RISC-V Reuse Made Easy by Interface Generation and Integration Automation

Ares Tahiraga, Said Llukaj, Wei Zhao, Endri Kaja, Sebastian Prebeck, Wolfgang Ecker
Infineon Technologies, Munich, Germany, <name>.<surname>@infineon.com

I. Introduction and problem statement

Intellectual Property (IP) reuse brought hardware design productivity a big step forward by simplifying reuse: with a single instance several thousand - if not hundreds of thousands of gates - can be inserted in the design. Unfortunately, IP reuse is not as simple as it seems: each IP must be understood, configured and then connected with the signals of the System-on-Chip (SoC). Despite de-facto standards for on-chip busses, scan signals, etc., integration is challenging in one major way: flexible interfaces such as APB, AHB, and AXI from the ARM family, allow for variations in implementation, offer numerous configuration choices, and can be often tailored to enhance performance, power, area (PPA), or to address specific safety and security needs.

For RISC-V, functional configuration is relatively easy due to its modular ISA [1] [2] and profiles that offer pre-configured feature sets [3]. However, the integration of RISC-V IPs remains challenging due to the fact that each RISC-V IP comes with its own communication and infrastructure demand.

II. Solution Approach

To tackle these challenges, we have developed generic modules to connect the RISC-V to its external world. Since modern Hardware Description Languages (HDLs) like SystemVerilog lack support for generic programming concepts, such as optional ports or automated object feature propagation, we have developed a Python-based code generation framework to provide these features as a generation step for HDLs [4]. This framework provides features comparable to Chisel or other Hardware Generation Languages (HGLs) [5], but is centered around its intermediate representation rather than being as a language on its own. This intermediate representation is an abstract time-agnostic model of the hardware, describing its design details. Ease of use is provided by a Python-integrated Domain Specific Language (DSL) for constructing the intermediate. This approach ensures that the formal specification of the design features is consistent with the intermediate model, making it easier to apply transformations to the intermediate model during the generator's development.

This generation framework is not only used to generate the RISC-V IP but also to generate the interface connections of the RISC-V to the SoC using introspection techniques also not supported by HDLs and HGLs.

III. Overview of the Generation Framework

Our framework is based on the Model-Driven Architecture (MDA) [6]. It performs model-to-model transformations, starting from a formalized specification and outputs HDL code. Based on this framework, several specialized generators have been built, including a RISC-V generator and a Bus generator.

On top of the framework stands a Metamodel which is the blueprint that defines the generator's configurable parameters and features. An instance of the Metamodel, called Model of Things (MoT), is used to capture the specific set of requirements describing an IP variant. There can be multiple MoTs, each describing a different design. The MoT for a RISC-V includes the non-privileged ISA, the privileged ISA, and implementation constraints. For example, one MoT may define the extension I, while another defines extensions I, M, and C.

To generate the variant, the MoT is read by the so-called Template of Design (ToD). The ToD encodes the construction logic and design decisions for the IP. The output of the ToD is the Model of Design (MoD), an intermediate time-agnostic non-simulatable model describing the (micro-)architecture of the design. The Template of View (ToV), implemented in Python, extracts information from the MoD and generates the final HDL view. [7]

IV. Bus interface generator

A. Overall Concept and Metamodel

The RISC-V generator itself can be developed in a such a way that it provides compatibility with different bus communication protocols. However, it is nearly impossible to determine in advance which protocol the RISC-V variants must be compliant to and providing support for all the possible protocols is a tedious work. Additionally, this approach risks spending efforts in supporting a protocol that will be never used.

Another approach, is to modify and adapt the RISC-V generator according to the integration need, once the integration requirements are known. This often requires manual intervention in the RISC-V generator to modify the

interfacing logic in order to make it compatible with the SoC interconnect system. Task such creating and renaming ports, ensuring correct encoding, assigning the proper signal size, etc., must be manually repeated for each new IP-to-interconnect combination.

This approach has several major drawbacks. Firstly, it defeats the purpose of using generators if the integration code needs to be manually modified for every use case. Secondly, maintenance of the generator and IP reuse becomes extremely difficult when having several versions of interfacing logic for different variants.

