MEEP

MareNostrum Experimental Exascale Platform

D6.3 -Emulated accelerator second release with full capability of inter-accelerator communication

Version 1.0

Document Information

Contract Number	946002
Project Website	https://meep-project.eu
Contractual Deadline	31/12/2022
Dissemination Level	Public (PU)
Nature	Other
Author	Teresa Cervero (BSC), Daniel J. Mazure (BSC)
Contributors	Alexander Kropotov (BSC), Francelly Cano Ladino (BSC)
Reviewers	Xavier Martorell (BSC), John Davis (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.

Change Log

Version	Description of Change
V 0.1	Initial draft for internal review
V0.2	Adding section for the ACME_EA accelerator
V 1.0	Typo corrections

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 (TÜBITAK BILGEM).

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

Index

1. Executive Summary

This document is part of a collection that describes the MareNostrum Experimental Exascale Platform (MEEP) Project. This project is organized into six work packages (WP1 to WP6). Of these, WP4, 5, and 6 cover technical content. Figure 1 shows the relationship between these technical WPs.

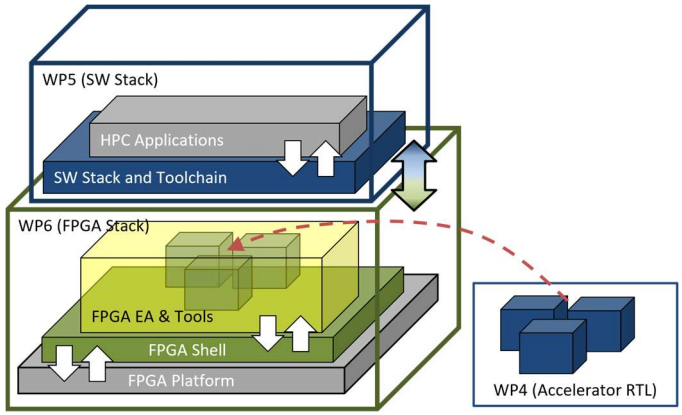

Figure 1. Layered structure of the MEEP project based on its work packages (WP4, WP5 and WP6)

- WP4 describes the Accelerated Compute and Memory Engine (ACME) accelerator architecture and its related RTL, verification, and performance modeling simulations.
- WP5 covers software, affecting 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.

As a whole, all these WPs together are focused on the development of a set of tools that enable the exploitation of the MEEP FPGA-based emulation platform as a hardware/software co-design laboratory.

In contrast to the information presented in previous deliverables, *D6.1 Platform definition and acquisition* and *D6.2 Emulated accelerator initial release* (M18), the scope of this deliverable D6.3 is going deeper into the communication capabilities, by extending the ones presented in previous deliverables, between emulated accelerators implemented on the FPGA.

D6.3 Emulated accelerator second release with full capability of inter-accelerator communication (M30). After the first release delivery of the MEEP FPGA Shell, this document presents a more advanced version of it, which provides communication capabilities for any emulated accelerator targeted in this project, but also beyond MEEP. As it is shown in Figure 2, one of the main advantages of using the MEEP FPGA Shell is the fact that it provides a flexible, scalable, and customizable wrapper for any kind of accelerator or system targeting the FPGAs.

All the achievements reported in this deliverable have been tested with different configurations of the ACME accelerator, including the second FPGA release of that accelerator (a many-core system).

In addition, the evolution of the continuous integration and continuous development (CICD) infrastructure, able to target the different FPGA designs under development, is presented.

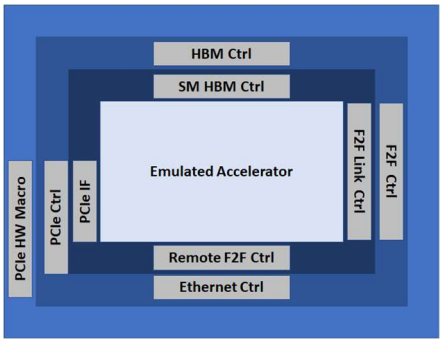

Figure 2. Layered structure of the MEEP FPGA Shell used as a communication wrapper for any targeted emulated accelerator

Three of the main elements of the FPGA Shell included in Figure 2 were explained in the deliverable D6.3: PCIe[1], HBM[2], and local FPGA-to-FPGA[3] (*F2F Ctrl & F2F Link*). As a consequence, only the remote FPGA-to-FPGA (*Ethernet Ctrl & Remote F2F Ctrl*) will be mentioned in this document. Indeed, this deliverable explains the two main communication mechanisms developed for using Ethernet: 1) Ethernet over PCIe and 2) Ethernet over QSFP.

2. Introduction

This report complements the information provided in previous deliverables *D6.1 Emulation Platform specification (M6)* and *D6.2 Emulated accelerator initial release (M18)*. On one hand, these previous documents presented and described the MEEP FPGA-Shell. A tool mainly focused on a project generation and implementation, in a reliable, reproducible, extensible, and automated way targeting different Alveo Boards. It moreover provides seamless and flexible communication capabilities to any design with the host, as part of the MEEP platform. On the other hand, this deliverable exploits the MEEP FPGA-Shell framework features, to enable the FPGA to FPGA communication through ethernet, using different mechanisms.

The FPGA Shell has been conceived as a wrapper for any design that wants to be emulated on an FPGA, initially targeting Alveo U280 and U55C but not limited to them. As it is shown in Figure 3 (left diagram), the FPGA Shell becomes a top module project (*FPGA_Shell Project*). From the hardware point of view, the final design project is composed of two main submodules: 1) the *Shell IPs,* which are all the communication IPs (i.e. Ethernet, Aurora, PCIe, etc.), and 2) the *Emulated Accelerator (EA)*, which includes the custom accelerator design to be implemented in the FPGA. In addition, from the software perspective, the final design project also includes software drivers as a layer in between the hardware and the operating system.

Figure 3 (right diagram) details the hard macros included in the U280/U55C FPGA: The PCIe macro, two QSFP ports for enabling Ethernet and FPGA to FPGA communication, two stacks

[1] PCIe: Deliverable D6.2, Section 4.1
[2] HBM: Deliverable D6.2, Section 4.2
[3] FPGA to FPGA communication: Deliverable D6.2, Section 4.3

of HBM memory with 8 channels per stack, and only for the U280, two blocks of DDR4 memory.

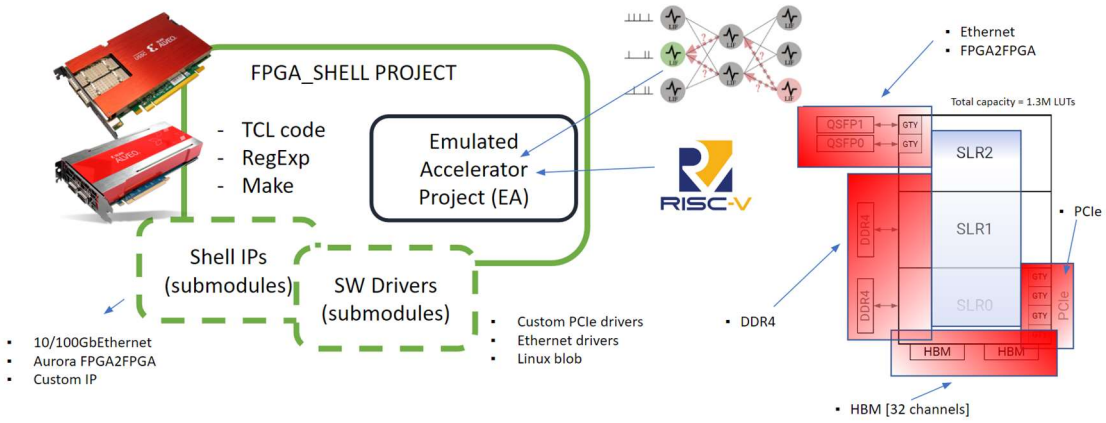

Figure 3: The MEEP-FPGA Shell project

The rest of this document is structured in the following sections:

