CONVOLVE

Seamless design of smart edge processors

GRANT AGREEMENT NUMBER: 101070374

Deliverable D6.3

Description SoC architecture, and the rapid design & prototyping environment

CONVOLVE

Title of the deliverable	Description SoC architecture, and the rapid design & prototyping environment
WP contributing to the deliverable	WP 6
Task contributing to the deliverable	Task 6.3
Dissemination level	PU – Public
Due submission date	30/04/2024
Actual submission date	30/04/2024
Author(s)	Ahmet Turan Erozan (BOS) Andre Guntoro (BOS) Ryan Antonio (KUL) Guilherme Paim (KUL) Moritz Scherer (ETHZ)
Internal reviewers	Tim Güneysu (RUB) Gaizka Eiguren Arza (TASE)

Document Version	Date	Change
V0.1	26/03/2024	Initial version with ToC
V1.0	24/04/2024	The finalized draft ready to internal revision
V1.1	06/05/2024	The final document

Deliverable Summary

This document provides the description of SoC architecture and rapid design & prototyping environment, providing an overview of the System-on-Chip (SoC) architecture, emphasizing the rapid design and prototyping environment. It outlines the system design flow, highlighting the crucial steps involved in the flow. The document also delves into the host and peripherals, discussing their integration and functionality within the SoC. Furthermore, it explores cluster instantiation, shedding light on the process of creating and configuring clusters within the architecture. Additionally, the document examines the accelerator template and integration, showcasing the integration of specialized accelerators into the SoC design. Overall, this comprehensive report offers valuable insights into the various components and design aspects of the SoC architecture, providing a solid foundation for further exploration and development.

1. Objectives

This document "D6.3 Description SoC architecture, and the rapid design & prototyping environment" is a deliverable of the Work package No.6 "Compositional architecture DSE and SoC generation".

WP6 Objectives

WP6 deals with automated compositional system architecture design space exploration (DSE) and system-on-chip (SoC) generation. This is done by providing a modular architecture template consisting of a RISC-V host with one or multiple machine learning (ML) and security accelerators.

The objectives of WP6 are defined as follows:

1) Provide a secure and modular RISC-V based SoC architecture template that eases the integration of multiple accelerators, managing control, synchronization, data exchange and run-time reconfiguration.
2) Create a SoC-level performance modelling framework for running ML applications on the targeted modular runtime configurable architectures, integrating the component models coming out of WP2.
3) Develop a rapid Design Space Exploration (DSE) framework to cycle quickly over ULP SoC and accelerator constellations, finding the optimal balance between design-time and run-time flexibility.
4) Realize an automated design time instantiation flow for optimal and run-time flexible SoC generation.

1.1.1. Deliverable D6.3 Objectives

The deliverable D6.3 of WP6 describes the definition of SoC and rapid design & prototyping environment within the context of CONVOLVE WP6. The objective of this deliverable is to provide a comprehensive overview of the System-on-Chip (SoC) architecture, with a specific focus on the rapid design and prototyping environment. It aims to outline the system design flow, elucidating the key steps involved in the design process. Additionally, the report aims to explore the integration and functionality of the host and peripherals within the SoC. It also seeks to examine the process of cluster instantiation, emphasizing the creation and configuration of clusters within the architecture. Furthermore, the report aims to showcase the integration of specialized accelerators through the accelerator template. By achieving these objectives, this report serves as a valuable resource for understanding the various components and design aspects of the SoC architecture, facilitating baseline for further exploration and development in this field.

WP6 Contribution to CONVOLVE's Objective

WP6 focuses on the modular SoC design and rapid deployment which makes the work package one of the contributors to achieve CONVOLVE's target to **reduce design time of edge AI hardware systems by 10x** by focusing on the faster design time of the SoC architecture and providing a design space exploration tool for rapid software-hardware co-design explorations. At the same time, WP6 is crucial to bring together all developed accelerators which are needed to achieve CONVOLVE's goal to **achieve 100x energy efficiency improvement** by providing an SoC template with standard interfaces to a set of ultra-low-power ML and security acceleration blocks which exploit novel architectures, microarchitectures, circuits and devices.

To achieve these goals, it is necessary to have customizable hardware acceleration blocks that can be parameterized during both design and run time using a standard interface. These blocks should allow for various configurations based on diverse application needs, including adjustments in supply voltage, clock frequency, data representation accuracy levels, parallelization degrees and dimensionality precision values. WP6 focuses on providing a modular and scalable SoC with such standardized interfaces such that the design acceleration blocks can be plugged easily to reduce the overall design time.

In addition to the RTL design itself, performance models and simulators must also be modifiable to enable fast exploration of the design-space without sacrificing compositional flexibility. WP6 focuses also on automated design-space exploration (DSE) and simulators using performance models of the hardware building blocks.

2. SoC Architecture Overview

Figure 1 provides an overview of the SoC template used in Convolve, which consists of two main domains: the support infrastructure domain and the L2-accelerators domain. The support infrastructure domain includes a RISC-V host, main memory, and peripherals. These components are connected through a high-speed on-chip interconnect, such as a network-on-chip (NoC) or AMBA AXI. This domain serves as the foundation of the SoC template. The L2-accelerators domain comprises a set of L2 Snitch Cluster Accelerator Extension (SNAX) clusters, which can contain the same type or a combination of different accelerators, but same standard interface. One example to SNAX cluster is shown on the right side of Figure 1. Within this cluster, general-purpose RISC-V cores can control the accelerators and share tightly coupled data memory (TCDM) with accelerators.

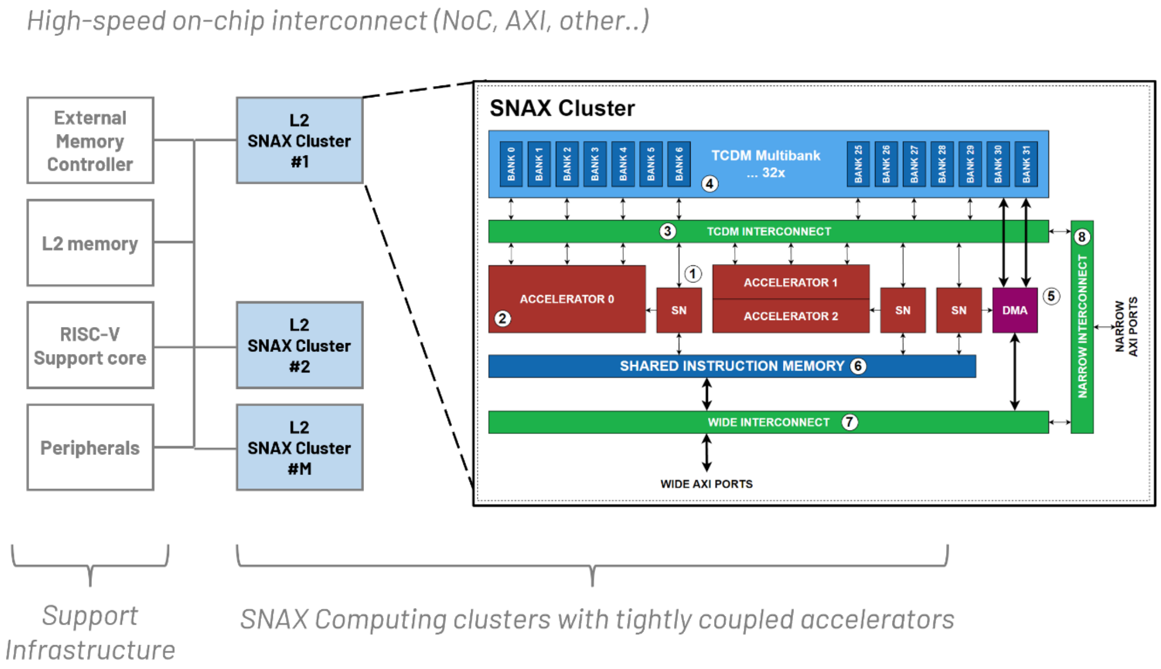


Figure 1: Overview of high-level SoC template (SN: Snitch, DMA: Direct Memory Access, TCDM: Tightly Coupled Data Memory)

The following features are included in the SNAX cluster, ordered based on numbers on the Figure 1. (1) A lightweight accelerator manager core implemented using a Snitch processor. (2) Custom accelerators capable of running specific kernels based on user requirements. (3) A tightly coupled data memory (TCDM) interconnect that connects the accelerator and Snitch cores to the memory. (4) A shared multi-bank memory accessible to all accelerators and cores. (5) A DMA core responsible for data movement between the wide and narrow AXI interconnects (7) and (8), transferring data to and from an L2 memory outside the cluster towards the local TCDM. (6) Shared instruction memory utilized by all Snitch cores. (7) A wide interconnect with a bandwidth of 512 bits, facilitating data transfer from L2 to TCDM. (8) A narrow interconnect enabling communication of shell data between TCDMs. The details of each part of the SNAX can be found in Deliverable D2.2 “Report on the Micro-architecture Design and Implementation”.

The current state of the SNAX platform allows users to explore basic architectural possibilities. With a complete HW-SW setup, users can also profile their system and obtain initial performance estimates. The integration of a new accelerator into the SNAX platform has been simplified for user convenience.

3. Rapid Design and Prototyping Environment

This Section introduces the newly developed rapid design and prototyping flow for the CONVOLVE project. Specifically, we first describe Cheshire, the SoC template used in CONVOLVE, and Bender and Solder, two software tools for managing IP dependencies and exploring AMBA AXI interconnect configurations.

3.1. System Design Flow

Architectural development, together with verification and physical implementation are often considered the main efforts of new system design. Such efforts often rely on an effective and structured management and integration of the components into the final design. Many challenges of designing a new SoC originate from the inherent fluidity of its initial specifications. System requirements may be incomplete, subject to change over the course of a project or conditioned to events unrelated to the project, necessitating a design approach that can accommodate ongoing modifications in functionalities and features. Such modifications can have a cascading effect, impacting other aspects of the design. For instance, replacing the CPU of the SoC host domain often necessitates adjustments to various parts of the SoC, triggering a re-evaluation of some of the components that have been already integrated. The process of new system design is inherently iterative.

The following sections introduce the complexities associated with rapid SoC prototyping. Each section focuses on the unique challenges encountered within specific SoC domains. Furthermore, these sections showcase the innovative solutions developed within the CONVOLVE project to overcome these challenges.

3.2. Host Domain Design Challenges

Developing SoCs (System-on-Chip) with design-time reconfigurability offers significant advantages, but it also presents unique challenges:

- **Design Partitioning**: Effectively partitioning the architecture of an SoC into well-defined features is crucial for iterative development and meeting evolving design requirements. An appropriate partitioning ensures ease of modification and enables independent optimization of individual features.
- **Standardized IP Interfaces**: Standardization of interfaces between different intellectual property (IP) blocks minimizes the effort required for integration. Consistent interfaces allow for minimally invasive architectural modifications and reconfiguration of IPs within the SoC.
- **Automated IP Management**: A robust framework for automated IP fetching and version control is essential. An effective IP management strategy streamlines the integration of IP dependencies and ensures compatibility throughout the design process.
- **Automated Script Generation**: Often, the design flow utilizes several Electronic Design Automation (EDA) tools from various vendors. A human friendly strategy for

generating scripts across these tools is necessary to automate design flow steps and track design files used in different phases of the project.

These challenges necessitate a multi-disciplinary approach that merges expertise in hardware design, software development, and system integration. Achieving an effective design partitioning requires deep understanding of hardware architecture and communication infrastructure design to ultimately ensure a good quality of result, i.e., flexible data flow across various configurations and, at the same time, efficient utilization of silicon area and low computational energy. Such expertise must be combined with In-depth understanding of software development principles, driver design for adaptable hardware for managing diverse configurations and guarantee usability of the system. Ultimately, a holistic view of system integration, encompassing hardware and software interactions is required for ensuring testability across different configurations, and usability of the SoC in the relevant application scenario.

Besides expertise, achieving design-time reconfigurability of an SoC requires the possibility to effectively modify the IPs that are integrated in the design. This might involve modifying IP parameters to enable specific configuration options, but it could be as invasive as entirely generating the IP source files from a configurable template.

In some cases, and very often when designing subsystems integrating CPUs, design-time reconfiguration might necessitate modifying both hardware and software components concurrently. The software stack needs to be adaptable to different hardware configurations. In this context, the design framework might need to provide tools or methodologies for generating configuration-specific software components or drivers. This must be combined with hardware and software co-simulation tools and methodologies that enable joint hardware-software exploration and optimization.

Ultimately, maintaining consistency between IP components becomes crucial. The framework needs robust version control mechanisms to ensure compatibility among IPs and avoid integration issues during the design process.

The next Section introduces the Cheshire SoC template, a foundational element within the design flow developed in the CONVOLVE project. Cheshire plays a critical role in addressing the challenges associated with design partitioning and efficient component interconnection. It achieves this through the utilization of standardized communication interfaces.

Building upon the foundation laid by Cheshire, we present Bender, a dedicated software tool designed to manage IP dependencies. Bender automates script generation for a majority of the EDA tools commonly employed in SoC development. This automation capability significantly reduces the overhead associated with IP dependency management.

