls and dynamic memory are supported in Coyote, our plan is to extend the *malloc()* implementation to also generate this kind of information.

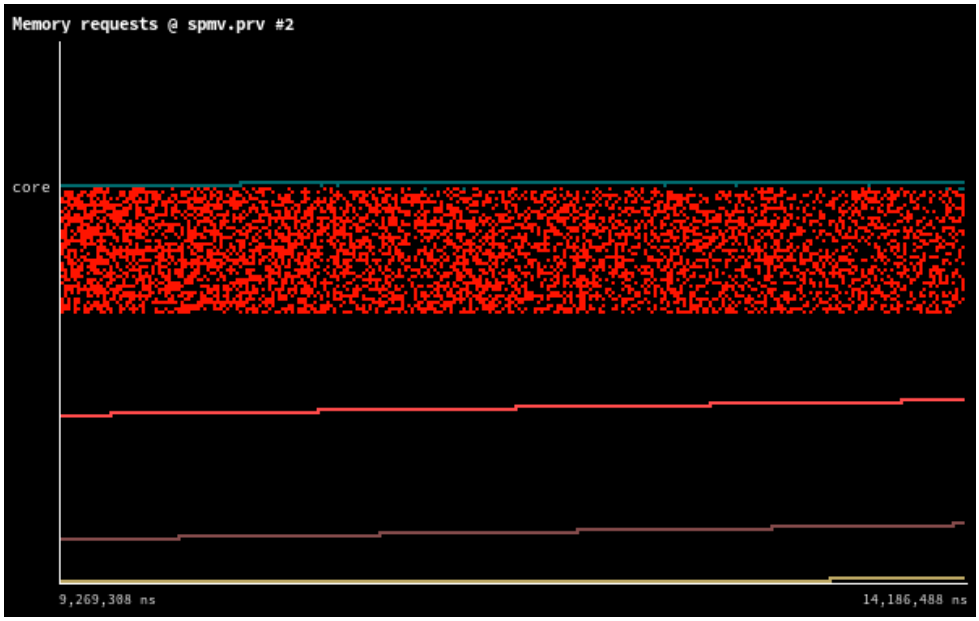
Examples of the memory access patterns of the four supported kernels are shown in Figure 91. For each of the applications, the x-axis represents time and the y-axis represents memory addresses. Each dot represents a memory request to an address, which is colored by the data structure that the request is for. If the simulation were for several cores, then several rows would be shown. In DGEMM, three different data structures are repeatedly traversed over different iterations. The red and green ones are the input matrices and the blue one is the output matrix. Detailed information, such as the correspondence between colors and data structures or the actual accessed address, can be interactively discovered using Paraver. In axpy, two data structures are iterated over sequentially in a streaming manner. In SpMV, four data structures are accessed sequentially while one is accessed unpredictably, associated to the sparsity of this kernel (input matrix was selected to very clearly show this particularly extreme behavior). In the case of somier, 4 different data structures are iterated over in order in phases. The memory access pattern for somier may be slightly different once Coyote supports coherence.



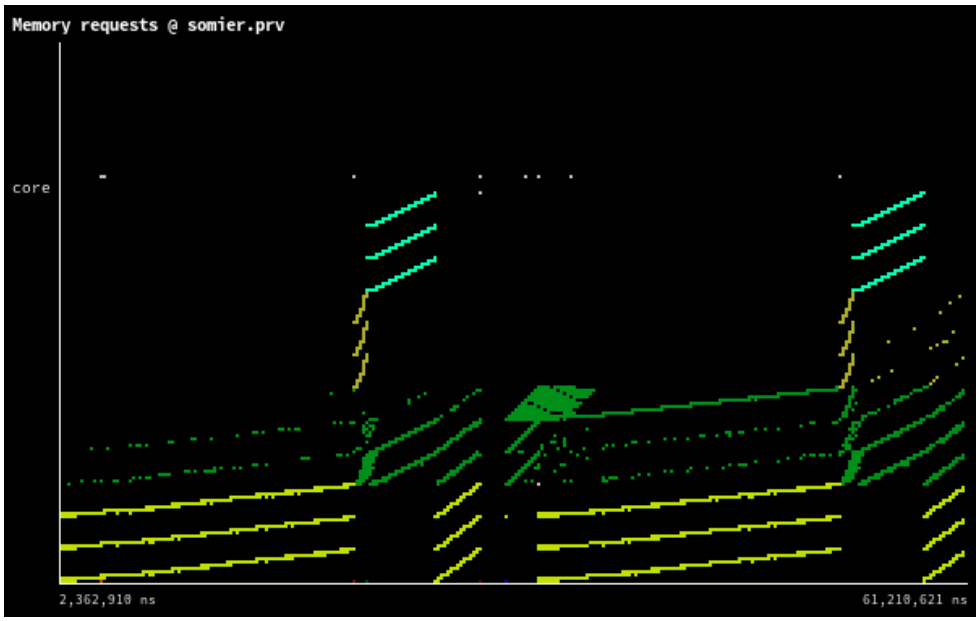

a) DGEMM


Memory requests @ axpy.prv
core
653,781 ns
36,677,140 ns

b) Axpy


Memory requests @ spmv.prv #2
core
9,269,388 ns
14,186,488 ns

c) SpMV




d) Somier

*Figure 91. Snapshot of the memory access pattern for the evaluated applications*

An equivalent analysis can be performed for other events such as L2 misses.