- Section 3: *Ethernet communication*. The scope of this section is focused on the different Ethernet solutions for the Alveo boards focusing, but not exclusively, on RISC-V embedded designs. There are two fundamental implementations for Ethernet:
  - The first one establishes an Ethernet connection between a host machine (that holds the Alveo board via PCIe) and an FPGA-Embedded RISC-V with a Linux operating system.
  - The second one is a more flexible Ethernet solution, which connects the FPGA-Embedded RISC-V with any other end-point via QSFP connection.
- Section 4: *ACME Emulated Accelerator* highlights the main differences between the first and the second release of the accelerator implemented on the FPGA. This latest release has been used for testing the communication capabilities of the FPGA Shell.
- Section 5: *Continuous Integration and Continuous Development process for the FPGA flow*. This section describes the development of a Continuous Integration Continuous Deployment (CICD) process that automates the whole FPGA flow for any design project that targets FPGAs used in the MEEP project.
- Section 6: *MEEP contributions to other projects from the FPGA perspective*. Finally, this section describes how the results of the FPGA efforts, in the context of the MEEP project, have been shared with other projects. WIth this, we demonstrate the long-term vision of the MEEP infrastructure, in terms of FPGA-based emulation platform, and its associated tool kit.

3. Ethernet Development

There are two main approaches for the Ethernet development:

1) Ethernet over PCIe, which enables an Ethernet based communication between the host and the FPGA board via PCIe, and

2) Ethernet over QSFP, which enables the communications between the FPGA-embedded OS and any other Ethernet system via QSFP.

As MEEP infrastructure targets up to 12 nodes, with 8 FPGA boards each, these two solutions combined provide full Ethernet connectivity on the system depicted in Figure 4.

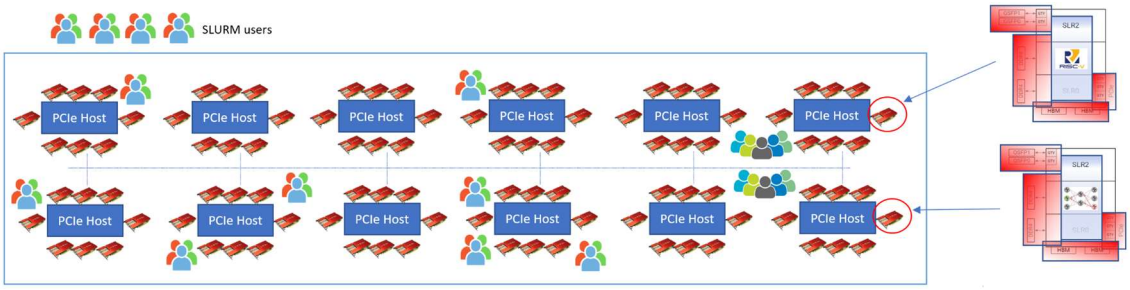

Figure 4: The MEEP nodes. 12 host with 8 FPGA boards connected via PCIe to each of them

Ethernet via PCIe provides full connectivity to all the 8 FPGA boards connected within the same node, whereas Ethernet via QSFP provides connectivity between FPGAs in different (or same) nodes. Full connectivity is also guaranteed using 100 Gb Ethernet switches that are not shown in the diagram for simplicity. In addition to this, the FPGA Shell provides an homogeneous and scalable communication infrastructure for all the FPGAs, independently from the EA architecture they are instantiating (represented on the right side of Figure 4).

3.1 Ethernet over PCIe

Configuring Ethernet over PCIe enables the communication between the host to any of the PCIe-connected FPGAs in the cluster. Figure 5 shows the configuration of a node within the MEEP large-scale FPGA-based machine.

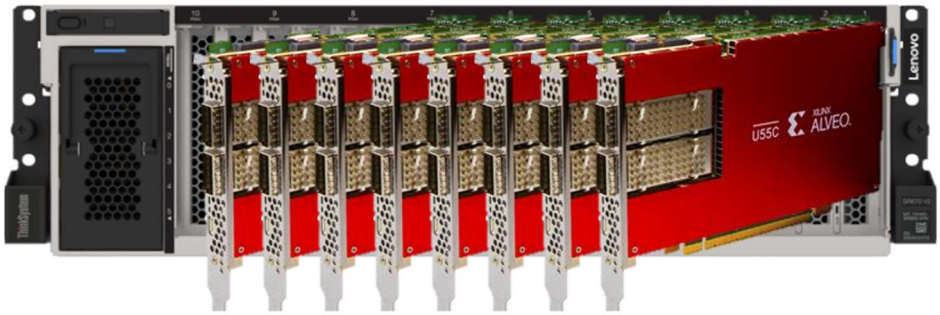

Figure 5: Large-scale MEEP infrastructure node with 8 U55C FPGAs

The solution consist of two main components, as it is depicted in Figure 6:

1) The host driver (*QDMA+ONIC driver kernel-level*): The final version of the driver has been developed within the MEEP project. It extends the QDMA driver provided by Xilinx [X.QDMA], also using the Xilinx *ONIC* project as reference [X.ONIC]. The QDMA driver creates an additional interface to the default QDMA/PCIe driver, especially dedicated for the ethernet communication. The modified QDMA driver (ONIC driver, from now on) sets up a Linux *netdev* device (the equivalent to an ethernet interface), and creates two queues to handle both the regular QDMA to memory transactions and the ethernet based transactions.
2) The embedded RISC-V driver (*MEEP-ETH device driver kernel-level*): The FPGA implementation consists of an embedded RISC-V processor with a Shared memory

space (*Data-xchg area*) for enabling the interchange of messages between the host (*Host side*) and the design implemented on the FPGA (*RISC-V side*).

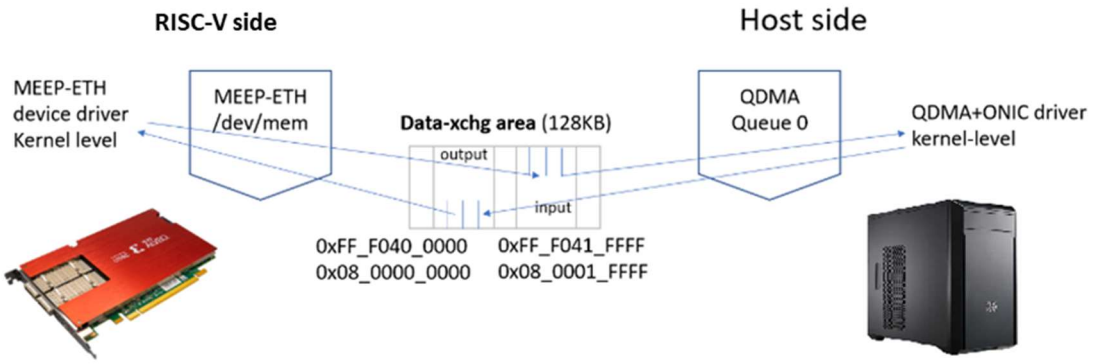

Figure 6: Shared memory diagram. The PCIe host and the embedded RISC-V processor use an exchange memory area for Ethernet packets

On one hand, the ONIC driver sets properties for the *netdev* device that implies direct connection between, at least two points, being the host one of them:

- *Point-to-point* communication: This is the basic communication property, since it establishes a direct communication mechanism between two devices.
- *No-arp* communication: In an environment where there is only a point-to-point connection, there is no need to resolve the MAC addresses, so the ARP protocol is disabled.
- *Multicast* communication: This property allows sending the same message across multiple devices at the same time.

The ONIC drivers are stored in the corresponding repository[4].

On the other hand, the embedded RISC-V Linux driver is included as part of an SDK repository[5]. This repo contains, not only this driver, but also other elements to complete a solution for design, develop and debug on an FPGA running Linux on a RISC-V core. Thus, this repository has the necessary submodules to build a binary that boots Linux:

1. Open SBI
2. Linux Kernel (Xilinx), including a set of drivers in which the modified version of ONIC for enabling the Ethernet is one of them.
3. Buildroot

In addition to the three submodules, the repository includes: 1) the embedded Ethernet-Over-PCie drivers (RISC-V side), 2) the Ethernet-over-QSFP drivers, and 3) an automatic mechanism based in GNU Make to generate different Linux configurations, depending on the features of the targeted RISC-V processor (i.e. Ariane, Lagarto Hun).