It is important to note, that while in this paper we are focusing on the RISC-V CPU, similar integration challenges also apply to other IPs such as DMAs, AI/ML accelerators, DSP units, etc. These components face comparable issues during integration, and a well-designed Interface Generator provides a unified solution to address these challenges across a variety of IPs.

To overcome these limitations, we propose a solution whose core concept is the separation of IP functionality from communication logic. The RISC-V generator is concerned only with the core functionalities. A separate Interface Generator is used to connect the communication signals of the IP to the external interconnect. While similar approach was introduced in [8] using a hard-to-debug multi-pass methodology, our solution is realized through a streamlined single-pass flow.

Our proposed solution is built on two key observations:

1. Common bus protocols share a unified semantic model: communication features are implemented by one or more signals, and different encodings select distinct modes. A reserved encoding indicates that the feature is inactive. While there are timing variations across protocols, these can also be effectively modeled.
2. Designer intent is known during generator development phase. The designer knows which signal is involved in communication of the CPU with the external SoC, i.e. which signals represent instruction fetch, load/store requests, data payloads, memory addresses, etc. The totality of these signals is, in some way, a defined communication protocol, one that can be described using the same feature and encoding vocabulary.

Based on these two points, we build the following Interface Metamodel as shown in Fig 1. It stores a collection of both RISC-V communication features (*Initiator_Interface*) and Bus communication features (*Bus_Interface*). The class *Feature_Options* contains, in turn, metadata about the signal that implements this feature, including its name, size, mode encodings, etc. The Interface generator will automatically identify corresponding features on both sides and connect their signals. Additional logic is inserted to translate the different encodings and resolve phase or handshake mismatches. The generator can also offload communication tasks from the CPU by synthesizing auxiliary components such as: address generators, buffers, or merged channels with arbitration logic (e.g., for Von Neumann architectures).

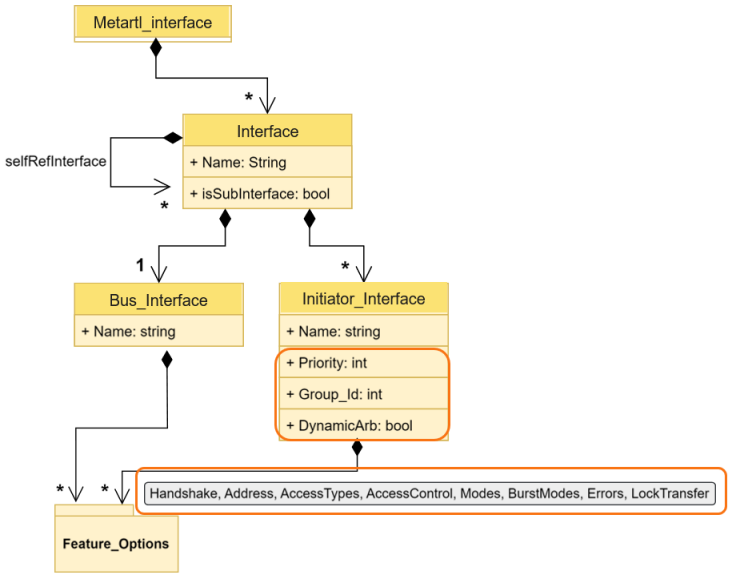

Figure 1 Generic Interface MetaModel

The MetaModel is the highest abstract representation of our solution and it consists of several classes, each playing a crucial role in the generator's functionality. We will go through each of them while highlighting their core functionality.

Root Class: The *metartl_interface* class serves as the root of the model, acting as the entry point for the generator.

Interface Class: The Root Class has a one-to-many relationship with the *Interface* class. Each *Interface* represents the point of connection of the IP with one external bus. The relation is one-to-many, since IPs can be connected to one more than one external bus, i.e., a CPU can be connected to the Instruction Memory and the Data Memory through two separate dedicated communication lines (see Fig. 2a). This relation allows building many interfaces from a single MoT. Furthermore, it enables the configuration of different protocols for each interface. For instance, Instructions are fetched via a simple AHB protocol, while Data is accessed with high throughput AXI protocol (see Fig. 2b).