### 18.3.2. Stalling requests and stall lengths

Identifying how the memory is accessed is important, but all the requests are not equally relevant. The key factor to guarantee performance is keeping the scalar cores and VPUs fed, so there is no need to stall the pipeline. Consequently, being able to characterize stalls in terms of the data that they are associated with, and their length is valuable to identify potential opportunities for improvement. This section goes through a stall analysis for a single core DGEMM as an example of the visualization capabilities of Coyote. An example for a bigger system is also shown at the end of the section.

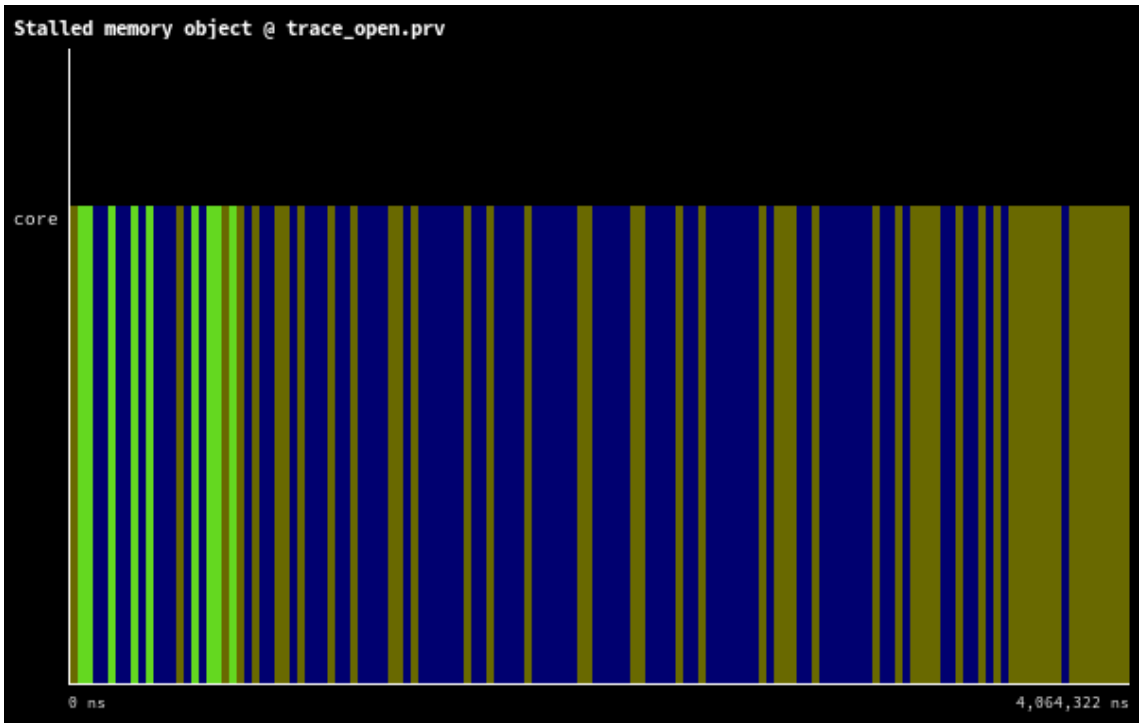



*Figure 92. Data structure stalling the core*

Figure 92 shows a visualization for the data structure that is stalling the scalar core. The x-axis represents time and each of the colored stripes represents a stall due to a particular data structure. The colors for the data structures match the ones shown in the previous Figures. The height of the stripes has no meaning in this Figure. However, as the timescale shows, the trace shown is for quite a long time, so looking at information at this level of detail necessarily hides certain information that just cannot be depicted. Figure 93 shows a zoom into a smaller region of the same trace (only ~2500 cycles instead of the 4000000 of the whole trace; please, note that the start and end times are shown at the bottom of the Figures), which indeed shows stalls that were not shown in the previous Figure 92, and also grey regions, which represent no stalls. The colors for the data structures match the ones shown in the access pattern analysis shown in Figure 91-a. Note that more colors appear on the histogram than the ones that were visible in the traffic access pattern. This is due to accesses to addresses that are infrequently used or even never accessed, such as functions that are defined in the code of the application but never called and, consequently, never fetched.