[4] https://gitlab.bsc.es/meep/FPGA_implementations/AlveoU280/xilinx_pcie_drivers
[5] https://gitlab.bsc.es/meep/meep-os/lagarto-openpiton-sdk/-/pipelines

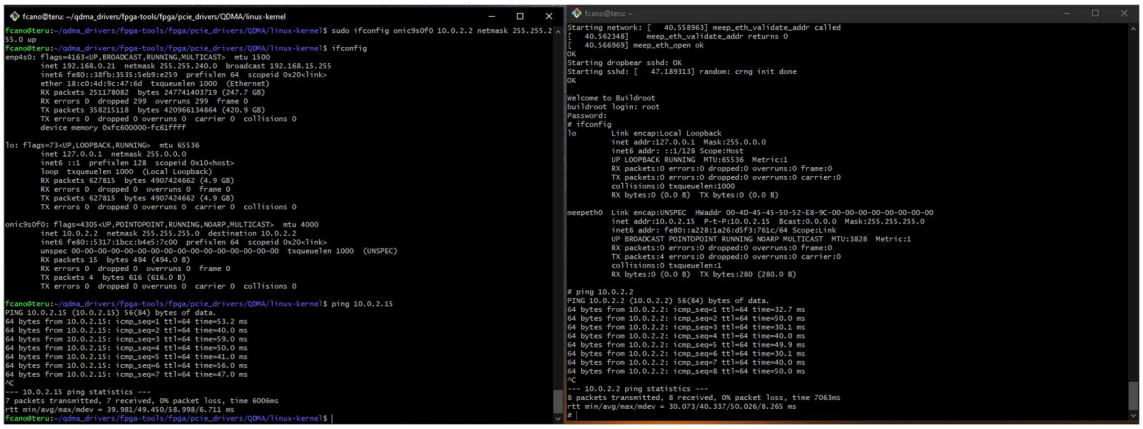

Figure 7: A point-to-point communication, using Ethernet connection over PCIe, between the host side (left side of the figure) and the RISC-V side on the FPGA (right side of the figure)

The Ethernet over PCIe mechanism has been tested and validated by running several tests that enables point-to-point communication between a host and a RISC-V design implemented on the FPGA. An example of these experiments is shown in Figure 7. Moreover, the experiments have been validated for the two FPGAs used in the MEEP project, Alveo U280 and U55C.

```
dmazure@teru ~/git_repo/xilinx_pcie_drivers/QDMA/linux-kernel/bin (meep-onic)
$ iperf3 -uc 10.0.2.15
Connecting to host 10.0.2.15, port 5201
[  5] local 10.0.2.2 port 39965 connected to 10.0.2.15 port 5201
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec   129 KBytes  1.06 Mbits/sec  35
[  5]   1.00-2.00   sec   129 KBytes  1.06 Mbits/sec  35
[  5]   2.00-3.00   sec   129 KBytes  1.06 Mbits/sec  35
[  5]   3.00-4.00   sec   125 KBytes  1.03 Mbits/sec  34
[  5]   4.00-5.00   sec   129 KBytes  1.06 Mbits/sec  35
[  5]   5.00-6.00   sec   129 KBytes  1.06 Mbits/sec  35
[  5]   6.00-7.00   sec   125 KBytes  1.03 Mbits/sec  34
[  5]   7.00-8.00   sec   129 KBytes  1.06 Mbits/sec  35
[  5]   8.00-9.00   sec   129 KBytes  1.06 Mbits/sec  35
```

Figure 8: iperf3 measured bandwidth for Ethernet over PCIe for communication between the FPGA and the local host.

Regarding the performance of this solution, as depicted in Figure 8, when using *iperf3* tool [IPERF3] the maximum measured bandwidth shows a value of 1.05Mb/s on average.

Our current solution focuses on correctness over performance. Future solutions will include performance enhancements which are still under development. From the HW point of view, we can enable interrupts to make the process more efficient.

3.2 Ethernet over QSFP

The Ethernet over QSFP solution refers to Ethernet links between FPGAs embedded RISC-V systems through the QSFP physical connectors. Unlike the Ethernet over PCIe solution, which involves development on the host side, this one only involves development on the FPGA side.

The Ethernet QSFP design uses the following pieces:

1) Xilinx's DMA based solution, with Scatter-Gather mode enabled. Although this Scatter-Gather mode is not required, it boosts performance when used. It creates a

queue of packet fragments that the DMA can manage itself without issuing interrupts, without CPU intervention, until the descriptor queue is finished.

2) A RISC-V system capable of mapping cacheable and non-cacheable regions.
3) A proper device-tree configuration, to map the different interfaces for allowing the operating system, Linux in our case, to interface with them.

Cache coherency is a challenge in combination with DMA-based solutions [DMA]. RISC-V ISA does not include the definition of flush or invalidate instructions. This implies relying on the cache hierarchy of the system to provide a mechanism to deal with those two situations.

A potential side effect of using DMA, when dealing with processors with data cache, is the possibility of data corruption problems because data in cache is no longer coherent with respect to the main memory. The problem occurs when a DMA transfer changes the contents of main memory that has been cached by the processor. The data stored in the cache will be the previous value. Being more precise, the problem happens when a DMA transfer changes the content of main memory addresses, which have been previously cached by the processor. However, when the cache is flushed the stale data will be written back to the main memory overwriting the new data stored by the DMA. The end result is that the data in main memory is not correct. A similar problem is faced when main memory starts transferring data to the DMA, and in parallel, the processor updates data in cache. In that case, there would be a data mismatch between cached data and main memory until a cache flush is executed. In this context, the data transferred from main memory to the DMA will be stale data instead of data updated by the processor. As with any memory corruption problem, these situations are notoriously difficult to track down because the behavior of the system may be unpredictable.

In order to avoid this cache coherence problem several techniques have been implemented in practice in different hardware system designs. One of them is to include *bus snooping* or *cache snooping* mechanisms as part of the system. The snooping hardware notices when an external DMA transfer refers to main memory using an address that matches data in the cache, and either flushes/invalidates the cache entry, or "redirects" the transfer so that the DMA transfers the correct data, and the state of the cache entry is updated accordingly. In systems with snooping, DMA drivers don't have to worry about cache coherency.

Another solution is more software oriented, by modifying the device driver to: 1) explicitly flush or invalidate the data cache before a transfer is initiated, depending on the address involved in the transaction; or 2) make data buffers available to bus mastering peripherals. This approach complicates the software and causes more transfers between cache and main memory; however, it does allow the application to use any arbitrary region of cached memory as a data buffer.

A third solution, and the approach we have implemented, is a combination of the above solutions. The driver explicitly issues flush commands to the hardware, which can interpret those commands to effectively flush the cache. OpenPiton [OP], which is the SoC framework we are using for our EA system, allows this approach because it has an internal flushing mechanism.

Summarizing, we have studied three possibilities:

1. The Kernel driver copies data from cached memory to uncached memory (DMA pool) and updates the DMA registers accordingly. This adds extra copies and likely wastes most of the DMA performance.
2. Flushing with OpenPiton. As RISC-V does not have any specific instruction for flushing the caches. Then, we have studied how to achieve this with OpenPiton itself, that documented several non-tested mechanisms to accomplish this [FLUSH]:
3. Add DMA coherency to OpenPiton: The DMA IP could eventually generate eviction/flushes in the OpenPiton cache hierarchy depending on the address the driver writes in the DMA registers, or similarly, depending on the AXI *AWADDR/ARADDR* where the *S2MM* (Stream to Memory Map) or *MM2S* (Memory Map to Stream) interfaces want to access.