Each *Interface* is designed to support a configurable number of initiators, represented through a one-to-many relationship with the *Initiator_Interface* class. Additionally, the one-to-one relationship with the *Bus_Interface* class facilitates the configuration of the communication logic. This modeling approach not only accommodates scenarios like von Neumann CPU architectures—where both instructions and data are fetched over a single bus (see Fig. 2c)—but also extends to modern CPUs with tightly integrated accelerators. These accelerators, which often require communication with the external environment, can be modeled by either assigning them a dedicated *Interface* or allowing them to share a common interface with the CPU (see Fig. 2d).

The *Interface class* also includes a Self-Reference Relation. This allows a parent interface to be composed of smaller, child interfaces. This relation is particularly useful when building sub-interfaces or channels that, when combined, form a complete parent interface. For example, the AXI protocol can be effectively modeled using this approach.

Initiator_Interface Class: The *Initiator Interface* class is more complex, and requires more attributes to capture generic behavior. In architectures where multiple Initiators share a single bus, known as *Unified Interfaces*, the connected modules will compete for the bus resource. To manage these scenarios, priorities are introduced to ensure coordinated access to the bus via an arbitration scheme. For a fine granular control on the bus resource, two additional attributes are introduced: *Group_Id* which allows for hierarchical priorities, where initiators with the same *Group_ID* belong to the same arbitration level, and a hierarchy of priorities can be implied not only between individual masters but also between groups of them; *DynamicArb* which allows for each initiator to dynamically raise its priority in specific circumstances in order to break deadlocks or avoid total starvations.

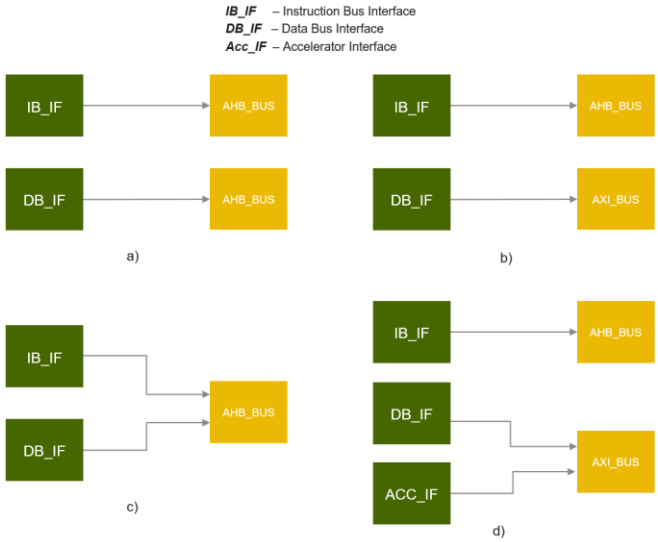

Figure 2 Different Intitator-to-Bus example combinations: a) Harvard Architecture with same bus protocol b) Harvard Architecture with different bus protocols c) von Neuman Architecture d) mixed Architecture

Bus Interface Class: The *Bus_Interface* class is relatively simple because the bus interface is treated as a single point-to-point communication. This approach eliminates the need to account for race conditions or handle any special circumstance.

Both, the protocol supported by the *Bus_Interface* and the communication needs of *Initiator Interface* are described in details with the help of *Feature Options*, explained in the next section.

B. Feature Options

The *Feature_Options* class provides an abstract representation of communication features. These features were identified by a thorough considerations of AMBA protocols, namely APB [9], AHB [10] and AXI [11]. Careful analysis was conducted in order to find all the common features among these protocols, in order to be able to model them in a single generator. The same feature set is then reused also for describing the communication needs of RISC-V CPUs. Each of these protocols is suitable for different SoC, fulfilling varying requirements and use cases.

As shown in Fig. 3, these protocols have an incremental complexity. APB is used for Low-speed peripherals and low power consumption. AHB is used in systems with mid to higher performance requirements, where memory and register accesses are frequent. AXI is used on high speed devices where low latency is a necessity.