3.3. Cheshire SoC Host

The design space exploration (DSE) developed in the CONVOLVE project is built around a modular SoC host system. This host acts as the central control unit, orchestrating critical functionalities that ensure the efficient operation of the System-on-Chip (SoC). The host developed in the context of this project is derived from the openly available (https://github.com/pulp-platform/cheshire). The block diagram describing the default architecture of Cheshire is reported in Figure 2.

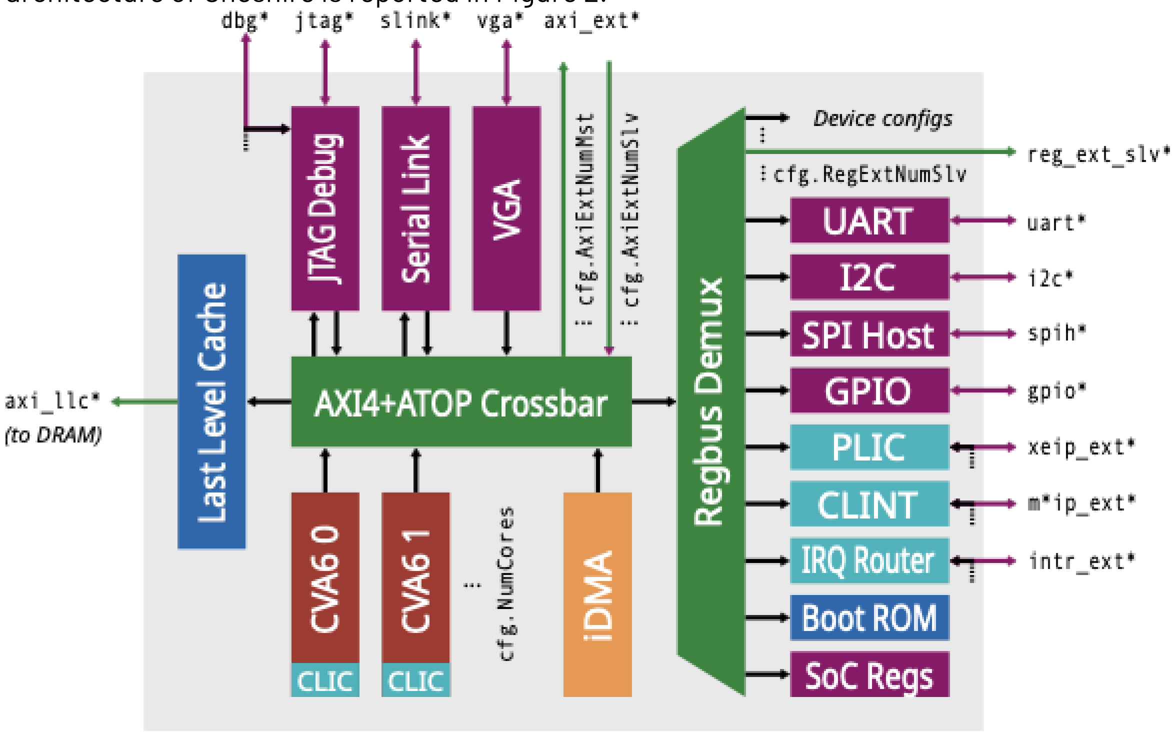


FIGURE 2: CHESHIRE SOC BLOCK DIAGRAM

By default, Cheshire uses a 64 Bit AXI crossbar that connects the Linux capable CVA6[1] core with the rest of the system, including various I/O peripherals like SPI, GPIO, I2C, UART, and JTAG. Cheshire features a SoC-level DMA which is responsible for efficient data movement between the various IP blocks. However, Cheshire is ultimately a flexible aggregator of IPs that provide an easy way to plug hardware accelerators, as the way it is partitioned allows for changing any component inside. The next sections provide an overview of how the various modules inside Cheshire can be adapted to different SoC requirements.

3.4. Core Complex

The Cheshire SoC template is built around a core complex hosting, by default, the CVA6 processor. These cores come pre-configured with hypervisor and Core Local Interrupt Controller (CLIC) support for advanced virtualization and interrupt handling capabilities.

Users can define the number of CVA6 cores instantiated within the SoC, ranging from a single core to a maximum of 31. Additionally, parameters like the depth of the CVA6 return address stack and the size of internal structures like the Branch Target Buffer (BTB) and Branch History Table (BHT) can be adjusted to fine-tune core behavior.

For communication with external memory, Cheshire utilizes the AXI4 bus protocol. Each CVA6 core acts as an independent AXI4 master, capable of initiating data transfers. To maintain data consistency across multiple cores, Cheshire employs a self-invalidation scheme. This

[1] https://github.com/openhwgroup/cva6

means cores automatically invalidate their cached copies of data whenever they modify that data, ensuring all cores have the latest version.

A unique aspect of Cheshire lies in its handling of RISC-V atomic instructions. These special instructions allow for synchronized memory access operations essential for multi-core programming. Cheshire implements a custom AXI4 extension that relies on user channels to facilitate these atomic operations. Each core and other AXI4 masters are assigned a dedicated user channel and a unique identifier within a designated user channel bit slice. This mechanism ensures efficient and coordinated execution of atomic instructions across multiple cores.

In the context of the CONVOLVE project, we plan to replace the RV64A CVA6 core with a low-power 32-bit RV32IMC configuration to maximize energy efficiency for a microcontroller class system as designed in CONVOLVE. The IMC configuration supports standard integer operations as well as multiplications, and improves cache performance and energy efficiency through the use of compressed instructions.

3.5. Interconnect

The Cheshire SoC template employs a hierarchical interconnect strategy to achieve a balance between performance and cost. Agents requiring high data transfer bandwidth are connected to the High-Performance AXI4 Crossbar. This component implements the AXI4 protocol. It supports Atomic Operations (ATOPs) for synchronized memory access crucial in multi-core systems. The AXI4 crossbar exposes various configurable parameters, allowing for fine-tuning aspects like data width, maximum in-flight transactions, and user channel allocation for RISC-V atomic operations. Additionally, Cheshire offers optional features for the AXI4 interconnect:

- AXI-RT: This adds traffic regulation units for individual AXI4 masters, enabling bandwidth and traffic control for real-time applications.
- BusErr: This integrates an error reporting mechanism for the AXI4 masters, reporting errors through a dedicated Regbus interface.

Agents requiring low data transfer bandwidth are connected to the AXI4 crossbar through a cost-effective Regbus demultiplexer. This component utilizes the Regbus protocol. While less performant than AXI4 due to limitations in burst transfers and pipelining, Regbus offers a simpler and significantly cheaper implementation. It provides access to various peripherals and configuration interfaces within the SoC. The Regbus protocol operates with a fixed 32-bit data width. This combined approach allows Cheshire to leverage the high-performance AXI4 for critical data paths while utilizing the cost-effective Regbus for peripheral interactions. This hierarchical structure significantly improves interconnect scalability, enabling efficient communication within the SoC while keeping the SoC interconnection complexity low.

Both the AXI4 crossbar and the Regbus are parametrized components. The parameters allow for customization of the AXI4 and Regbus components, tailoring their behavior to specific design requirements. Additionally, the configurable number of external ports on both AXI4 and Regbus facilitates integration with external memory systems employed by surrounding SoCs.

3.6. IO Peripherals

The Cheshire SoC template offers a rich set of peripherals for designers to integrate into their custom SoCs. Here's a breakdown of some key components:

- VGA Controller: This peripheral allows for displaying video frames stored in memory on a VGA interface. It operates autonomously, fetching data through an AXI4 manager port.
- OpenTitan Peripherals: The I2C host, SPI host, and GPIO interface leverage OpenTitan IP blocks [A reference needed here], adapted for use within the PULP platform. These peripherals maintain compatibility with OpenTitan's device interface functions (DIFs) with minor modifications.
- UART: This serial communication interface is compatible with the industry-standard TI 16750, ensuring seamless integration with popular operating systems like OpenSBI, U-Boot, and Linux. Notably, Cheshire exposes the modem access control functionality for systems requiring such features.

The Cheshire SoC template integrates a RISC-V compliant debug module to aid in development and troubleshooting. This module supports the JTAG protocol for external debugging and offers functionalities for both internal and external processor cores (harts). Additionally, it provides access to the system bus for memory inspection and manipulation.

3.7. Memory Hierarchy and Access

The Cheshire SoC template relies on external memory for its main storage. However, several internal components play crucial roles in memory management and the boot process:

- Last Level Cache (LLC): This on-chip cache acts as a buffer between the SoC and external memory (typically DRAM). It caches all memory accesses unless explicitly bypassed. Notably, the LLC can be configured to function as scratchpad memory (SPM) during the initial boot stage. This allows the boot ROM to utilize the LLC as temporary working memory for loading code from external storage. The LLC can be entirely omitted if a more complex external memory system is used. However, an external scratchpad memory would still be necessary for booting bare-metal code from local memory.
- Boot ROM: This embedded ROM contains the initial code executed by the SoC after reset. Its primary function is to safely and efficiently load a main program from an external source into memory.
- iDMA Engine: This hardware component implements high-speed data transfers between any two memory locations within the system. This component supports two-dimensional data transfers efficiently.

To reduce the memory access energy and shrink the overall power envelope of the SoC domain, we plan to replace the LLC with a dedicated L2 scratchpad memory.

3.8. Interrupt Routing

The Cheshire SoC employs a flexible RISC-V interrupt architecture to efficiently manage interrupts from both internal components and external devices. This system allows for routing and directing these interrupts to various controllers and targets within the SoC.

The internal interrupt map is statically defined, simplifying interrupt handling for internal components. Conversely, the handling of external interrupts can be customized based on the surrounding system configuration. This allows the Cheshire SoC to adapt to various setups involving different external devices and peripherals.

The Interrupt Routing Process works as follows: All interrupt sources, internal and external, are first collected into a single pool. If the interrupt router is enabled, it takes over, routing and multiplexing the interrupts efficiently. When the router is disabled, the interrupts are directly distributed (fanned out) to the interrupt targets. It's important to note that targets may have limitations on the number of interrupts they can handle. Any exceeding interrupts are simply ignored in such scenarios.

The Cheshire SoC offers a range of interrupt targets, including:

- Core-Local Interrupter (CLINT): This internal component groups all interrupt requests from a single core, primarily handling inter-processor and timer interrupts.
- Shared Platform-Level Interrupt Controller (PLIC): This controller acts as a central hub for interrupt handling within the entire platform.
- Optional Core-Local Interrupt Controller (CLIC): Each CVA6 core can optionally have its own CLIC, providing another target for interrupts.
- External Interrupt Targets: The system can be configured with a user-defined number of external targets, each with its own capabilities for handling interrupts.

The PLIC and grouped CLINT can be configured to manage interrupts for external processor cores (harts) that lack their own internal interrupt controllers. This allows the Cheshire SoC to seamlessly integrate with various system configurations.

3.9. Bender

Bender (https://github.com/pulp-platform/bender) is a powerful solution for managing dependencies within hardware design projects, specifically targeting Intellectual Property (IP) integration. This dependency management tool enables a streamlined workflow by describing the definition of dependencies between IPs and the source file compatibility with various simulation and synthesis tools.

Bender prioritizes user autonomy. It eschews restrictions on specific Electronic Design Automation (EDA) tools, workflows, or IP directory structures. Additionally, being compiled as a single executable, it can be easily integrated into established development pipelines, e.g., based on Makefiles.

Reproducibility is another goal of Bender. Once dependencies are fetched It tracks the git hash for each dependency within a lock file. This record allows for the reconstruction of a project's source code even after a significant timeframe has elapsed, guaranteeing a reliable historical record, crucial when multiple parts of the design change concurrently.