The first solution is the easiest one but inefficient. The second solution is affordable, and it was what we finally implemented. The third solution would be the best from the performance point of view, adding extra features to OpenPiton that would be welcomed by the community. This is a significant amount of work and left to future (community) work.

The ACME_EA needs to define a shared memory space between the DMA engine and the CPU. Linux will use it as a memory pool to store the buffer descriptors, but it does not store the Ethernet packets. These descriptors will be used by the Statter-Gather engine in the Xilinx DMA IP to generate the necessary transfers and the consequent interrupts.

The DMA pool needs to be placed in a non-cacheable region in the system to avoid coherency issues at the buffer descriptor level. Still, Linux internally manages the Ethernet packet in the main memory, and coherency issues will eventually appear when processing them.

Finally, as mentioned above, we have implemented a flush mechanism shared both at the software driver and hardware level. The driver does specific writes to the cache hierarchy system, which has registers to perform flushes based on either cache lines or physical addresses. We have seen how the flush mechanism is unavoidable for making the Ethernet point-to-point communication works without errors. A successful test of this mechanism is shown in Figure 9. This image represents a ping command between two different U55C FPGAs.

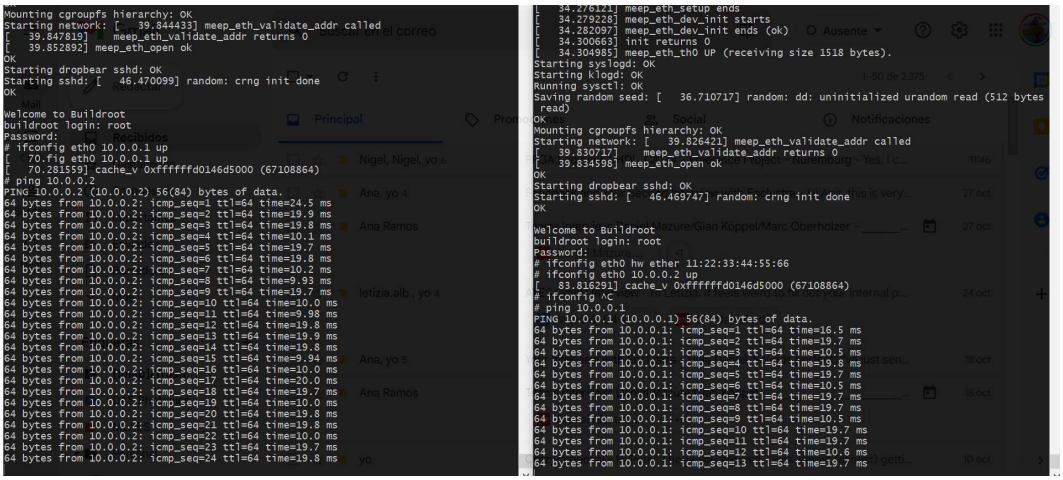

Figure 9: Ping between two embedded FPGA (U55C) -RISC-V machines

3.2.1 10Gb Ethernet solution

The 10GbE solution is an Ethernet solution based on two main components:

1) Xilinx DMA [X.DMA].
2) An open-source 10GbE IP from Aleix Forencich [10GAF].

Hierarchy

Name	CLB LUTs (1303680)	CLB Registers (2607360)	CARRY8 (162960)	F7 Muxes (651840)	F8 Muxes (325920)	CLB (162960)	LUT as Logic (1303680)	LUT as Memory (600960)	Block RAM Tile (2016)
system_top	137306	123398	2234	6233	1650	26563	128692	8614	149
meep_shell_inst (meep_shell)	65045	74795	681	2182	456	13848	58326	6719	95
APB_rst_or (meep_shell_APB_rst_or_0)	1	0	0	0	0	1	1	0	0
axi_gpio_0 (meep_shell_axi_gpio_0_0)	38	52	0	0	0	11	38	0	0
axi_xbar_pcie (meep_shell_axi_xbar_pcie_0)	2410	2868	10	40	0	630	2305	105	0
axi_xbar_pcie_lite (meep_shell_axi_xbar_pcie_	0	0	0	0	0	0	0	0	0
clk_wiz_1 (meep_shell_clk_wiz_1_0)	0	0	0	0	0	0	0	0	0
Ethernet10Gb_qsfp1 (Ethernet10Gb_qsfp1_ir	8909	10875	50	92	32	2104	8656	253	12
axi_dma_0 (meep_shell_axi_dma_0_0)	4021	6855	34	59	0	1095	3818	203	9
axi_interconnect_0 (meep_shell_axi_interc	1784	2144	10	32	32	390	1736	48	0
MEEP_10Gb_Ethernet_qsfp1 (meep_shell_	2866	1477	6	1	0	596	2866	0	3
proc_sys_reset_eth (meep_shell_proc_sys_	17	37	0	0	0	10	16	1	0
smartconnect_0 (meep_shell_smartconnec	223	362	0	0	0	92	222	1	0
xlconcat_irq (meep_shell_xlconcat_irq_0)	0	0	0	0	0	0	0	0	0

Figure 10: Utilization report highlighting the 10GbE module

The resource utilization report for this solution is shown in Figure 10. As it might be seen, the main source of resource consumption is the Xilinx DMA, which uses around 4K LUTs, whereas the 10GbE IP uses around 2K LUTs, which can be considered a low amount of LUTs.

```
# iperf3 -uc 10.0.0.2 -bidir
Connecting to host 10.0.0.2, port 5201
[  5] local 10.0.0.1 port 34865 connected to 10.0.0.2 port 5201
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.03   sec   339 KBytes  2.70 Mbits/sec  240
[  5]   1.03-2.03   sec   325 KBytes  2.68 Mbits/sec  230
[  5]   2.03-3.02   sec   325 KBytes  2.69 Mbits/sec  230
[  5]   3.02-4.00   sec   325 KBytes  2.70 Mbits/sec  230
[  5]   4.00-5.03   sec   339 KBytes  2.70 Mbits/sec  240
[  5]   5.03-6.02   sec   325 KBytes  2.70 Mbits/sec  230
[  5]   6.02-7.00   sec   325 KBytes  2.70 Mbits/sec  230
[  5]   7.00-8.03   sec   339 KBytes  2.70 Mbits/sec  240
[  5]   8.03-9.02   sec   325 KBytes  2.70 Mbits/sec  230
```

Figure 11: iperf3 measured bandwidth for the 10 GbE IP for FPGA to FPGA communication.

To test the bandwidth, the *iperf3* tool has been used, and according to the data shown in Figure 11, the measured bandwidth reflects 2.70Mb/s. Again, the focus is on correctness and not performance. There are several performance improvements that can be made in the future to increase the bitrate.

3.2.2 100Gb Ethernet solution

The 100GbE solution is an Ethernet solution based on two main components:

1. Xilinx DMA
2. Xilinx CMAC 100GbE IP [X.CMAC]. This IP is a hard macro in the Alveo U280 and U55C FPGAs.

The resource utilization, depicted in Figure 12, shows that Xilinx DMA consumes rather more resources for 100Gb IP (20K LUTs vs 4K LUTs). As expected, to support 100Gb data rates, the DMA engine needs to be more productive, providing concurrent transmission of transmit

(Tx) and receive (Rx) data with no stalls from/to AXI-MM to/from AXI-Stream channels through 512-bit bus width at 322MHz.