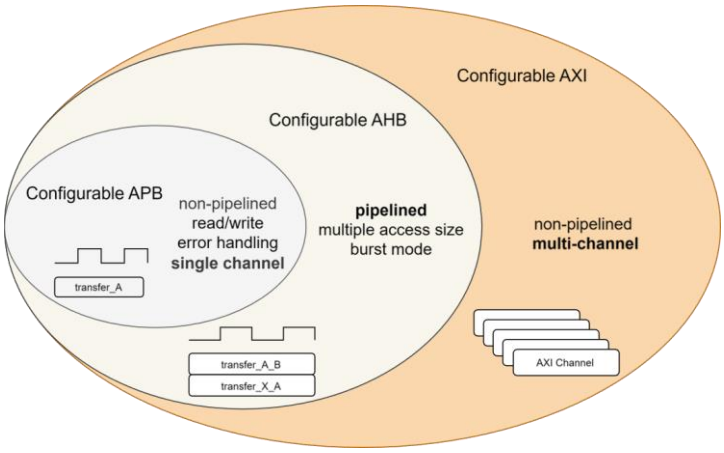

Figure 3 Incremental Complexity of supported protocols in Interface Generator

Before exploring the specifics of each feature, it is important to first introduce two key classes that serve as the foundation for building our models: the *Signal* and the *Encoding* classes.

Signal class is used to define signals or groups of signals that are associated with a specific feature (*DataSignal*) or specific encodings (*ControlSignal*). Each signal is defined by its unique characteristics, such as *phase, size, and default value*. Among these, the phase is especially critical, as it determines whether the transfer is pipelined or non-pipelined.

Encoding class is closely related to the *Signal* class and it allows for the configuration of special signal values. Via the encoding class, users can specify the value for which the signal activates a given feature. For example, *Feature Request* is activated when *Signal WRITE* has *Encoding* of *1'b1*.

The complete overview on the supported features is shown in Fig. 4 and details are given below.

Handshake Feature is built around two primary classes: *Request* and *Ready*. The *Request* class defines the required signals and their encodings, encompassing the process by which an *Initiator* initiates a transfer request. Complementing this, the *Ready* class ensures proper functioning of the handshake mechanism by indicating a module's readiness to respond to a request. It guarantees accurate coordination, preventing information loss or mismatched transactions.

Access Types: The *AccessTypes* class defines whether the transaction intends to perform a Read or a Write operation.

Access Controls: The *AccessControls* class specifies the size of data for bus transfers. This model allows for configurable data sizes, including Byte, HalfWord, Word, and DoubleWord transfers, with each size defined independently.

Addressing: The *Address* class represents the presence of an address signal, which is essential in all bus communications that rely on address-based protocols. This signal does not require any specific encoding since its dynamic values are only propagated from the IP to the bus..

Burst Modes: The *BurstModes* class facilitates efficient communication by offering a range of burst transfer options. Each transfer mode is associated with a specific signal and encoding. Additionally, the Bus Interface Generator incorporates an address generator component, which enhances burst transfer operations by managing address computations, thereby relieving the master of this responsibility.

Modes: The *Modes* class specifies the various states or modes that the communication line can operate in, i.e IDLE, BUSY, SEQUENTIAL, NONSEQUENTIAL, etc.