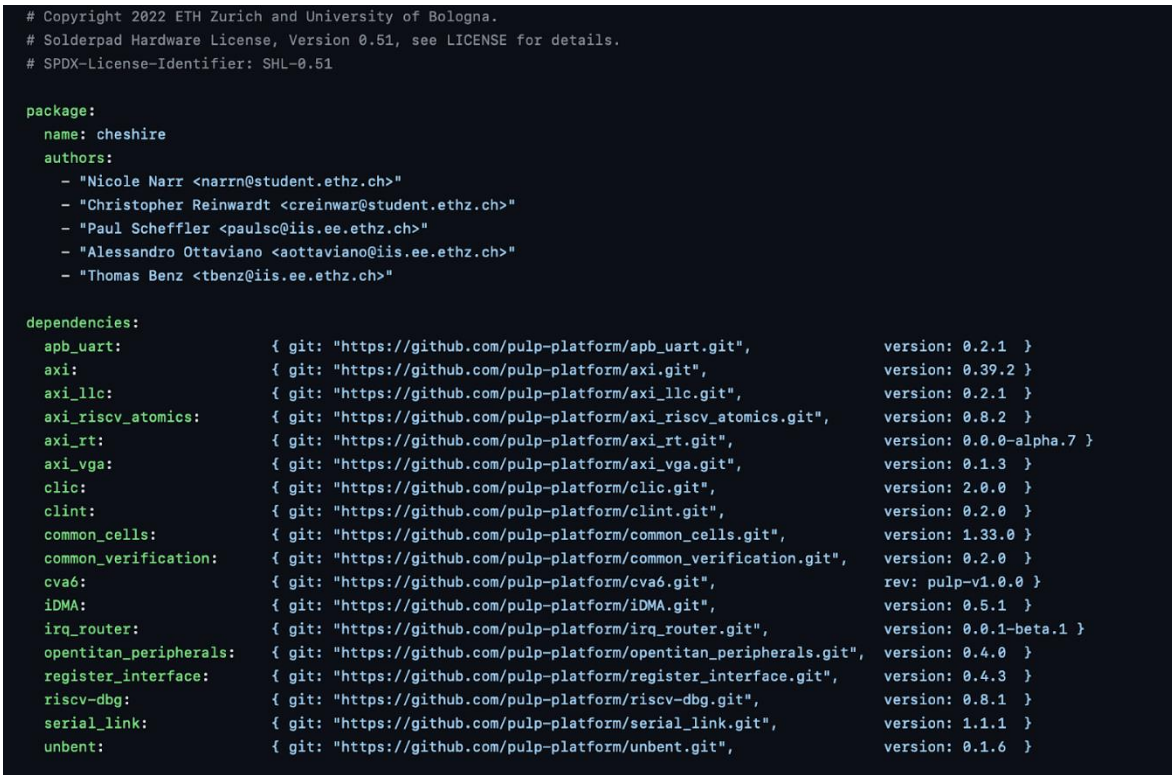
Bender operates in a three-tiered approach:

- **Source File Collection**: Bender efficiently gathers source files from hardware IPs, maintaining the correct order (e.g., ensuring package declarations precede usage) and supporting multiple hardware description languages like SystemVerilog and VHDL. It facilitates the organization of files into logical groups and manages, defines and includes directories for each group.
- **Dependency Management**: Bender tracks dependencies between IPs and streamlines the process of local checkouts for the necessary source files. It supports transitive dependencies (dependencies of dependencies) and enforces strict adherence to semantic versioning for clear communication of compatibility changes. Notably, Bender deviates from traditional package managers by not relying on a central registry. This caters to the often confidential nature of IPs within a project.
- **Tool Script Generation**: Bender further optimizes the development workflow by generating scripts for various tools. These scripts can be source file listings or compilation scripts tailored to specific tools.

The package manifest describes the package, its metadata, its dependencies, and its source files. All paths in the manifest may be relative, in which case they are understood to be relative to the directory that contains the manifest.

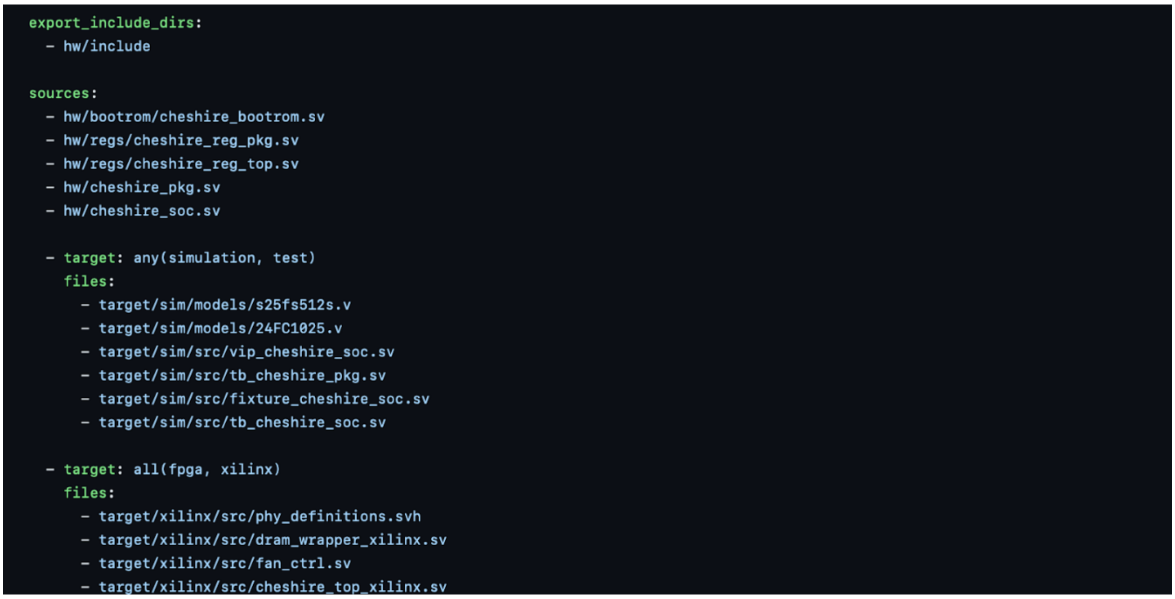
3.10. Bender Example

We show an example for the bender configuration of Cheshire, the basis of the SoC used in the CONVOLVE project. All fundamental IP blocks, including the CVA6 cores, iDMA, debug unit and CLINT are declared as dependencies. Next, by running *bender checkout*, the corresponding dependencies are downloaded from their open-source repositories. Examples of the Bender.yml file are shown in Figures 3 and 4.



```
# Copyright 2022 ETH Zurich and University of Bologna.
# Solderpad Hardware License, Version 0.51, see LICENSE for details.
# SPDX-License-Identifier: SHL-0.51

package:
  name: cheshire
  authors:
    - "Nicole Narr <narrn@student.ethz.ch>"
    - "Christopher Reinwardt <creinwar@student.ethz.ch>"
    - "Paul Scheffler <paulsc@iis.ee.ethz.ch>"
    - "Alessandro Ottaviano <aottaviano@iis.ee.ethz.ch>"
    - "Thomas Benz <tbenz@iis.ee.ethz.ch>"

dependencies:
  apb_uart:                 { git: "https://github.com/pulp-platform/apb_uart.git",               version: 0.2.1  }
  axi:                      { git: "https://github.com/pulp-platform/axi.git",                    version: 0.39.2 }
  axi_llc:                  { git: "https://github.com/pulp-platform/axi_llc.git",                version: 0.2.1  }
  axi_riscv_atomics:        { git: "https://github.com/pulp-platform/axi_riscv_atomics.git",      version: 0.8.2  }
  axi_rt:                   { git: "https://github.com/pulp-platform/axi_rt.git",                 version: 0.0.0-alpha.7 }
  axi_vga:                  { git: "https://github.com/pulp-platform/axi_vga.git",                version: 0.1.3  }
  clic:                     { git: "https://github.com/pulp-platform/clic.git",                   version: 2.0.0  }
  clint:                    { git: "https://github.com/pulp-platform/clint.git",                  version: 0.2.0  }
  common_cells:             { git: "https://github.com/pulp-platform/common_cells.git",           version: 1.33.0 }
  common_verification:      { git: "https://github.com/pulp-platform/common_verification.git",    version: 0.2.0  }
  cva6:                     { git: "https://github.com/pulp-platform/cva6.git",                   rev: pulp-v1.0.0 }
  iDMA:                     { git: "https://github.com/pulp-platform/iDMA.git",                   version: 0.5.1  }
  irq_router:               { git: "https://github.com/pulp-platform/irq_router.git",             version: 0.0.1-beta.1 }
  opentitan_peripherals:    { git: "https://github.com/pulp-platform/opentitan_peripherals.git",  version: 0.4.0  }
  register_interface:       { git: "https://github.com/pulp-platform/register_interface.git",     version: 0.4.3  }
  riscv-dbg:                { git: "https://github.com/pulp-platform/riscv-dbg.git",              version: 0.8.1  }
  serial_link:              { git: "https://github.com/pulp-platform/serial_link.git",            version: 1.1.1  }
  unbent:                   { git: "https://github.com/pulp-platform/unbent.git",                 version: 0.1.6  }
```

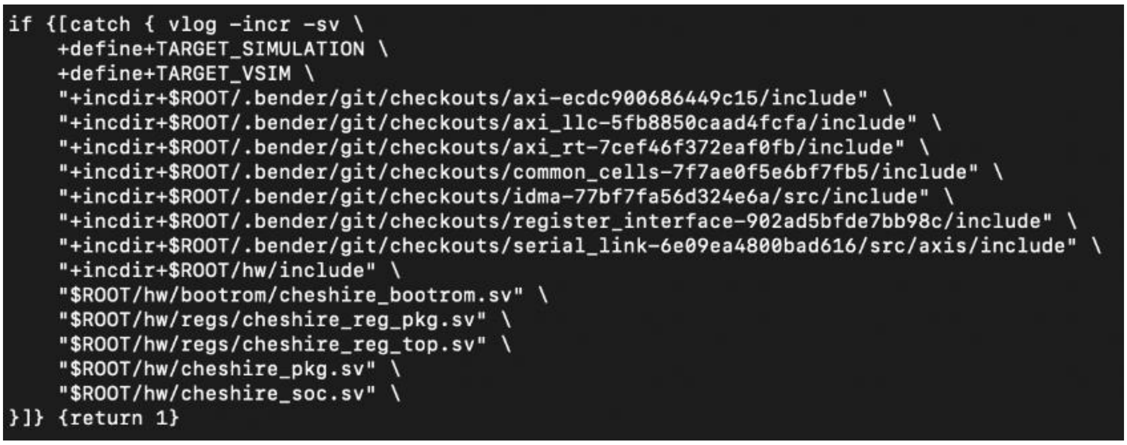
FIGURE 3: BENDER.YML DEPENDENCY MANAGEMENT FILE SHOWING THE INTEGRATION OF VARIOUS HARDWARE IPS FROM GITHUB.



```
export_include_dirs:
  - hw/include

sources:
  - hw/bootrom/cheshire_bootrom.sv
  - hw/regs/cheshire_reg_pkg.sv
  - hw/regs/cheshire_reg_top.sv
  - hw/cheshire_pkg.sv
  - hw/cheshire_soc.sv

  - target: any(simulation, test)
    files:
      - target/sim/models/s25fs512s.v
      - target/sim/models/24FC1025.v
      - target/sim/src/vip_cheshire_soc.sv
      - target/sim/src/tb_cheshire_pkg.sv
      - target/sim/src/fixture_cheshire_soc.sv
      - target/sim/src/tb_cheshire_soc.sv

  - target: all(fpga, xilinx)
    files:
      - target/xilinx/src/phy_definitions.svh
      - target/xilinx/src/dram_wrapper_xilinx.sv
      - target/xilinx/src/fan_ctrl.sv
      - target/xilinx/src/cheshire_top_xilinx.sv
```

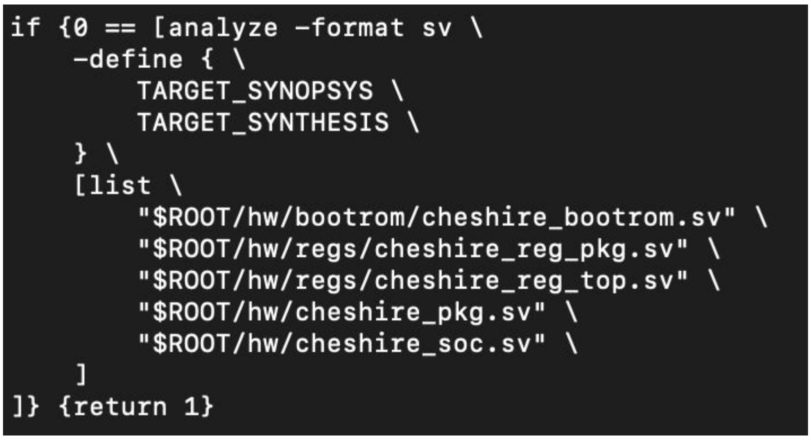
FIGURE 4: BENDER.YML DEPENDENCY MANAGEMENT FILE SHOWING THE FILES AND THEIR COMPILATION ORDER FOR THE CHESHIRE PROJECT.

Using these checked out dependencies, simulation and elaboration scripts can be auto-generated by generating for one the numerous supported targets like vsim, vcs & verilator for simulation and synopsys, genus, and vivado for synthesis targets, where the command is structured `bender script TARGET`. Figures 5 and 6 show automatically generated simulation elaboration and synthesis elaboration scripts for Questasim and Synopsys respectively.



```
if {[catch { vlog -incr -sv \
    +define+TARGET_SIMULATION \
    +define+TARGET_VSIM \
    "+incdir+$ROOT/.bender/git/checkouts/axi-ecdc900686449c15/include" \
    "+incdir+$ROOT/.bender/git/checkouts/axi_llc-5fb8850caad4fcfa/include" \
    "+incdir+$ROOT/.bender/git/checkouts/axi_rt-7cef46f372eaf0fb/include" \
    "+incdir+$ROOT/.bender/git/checkouts/common_cells-7f7ae0f5e6bf7fb5/include" \
    "+incdir+$ROOT/.bender/git/checkouts/idma-77bf7fa56d324e6a/src/include" \
    "+incdir+$ROOT/.bender/git/checkouts/register_interface-902ad5bfde7bb98c/include" \
    "+incdir+$ROOT/.bender/git/checkouts/serial_link-6e09ea4800bad616/src/axis/include" \
    "+incdir+$ROOT/hw/include" \
    "$ROOT/hw/bootrom/cheshire_bootrom.sv" \
    "$ROOT/hw/regs/cheshire_reg_pkg.sv" \
    "$ROOT/hw/regs/cheshire_reg_top.sv" \
    "$ROOT/hw/cheshire_pkg.sv" \
    "$ROOT/hw/cheshire_soc.sv" \
}]} {return 1}
```

Figure 5: Questasim elaboration script automatically generated by Bender.



```
if {0 == [analyze -format sv \
    -define { \
        TARGET_SYNOPSYS \
        TARGET_SYNTHESIS \
    } \
    [list \
        "$ROOT/hw/bootrom/cheshire_bootrom.sv" \
        "$ROOT/hw/regs/cheshire_reg_pkg.sv" \
        "$ROOT/hw/regs/cheshire_reg_top.sv" \
        "$ROOT/hw/cheshire_pkg.sv" \
        "$ROOT/hw/cheshire_soc.sv" \
    ]
]} {return 1}
```

Figure 6: Synopsys elaboration script automatically generated by Bender.