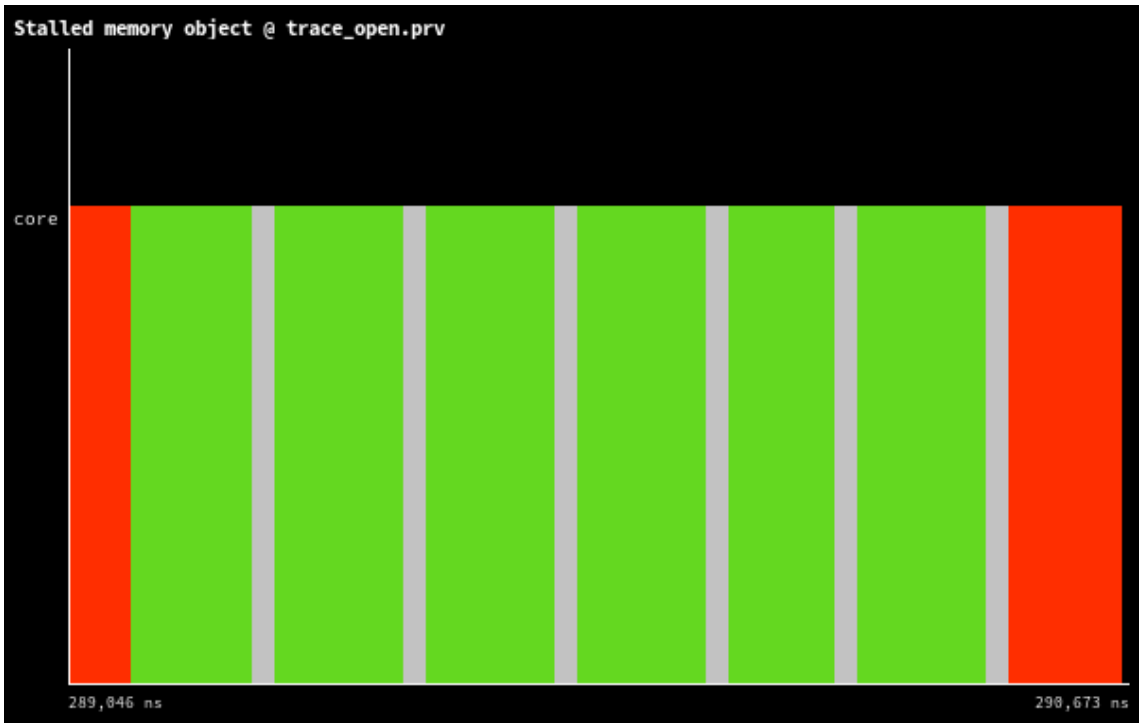



*Figure 93. Zoom into a smaller region of the visualization for the data structure stalling the core*

In order to obtain summarized views, paraver offers histograms. The one shown in Figure 94 shows the number of stalls produced by each data structure. The colors in the top row represent the data structures, matching the colors in the stall and data access pattern visualizations shown earlier. The colors in the bottom represent the number of stalls for each data structure in a scale that goes from bright green (lowest) to dark blue (highest). These colors would be equivalent to the height of the bars in a traditional histogram. This Figure clearly shows that the data structures producing the most stalls are the red and green one, which correspond to the input matrices as explained in the data access pattern section.

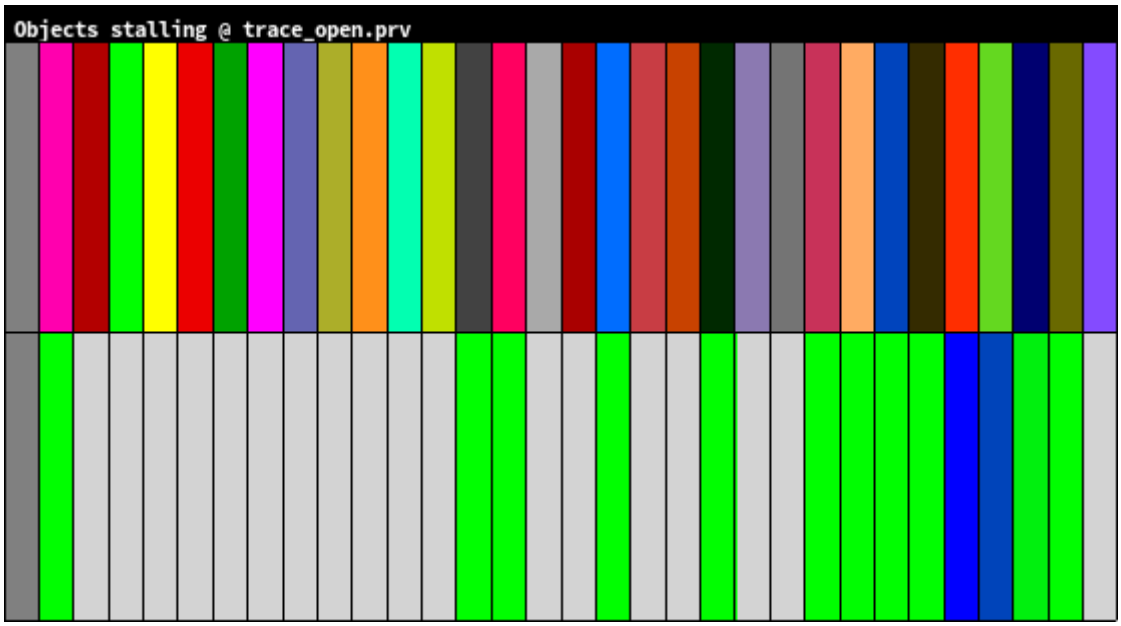



*Figure 94. Histogram for the number of stalls per data structure*

We can also look at memory stalls from a different point of view, which is their length. This is shown in Figure 95. The x-axis represents time, while the y-axis has no meaning. Colors show the length of the stalls over time, ranging from bright green (1 cycle) to dark blue (692 cycles), which represent short to long stalls. This visualization is useful to identify different phases in the kernel, but, as in the one earlier, it also hides some details unless interactively zoomed in.