Hierarchy

Name	CLB LUTs (1303680)	CLB Registers (2607360)	CARRY8 (162960)	F7 Muxes (651840)	F8 Muxes (325920)	CLB (162960)	LUT as Logic (1303680)	LUT as Memory (600960)	Block RAM Tile (2016)	URAM (960)	DSPs (9024)	Bonded IOB (624)
clk_wiz_1 (meep_shell_clk_wiz_1_0)	0	0	0	0	0	0	0	0	0	0	0	0
Eth100GbSyst_w_hbm (meep_shell_Eth100GbSyst_w_hb	39858	62585	1214	1145	286	12905	36729	3129	38	0	0	0
inst (meep_shell_Eth100GbSyst_w_hbm_0_Eth_CMAC	39858	62585	1214	1145	286	12905	36729	3129	38	0	0	0
axi_reg_slice_rx (meep_shell_Eth100GbSyst_w_hbr	725	2499	0	0	0	1112	388	337	0	0	0	0
axi_reg_slice_tx (meep_shell_Eth100GbSyst_w_hbr	644	2221	0	0	0	843	340	304	0	0	0	0
axi_timer_0 (meep_shell_Eth100GbSyst_w_hbm_0_	292	240	40	0	0	69	292	0	0	0	0	0
concat_intc (meep_shell_Eth100GbSyst_w_hbm_0_	0	0	0	0	0	0	0	0	0	0	0	0
ctl_tx_send_idle (meep_shell_Eth100GbSyst_w_hbi	0	0	0	0	0	0	0	0	0	0	0	0
ctl_tx_send_lfi (meep_shell_Eth100GbSyst_w_hbm_	0	0	0	0	0	0	0	0	0	0	0	0
ctl_tx_send_rfi (meep_shell_Eth100GbSyst_w_hbm	0	0	0	0	0	0	0	0	0	0	0	0
dma_connect_rx (meep_shell_Eth100GbSyst_w_hb	1146	983	8	0	0	296	741	405	0	0	0	0
dma_connect_sg (meep_shell_Eth100GbSyst_w_hl	1319	1550	16	32	0	290	850	469	0	0	0	0
dma_connect_tx (meep_shell_Eth100GbSyst_w_hb	695	1355	8	0	0	265	340	355	0	0	0	0
dma_loopback_fifo (meep_shell_Eth100GbSyst_w_	391	1248	0	0	0	227	55	336	0	0	0	0
eth100gb (meep_shell_Eth100GbSyst_w_hbm_0_E	8958	28951	1098	596	214	4504	8638	320	0	0	0	0
eth_dma (meep_shell_Eth100GbSyst_w_hbm_0_Et	19851	14844	38	413	37	4245	19587	264	21	0	0	0
eth_loopback_fifo (meep_shell_Eth100GbSyst_w_h	389	1248	0	0	0	337	53	336	0	0	0	0
ext_rstn_inv (meep_shell_Eth100GbSyst_w_hbm_0	1	0	0	0	0	1	1	0	0	0	0	0

Figure 12: Utilization report highlighting the 100GbE module

The performance results obtained for the 100Gb solution under Linux are measured by using the diagnostic utility *iperf3*. This tool directly accesses the IP by using the Ethernet driver. At this point in time the driver is focused on correctness instead of performance:

- Diagnostic utility on non-cached HBM-based DMA pool: ~ 10 Gb/s
- Diagnostic utility on cached HBM-based DMA pool: ~ 415 Mb/s
- Linux *iperf3* utility on non-cached HBM-based DMA pool: ~1 Mb/s

3.2.3 100Gb Ethernet via Switch

We tested the 100GbE via a switch by using the E4 [E4IF] infrastructure setup, as a sandbox for validating the communication mechanism.

The 100GbE solution has been successfully tested using two u55c boards connected in the same host via PCIe, where their QSFP connectors were both connected to a 100Gb Switch (Figure 13). This test was performed with a bare metal application running without Linux, thus validating our HW implementation.

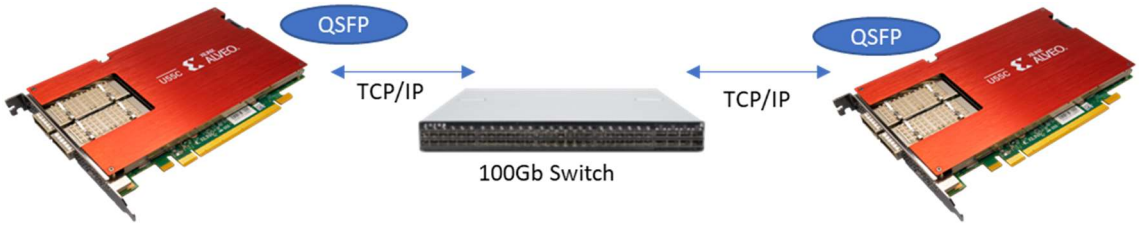

Figure 13: TCP/IP communication between two U55C Alveo boards using a 100GbE switch

4. ACME Emulated Accelerator (ACME_EA)

The ACME emulated accelerator presented in this section is an FPGA implementation of the RTL system, developed in the technical WP4. It is a many-core system with the capability of running many scalar cores and vector accelerators with the capability of being fully communicated with other accelerator modules.

The interconnection network between the cores (scalar core+vector accelerator) is based on the OpenPiton framework, and the computation core as part of the Tile is an evolution of the first FPGA release of the accelerator (DVINO). The high-level view of the ACME architecture, when it is configured in a 2x2 topology, is depicted in Figure 14.

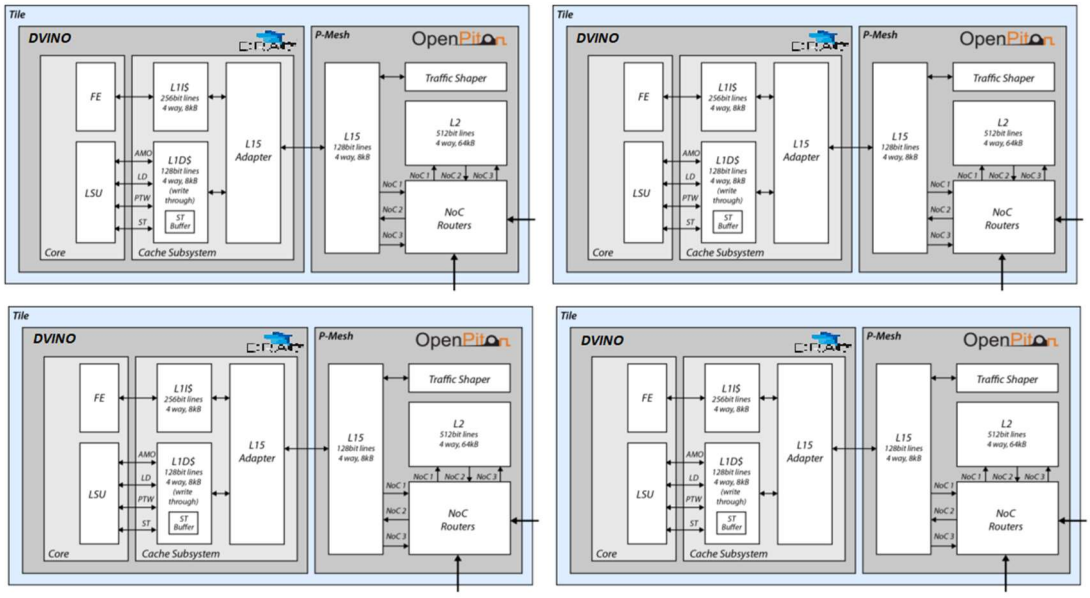

Figure 14: Emulated Accelerator prototype. Consist of 4 DVINO tiles, each composed by 1 RISC-V core (Lagarto) with a VPU with two lanes

More in detail, the scalar core used in DVINO is Lagarto Hun; a RISC-V core developed in BSC and evolved in MEEP. The scalar core instantiates the MEEP.VPU v2.2.1, which uses the OVI as the communication interface, and is configured by default with 2-Lanes configuration. This yields a total of 4 RISC-V cores with their corresponding VPU each, and a total of 8 vector Lanes in the system. The resource usage for this configuration is presented in Figure 15. In terms of Look-Up Tables (LUT), the critical resource in our designs, this EA design uses 733,476 LUTs, 56% of the total. The most demanding module is by far the VPU, which uses approximately 50KLUTs per lane.