Using Bender to generate elaboration scripts automatically not only avoids tedious manual scripting of boilerplate code, but also avoids errors in elaboration order and dependency management, which can cause hard to trace bugs.

3.11. Results and Future Work

We have specified the final SoC template to be used in the CONVOLVE project based on the Cheshire SoC. We have also integrated Cheshire with Bender, which allows us to easily add new accelerator IPs, generate elaboration and simulation scripts automatically, and manage versions of shared dependencies between all IPs.

In future work, we will update the Cheshire SoC with a 32-bit CVA6 core and replace the LLC with an L2 scratchpad memory.

4. Cluster Instantiation

Integrating multiple accelerator cluster Intellectual Properties (IPs) into System-on-Chips (SoCs) presents numerous challenges. This Section gives an overview of the challenges we address in the CONVOLVE project through automated tooling for IP integration on the cluster level.

The CONVOLVE project's SoC template offers three integration possibilities, L0, L1, and L2 accelerators, as explained in Deliverable D6.1. The protocol used for L2 accelerators, Advanced eXtensible Interface (AXI), is well-suited to integrate multiple concurrent Snitch and SNAX clusters, as it supports high data rates through features like burst transfers and split transactions. In AXI, data transfers are decoupled through split transactions into address/control and data phases, simultaneously allowing multiple outstanding transactions. This significantly improves the utilization of the SoC's main bus, as the interface can handle new requests even before the current transaction is completed.

In the following, we describe the challenges of integrating clusters within SoCs and introduce the Solder tool, which helps addressing the challenges.

4.1. Accelerator IP Integration Challenges

While most IPs in the SoC operate in a single synchronous clock domain, the accelerators are intended to provide high throughput, meaning the SoC must accommodate the accelerator's frequency requirements. As such, the accelerators in the CONVOLVE project must be able to operate at frequencies different from those of the main SoC.

To communicate with the SoC, compute clusters must cross clock domains. Clock domain crossings (CDCs) are an important challenge, as improper synchronization may lead to metastability, a condition that occurs when a flip-flop receives a change of its input signal close to the transition edge of its clock signal, violating the setup condition. In these conditions, the flip-flop may enter an undefined state for an unpredictable duration, potentially leading to data corruption and system instability.

Addressing metastability is challenging, as metastability is a condition that arises non-deterministically, making it hard to capture in traditional digital verification flows. An example of a metastable condition is shown in the Figure 7.

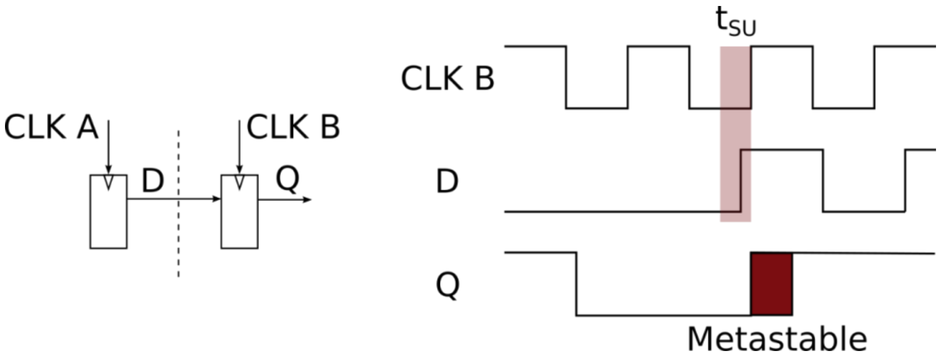


FIGURE 7: EXAMPLE OF A METSTABLE OPERATING CONDITION.

Besides the challenge of correctly identifying and constraining CDCs, every cluster has to be assigned a unique, non-overlapping address range. This address range is used to map all globally addressable registers and memories of the cluster, making them accessible from the SoC or other clusters. This address range must be propagated to every endpoint of the AXI interconnect; this process is error-prone, especially when integrating multiple cluster IPs using differently sized L1 memories or when exploring different cluster configurations.

Correctly propagating address ranges through interconnects becomes more challenging when exploring different interconnect topologies. The choice of interconnect topologies ranges from fully flat crossbar configurations to deeply hierarchical routing trees, which allows to trade-off throughput and contention with the area and routing congestion of the interconnect. This is especially relevant for multi-tile SoC architectures, where many accelerators must communicate with the SoC's memories and peripherals, as the area and energy cost of full crossbar configurations scale quadratically with the number of AXI masters.

Scaling interconnects hierarchically solves this problem but requires tedious reconfiguration of low-level Register Transfer Level (RTL) code. Moreover, finding an optimal configuration trade-off for interconnects is a challenging design problem and requires design space exploration.

In summary, the key challenges for generating scalable interconnects for multi-tile AI SoCs lie in three key problems:

- Verifying correctness of address ranges for interconnect endpoints
- Rapid reconfiguration of complex, multi-tiered hierarchical interconnects
- Correct insertion and constraining of clock domain crossings

In the next Section, we present our CONVOLVE work on Solder, an interconnect generation tool for AXI interconnects, which addresses the core challenges that present itself in heterogeneous multi-tile SoCs.

4.2. Solder

As discussed in the previous Section, solving the various design challenges of multi-tile SoCs requires a structured approach to interconnect design. To address these challenges, we developed Solder, a tool to generate complex interconnects automatically.

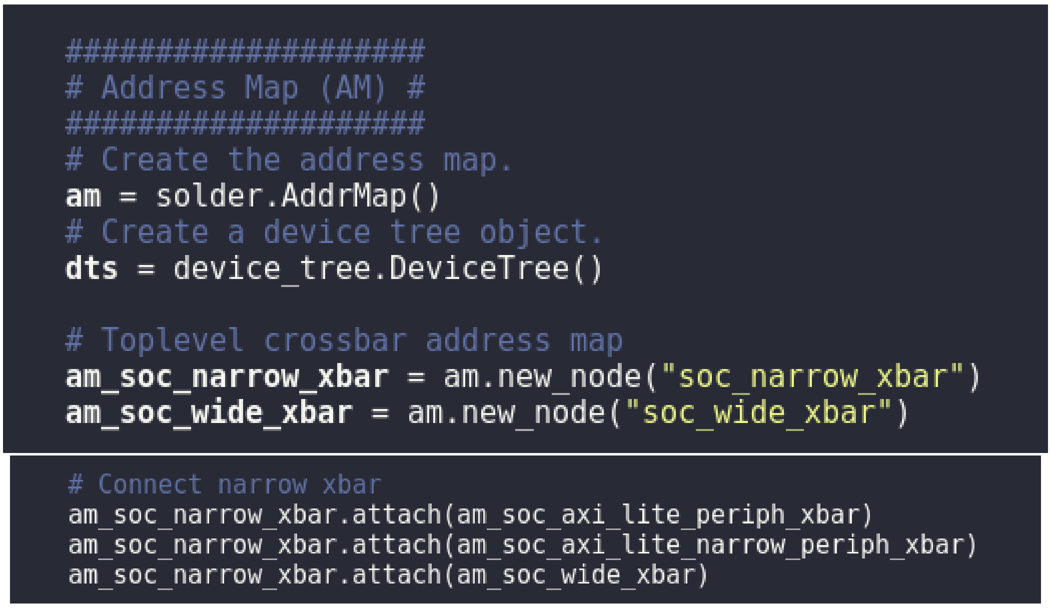
Solder is a Python application that instantiates hand-optimized, open-source System Verilog AXI modules according to topology specifications from JSON files. Solder uses the Mako template system publicly available as a Python library, to instantiate various AXI modules.

We break down the design time benefits of Solder into two sections; First, we show how Solder can be used to construct AXI interconnects and verify address maps, generating correct-by-construction interconnects. Next, we explain how Solder can automatically instantiate appropriate CDC IPs. Finally, we go over applications of Solder in Design Space Exploration (DSE) for AXI interconnects.

4.2.1. Solder Rapid Prototyping Flow

The core abstraction in Solder is the *AddrMap* graph, which models the connections between AXI crossbars and connections with AXI endpoint IPs.

To generate an interconnect with Solder, the designer first provides an addressmap describing each endpoint's address range. Using this addressmap, and a JSON-based configuration for the individual crossbars used in the design, Solder offers a high-level descriptive interface that allows users to connect endpoints to the other nodes, like AXI crossbars. An example of the high-level API used to instantiate interconnect components and attach IPs is shown in Figure 8.



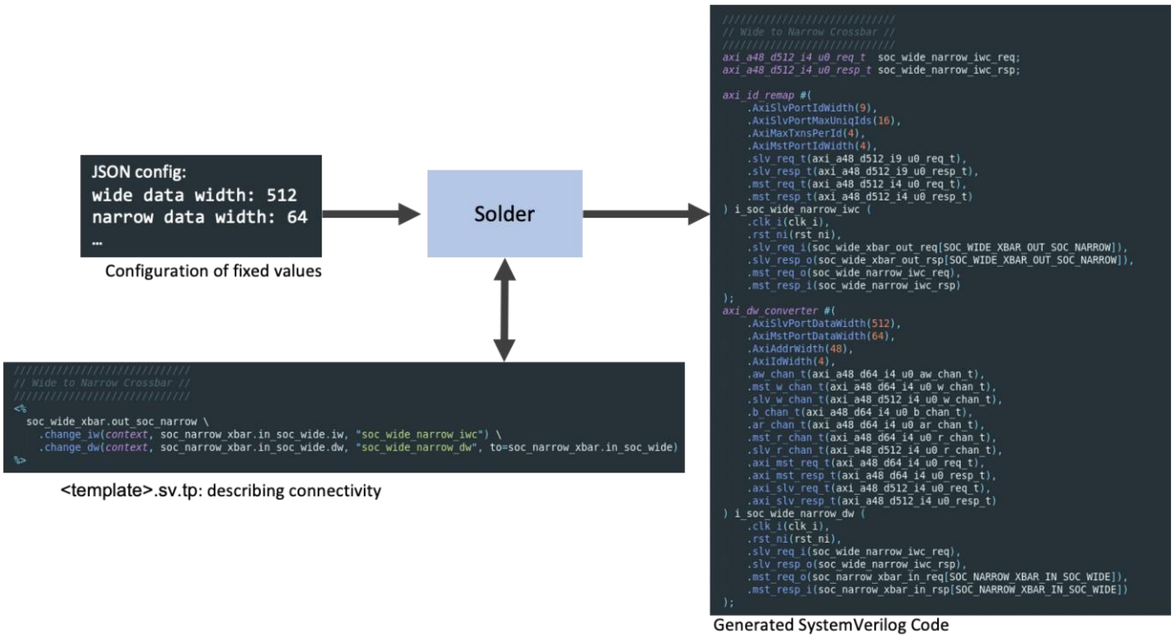
```
####################
# Address Map (AM) #
####################
# Create the address map.
am = solder.AddrMap()
# Create a device tree object.
dts = device_tree.DeviceTree()

# Toplevel crossbar address map
am_soc_narrow_xbar = am.new_node("soc_narrow_xbar")
am_soc_wide_xbar = am.new_node("soc_wide_xbar")
```

```
# Connect narrow xbar
am_soc_narrow_xbar.attach(am_soc_axi_lite_periph_xbar)
am_soc_narrow_xbar.attach(am_soc_axi_lite_narrow_periph_xbar)
am_soc_narrow_xbar.attach(am_soc_wide_xbar)
```

FIGURE 8: EXAMPLE OF A SOLDER SCRIPT TO MODEL THE NARROW INTERCONNECT OF AND SOC.

During the construction of this graph, address rules are generated and propagated through the interconnect structure; this is critical to avoid address map conflicts and ensure correct communication between all endpoints.

