SCAL: An Open-Source Scalable Core Adaptation Layer for Interfacing RISC-V ISA Extensions

Brindusa Mihaela Damian-Kosterhon, Florian Meisel, Andreas Koch
Embedded Systems and Applications Group
Technical University of Darmstadt
Darmstadt, Germany
damian@esa.tu-darmstadt.de, meisel@esa.tu-darmstadt.de, koch@esa.tu-darmstadt.de

***Abstract*—Instruction Set Architecture eXtensions (ISAX) are one of the key methods to improve the performance of software-programmable processors. However, adding them into processor cores is cumbersome and becomes even more challenging if the ISAX needs to be *portable* among multiple *different* cores. Existing extension interfaces aim to alleviate these difficulties. However, they lack an *abstraction layer* that encapsulates commonly used adaptation mechanisms for mediating between the base core and the ISAX. Typical adaptation issues include latency matching, shared resource management, or hazard handling.**

For this purpose, we propose the Scalable Core Adaptation Layer (SCAL), which builds on an existing open-source interface, SCAIE-V. By using SCAL, we show that for ISAXes reading and returning a result, their interface width and semantics remain unchanged independent of their latency, number, or base core. We demonstrate this for four cores with different microarchitectures. Moreover, the ISAX is agnostic of its execution mode (in-pipeline or in parallel to the main pipeline), and its datapath does not change even if the result is written to the memory, program counter, or a new internal state element. This brings us closer to fast ISAX development and integration into multiple cores.

***Index Terms*—SCAIE-V, ISAX, Portability, RISC-V**

I. Introduction

Extending an existing Instruction Set Architecture (ISA) is an attractive approach to obtaining faster and more power-efficient designs for specific applications. Yet, the tremendous advantage of custom instructions comes with high expenses like engineering effort, chip area, frequency drop, or license costs. To make Instruction Set Architecture eXtensions (ISAXes) more accessible, multiple cores already provide specialized interfaces for ISAX integration.

Some commercial tools for ISAX integration are available, such as CodAL [1]. However, these are proprietary and not portable to non-vendor cores. The open-source community also provides ISAX interfaces. Some are general-purpose [2], while others are tailored for a specific application, like vector processing [3]. Yet, the latter can reuse the core's ALU, while other solutions are similar to a separate execution unit [4]–[6]. Such implementations usually return a result but don't allow LSU access. RoCC [7] gives access to the L1 cache. [5] has the limitation that an ISAX must take at least two cycles, and [8] stalls the core until the ISAX is finished. Still, the most common limitation is the lack of support for branching instructions [2], [5], [8]–[10]. Moreover, they were not evaluated on several open-source cores. Thus, porting an application with ISAXes to another processor is non-trivial. To approach this issue, a portable interface that can be used across different microarchitectures was presented in [11]. We base our paper on this work and further extend the technology to reduce the effort to port the interface to new cores, and also ease ISAX generation.

SCAIE-V [11] provides a tight ISAX integration, allowing better ISAX-core communication than more loosely coupled approaches. Through this interface, ISAXes can read or write the register file, read or update the program counter, read instruction fields, or start memory transactions. Some of these operations can be moved between earlier or later pipeline stages (for better ISAX timing) or even be decoupled from the processor (for better parallelism). The interface relies on signal taps, which are automatically inserted into the base core. Moreover, on top of these raw signals, the interface defines higher-level functionality, which is required for each core, like hazard handling. We observed many commonalities that can be factored out into a reusable abstraction layer. This layer, the Scalable Core Adaptation Layer (SCAL), is shown in Fig. 1 and is the topic of this work.

There are multiple reasons for a common abstraction layer. SCAL can ease the support of new cores, and ease adding new interface capabilities for all cores. Unifying much of the per-core logic into SCAL also simplifies the verification problem, as SCAL can be verified independently of the base cores. Moreover, an abstraction layer can also facilitate ISAX development. Some ISAXes require instruction-internal state. Similar to the core's main register file, such state also needs hazard-handling mechanisms. This can be abstracted into SCAL. Similar abstractions can be introduced for the writeback, both of in-pipeline as well as decoupled ISAXes. We identified multiple points where such per-ISAX logic can be abstracted into a shared adaptation layer, enabling easier ISAX generation, as an HLS tool (e.g., [12]) no longer has to handle these integration complexities.