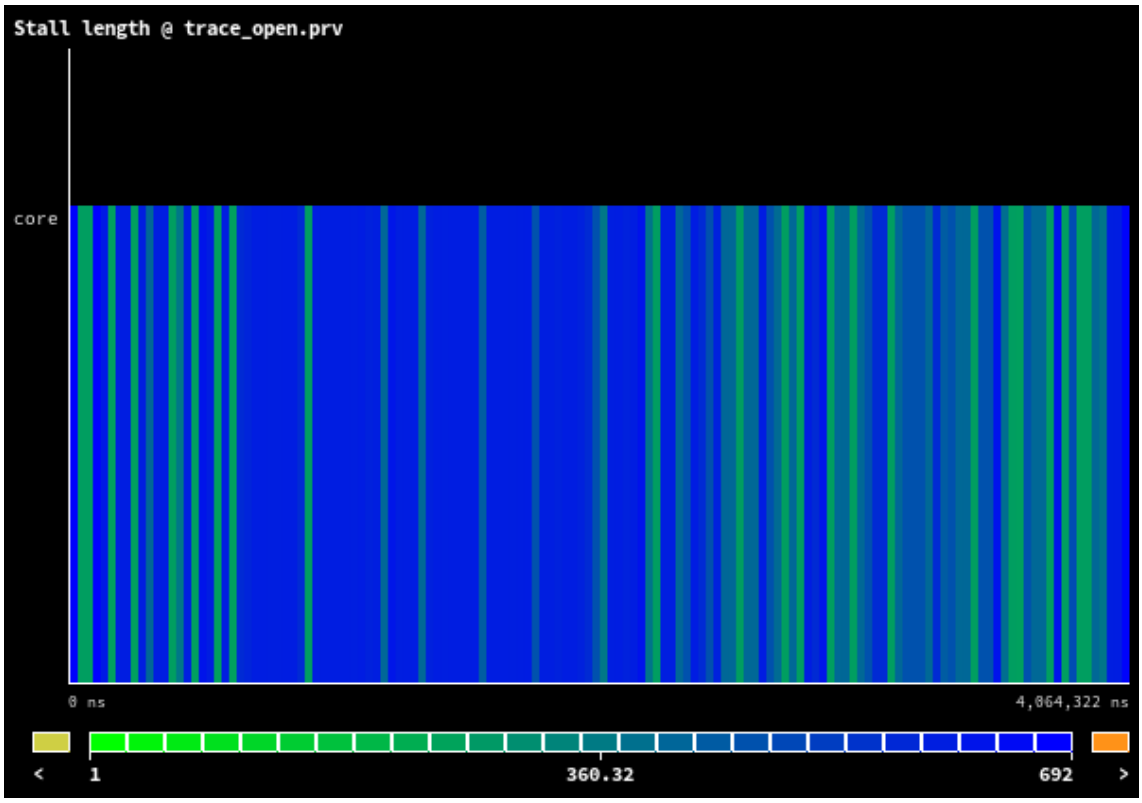



*Figure 95. Stall length representation*

Figure 96 shows a summary for the stall length to get an actual overview of the application. This histogram represents the frequency of the stalls for a group of lengths, in bins of 50 cycles. Consequently, the leftmost (and bluemost) bin represents stalls that last from 0 to 49 cycles, the next one stalls that last 50 to 99 cycles... and so on. Again, bright green means very infrequent, while dark blue means very frequent. This Figure shows that most stalls are short (0-49, which is within the limits of an L2 hit with normal contention as per the configuration in Coyote), but there is also a cluster of frequent stalls that last between 150 and 300 cycles. Missing in the L2, crossing the NoC, accessing the HBM (if there is no contention) and crossing the NoC takes approximately 300 cycles as per the configuration. Stalls with lengths between longer than 0-49 and shorter than 250-300 correspond to hits in the L2 that have to wait in queues due to contention in the L2 queue (which are shown as fairly frequent). Stalls longer than 250-300 are delayed by contention in the memory controller. These are shown as infrequent, but some of the bins are not completely bright if inspected closely.

These observations led to a deeper analysis and optimization regarding HBM banks. The result of the optimization regarding stalls is shown in Figure 97, which depicts exactly the same visualization as the one in Figure 96. As shown, the stalls now cluster nicely around the L2 hit and L2 miss+NoC+HBM+NoC latencies, being all the others infrequent. The contention in the L2 is now not present and the region for the contention in HBM is actually bright green.