While generating interconnects, Solder inserts several standard AXI IPs in appropriate locations automatically; for example, when connecting a wide AXI crossbar with a large transfer size to a narrow AXI crossbar, Solder inserts AXI width and index converters automatically, which take care of converting transfer sizes for correct communication, as shown in Figure 9.



```
JSON config:
wide data width: 512
narrow data width: 64
…
```

Configuration of fixed values
Solder

```
//////////////////////////////
// Wide to Narrow Crossbar //
//////////////////////////////
<%
  soc_wide_xbar.out_soc_narrow \
    .change_iw(context, soc_narrow_xbar.in_soc_wide.iw, "soc_wide_narrow_iwc") \
    .change_dw(context, soc_narrow_xbar.in_soc_wide.dw, "soc_wide_narrow_dw", to=soc_narrow_xbar.in_soc_wide)
%>
```

<template>.sv.tp: describing connectivity

```
//////////////////////////////
// Wide to Narrow Crossbar //
//////////////////////////////
axi_a48_d512_i4_u0_req_t  soc_wide_narrow_iwc_req;
axi_a48_d512_i4_u0_resp_t soc_wide_narrow_iwc_rsp;

axi_id_remap #(
    .AxiSlvPortIdWidth(9),
    .AxiSlvPortMaxUniqIds(16),
    .AxiMaxTxnsPerId(4),
    .AxiMstPortIdWidth(4),
    .slv_req_t(axi_a48_d512_i9_u0_req_t),
    .slv_resp_t(axi_a48_d512_i9_u0_resp_t),
    .mst_req_t(axi_a48_d512_i4_u0_req_t),
    .mst_resp_t(axi_a48_d512_i4_u0_resp_t)
) i_soc_wide_narrow_iwc (
    .clk_i(clk_i),
    .rst_ni(rst_ni),
    .slv_req_i(soc_wide_xbar_out_req[SOC_WIDE_XBAR_OUT_SOC_NARROW]),
    .slv_resp_o(soc_wide_xbar_out_rsp[SOC_WIDE_XBAR_OUT_SOC_NARROW]),
    .mst_req_o(soc_wide_narrow_iwc_req),
    .mst_resp_i(soc_wide_narrow_iwc_rsp)
);
axi_dw_converter #(
    .AxiSlvPortDataWidth(512),
    .AxiMstPortDataWidth(64),
    .AxiAddrWidth(48),
    .AxiIdWidth(4),
    .aw_chan_t(axi_a48_d64_i4_u0_aw_chan_t),
    .mst_w_chan_t(axi_a48_d64_i4_u0_w_chan_t),
    .slv_w_chan_t(axi_a48_d512_i4_u0_w_chan_t),
    .b_chan_t(axi_a48_d64_i4_u0_b_chan_t),
    .ar_chan_t(axi_a48_d64_i4_u0_ar_chan_t),
    .mst_r_chan_t(axi_a48_d64_i4_u0_r_chan_t),
    .slv_r_chan_t(axi_a48_d512_i4_u0_r_chan_t),
    .axi_mst_req_t(axi_a48_d64_i4_u0_req_t),
    .axi_mst_resp_t(axi_a48_d64_i4_u0_resp_t),
    .axi_slv_req_t(axi_a48_d512_i4_u0_req_t),
    .axi_slv_resp_t(axi_a48_d512_i4_u0_resp_t)
) i_soc_wide_narrow_dw (
    .clk_i(clk_i),
    .rst_ni(rst_ni),
    .slv_req_i(soc_wide_narrow_iwc_req),
    .slv_resp_o(soc_wide_narrow_iwc_rsp),
    .mst_req_o(soc_narrow_xbar_in_req[SOC_NARROW_XBAR_IN_SOC_WIDE]),
    .mst_resp_i(soc_narrow_xbar_in_rsp[SOC_NARROW_XBAR_IN_SOC_WIDE])
);
```

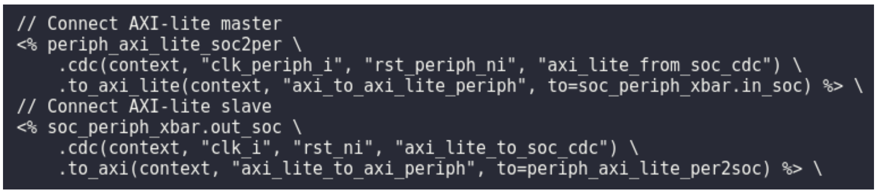
FIGURE 9: OVERVIEW OF THE SOLDER PROTOTYPING FLOW: USING A JSON CONFIG, SOLDER INSERTS AXI IPS INTO SYSTEM VERILOG SOURCE FILES.

Using this functionality, Solder supports rapid prototyping of AXI crossbar widths without tedious manual adaption of RTL code, enabling DSE on the SoC level. Solder supports arbitrary AXI interconnect configurations, allowing to prototype and integrate hierarchical interconnect structures easily. Solder further supports automatic protocol conversion between AXI and AXI Lite and the insertion of CDC IPs to synchronize clock domain crossings.

4.2.2. CDC and Protocol Adapter Instantiation

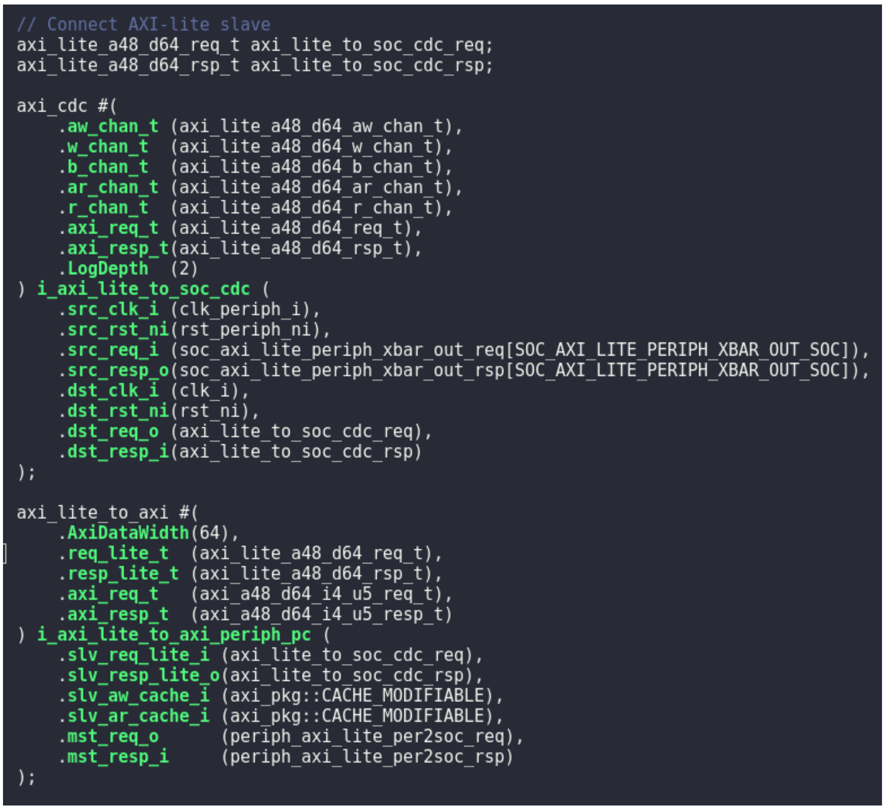
A critical aspect of multi-tile heterogeneous SoCs developed in the CONVOLVE project is clock domain crossing synchronization of asynchronous IP blocks, like clusters. Especially for wide buses, synchronization IPs must be carefully designed and well-constrained to avoid metastability effects.

Solder facilitates implementing CDCs by exposing a high-level API to specify the insertion of CDC IPs. An example of the configuration and the generated System Verilog to instantiate a CDC FIFO is shown in Figures 10 and 11.



```
// Connect AXI-lite master
<% periph_axi_lite_soc2per \
    .cdc(context, "clk_periph_i", "rst_periph_ni", "axi_lite_from_soc_cdc") \
    .to_axi_lite(context, "axi_to_axi_lite_periph", to=soc_periph_xbar.in_soc) %> \
// Connect AXI-lite slave
<% soc_periph_xbar.out_soc \
    .cdc(context, "clk_i", "rst_ni", "axi_lite_to_soc_cdc") \
    .to_axi(context, "axi_lite_to_axi_periph", to=periph_axi_lite_per2soc) %> \
```

FIGURE 10: OVERVIEW OF SOLDER DESCRIPTIONS OF AXI PROTOCOL ADAPTERS THAT ARE AUTOMATICALLY CONFIGURED USING THE JSON CONFIG.



```
// Connect AXI-lite slave
axi_lite_a48_d64_req_t axi_lite_to_soc_cdc_req;
axi_lite_a48_d64_rsp_t axi_lite_to_soc_cdc_rsp;

axi_cdc #(
    .aw_chan_t (axi_lite_a48_d64_aw_chan_t),
    .w_chan_t  (axi_lite_a48_d64_w_chan_t),
    .b_chan_t  (axi_lite_a48_d64_b_chan_t),
    .ar_chan_t (axi_lite_a48_d64_ar_chan_t),
    .r_chan_t  (axi_lite_a48_d64_r_chan_t),
    .axi_req_t (axi_lite_a48_d64_req_t),
    .axi_resp_t(axi_lite_a48_d64_rsp_t),
    .LogDepth  (2)
) i_axi_lite_to_soc_cdc (
    .src_clk_i (clk_periph_i),
    .src_rst_ni(rst_periph_ni),
    .src_req_i (soc_axi_lite_periph_xbar_out_req[SOC_AXI_LITE_PERIPH_XBAR_OUT_SOC]),
    .src_resp_o(soc_axi_lite_periph_xbar_out_rsp[SOC_AXI_LITE_PERIPH_XBAR_OUT_SOC]),
    .dst_clk_i (clk_i),
    .dst_rst_ni(rst_ni),
    .dst_req_o (axi_lite_to_soc_cdc_req),
    .dst_resp_i(axi_lite_to_soc_cdc_rsp)
);

axi_lite_to_axi #(
    .AxiDataWidth(64),
    .req_lite_t  (axi_lite_a48_d64_req_t),
    .resp_lite_t (axi_lite_a48_d64_rsp_t),
    .axi_req_t   (axi_a48_d64_i4_u5_req_t),
    .axi_resp_t  (axi_a48_d64_i4_u5_resp_t)
) i_axi_lite_to_axi_periph_pc (
    .slv_req_lite_i (axi_lite_to_soc_cdc_req),
    .slv_resp_lite_o(axi_lite_to_soc_cdc_rsp),
    .slv_aw_cache_i (axi_pkg::CACHE_MODIFIABLE),
    .slv_ar_cache_i (axi_pkg::CACHE_MODIFIABLE),
    .mst_req_o      (periph_axi_lite_per2soc_req),
    .mst_resp_i     (periph_axi_lite_per2soc_rsp)
);
```

FIGURE 11: OVERVIEW OF THE AUTOMATICALLY GENERATED PROTOCOL ADAPTERS AND CDCS USING SOLDER.

4.3. Results and Future Work

We have implemented Solder, a Python-based tool for generating AXI interconnects. We have tested Solder on several important topologies relevant to multi-tile SoCs. Thanks to the reconfigurability and high-level API, Solder reduces the design- and evaluation time for AXI interconnects significantly, allowing for faster iterations, and, ultimately, higher quality of results in interconnect design.

For future work, we will integrate the Solder prototyping flow with the Cheshire SoC template to enable rapid prototyping on the SoC level, including the integration of Cluster IPs.

4.4. Accelerator Template and Integration

Integrating accelerators is challenging due to the variety of customized kernels and customized designs of each accelerator. Specifically, we have two main problems with integrating multiple accelerators: (1) There does not exist a standard interface that enables smooth accelerator integration at the hardware level while being trans- parent in software. We need this for controlling accelerators with ease. (2) Every accelerator requires custom data and memory layouts to support accelerator- specific data access patterns for efficiency reasons.

To solve these challenges, we developed the SNAX platform to provide uniformity at the hardware and software interfaces and supporting modules to handle the customized data accesses of accelerators. It is an extension of the existing Snitch cluster, but with heavy modifications that can provide convenience to accommodate multiple heterogeneous accelerators.

4.5. SNAX Platform

Figure 12 shows an overview of the entire SNAX platform system. We divide it into three sections: (1) SNAX-Chisel contains Chisel generated modules and support for quick and parametrized accelerator designs, (2) SNAX-Development serves as an intermediate test bench for testing and verification of designs before it connects to the actual cluster, and (3) the SNAX cluster, which is the main compute cluster containing the memories, peripherals, cores, and AXI interconnects for scalability. We've organized the platform in these three sections to make the development modular.

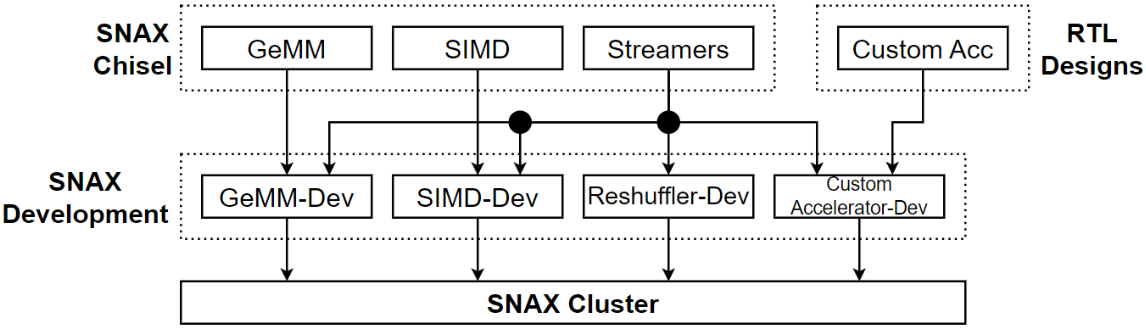


FIGURE 12: OVERVIEW OF SNAX PLATFORM SHOWING DIFFERENT SECTIONS OF DEVELOPMENT

3.10 SNAX Chisel

SNAX Chisel contains chisel generated modules and accelerators made for the SNAX cluster. Since this system upholds the idea of open-source development, Chisel is a powerful open-source to streamline parametrized designs. Parametrized designs are seen in several accelerators like the general matrix-matrix (GeMM) or single input multiple data (SIMD) accelerators. These accelerators can be parametrizable like the bandwidths, the number of processing elements, and the number of I/O ports. For example, the GeMM accelerator shown in Figure 13, can have parametrized sizes for its processing elements.