Name	CLB LUTs (1303680)	CLB Registers (2607360)	CARRY8 (162960)	F7 Muxes (651840)	F8 Muxes (325920)	CLB (162960)	LUT as Logic (1303680)	LUT as Memory (600960)	Block RAM Tile (2016)	URAM (960)	DSPs (9024)
N system_top	733476	601721	16331	53138	12888	143601	720403	13073	275.5	15	188
meep_shell_inst (meep_shell)	65212	76059	681	2182	456	15172	58493	6719	95	7	0
openpiton_wrapper_inst (openpiton_wrapper)	666535	525642	15474	50956	12432	129147	660181	6354	180.5	8	188
ACME_OP (system)	666535	525642	15474	50956	12432	129147	660181	6354	180.5	8	188
chip (chip)	648849	513659	15412	50718	12432	125875	643901	4948	166	8	188
noc1 (pronoc_noc)	3894	2757	0	40	4	723	3370	524	0	0	0
noc2 (pronoc_noc__parameterized0)	4249	2989	0	40	4	762	3685	564	0	0	0
noc3 (pronoc_noc__parameterized1)	4086	2982	0	42	4	773	3522	564	0	0	0
rst_sync (synchronizer)	1	2	0	0	0	2	1	0	0	0	0
tile0 (tile)	159128	126235	3853	12649	3105	33383	158304	824	41.5	2	47
tile1 (tile_181)	159161	126227	3853	12649	3105	29173	158337	824	41.5	2	47
tile2 (tile_182)	159179	126235	3853	12649	3105	29192	158355	824	41.5	2	47
tile3 (tile_183)	159163	126232	3853	12649	3105	32683	158339	824	41.5	2	47
cgni_blk1 (credit_to_valrdy_184)	51	8	0	0	0	11	11	40	0	0	0
cgni_blk2 (credit_to_valrdy_185)	48	8	0	0	0	10	8	40	0	0	0
cgni_blk3 (credit_to_valrdy_186)	53	8	0	0	0	12	13	40	0	0	0
cgno_blk1 (valrdy_to_credit)	10	8	0	0	0	3	10	0	0	0	0
cgno_blk2 (valrdy_to_credit_187)	9	8	0	0	0	2	9	0	0	0	0
cgno_blk3 (valrdy_to_credit_188)	9	8	0	0	0	2	9	0	0	0	0
g_lagarto_m20_core.core (vas_tile_co	138437	112219	3762	11032	2609	28612	137901	536	32	0	47
genblk1[0].i_irq_sync (synchronizer	0	0	0	0	0	0	0	0	0	0	0
genblk1[1].i_irq_sync (synchronizer	0	2	0	0	0	2	0	0	0	0	0
i_csr_regfile (csr_regfile)	8066	7002	1093	178	12	2654	8066	0	0	0	0
i_ipi_sync (synchronizer_191)	0	0	0	0	0	0	0	0	0	0	0
i_sync (synchronizer_192)	3	2	0	0	0	3	3	0	0	0	0
i_timer_sync (synchronizer_193)	0	2	0	0	0	2	0	0	0	0	0
lagarto_m20 (top_drac)	130374	105195	2667	10854	2597	26836	129838	536	32	0	47
datapath_inst (datapath)	22029	10343	517	1305	513	5113	22029	0	0	0	25
i_cache_subsystem (wt_cache_s	6100	2584	96	692	65	1487	6100	0	32	0	0
i_mmu (mmu)	1846	2746	6	0	0	557	1846	0	0	0	0
icache_interface_inst (icache_int	17	66	2	0	0	30	17	0	0	0	0
lagarto_dcache_interface_inst (l	886	142	0	128	0	418	886	0	0	0	0
vpu_inst (multi_lane_wrapper)	98484	87826	2045	8729	2019	20096	97948	536	0	0	22
l2 (l2)	14613	7397	75	637	64	2707	14613	0	5.5	2	0
l15 (l15_wrap)	5779	6327	16	979	432	1485	5611	168	4	0	0
rtap (rtap)	100	0	0	0	0	38	100	0	0	0	0
uncore_config (config_regs)	57	241	0	1	0	51	57	0	0	0	0
chipset (chipset)	17680	11976	62	238	0	3477	16274	1406	14.5	0	0

Figure 15. Vivado resource utilization window for a 2x2 manycore configuration. Each tile has a DVINO with 2 Lanes per VPU

This accelerator interfaces with the FPGA Shell devices through the defined I/O space of the system, implemented with a crossbar that addresses the requests of the RISC-V system to the corresponding Shell device. This is the case of the presented Ethernet implementations, the UART, and the HBM. Figure 16 represents the relationship between the Emulated Accelerator and the MEEP FPGA Shell.

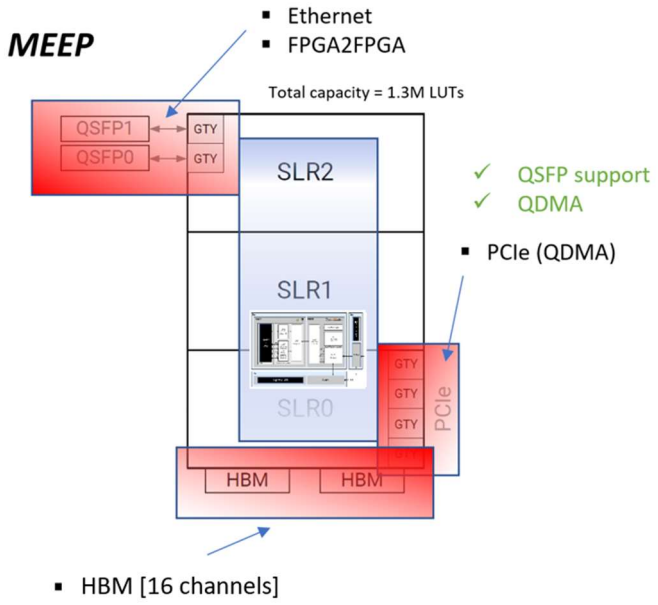

Figure 16. FPGA Shell containing a basic EA configuration within.

When the EA is clocked to 100MHz, scaling up the number of Tiles in the system eventually creates timing issues, as the tool faces congestion issues. Relaxing timing constraints (i.e, 50MHz) increases the possibilities of spreading the logic to further zones of the FPGA, thus making a 4x4 configuration feasible and generally maximizing the use of the FPGA resources.

4.1 Second FPGA release of the ACME_EA accelerator

The results shown in previous sections on the FPGA have been obtained using the more advanced version of the ACME_EA accelerator, which includes advanced features with respect to the first release presented in the deliverable D6.2, Section 3. Table 1 collects the main information for comparing the evolution of the ACME accelerator from the first to the second release.

Table 1. Characterization of the first and second FPGA release of the ACME_EA accelerator

Description		**ACME_EA FPGA 1st release**	**ACME_EA FPGA 2nd release**
		M18	**M36**
Architecture: computation		single core	many-core
Arch: communication		Bus	NoC (Routers support: OpenPiton and ProNoC routers)
core	**ACME VAS Tile core**	scalar core + VPU	scalar core + VPU
	scalar core	RV64IMA (5 stages) In-order	RV64GC (6 stages) In-order
	VPU	MEEP.VPU v1.0 (See Appendix I)	MEEP.VPU v2.2.1 (See Appendix I)
vector support		yes	yes
Configurable L2 size		yes	yes
Scalable number of Tiles		no	yes (2x2* 2D-mesh)
Language		SystemVerilog and Chisel	SystemVerilog
Linux OS support		yes (buildroot)	yes (Fedora)
Host/Device communication		PCIe	PCIe, Ethernet over PCIe
MEEP FPGA Shell support		yes	yes
Memory Controllers (MCs)		N/A	support for multiple MCs

Clock frequency	50 MHz	50MHz and 100MHz*
M36 is the evolution of M18, and it is under development by the RTL team. *100MHz is only possible when the many-core system only includes the scalar core in each of the Tiles. Running the system at 50MHz allows the manycore system to close timing with a 4x4 configuration, when the tile includes only the scalar core.		

4.1.1 FPGA system release

The Emulated Accelerator releases are available in the MEEP Gitlab repository:

https://gitlab.bsc.es/meep/FPGA_implementations/AlveoU280/fpga_shell/-/wikis/MEEP-FPGA-Releases

The site above describes the main features of the different releases, a link to the source code and the link for the bitstreams.