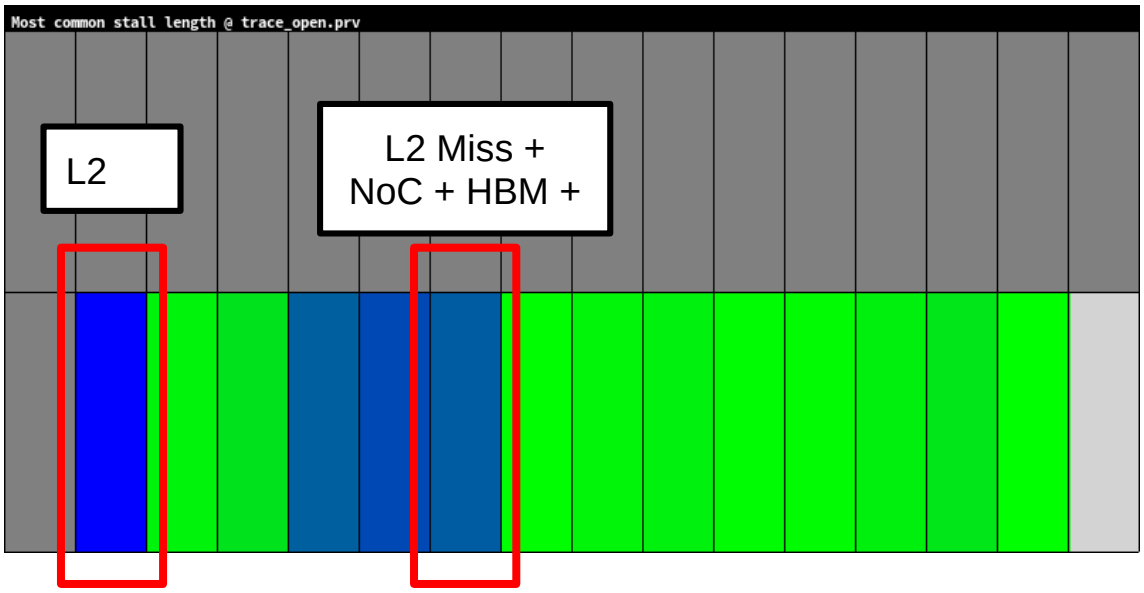



*Figure 96. Histogram of the stall lengths*

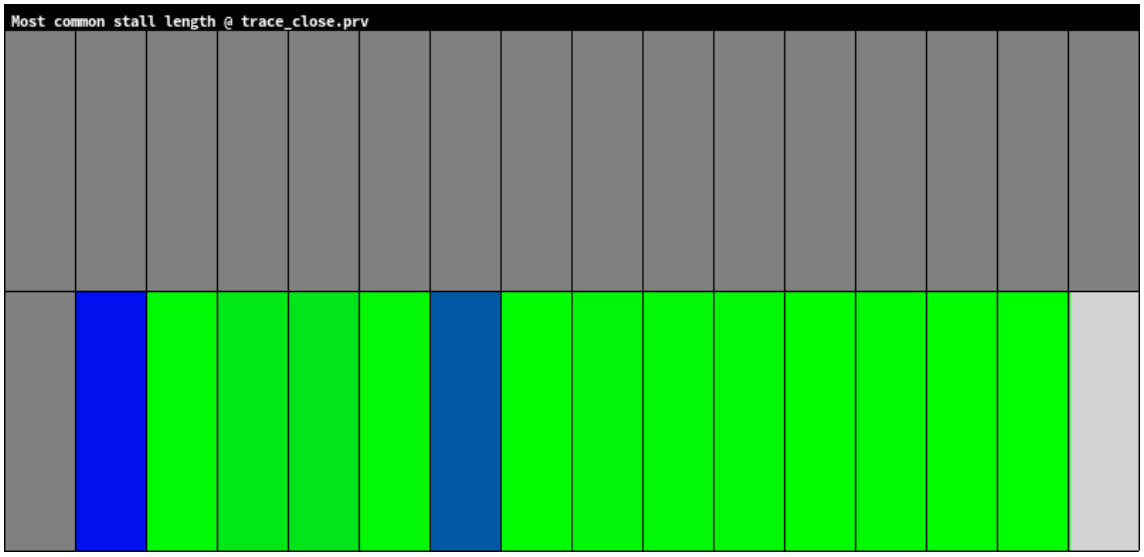



*Figure 97. Histogram of the stall lengths (after optimization)*

As an example of what these visualizations look like for bigger scale systems, Figures C.44 and C.45 show stall lengths and a stall histogram for a 128 core simulation. Now the behavior of individual cores is not visible due to the scale, but the colors in Figure 98 still let us identify clear phases of shorter stalls separated by bursts of longer ones, and some cores being affected by longer stalls more often than others. Figure 99, in turn, shows that cores have different most frequent stall lengths, depicted as the “broken” blue line. This is because this simulation is for a fully shared configuration and some cores are accessing local L2 partitions instead of remote ones more often than others. A shaded region that is related to congestion/contention is also shown near the line for the most common stall length.