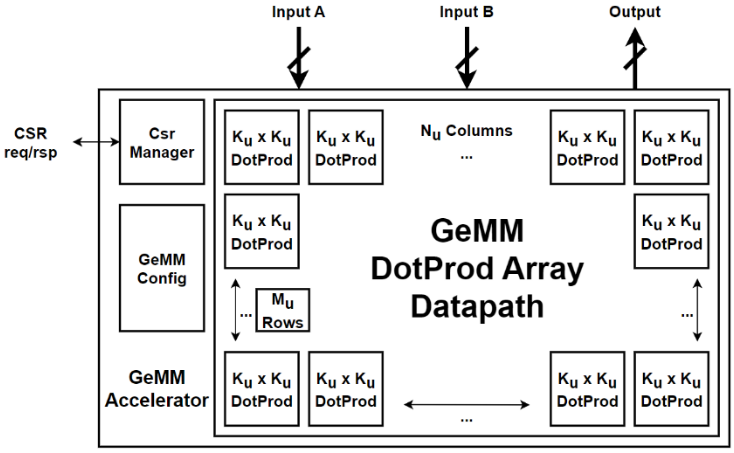


FIGURE 13: 2-D DIGITAL GEMM ACCELERATOR WITH PARAMETRIZABLE SIZES

Another example can be based on the SIMD post-processing unit, which could have any kernel inside it, as shown in Figure 15.

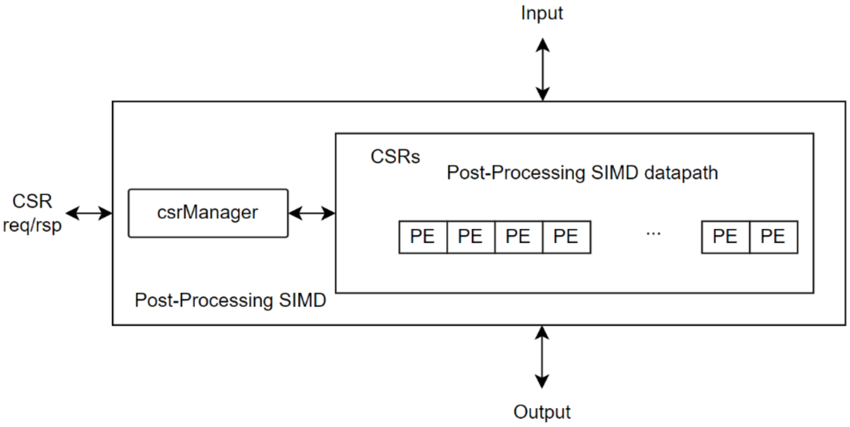


FIGURE 14: SIMD ACCELERATOR WITH PARAMETRIZABLE PROCESSING ELEMENTS

In both cases, we use chisel to generate these designs which can be conveniently hooked up into our SNAX cluster later. The GeMM is available at GitHub: https://github.com/KULeuven-MICAS/snax-gemm while the SIMD is also available at GitHub: https://github.com/KULeuven-MICAS/snax-postprocessing-simd

Efficiently loading and storing data between memory and accelerators is crucial for the dataflow of accelerators while accommodating diverse data access patterns. To address this, we developed a design-time configurable streamer generator that is further run-time programmable. This generator produces hardware streamers that continuously manage multiple data streams between memory and accelerator data path, in a programmable manner. This module aims to alleviate the design complexity of accelerator designers, since this already provides the data access decoupling the design of the accelerator data path. Figure 15 shows this streamer accelerator.

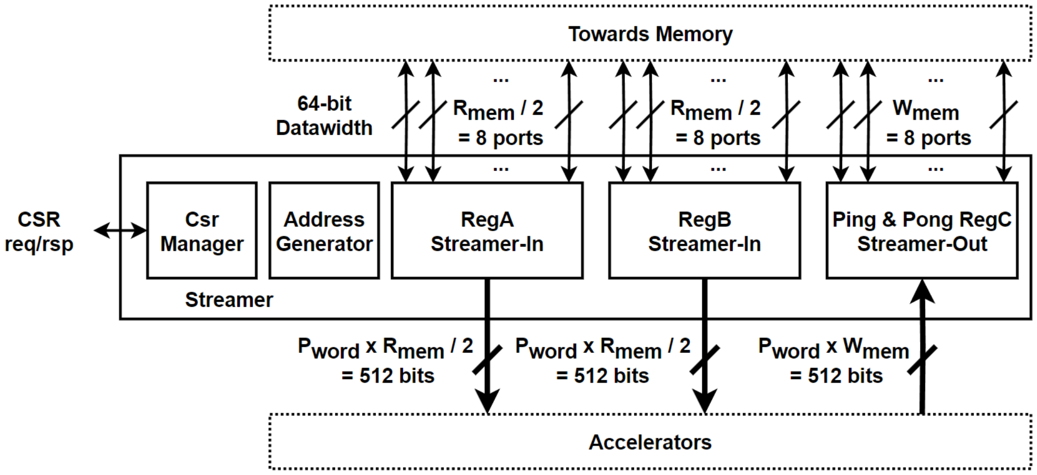


FIGURE 15: STREAMER MODULE TO DECOUPLE ACCELERATOR DATA PATH FROM MEMORY ACCESSES

The microarchitecture of the data streamer comprises of four main components: the data movers (read and write), address generation unit, FIFO buffers, and the CSR manager. Data readers manage data loading, while data writers handle data storing to and from the shared memory ports. Each data mover includes an individually programmable address generation unit (AGU) for autonomous operation. FIFO buffers employ a prefetch mechanism to mitigate data contentions, ensuring continuous data fetching for accelerator ports hiding contention occurrences. Lastly, the CSR manager oversees the CSR commands, which allow the CPU to program for each streamer port the base memory R/W address, as well as the memory stride. It also has a double-buffered CSR for pre-loading CSR configurations while an ongoing task is running. The interface towards the accelerator is a simple data channel with a decoupled interface (valid-ready protocol).

The streamer developed was aimed for users to focus more on the data path of the accelerator. Hence, we highly recommend using our parametrizable streamers when integrating accelerators to our system. The streamer module is available at our GitHub page: https://github.com/KULeuven-MICAS/snax-streamer

It should be noted that in the same level of the SNAX Chisel, some other accelerator designers may have their own custom accelerator design in RTL or some other method. This is just to visualize that at this level, we focus more on the stand-alone accelerator design. We've considered the possibilities that some designers may have their own accelerator implementation strategy.

4.6 SNAX Development

The second section is the SNAX Development which is meant for intermediate verification before plugging into the actual SNAX cluster. Specifically, users need to (1) make a wrapper that complies with the interface of the SNAX cluster, and (2) work on some rigorous verification that's easier to monitor. It emulates the SNAX cluster shell but without all the additional peripherals and control (e.g., CPU core, AXI interconnects, shared peripherals … etc.). It only contains a tightly coupled data memory (TCDM) interconnect, a memory-subsystem, and a Cocotb driver. Figure 16 shows the simplified SNAX development system:

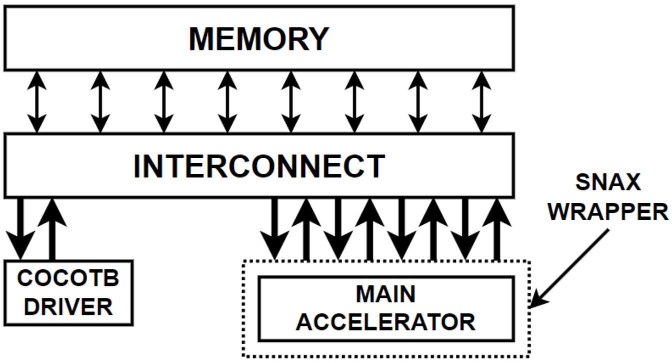


FIGURE 16: SNAX DEVELOPMENT DIAGRAM

The memory is reconfigurable like how it is from the SNAX cluster (see next subsection). We can configure the number of banks, the number of data widths, and the total memory size. The interconnect is a complex interconnect hooked to the accelerator to handle memory requests and data contentions. The accelerator is any custom accelerator a user may integrate, and the Cocotb driver is a Python package[1] used for driving stimuli.

In the SNAX development component, users can do better verification of their accelerator system without having to go through the complexity of the software setup of the SNAX cluster. This is especially useful for hardware designers who just need to focus on what happens on the signal level which are not immediately transparent to the software side. The Cocotb driver is the most tool for verification because the stimulus is driven by a Python program which is easier to driver than in vanilla Verilog testbench stimuli.

Moreover, there is a set of utility Python scripts for generating stimuli. Since the SNAX cluster uses a register-mapped interface to control the accelerators, then writing or reading registers just uses simple register commands. We also provide an emulation of a DMA transfer from the Cocotb driver as well.

The other goal of the SNAX-dev is to prepare the immediate wrappers for easy integration into the SNAX cluster. There are examples that show how to re-map signals from a custom designed accelerator to comply with the simplified interface of the SNAX cluster. With the wrappers already prepared by the SNAX-dev level, integrating into the SNAX cluster is easy since it complies with the custom interface.

The idea is that after development and testing is done in the SNAX-dev, *benderizing* the components into the SNAX cluster makes it convenient for integration. We just need to update the Bender file from the SNAX cluster, and it automatically pulls in the components from the SNAX-dev repositories.

From Figure 12, we can see that each GeMM, and SIMD accelerators each have their own SNAX-dev setups. Interestingly, each setup uses the SNAX chisel generated hardware. Especially the streamer which is an integral part of the accelerator because both GeMM and SIMD are just data path components. There exists a reshuffle and custom accelerators that are not developed with Chisel, but with vanilla Verilog code. However, since the streamer is important for accessing data, we can combine the custom accelerators with the generated streamers.

4.6. SNAX Cluster

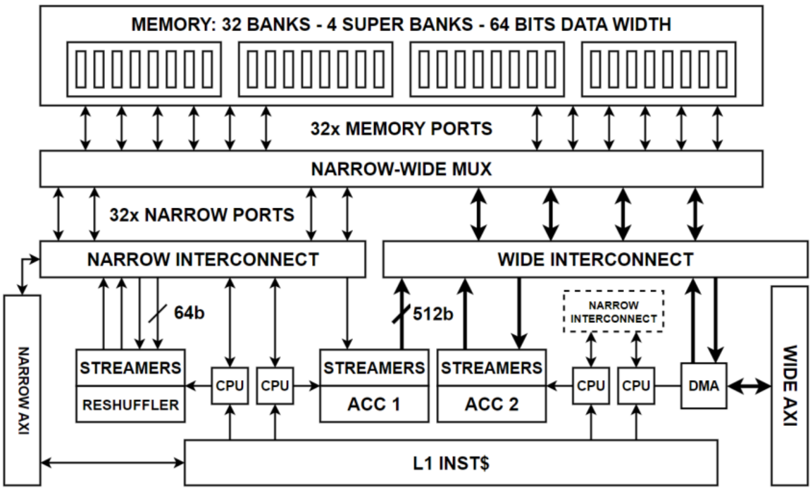


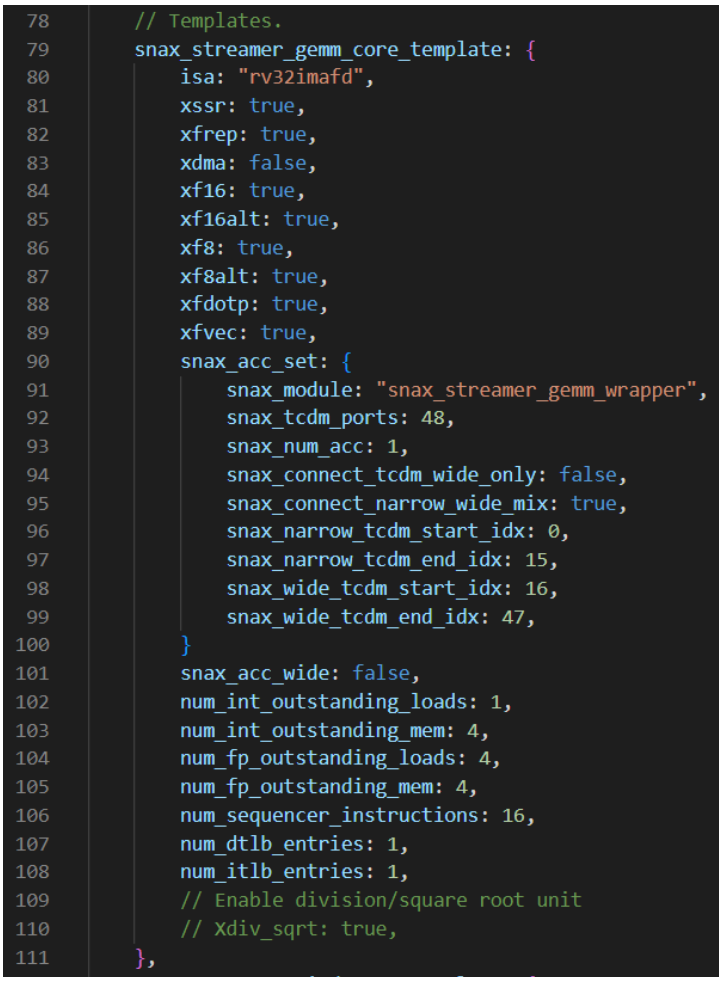
Figure 18: SNAX accelerator-centric compute cluster

The SNAX cluster is the main system that integrates all accelerators and has all the peripherals added to it. Figure 18 shows this. It contains the main memory, a complex TCDM interconnect, Snitch cores for controlling the accelerators, an instruction cache that shared with all the Snitch cores, a DMA to transfer data from an external memory and into the TCDM memory, and AXI interconnects for communicating outwards of the cluster.