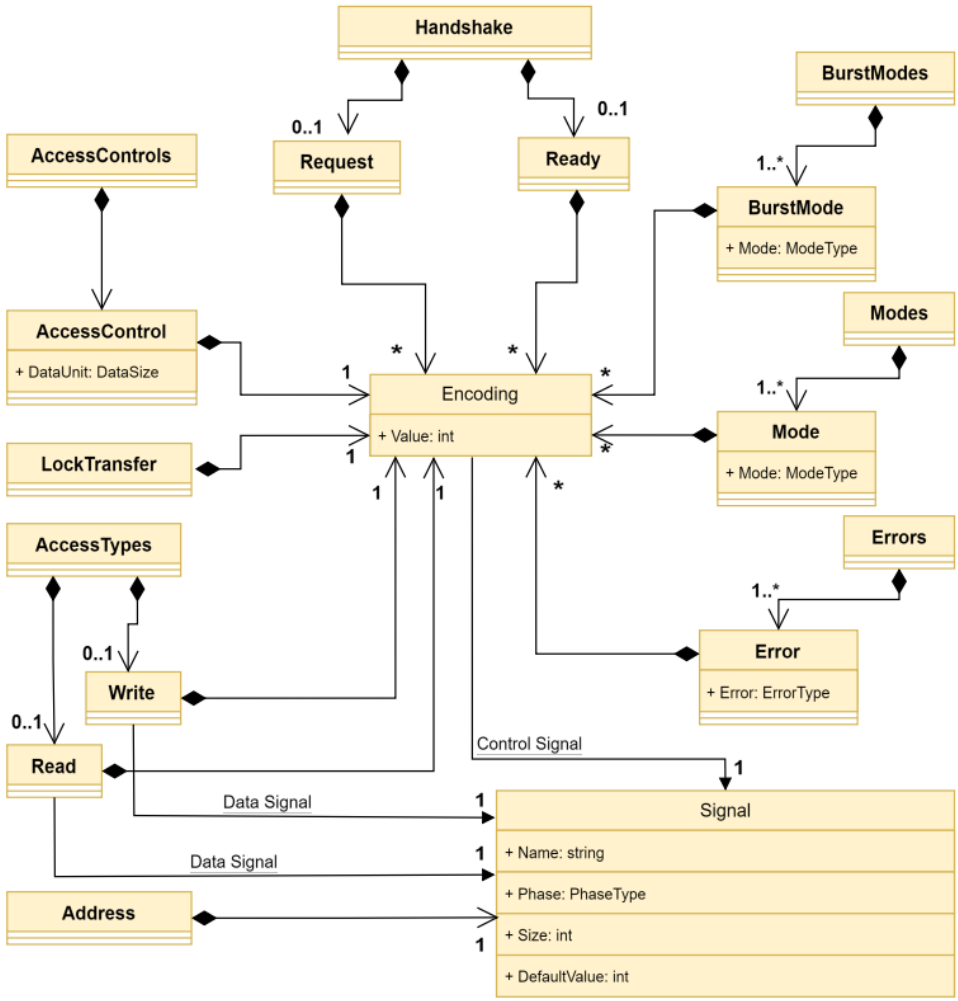

Figure 4 Supported Features Metamodel

Error: The *Errors* class is designed to ensure transfer accuracy by representing and managing erroneous behaviors, as well as the associated signals and their encodings for each protocol.

Lock Transfer: The *LockTransfer* class is essential for signaling atomic accesses and safeguarding data consistency and integrity within a multi-master configuration.

C. Hardware Architecture of Generated Interface

The internal logic of the generator performs the feature mapping from initiator towards the bus. The generator is built to support all valid configurations, manage the design implementation of various features, and produce well-structured, embedded interfaces for each module, initiator, and bus.

Fig. 5 provides a view of the generated hardware components, illustrating the mapping of multiple descriptive initiator interfaces that are bus-agnostic to a single unified bus interface that finally is connected to the external communication bus. The main components are:

- *Bridge* is responsible to collect the signals of the initiator and map them to the respective signal of the bus, based on the common feature the signals support. The Bridge is also responsible of translating the initiators signal value from its encoding to the respective bus encoding. When the phase of a specific feature differs between the initiator and the bus, the bridge takes care of synchronizing the respective signals by introducing delays and stalling the pipeline to ensure timing alignment.
- *Arbitration_Unit* is tasked with managing priority resolution and granting access to the shared bus resource in situations where multiple initiators share a unified interface. The generator is specifically designed to handle the full range of coordination rules defined by supported bus protocols. This involves rigorous

consideration of how transfers are initiated, how wait states are managed and communicated, and how the completion of transfers is signaled.

- *Master_Select* component ensures correct routing of the signals belonging to the selected bridge that has been granted access from the Arbitration_Unit.