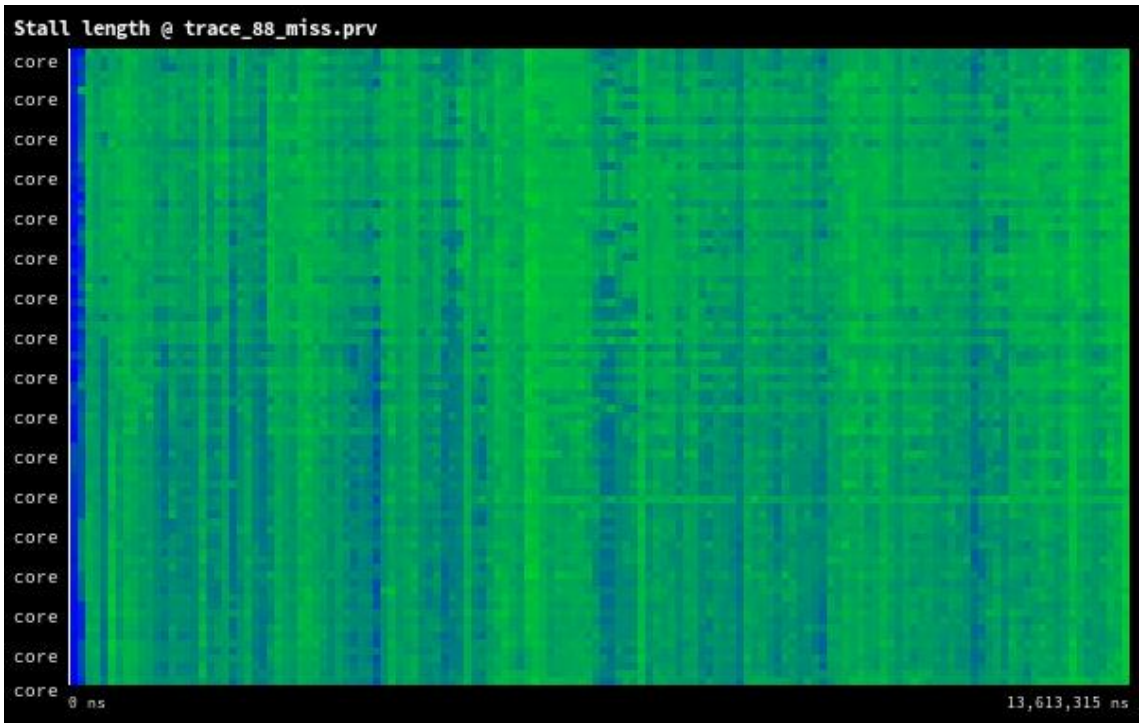



*Figure 98. Stall length representation over time for 128 cores*

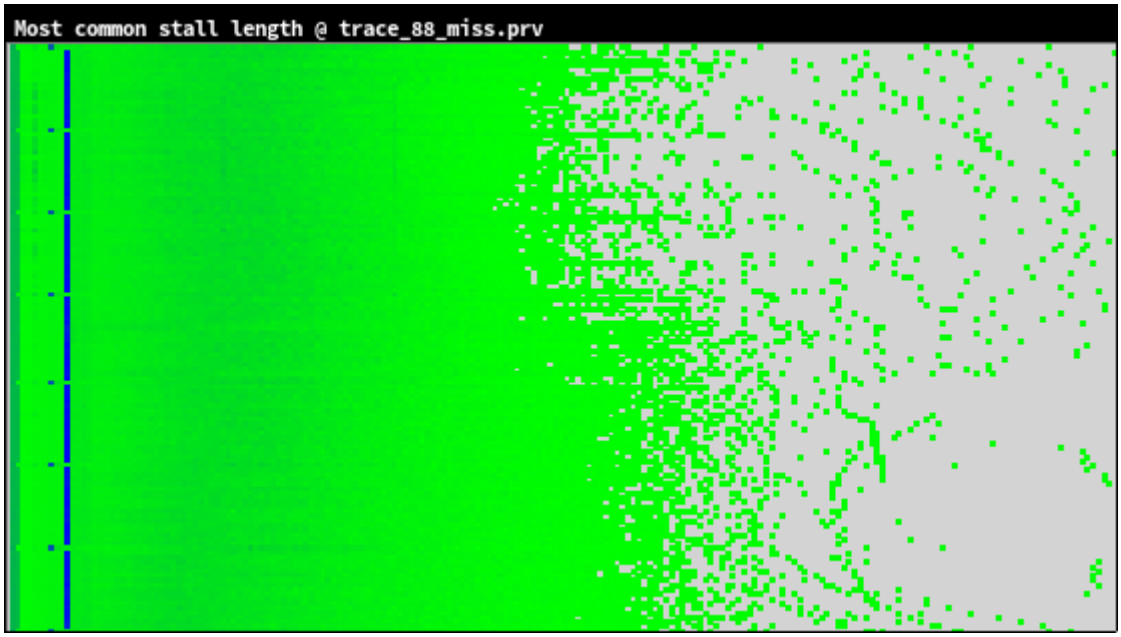



*Figure 99. Stall length histogram for 128 cores*

### 18.3.3. Cache bank usage

The results presented in section 4.1.3 showed that the cache data mapping policy used has a significant impact on the performance for DGEMM. To better understand the reason behind it, we generated traces for DGEMM using both data mapping policies and analyzed them using Paraver, generating Figures 100 and 101 for set-interleaving and page-to-bank, respectively.

Cores are represented in the y-axis, while cache banks are shown in the x-axis. The color in square [*x*,*y*] represents the time spent by the core *y* waiting for requests that will be serviced by bank *x*. Dark blue represents a long time, while bright green represents a comparatively shorter time.

As made clear by the Figures, in the set-interleaving data mapping policy, 4 out of 16 banks incur in much longer stalls. This is due to contention that in turn exhausts the MSHRs in the banks. This causes the cores to wait longer, which increases the execution time of the applications. On the contrary in the page-to-bank data policy, the stalls are evenly spread out among the banks.