The SNAX cluster has several design-time configurations, but a few are notable ones are:

- Number of Snitch cores.
- Number of accelerators controlled per each Snitch core.
- List of accelerators to attach.
- Number of accelerator ports, bandwidth of each port, and how to connect to the complex TCDM.
- Memory size and number of banks.

These notable configurations are the basics for designing the architecture. Since SNAX is an extension of the original Snitch cluster, the cluster is generated through a template with a set of configurations. Figure 19 shows a sample template configuration that also includes other customizable features:



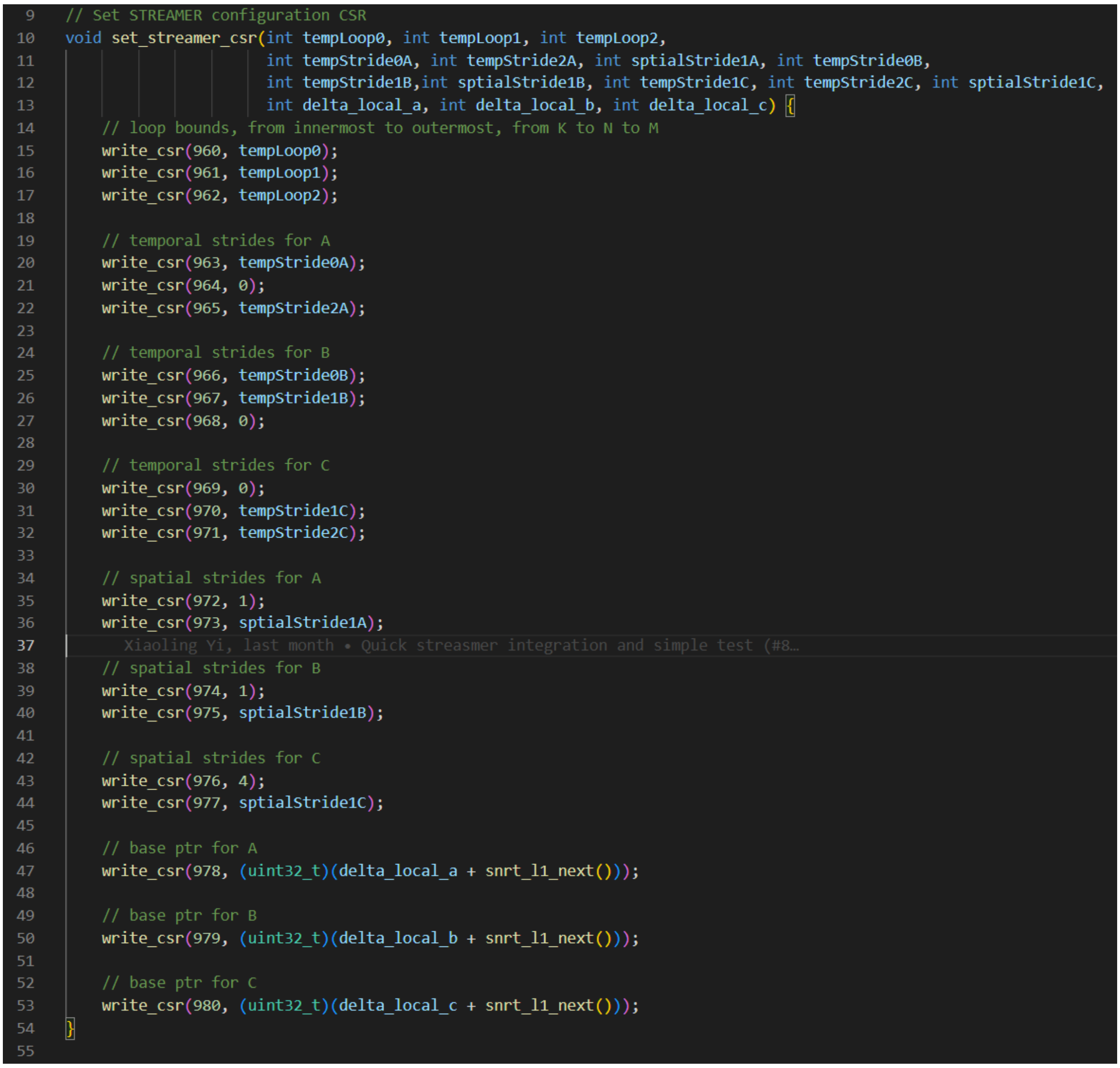
```
    // Templates.
    snax_streamer_gemm_core_template: {
        isa: "rv32imafd",
        xssr: true,
        xfrep: true,
        xdma: false,
        xf16: true,
        xf16alt: true,
        xf8: true,
        xf8alt: true,
        xfdotp: true,
        xfvec: true,
        snax_acc_set: {
            snax_module: "snax_streamer_gemm_wrapper",
            snax_tcdm_ports: 48,
            snax_num_acc: 1,
            snax_connect_tcdm_wide_only: false,
            snax_connect_narrow_wide_mix: true,
            snax_narrow_tcdm_start_idx: 0,
            snax_narrow_tcdm_end_idx: 15,
            snax_wide_tcdm_start_idx: 16,
            snax_wide_tcdm_end_idx: 47,
        }
        snax_acc_wide: false,
        num_int_outstanding_loads: 1,
        num_int_outstanding_mem: 4,
        num_fp_outstanding_loads: 4,
        num_fp_outstanding_mem: 4,
        num_sequencer_instructions: 16,
        num_dtlb_entries: 1,
        num_itlb_entries: 1,
        // Enable division/square root unit
        // Xdiv_sqrt: true,
    },
```

Figure 19: Example configuration for a SNAX cluster

From this configuration, we have a SNAX accelerator set that describes what accelerator to add, the number of TCDM ports it uses, the number of accelerators is controlled with one core, and configurations to indicate how it connects to the TCDM interconnect.

It is worth noting that there are two kinds of TCDM interconnects, a narrow interconnect that has a low bandwidth (64 bits per port) and a wide interconnect that has a higher bandwidth (512 bits per port). The narrow interconnect has fine-grained access to the memory but has higher interconnect complexity as the number of accelerator or core ports increases. The wide interconnect has a higher granularity of access making it less flexible since it needs to access data in super banks (as indicated by the sub boxes in the memory). The wide interconnect: however, has a smaller number of ports to manage and therefore has less circuit complexity.

With the accelerators prepared in the SNAX development phase, these are easily integrated into the system. The SNAX cluster uses RISCV CSR instructions to control the accelerators. We have examples of prepared libraries showing functions with inline assembly instructions in C code to control the accelerators. Because of the register-mapped interface and the CSR instruction control, we can conveniently control the accelerators by just modifying the register configurations. The Figure below shows a code snippet of the SNAX GeMM library showing how to program the accelerator:



```
// Set STREAMER configuration CSR
void set_streamer_csr(int tempLoop0, int tempLoop1, int tempLoop2,
                      int tempStride0A, int tempStride2A, int sptialStride1A, int tempStride0B,
                      int tempStride1B,int sptialStride1B, int tempStride1C, int tempStride2C, int sptialStride1C,
                      int delta_local_a, int delta_local_b, int delta_local_c) {
    // loop bounds, from innermost to outermost, from K to N to M
    write_csr(960, tempLoop0);
    write_csr(961, tempLoop1);
    write_csr(962, tempLoop2);

    // temporal strides for A
    write_csr(963, tempStride0A);
    write_csr(964, 0);
    write_csr(965, tempStride2A);

    // temporal strides for B
    write_csr(966, tempStride0B);
    write_csr(967, tempStride1B);
    write_csr(968, 0);

    // temporal strides for C
    write_csr(969, 0);
    write_csr(970, tempStride1C);
    write_csr(971, tempStride2C);

    // spatial strides for A
    write_csr(972, 1);
    write_csr(973, sptialStride1A);
      Xiaoling Yi, last month • Quick streasmer integration and simple test (#8…
    // spatial strides for B
    write_csr(974, 1);
    write_csr(975, sptialStride1B);

    // spatial strides for C
    write_csr(976, 4);
    write_csr(977, sptialStride1C);

    // base ptr for A
    write_csr(978, (uint32_t)(delta_local_a + snrt_l1_next()));

    // base ptr for B
    write_csr(979, (uint32_t)(delta_local_b + snrt_l1_next()));

    // base ptr for C
    write_csr(980, (uint32_t)(delta_local_c + snrt_l1_next()));
}

```

FIGURE 20: EXAMPLE PROGRAM FOR CONFIGURING THE GEMM

To demonstrate one of our earlier prototypes, we combined three accelerators: a GeMM, a SIMD rescaler, and a data reshuffle. The Figure below shows this architecture:

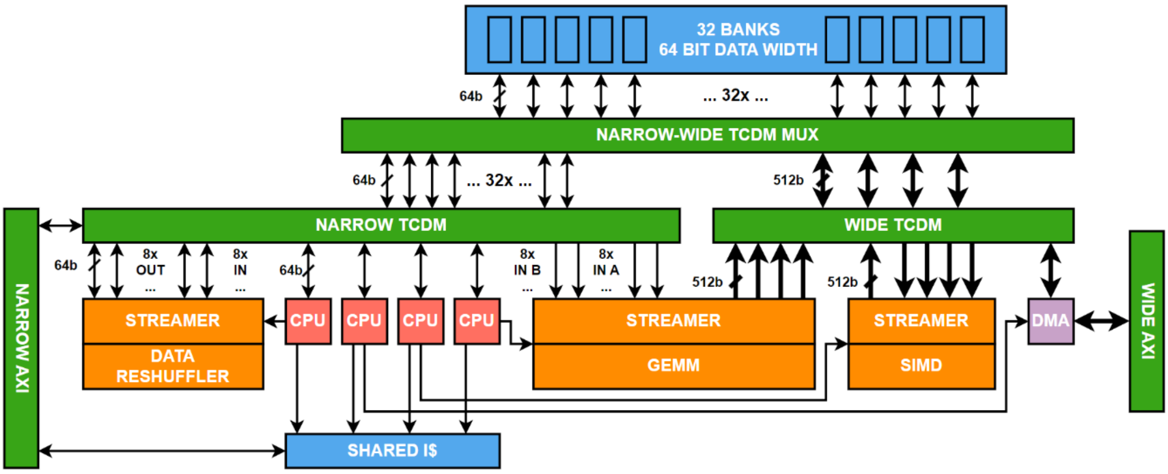


FIGURE 21: AN EXAMPLE INCLUDING A NARROW STREAMER-RESHUFFLER, NARROW-WIDE STREAMER-GEMM AND WIDE STREAMER-SIMD.

In this prototype, the GeMM does an 8x8x8 (8x8 matrices with 8-bit integers each element). It has two input ports, where each takes in 512 bits of 8x8x8 matrices each port. It produces an 8x8 matrix with 32-bits per element, resulting in a 2,048 bits bandwidth for the output. To reduce the complexity of the narrow interconnect, we maintained the inputs of the GeMM on the narrow interconnect and placed its output on the wide interconnect instead. Since the output of the GeMM occupies all 32-banks (each bank is 64 bits) then it makes sense to place it on the wide interconnect instead. The SIMD is a rescale accelerator that takes the 8x8x32 and rescales it back to 8x8x8 to be used in another subsequent matrix multiplication of size 8x8x8. The data reshuffler is a supporting module that is aimed for transposing matrices. For matrix sizes that are larger than 8x8, (e.g., 16x16 or 32x32) we need to divide the matrix into sub-matrices. In operations that require transposes like AxB^T we need to have a dedicated reshuffle to handle element-wise transpositions. This example demonstrates how we can easily integrate multiple heterogeneous accelerators in a single shared memory system.

With the complete SNAX platform, we re-iterate the main sections: (1) SNAX Chisel contains Chisel generated hardware accelerators, and this is where we generate our accelerators. If a user has their own custom accelerator, they can utilize the parametrizable streamer to be augmented into the main datapath. (2) The SNAX development is used as an intermediate verification step. The goal is to create a wrapper that connects correctly into SNAX cluster and to do more rigorous verification before the final integration. Users can also test and monitor the mechanisms in the SNAX development phase. Lastly, (3) is the SNAX cluster with design-time customization on how to connect the accelerators into Snitch cores and into the TCDM memory. With the uniformity of the hardware and software interfaces, and with the customization of data accesses the SNAX platform provides a convenient way for users to integrate their custom accelerators.

5. Conclusion

This report provides a comprehensive overview of SoC architecture and the rapid design and prototyping environment. The report began by outlining the objectives, setting the foundation for the subsequent discussions. It then delved into the SoC architecture overview, explaining the key elements and interconnectivity within the system. The rapid design and prototyping environment were thoroughly explored, highlighting the tools and methodologies employed to accelerate the design cycle.

The significance of SoC architecture lies in its ability to integrate diverse components and subsystems into a cohesive system. The support infrastructure domain, interconnected with L2-clusters, enables rapid SoC generation. This architecture facilitates the integration of various accelerators, such as RISC-V cores with various accelerators sharing tightly coupled data memory (TCDM).