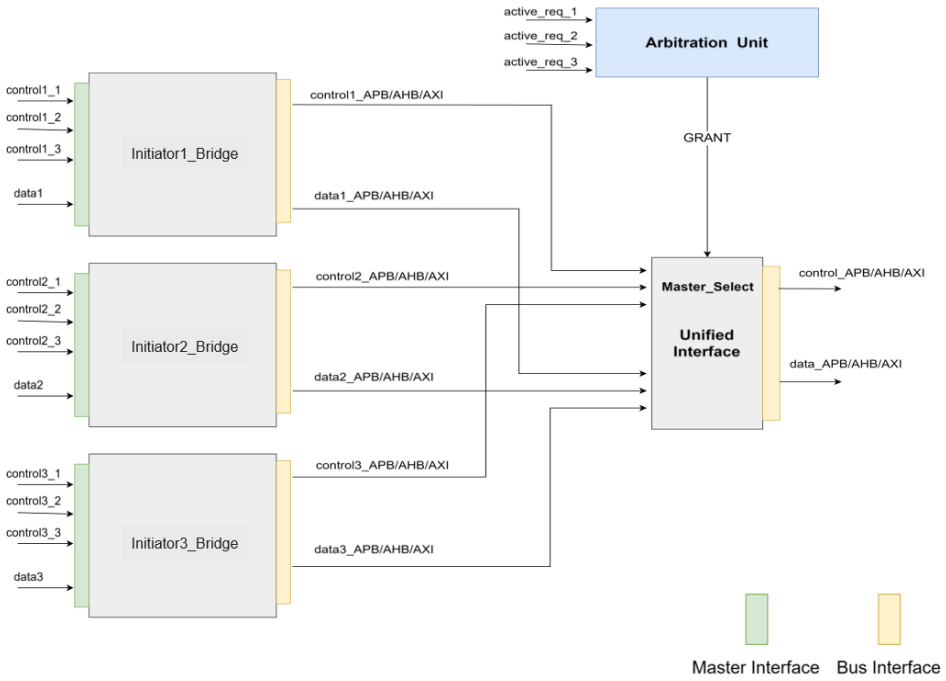

Figure 5 Architecture of Unified Interfaces

V. AUTOMATED INTEGRATION AND CONNECTION

Fig 6 shown a visualization diagram of the automatic integration and connection flow of RISC-V cores into a given SoC. For automatic integration of the RISC-V with a SoC interconnect, the interface generator requires information about the communication signals involved on both sides.

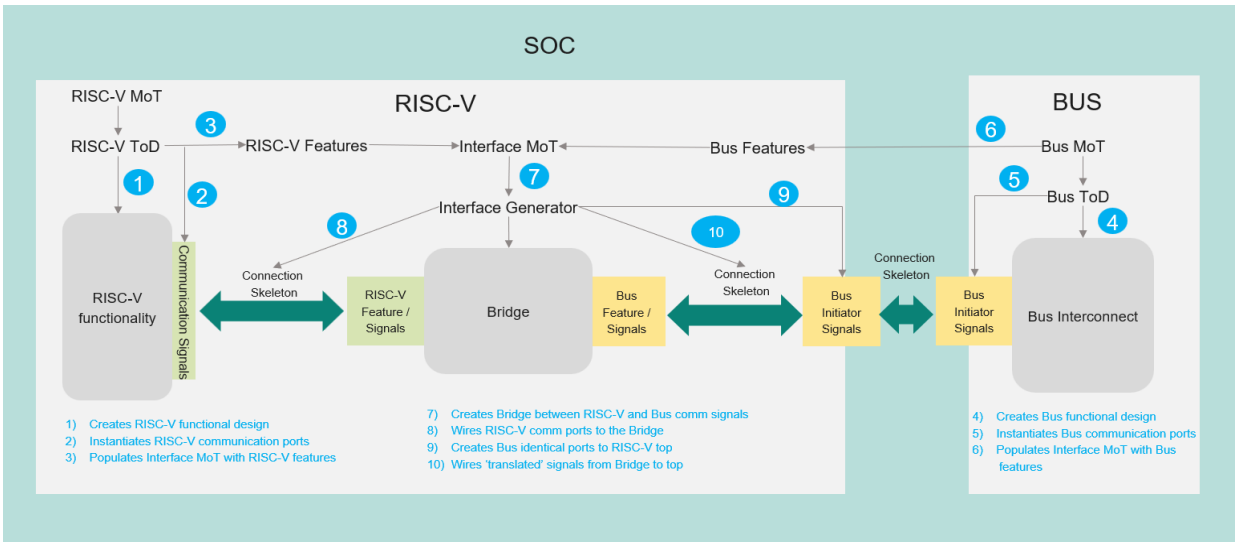

Figure 6 Overall Flow of RISC-V IP Generation and Integration

From the RISC-V side, the RISC-V ToD specifies the communication features in the Generic Interface MoT, populating the *Initiator_Interface* object. From the Bus side, the communication features are already defined and

encoded in the Bus MoT that was used to generate the bus interconnect. The Interface generator just needs to read this MoT, and populate its own *Bus_Interface* object. It shows how the Generic Interface MoT is built automatically with RISC-V and Bus Features. Finally, it connects the RISC-V communication signals via the Bridge to the top of the RISC-V wrapper and from there to the Bus interconnect.