Cores stalling for comparatively lesser time for a request that will be serviced by the bank

Cores stalling for comparatively more time for a request that will be serviced by the bank

Cores stalling for the maximum time for a request that will be serviced by the bank

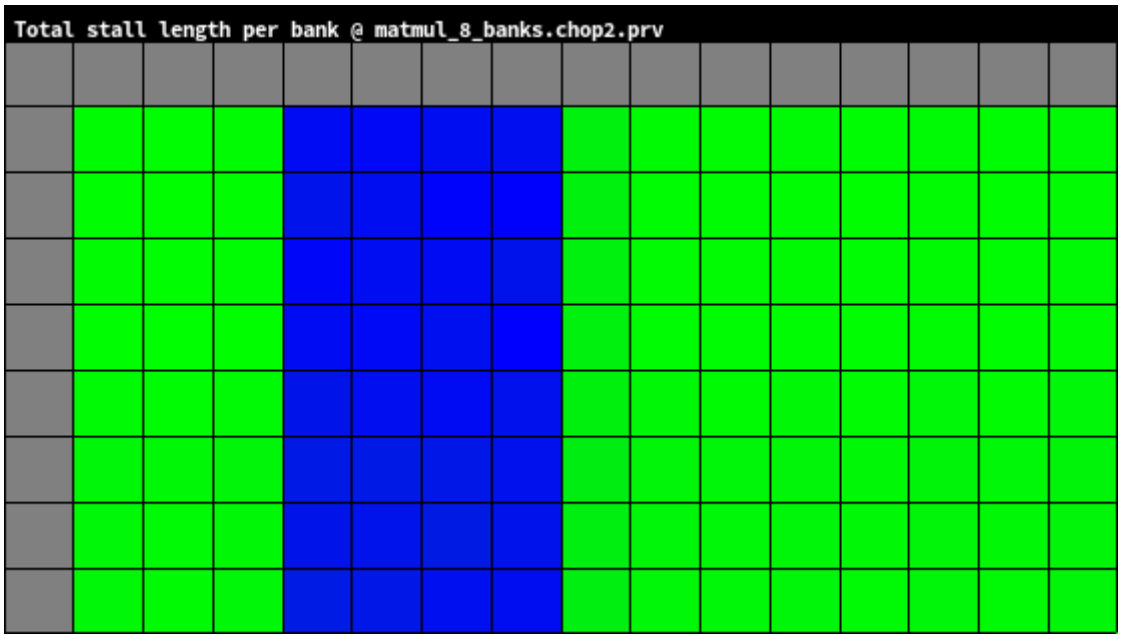



*Figure 100. Histogram for the stalls using set-interleaving*


*Figure 101. Histogram for the stalls using page-to-bank*

# 19. Conclusions and next steps

This document shows the new features added to Coyote between M9 and M18. These include new capabilities such as variable instruction latencies, memory controllers or extensive modeling for the NoC, and extended visualization features through Paraver to obtain insights about how the architecture behaves. All this, together with the ongoing work regarding the MCPU, represent a first approach to the modeling of ACME. This has served two purposes: (1) exploiting the features of Coyote for debugging purposes, and (2) getting preliminary insights of the behavior of the ACME accelerator.

Despite the fact that we are aware of the need for running more exhaustive simulations, the ones shown in this document seem to indicate that ACME is correctly dimensioned and the NoC is not one of its main bottlenecks. This notion may enable the use of narrower links. However, NoC usage needs to be re-evaluated as new features are added to Coyote and a more accurate modeling of the system is achieved.

In sum, the foundation for great simulation is set and initial insights have been drawn. More simulation is to come and, as a consequence, learning about the behavior of ACME, paving the road for successful future work regarding the other hardware-related MEEP tasks.

## 19.1. Future work

Next steps are being determined by simulation itself. Some have already been outlined throughout this document. Coherence will be modeled to better capture the behavior of applications and add new different classes of traffic to the NoC. The modeling of the interface between the memory subsystem and the NoC will be improved to prevent the issues identified during simulation. The results have also shown that address mapping and data placement policies have a significant impact on performance, which will be studied in depth, as this is a clear opportunity for

improvement at negligible cost. This also leads to data management in the MCPU. An analysis of how/when/where data is moved by the MCPU will be carried out, which will produce a new NoC traffic analysis, considering both Memory Tile-VAS Tile and Memory Tile-Memory Tile interactions. This will also generate an interesting analysis regarding how the Scratchpad is managed as part of the L2. Also as a result of the MCPU evaluation, a deeper analysis for SpMV will be carried out using different matrices, in order to generate different memory access patterns as a result of the different sparsities. An approximation of the OpenPiton NoC, which is the baseline in RTL, will be modeled in Coyote. Finally, the other major element of ACME that has not been mentioned throughout this document, the systolic array, will be modeled in Coyote.

# 21. References


[A1] lowRISC. “Untethered lowRISC v0.2”. https://lowrisc.org/docs/untether-v0.2 [Online; accessed 30-June-2021].