The rapid design and prototyping environment play a crucial role in reducing time-to-market and enhancing productivity. By utilizing advanced tools and methodologies, designers can efficiently generate their SoC designs while reducing the risk of potential issues thanks to reused IPs.

It is important to acknowledge the limitations of this report, such as the focus on an initial architecture and its prototyping environment. Further research can explore improved architectures to overcome challenges such as processing, interconnect and memory bottlenecks.

In conclusion, the report provides detailed information on SoC architecture and the rapid design and prototyping tools and environment in enabling efficient and effective SoC designs. By leveraging these approaches, designers can navigate the complexities of modern chip design and deliver innovative solutions in a timely manner.

6. References

[1] N. Bruschi, G. Haugou, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, "GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors," in *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Institute of Electrical and Electronics Engineers Inc., 2021, pp. 409–416. doi: 10.1109/ICCD53106.2021.00071.

[2] N. Bruschi *et al.*, "Scale up your In-Memory Accelerator: Leveraging Wireless-on-Chip Communication for AIMC-based CNN Inference," in *Proceeding - IEEE International Conference on Artificial Intelligence Circuits and Systems, AICAS 2022*, Institute of Electrical and Electronics Engineers Inc., 2022, pp. 170–173. doi: 10.1109/AICAS54282.2022.9869996.

[3] A. Garofalo *et al.*, "A 1.15 TOPS/W, 16-Cores Parallel Ultra-Low Power Cluster with 2b-to-32b Fully Flexible Bit-Precision and Vector Lockstep Execution Mode," in *ESSCIRC 2021 - IEEE 47th European Solid State Circuits Conference, Proceedings*, Institute of Electrical and Electronics Engineers Inc., Sep. 2021, pp. 267–270. doi: 10.1109/ESSCIRC53450.2021.9567767.

[4] "STM32L4R5xx STM32L4R7xx STM32L4R9xx." [Online]. Available: www.st.com

[5] "This is information on a product in full production. STM32U575xx Ultra-low-power Arm ® Cortex ®-M33 32-bit MCU+TrustZone ® +FPU, 240 DMIPS, up to 2 MB Flash memory, 786 KB SRAM Datasheet-production data Features Includes ST state-of-the-art patented technology Ultra-low-power with FlexPowerControl Core • Arm ® 32-bit Cortex ®-M33 CPU with TrustZone ® , MPU, DSP, and FPU," 2023. [Online]. Available: www.st.com

[6] J. Yue *et al.*, "7.5 A 65nm 0.39-to-140.3TOPS/W 1-to-12b Unified Neural Network Processor Using Block-Circulant-Enabled Transpose-Domain Acceleration with 8.1× Higher TOPS/mm2and 6T HBST-TRAM-Based 2D Data-Reuse Architecture," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 138–140. doi: 10.1109/ISSCC.2019.8662360.

[7] Z. Yuan *et al.*, "A Sparse-Adaptive CNN Processor with Area/Performance balanced N-Way Set-Associate PE Arrays Assisted by a Collision-Aware Scheduler; A Sparse-Adaptive CNN Processor with Area/Performance balanced N-Way Set-Associate PE Arrays Assisted by a Collision-Aware Scheduler," 2019.

[8] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo, "7.7 LNPU: A 25.3TFLOPS/W Sparse Deep-Neural-Network Learning Processor with Fine-Grained Mixed Precision of FP8-FP16," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 142–144. doi: 10.1109/ISSCC.2019.8662302.

[9] J. Song *et al.*, "7.1 An 11.5TOPS/W 1024-MAC Butterfly Structure Dual-Core Sparsity-Aware Neural Processing Unit in 8nm Flagship Mobile SoC," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 130–132. doi: 10.1109/ISSCC.2019.8662476.

[10] S. Ryu *et al.*, "A 44.1TOPS/W Precision-Scalable Accelerator for Quantized Neural Networks in 28nm CMOS," in *2020 IEEE Custom Integrated Circuits Conference (CICC)*, 2020, pp. 1–4. doi: 10.1109/CICC48029.2020.9075872.

[11] Z. Yuan *et al.*, "14.2 A 65nm 24.7µJ/Frame 12.3mW Activation-Similarity-Aware Convolutional Neural Network Video Processor Using Hybrid Precision, Inter-Frame Data Reuse and Mixed-Bit-Width Difference-Frame Data Codec," in *2020*

IEEE International Solid- State Circuits Conference - (ISSCC), 2020, pp. 232–234. doi: 10.1109/ISSCC19947.2020.9063155.

[12] Z. Li *et al.*, "An 879GOPS 243mW 80fps VGA Fully Visual CNN-SLAM Processor for Wide-Range Autonomous Exploration," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 134–136. doi: 10.1109/ISSCC.2019.8662397.

[13] Y.-C. Lo *et al.*, "Physically Tightly Coupled, Logically Loosely Coupled, Near-Memory BNN Accelerator (PTLL-BNN)," in *ESSCIRC 2019 - IEEE 45th European Solid State Circuits Conference (ESSCIRC)*, 2019, pp. 241–244. doi: 10.1109/ESSCIRC.2019.8902909.

[14] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: An Energy-Efficient Deep Neural Network Accelerator With Fully Variable Weight Bit Precision," *IEEE J Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, 2019, doi: 10.1109/JSSC.2018.2865489.

[15] B. Moons, D. Bankman, L. Yang, B. Murmann, and M. Verhelst, "BinarEye: An always-on energy-accuracy-scalable binary CNN processor with all memory on chip in 28nm CMOS," in *2018 IEEE Custom Integrated Circuits Conference (CICC)*, 2018, pp. 1–4. doi: 10.1109/CICC.2018.8357071.

[16] I. A. Papistas *et al.*, "A 22 nm, 1540 TOP/s/W, 12.1 TOP/s/mm2 in-Memory Analog Matrix-Vector-Multiplier for DNN Acceleration," in *2021 IEEE Custom Integrated Circuits Conference (CICC)*, 2021, pp. 1–2. doi: 10.1109/CICC51472.2021.9431575.

[17] H. Jia *et al.*, "15.1 A Programmable Neural-Network Inference Accelerator Based on Scalable In-Memory Computing," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, 2021, pp. 236–238. doi: 10.1109/ISSCC42613.2021.9365788.

[18] P.-C. Wu *et al.*, "A 28nm 1Mb Time-Domain Computing-in-Memory 6T-SRAM Macro with a 6.6ns Latency, 1241GOPS and 37.01TOPS/W for 8b-MAC Operations for Edge-AI Devices," in *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, 2022, pp. 1–3. doi: 10.1109/ISSCC42614.2022.9731681.

[19] J. Yue *et al.*, "15.2 A 2.75-to-75.9TOPS/W Computing-in-Memory NN Processor Supporting Set-Associate Block-Wise Zero Skipping and Ping-Pong CIM with Simultaneous Computation and Weight Updating," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, 2021, pp. 238–240. doi: 10.1109/ISSCC42613.2021.9365958.

[20] J. Yue *et al.*, "14.3 A 65nm Computing-in-Memory-Based CNN Processor with 2.9-to-35.8TOPS/W System Energy Efficiency Using Dynamic-Sparsity Performance-Scaling Architecture and Energy-Efficient Inter/Intra-Macro Data Reuse," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 234–236. doi: 10.1109/ISSCC19947.2020.9062958.

[21] Q. Liu *et al.*, "33.2 A Fully Integrated Analog ReRAM Based 78.4TOPS/W Compute-In-Memory Chip with Fully Parallel MAC Computing," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 500–502. doi: 10.1109/ISSCC19947.2020.9062953.

[22] Z. Chen, X. Chen, and J. Gu, "15.3 A 65nm 3T Dynamic Analog RAM-Based Computing-in-Memory Macro and CNN Accelerator with Retention Enhancement, Adaptive Analog Sparsity and 44TOPS/W System Energy Efficiency," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, 2021, pp. 240–242. doi: 10.1109/ISSCC42613.2021.9366045.

[23] E. Flamand *et al.*, "GAP-8: A RISC-V SoC for AI at the Edge of the IoT," in *29th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2018, Milano, Italy, July 10-12, 2018*, IEEE Computer Society, 2018, pp. 1–4. doi: 10.1109/ASAP.2018.8445101.

[24] GreenWaves, "GAPuino."

[25] Syntiant, "NDP120."

[26] D. Rossi *et al.*, "Vega: A Ten-Core SoC for IoT Endnodes with DNN Acceleration and Cognitive Wake-Up from MRAM-Based State-Retentive Sleep Mode," *IEEE J Solid-State Circuits*, vol. 57, no. 1, pp. 127–139, Jan. 2022, doi: 10.1109/JSSC.2021.3114881.

[27] I. Miro-Panades *et al.*, "SamurAI: A 1.7MOPS-36GOPS Adaptive Versatile IoT Node with 15,000× Peak-to-Idle Power Reduction, 207ns Wake-Up Time and 1.3TOPS/W ML Efficiency," in *2020 IEEE Symposium on VLSI Circuits*, 2020, pp. 1–2. doi: 10.1109/VLSICircuits18222.2020.9163000.

[28] M. Molendijk, F. de Putter, M. Gomony, P. Jääskeläinen, and H. Corporaal, "BrainTTA: A 35 fJ/op Compiler Programmable Mixed-Precision Transport-Triggered NN SoC," Nov. 2022, [Online]. Available: http://arxiv.org/abs/2211.11331

[29] M. Scherer, A. Di Mauro, G. Rutishauser, T. Fischer, and L. Benini, "A 1036 TOp/s/W, 12.2 mW, 2.72 μJ/Inference All Digital TNN Accelerator in 22 nm FDX Technology for TinyML Applications," in *25th IEEE Symposium on Low-Power and High-Speed Chips and Systems, COOL Chips 2022 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., 2022. doi: 10.1109/COOLCHIPS54332.2022.9772668.

[30] V. Jain, S. Giraldo, J. De Roose, L. Mei, B. Boons, and M. Verhelst, "TinyVers: A Tiny Versatile System-on-Chip With State-Retentive eMRAM for ML Inference at the Extreme Edge," *IEEE J Solid-State Circuits*, pp. 1–12, Jan. 2023, doi: 10.1109/jssc.2023.3236566.

[31] P. Houshmand *et al.*, "DIANA: An End-to-End Hybrid DIgital and ANAlog Neural Network SoC for the Edge," *IEEE J Solid-State Circuits*, vol. 58, no. 1, pp. 203–215, Jan. 2023, doi: 10.1109/JSSC.2022.3214064.

[32] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020, doi: 10.1145/3361682.

[33] F. Conti *et al.*, "22.1 A 12.4TOPS/W @ 136GOPS AI-IoT System-on-Chip with 16 RISC-V, 2-to-8b Precision-Scalable DNN Acceleration and 30%-Boost Adaptive Body Biasing," in *2023 IEEE International Solid- State Circuits Conference (ISSCC)*, IEEE, Feb. 2023, pp. 21–23. doi: 10.1109/ISSCC42615.2023.10067643.

[34] Y. Tortorella, L. Bertaccini, D. Rossi, L. Benini, and F. Conti, "RedMulE: A Compact FP16 Matrix-Multiplication Accelerator for Adaptive Deep Learning on RISC-V-Based Ultra-Low-Power SoCs," in *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe*, in DATE '22. Leuven, BEL: European Design and Automation Association, 2022, pp. 1099–1102.

[35] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, "ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1160–1174, 2021, doi: 10.1109/TC.2021.3059962.

[36] A. Symons, L. Mei, S. Colleman, P. Houshmand, S. Karl, and M. Verhelst, "Towards Heterogeneous Multi-core Accelerators Exploiting Fine-grained Scheduling of

Layer-Fused Deep Neural Networks," Dec. 2022, [Online]. Available: http://arxiv.org/abs/2212.10612

[37] H. E. Sumbul *et al.*, "System-Level Design and Integration of a Prototype AR/VR Hardware Featuring a Custom Low-Power DNN Accelerator Chip in 7nm Technology for Codec Avatars," in *Proceedings of the Custom Integrated Circuits Conference*, Institute of Electrical and Electronics Engineers Inc., 2022. doi: 10.1109/CICC53496.2022.9772810.

[38] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings - International Symposium on Computer Architecture*, Institute of Electrical and Electronics Engineers Inc., Jun. 2017, pp. 1–12. doi: 10.1145/3079856.3080246.

[39] K. Seshadri, B. Akin, J. Laudon, R. Narayanaswami, and A. Yazdanbakhsh, "An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks," in *Proceedings - 2022 IEEE International Symposium on Workload Characterization, IISWC 2022*, Institute of Electrical and Electronics Engineers Inc., 2022, pp. 79–91. doi: 10.1109/IISWC55918.2022.00017.

[40] H. Liao *et al.*, "Ascend: A Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : stry Track Paper," in *Proceedings - International Symposium on High-Performance Computer Architecture*, IEEE Computer Society, Feb. 2021, pp. 789–801. doi: 10.1109/HPCA51647.2021.00071.

[41] E. Talpes *et al.*, "Compute solution for tesla's full self-driving computer," *IEEE Micro*, vol. 40, no. 2, pp. 25–35, Mar. 2020, doi: 10.1109/MM.2020.2975764.