VI. EXPERIMENTAL RESULTS

To emphasize the ease of use and efficiency of this flow, we provide an industrial case study involving the integration of an in-house 5-stage RISC-V core into an AHB-based SoC.

Initially, the MoT is configured with details from the bus architecture. In the selected scenario, the target SoC consists of two buses: an instruction bus and a data bus. Consequently, the IP requires two interfaces. Both buses adhere to the AHB-Lite protocol and support an identical set of features, which are summarized in Table. 1 below.

Bus transaction initiation is managed by the HTRANS signal, while the HREADY signal indicates either successful communication or wait states. The HWRITE signal determines the type of access, with HWRITE=1 representing a WRITE operation and HWRITE=0 indicating a READ operation. The size of the data transfer is encoded using the HSIZE signal. Targeted address information is specified in the HADDR signal, and the HRESP signal denotes any errors that may occur during transactions. Additionally, the transfer mode, such as SEQUENTIAL or BURST, is represented by the HTRANS signal.

Table 1 Modeling of AHB-Lite in features

Feature	Control Signal	Data Signal
Handshake_Request	HTRANS	N/A
Handshake Ready	HREADY	N/A
Access_Write	HWRITE	HWDATA
Access_Read	HWRITE	HRDATA
AccessControls	HSIZE	N/A
Mode	HTRANS	N/A
Address	HADDR	N/A
Error	HRESP	N/A

Similarly, we need to describe also the initiator interfaces that need to be connected. The RISCV 5-stage core has two initiators: one originating from the Prefetcher and the other from Execute stage. The Prefetcher Initiator must be connected to the Instruction Bus while the Execute Initiator to the Data Bus. The respective configuration is summarized in Table 2. Both initiators share a comparable configuration, with the exception of the Prefetcher lacking the write access feature, as it only fetches instructions and never needs to write to the Instruction memory. This difference is highlighted in blue within the table.

When the Prefetcher is ready to fetch a new instruction, it asserts the IB_ren_out signal, which is subsequently translated into the corresponding HTRANS encoding to initiate a transfer request. The response from the HREADY signal is then routed back to the Prefetcher via the IB_ext_en_in signal. As the Prefetcher does not perform write operations, it does not utilize any signals associated with this functionality. Consequently, the HWRITE signal is always set to zero from the bus perspective.

Table 2 Prefetcher/Execute Initiator Configuration

Feature	Control Signal		Data Signal	
	Prefetcher	*Execute*	*Prefetcher*	*Execute*
Handshake_Request	IB_ren_out	DB_wen_out or DB_ren_out	None	None
Handshake Ready	IB_ext_en_in	DB_ext_en_in	None	None
Access_Write	None	DB_wen_out	None	DB_wdata_out
Access_Read	IB_ren_out	DB_ren_out	IB_rdata_in	DB_data_in
AccessControls	IB_access_size_out	DB_access_size_out	None	None
Mode	None	None	None	None
Address	IB_addr_out	DB_addr_out	None	None
Error	IB_accessfault_in	DB_accessfault_in	None	None

The IB_addr_out signal specifies the address to be fetched, which can represent the next instruction address, the branch target address, or the exception handler address. In the event of a Bus Error, the IB_accessfault_in signal is

asserted. Once the instruction is fetched from the Bus via the HRDATA signal, it is delivered to the Prefetcher through the IB_rdata_in signal. The IB_access_size_out signal remains constant and indicates that only WORD-sized transfers are supported. Since Burst transfers are not enabled in this configuration, no signal is assigned to the Mode feature. As such, the HTRANS signal defaults to the encodings for NONSEQUENTIAL and IDLE transfers.

The behavior of the Execute Initiator is similar. The interface signals are controlled by the logic that manages LOAD and STORE instructions. To signal a bus request and specify the type of operation, the DB_wen_out or DB_ren_out signals are asserted. Specifically, DB_wen_out is asserted during STORE instructions to indicate a WRITE operation, while DB_ren_out is asserted during LOAD instructions to signal a READ operation.