[A2] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, Luca Benini. Ara: A 1GHz+ Scalable and Energy-Efficient RISC-V Vector Processor with Multi-Precision Floating Point Support in 22nm FD-SOI. https://ieeexplore.ieee.org/document/8918510

[A3] Roger Espasa, Pedro Marcuello, Alberto Moreno, Sebastiano Pomata, AVISPADO - VPU interface, SemiDynamics Technology Services SL, Version 1.03. https://github.com/semidynamics/OpenVectorInterface

[A4] J. Abella et al., "An Academic RISC-V Silicon Implementation Based on Open-Source Components," 2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS), 2020, pp. 1-6, doi: 10.1109/DCIS51330.2020.9268664. https://ieeexplore.ieee.org/document/9268664

[B1] lowRISC. “Untethered lowRISC v0.2”. https://lowrisc.org/docs/untether-v0.2 [Online; accessed 30-June-2021]

[B2] https://www.gigabyte.com/Enterprise/GPU-Server/G482-Z54-rev-100#Specifications

[B3] Barcelona Supercomputing Center. 2021.https://www.bsc.es/support/Nord3-ug.pdf

[B4] Barcelona Supercomputing Center. 2021. https://www.bsc.es/support/MareNostrum4-ug.pdf

[B5] The European Processor Initiative (EPI) https://www.european-processor-initiative.eu/

[B6] Jenkins. 2021. https://www.jenkins.io

[B7] Spike - RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim

[B8] Roger Espasa, Pedro Marcuello, Alberto Moreno, Sebastiano Pomata, AVISPADO - VPU interface, SemiDynamics Technology Services SL, Version 1.03. https://github.com/semidynamics/OpenVectorInterface

[B9] Mentor. 2020. *Questa® Verification IP Reference Manual for the ARM® AMBA® 4 AXI Protocol*. 2020.1 ed. N.p.: Siemens.

[B10] Google. 2021. “riscv-dv.” https://github.com/google/riscv-dv

[B11] OpenHW Group https://www.openhwgroup.org

[B12] Multicore Simulator (MDEV) from Imperas: https://www.imperas.com/dev-virtual-platform-development-and-simulation#Feature

[C3] T.M Pinkston: Deadlock characterization and resolution in interconnection networks. Deadlock Resolution in Computer-Integrated Systems pp. 445–492 (2004)

[C4] BookSim Interconnection Network Simulator (Jun 2014), https://github.com/booksim/booksim2

[C5] The RISC-V proxy kernel https://github.com/riscv/riscv-pk

[C6] The Paraver visualization tool. https://tools.bsc.es/paraver

[C7] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. OpenPiton: An open source manycore research framework. In Proc. of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, pages 217–232. ACM, 2016. DOI: 10.1145/2872362.2872414.

[C8] OpenPiton Microarchitecture Specification. Wentzlaff parallel research group. Version 1.0. https://parallel.princeton.edu/openpiton/docs/micro_arch.pdf

[C9] OpenPiton with RISC-V Cores. A Hands-On Tutorial with the Open Source Manycore Processor. Princeton University and ETH Zürich. http://parallel.princeton.edu/openpiton/tutorial_slides/micro19/5-openpitonariane-noc.pdf

[C10] Natalie Enright Jerger; Tushar Krishna; Li-Shiuan Peh; Margaret Martonosi, On-Chip Networks: Second Edition, Morgan & Claypool, 2017.

[C11] William James Dally; Brian Patrick Towles, Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers Inc.2004.

[C12] Sang-Woo Jun, Shuotao Xu, Arvind, Terabyte Sort on FPGA-Accelerated Flash Storage, Department of Electrical Engineering and Computer Science Massachusetts Institute of Technology

[C13] e6500 Core Reference Manual. freescale.com https://www.nxp.com/docs/en/reference-manual/E6500RM.pdf

[C14] BST: Bypass Simulation Toolset. https://bitbucket.org/gicuc/booksim-unican/src

[C15] The Spike simulator https://github.com/riscv/riscv-isa-sim

# 22. List of acronyms

**ACME** Accelerator Compute and Memory Engine

**BFM** Bus Functional Model

**CGMT** Coarse-Grain Multithreading

**CPU** Computation Processing Unit

**CSR** Control Status Register

**DMA** Direct Memory Access

**DTC** Direct Transform Cosine

**DUT** Device Under Test

**DV** Design Verification

**EA** Emulated Accelerator

**EPI** European Processor Initiative

**FGMT** Fine Grain MultiThreading

**FPGA** Field Programmable Gate Array

**HBM** High Bandwidth Memory

**HPC** High Performance Computing

**IDCT** Inverse Discrete Cosine Transform

**IP** Intellectual Property

**ISA** Instruction Set Architecture

**ISS** Instruction Set Simulator

**IQ** Inverse Quantization

**JTLB** Jumbo Translation Lookaside Buffer

**LLC** Last Level Cache**LSU** Load and Store Unit

**MC** Memory Controller

**MCPU** Memory Controller CPU

**MEEP** MareNostrum Experimental Exascale Platform

**MMU** Memory Management Unit

**NoC** Network on Chip

**OCP** Open Compute Projects

**OVI** Open Vector Interface

**OS** Operating System

**RAW** Read After Write

**RISC** Reduced Instruction Set Computer Architecture

**RISC-V** an open RISC architecture managed by RISC-V International

**RTL** Register Transfer Level (Hardware Description Language)

**SA** Systolic Array

**TLM** Transaction Level Modelling

**TVM** Test Virtual Machine

**UVC** Universal Verification Component

**UVM** Universal Verification Methodology

**VAS** Vector And Systolic

**VPU** Vector Processing Unit