5. Continuous Integration and Continuous Delivery for the FPGA flow

Continuous integration and Continuous Delivery (CICD) is a common practice in the development of software projects. It is quite hard to imagine a project that does not consider its use as part of the delivery flow. Despite the different nature of software and hardware projects, there is no reason why not to use CICD in the latter. Going further, it is perfectly possible to apply the same concept to an FPGA environment, where the use of CICD flows is even less common and mature.

The MEEP project has prioritized a CICD system that robustly supports the whole FPGA flow for any hardware system design, by adding repeatability, reliability, flexibility, and scalability properties to the delivery process.

The MEEP CICD flow creates the final software release (binaries) and hardware releases (bitstreams) starting from scratch: The process consists of cloning the sources in a clean space and then automating the necessary steps to build the final binary (or binaries). This automation is run by a set of scripts, a runner(s) that executes them, and a server that orchestrates the process in combination with the other two.

The main difference in the FPGA environment with respect to a pure software one is the nature of the sources (RTL code), and likely some script-level software that carries on the building of the FPGA system. In MEEP, mainly TCL scripts are used to execute the vendor specific tasks (Xilinx/AMD), python, and bash scripting. This combination eases the scalability and reproducibility of the different projects. That enables the possibility of building the same project with a different set of parameters that yield a completely different system, always from scratch.

It would be perfectly correct, and useful, to stop the CICD execution after successfully synthesizing some RTL code. However, the FPGA flow proposed in MEEP extends this approach to complete all the steps, including the effective implementation of a design on the FPGA; which means the place and route, and the bitstream generation. Even further, the infrastructure supports targeting different FPGA boards (Alveo U280, Alveo U55C, Alveo U200, and VCU128), programming them and running different benchmarks to validate the implemented designs, or even check improvements with respect to previous designs, in terms of performance, utilization, etc.

From the project perspective, this whole ecosystem is not only one repository with all the pieces, but a cross-project system, which involves RTL development, FPGA development, and SW development. This interconnection between different projects obliges: 1) to set some policies that define the boundaries among them, and 2) configure events that trigger the different pipelines on the different repositories.

In order to detail the MEEP FPGA flow, it is important to take in consideration that the MEEP project is designing and developing the ACME accelerator. As a proof of concept, and to test the whole infrastructure, the scaled down version of ACME is based on: 1) a computation engine, where the core is the *VAS Tile core* module, 2) a memory engine, which is a module close to memory, and 3) the communication infrastructure based on the OpenPiton framework. More information about the design in the deliverable *D4.2 FPGA RTL revision 1 release. Complete verification environment.*

Moreover, another piece to implement any design in the MEEP infrastructure is the FPGA-Shell, which wraps any hardware design and provides communication capabilities with the host and other FPGAs.

All the steps involved in the MEEP FPGA flow are shown in Figure 17.

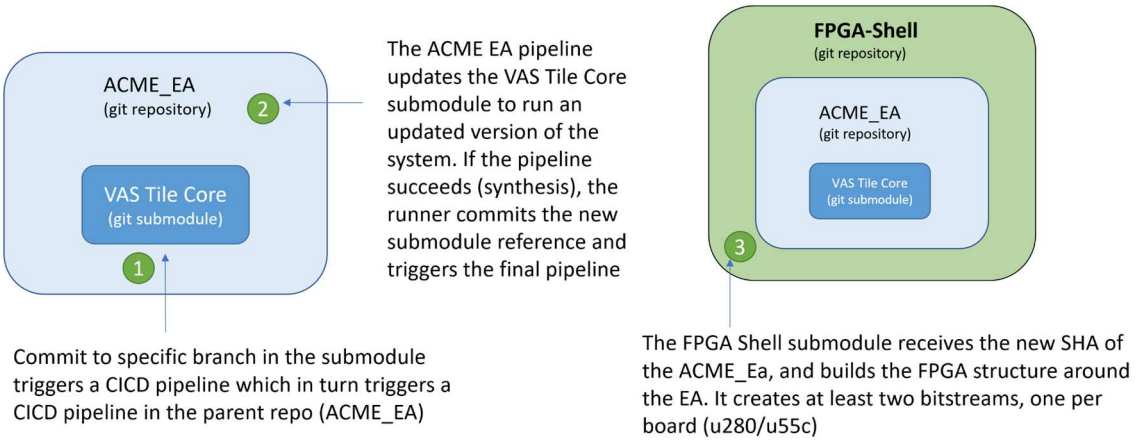

Figure 17: multi project CICD high level diagram

As stated, ACME_EA is a flexible project that allows different manycore configurations, interfaces, and features. For instance, it allows selecting which RISC-V Core to implement in the system, how many of them and the system topology in the manycore architecture. The RISC-V CPU is integrated in the ACME_EA file system as a git submodule. The regular CICD flow works on top of one repository, so in a multi-project, multi-repository context, we need to implement a mechanism to communicate the different repository pipelines between them. This mechanism is the "trigger" feature. Some jobs in a pipeline can have the goal to be the triggering source of a downstream pipeline. As depicted in the Figure 17 (step 1), when changes in the RISC-V submodule are committed to a specific branch, this triggers the ACME_EA main repository pipeline. In this structure, ACME_EA is the parent repository, since it is the one containing the submodule.

When triggered, the ACME_EA repository contains an outdated *SHA* hash for the submodule. One of the jobs in the pipeline is to update the submodule to be the one that triggered the ACME_EA pipeline. To achieve this, the submodule pipeline passes the SHA of the commit as a variable for the ACME_EA pipeline. This way, the ACME_EA pipeline can update the submodule and do the necessary steps to complete the building flow. As a final job, if the ACME_EA pipeline succeeds, the job commits the ACME_EA repository with the updated submodule, creating a new release version.

Simulating the design and doing the FPGA synthesis of it are the main jobs in the ACME_EA pipeline, which overall verifies that the RTL is FPGA compliant. Thus, in case the RTL is inferring latches or creating timing loops, a job in the pipeline detects them and makes the pipeline fail. The detail of the whole ACME_EA flow is depicted in Figure 18, where all the different CICD steps are included:

- *simulation*: Simulates ACME_EA with the updated VAS tile core. This job is calling git to get the VAS commit SHA that has been passed by the parent pipeline.
- *synthesis*: FPGA synthesis of the design to know if it is compliant with the FPGA requirements.

- *validation*: This step checks whether there are RTL errors, critical warnings or failures on the synthesized design.
- *deploy-openpiton*: Stores the generated artifacts in the servers, making them available for later study (routing, timing, etc). A second job in the same stage pushes to the ACME_EA repo with the updated and successfully verified VAS submodule, creating new commit SHA that will be passed down to the FPGA Shell pipeline
- *push-shell*: This is the step that triggers the CICD associated with the FPGA-Shell repository. It takes the new SHA created in the parent pipeline.

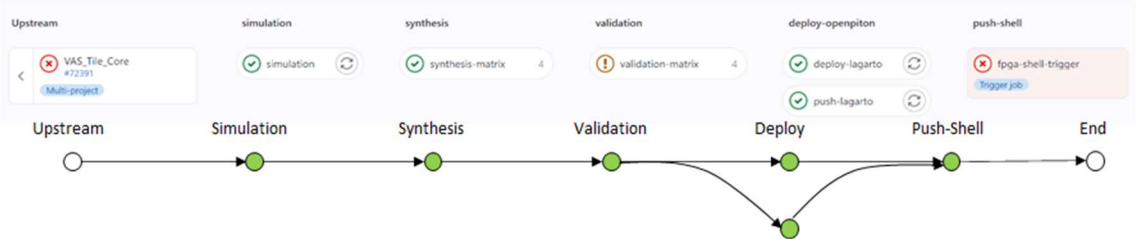

Figure 18: ACME_EApipeline

As it is shown in Figures 18 and 19, the process creates new triggers (*fpga-shell-trigger* in Figure 18, and *OpenPiton to FPGA Shell* in Figure 19). This time, along the updated and successful ACME_EA repository, the trigger acts on the FPGA Shell project, which gets the newly (and automatically) generated commit SHA for the ACME_EA. From the FPGA Shell perspective, ACME_EA is an IP, which is dropped inside a set of interfaces, mainly PCIe and HBM (see Figure 3). This new FPGA pipeline has a configuration of jobs that goes through the project creation to the bitstream generation.