The DB_access_size_out signal is determined by the logic that identifies the specific type of the load/store instruction—byte, halfword, or word—and is encoded accordingly. The DB_addr_out signal is driven by the calculated absolute address, resolved based on the offsets and base addresses of the load/store instructions. For STORE instructions, the data to be written to memory is provided via the DB_wdata_out signal. Conversely, the DB_data_in signal carries data fetched during LOAD instructions.

The result of the transmission is communicated back to the Execute Initiator using the DB_ext_in signal in the case of a successful operation, or via the DB_accessfault_in signal to indicate a failed operation, each driven by HREADY and HRESP respectively.

Now that the first MoT has been established, the simplicity and adaptability of our flow become evident. To adapt the core to a single-bus SoC, one simply needs to remove one of the Bus Interfaces. To enable Burst Mode functionality, it is merely a matter of adding the necessary signals to the MoT in the Initiator side. Similarly, integrating the core into an AXI-based SoC only requires updating the bus configuration, without any additional effort needed to modify the RISC-V core itself. Unlike conventional HDL flows, which demand considerable manual effort to update the RTL description, add new logic, and modify ports for each design adaptation, our flow transforms these complex tasks into effortless configuration adjustments managed directly by the generator, significantly improving design efficiency.

VII. CONCLUSIONS

This automatic integration enables several established flows that help ensure high quality of the RISCV generator, accelerate design cycles and have a detailed profiling of the supported variants. These flows include:

- *Testing and regression*: Run software to validate functional correctness across variants.
- *Benchmarking*: Evaluate the performance of internal and external cores for comparison
- *Design Space Exploration:* Sweep RISC-V microarchitectures and bus protocols to assess PPA/performance trade-offs.
- *Faster product integration:* reduces manual effort during integration to external products.

A simple sweeper script can iterate over RISC-V MoTs and trigger full generation and integration per variant. Without automated integration, such continuous exploration and validation would require continuous human intervention slowing the timeline.

REFERENCES

[1] RISC-V International, “The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA,” Version 20250508, 2025. Available at: riscv.org

[2] RISC-V International, “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture,” Version 20250508, 2025. Available at: riscv.org

[3] “RISC-V RVA23—A Major Milestone," Electronic Design, [online]. Available: electronicdesign.com. [Accessed: 05-09-2025].

[4] Schreiner, J., Gerl, D., Kunzelmann, R., Sinha, P. K., & Ecker, W. (2024). The Argument for Meta-Modeling-Based Approaches to Hardware Generation Languages. arXiv preprint arXiv:2404.05599.

[5] Lockhart, D., & Batten, C. (2014, February). Hardware generation languages as a foundation for credible, reproducible, and productive research methodologies. In Workshop on Reproducible Research Methodologies (REPRODUCE).

[6]Truyen, F.: The Fast Guide to Model Driven Architecture [online]. Available: https://www.omg.org. [Accessed: 12-01-2026].

[7] Schreiner, Johannes ; Ecker, Wolfgang. / Digital hardware design based on metamodels and model transformations. VLSI-SoC: System-on-Chip in the Nanoscale Era – Design, Verification and Reliability - 24th IFIP WG 10.5/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2016. editor / Ricardo Reis ; Thomas Hollstein ; Jaan Raik ; Sergei Kostin ; Anton Tsertov ; Ian O’Connor. Springer New York LLC, 2017. pp. 83-107 (IFIP Advances in Information and Communication Technology)

[8] J. Schreiner, V. R. Gontia, S. Prebeck and W. Ecker, "Generator IP-reuse and Automated Infrastructure Generation for Model-based Full-Chip Generation," MBMV 2023; 26th Workshop, Freiburg, 2023, pp. 1-12.

[9] ARM, "AMBA APB Protocol Specification", ARM Limited, 2023. [Online]. Available at: https://developer.arm.com/documentation/ihi0024/latest/ [Accessed: 27-10-2025]

[10] ARM, "AMBA AHB Protocol Specification", ARM Limited, 2021. [Online]. Available at: https://developer.arm.com/documentation/ihi0033/latest/ [Accessed: 27-10-2025]

[11] ARM, "AMBA AXI Protocol Specification", ARM Limited, 2025. [Online]. Available at: https://developer.arm.com/documentation/ihi0022/latest/ [Accessed: 27-10-2025]