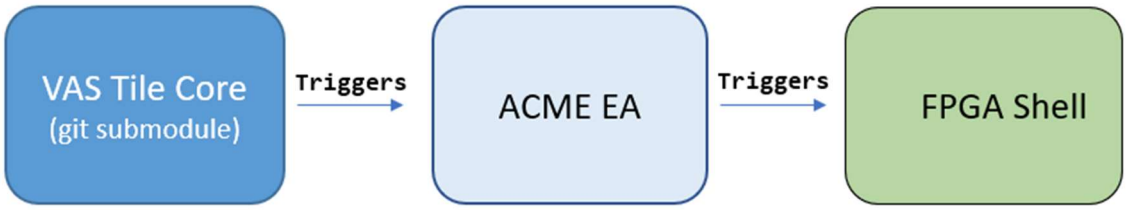

Figure 19: Multi Project CICD trigger chain

Typical FPGA flow steps are themselves jobs in the pipeline, namely synthesis and place and route. The synthesis in this case comprehends all the interfaces that the release configuration may need (Ethernet, Aurora, etc.), differentiating this synthesis from the synthesis carried on in the ACME_EA pipeline in the number of involved elements. In the ACME_EA pipeline, there is no need to add the whole set of interfaces, as the main goal is to verify the RTL developed in the submodules, and in ACME_EA itself.

The last two jobs in the FPGA Shell pipeline are: 1) programming the FPGA, and 2) testing the final design. For this final step, different benchmarks might be executed, and in the end the output is read. If the read output matches the expected output, the pipeline passes. An example of this is booting Linux on the FPGA. If Linux boots, there are different messages during the booting that demonstrate that the FPGA design is operational and a successful build.

Lastly, the *deploy* stage takes place (Figure 20). Here, the pipeline stores the different artifacts created by the runners on the workstations. This is useful to check the main aspects of the

design, getting the FPGA reports offline, or for comparing the details between different implementations. When this process runs on the FPGA Shell master branch, an extra job uploads the bitstreams to the MEEP official release site.

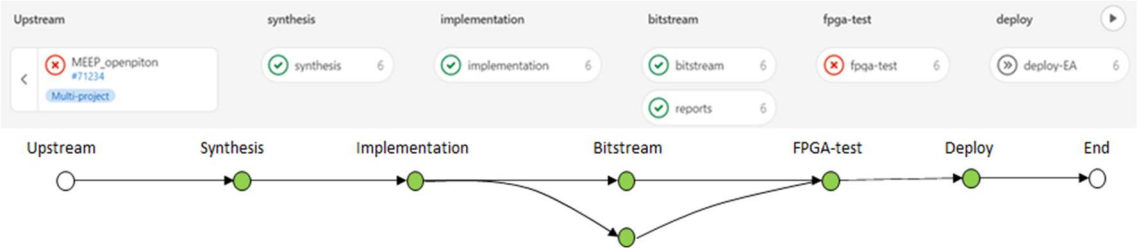

Figure 20: FPGA Shell pipeline

6. MEEP Contributions to other projects from the FPGA perspective

The design efforts that we have described in previous sections are exploited in other projects. Such is the case of DRAC [DRAC], eProcessor [EPROC] and EPI [EPI]. The MEEP Shell project is now open-source and available on GitHub [MEEP.GH]. In addition, some students from the University of Santa Barbara have used it for internal purposes in combination with OpenPiton.

1) **eProcessor** is using part of the CICD code developed for MEEP to implement its CICD flow. The whole FPGA project can be generated using the MEEP FPGA Shell flow.
2) **EPI** is using the 10Gb Ethernet solution, and we also ported the design from the VC128 board to the Alveo U280 and U55C boards.
3) **DRAC** is also using the 10Gb Ethernet solution in combination with the MEEP FPGA-Shell approach.

On the other hand, MEEP development have been supported by other projects:

1) EPI: Leveraging the Linux Ethernet driver they implemented, as a reference to insert our modifications.
2) OpenPiton: Establishing technical discussions with the main maintainer of the project for clarifying some low-level details of the project.

7. Conclusion

With the work described in this deliverable two kinds of Ethernet communications have been enabled for the MEEP project: 1) Ethernet over PCIe, and 2) Ethernet over QSFP. This opens a broad set of possibilities, not only for MEEP but for all the different projects that wish to target a Xilinx Alveo board. In addition, the CICD flow is released as part of the MEEP FPGA Shell, again, enabling this flow for any project based on it.

The Ethernet solutions described have still some room for improvement. The Ethernet over PCIe solution could benefit from HW interrupts to increase performance, and the driver on the RISC-V side could use a device tree entry for a better Linux integration. In addition, the Ethernet over QSFP solution is not achieving high speed rates when working along with a

RISC-V and a Linux driver, whereas within a bare-metal environment and with Microblaze, the Ethernet achieved speed is around 22Gb/s.

8. References

[X.QDMA] https://github.com/Xilinx/dma_ip_drivers

[X.ONIC] https://github.com/Xilinx/open-nic

[DMA] http://www.pebblebay.com/a-guide-to-using-direct-memory-access-in-embedded-systems-part-two/

[RVISA]. : https://github.com/ /riscv-isa-manual/issues/255

[OP] https://github.com/PrincetonUniversity/openpiton

[IPERF3] https://iperf.fr

[FLUSH] https://groups.google.com/g/openpiton/c/E3wGTlzx-bg/m/zGk5LvIRBgAJ

[E4IF] https://www.e4company.com/

[10GAF] https://github.com/alexforencich/verilog-ethernet/tree/master/rtl

[X.DMA] Xilinx manual: https://github.com/Xilinx/dma_ip_drivers

[X.CMAC] https://docs.xilinx.com/r/en-US/pg203-cmac-usplus

[DRAC] https://drac.bsc.es/en/home

[EPROC] https://eprocessor.eu/

[MEEP.GH] https://github.com/MEEPproject

[EPI] https://www.european-processor-initiative.eu/

[OVI] Specs: https://github.com/semidynamics/OpenVectorInterface

Appendix I

Table A shows the MEEP VPU characteristics used in each of the FPGA releases.

MEEP VPU features		
	MEEP VPU v1.1 (FPGA 1st release)	**MEEP VPU v2.2.1 (FPGA 2nd release)**
Number of vector lanes	Up to 16 (grouped in vector-lane pairs for smaller configurations (2-4-8))	
Maximum vector length	128 elements x 64 bits	*ACME-classic* mode 128 elements x 64 bits *ACME* mode 512 elements x 64 bits
Number of FMAs	1 Fused Multiply Accumulate (FMA) unit per lane (2 DP FLOP/cycle)	
FP operation support	Support for 64- and 32-bit FP operation	
Integer operation support	Support for 64-, 32-, 16-, and 8-bit integer operations, signed and unsigned	
Vector Register File (VRF)	VRF number of banks: 5. N of physical vector registers: 40. Single-Port limited access.	VRF number of banks: 4. N of physical vector registers: 32 Dual-Port access. Redesign of lane control logic to leverage VRF concurrent read/write accesses.
RISC-V vector version	RVV v0.7.1	
Core's Interface	OVI 1.0 [OVI]	
Memory's interface	OVI 1.0	OVI 1.0 & Direct Memory Access
Direct access to L2	Through OVI	Through OVI & Long Vector Register File
Execution modes support	*ACME-classic* mode vector lanes config: 2,4,8,16	*ACME-classic* mode vector lanes config: 2,4,8,16 *ACME* mode vector lanes config: 2, 16

Table A. MEEP VPU characteristics for each of the FPGA releases