To sum up, we present the Scalable Core Adaptation Layer (SCAL), which makes the following contributions to the problem space of extending processors with ISAXes:

- easier ISAX generation
- improved ISAX portability across cores
- better area efficiency by sharing interface logic across ISAXes

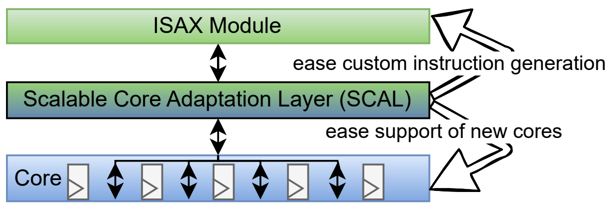
ISAX Module
ease custom instruction generation
Scalable Core Adaptation Layer (SCAL)
ease support of new cores
Core

Fig. 1. Proposed configuration with a middle adaptation layer to ease ISAX generation and core support.

- reduced complexity within the base core
- reduced maintenance and verification effort by refactored logic

II. Fundamentals and Terminology

Before we discuss our architecture, we will first establish fundamental concepts. By ISAX, we mean a single custom instruction. The individual interactions that occur between the core and an ISAX during the latter's execution are called *nodes* or *operations*. These will be scheduled to take place in the base core's pipeline *stages*. Note that not all the cores have a pipeline microarchitecture, such as PicoRV32, which is a multicycle core. Here, we will use the term *stage* to specify the cycle number in which a node is scheduled to occur, defining that the *fetch* of a new instruction occurs in Stage/Cycle 0.

SCAIE-V employs different operations, which will be abbreviated here as follows: register file reads (**`RdRS`**) and writes (**`WrRD`**), program counter (PC) reads (**`RdPC`**) and updates (**`WrPC`**), memory reads (**`RdMem`**) and writes (**`WrMem`**), accessing the instruction fields (**`RdInstr`**), and accessing stall signals (**`RdStall`**, **`WrStall`**) and flush signals (**`RdFlush`**, **`WrFlush`**). The **`RdIValid_X_n`** bit informs if an instruction *X* is currently in stage *n*. Write (**`Wr`**) interfaces are from ISAX to core, while read (**`Rd`**) interfaces have the opposite direction.

III. SCAL's Architectural Concepts

To ease ISAX generation and reduce the SCAIE-V footprint within the core, we move more interface logic into the potentially reusable SCAL layer. The following subsections present SCAL's capabilities.

A. Multi-ISAX Arbitration

When multiple ISAXes are needed for a certain application, it is desirable to generate each of them independently. This enables faster parallel generation of ISAXes and allows for different suppliers to provide them separately. SCAL then has the responsibility to arbitrate between them. This is done to manage interfaces that represent ISAX results and are inputs to the core (e.g., **`WrPC`**, **`WrMem`**, **`WrRD`**).

Apart from multiplexing this "data plane," SCAL also computes the "control plane" in the form of valid bits. For example, in the case of a **`RdMem`** operation, two mandatory valid bits exist between SCAL and the core: the request valid bit and the *custom address valid* bit. The first one is used to start a memory transaction and is computed using the opcode and the optional valid bit from an ISAX. The custom address valid bit refers to the memory address given by the ISAX. This bit is set to 0 if the ISAX uses the standard "offset+base register" address (allowing reuse of the base core's hardware).

Depending on the core's implementation, the valid bits discussed here are sometimes needed by the core in *earlier* stages than the ISAX provides in its schedule. One such example is a memory transfer in the non-pipelined multi-cycle PicoRV32 core. The user may schedule it in cycle 2, but to transition to the load/store FSM state correctly, the core must know in *advance* whether the instruction requires memory access. For this, SCAL informs the core based on **`RdIValid`**. Correspondingly, we have moved the generation of the **`RdIValid`** bits out of the core and into SCAL.

B. Reconstructing SCAIE-V Signals from Limited Core State

In some cases, SCAL requires signals that are either unavailable in a particular stage of the base core or are invalid when SCAL would typically sample them. A sample scenario is Piccolo, where the core does not pipeline operands (**`RdRS`**) into the last stage; an example of incorrect data is PicoRV32, where the operands exist but can be overwritten by a memory request in the third cycle.

To cover this, SCAL instantiates registers and shifts this data in sync with the main pipeline into the stage in which the ISAX requires it. SCAL also uses this mechanism for instruction decoding if **`RdInstr`** is missing in a specific stage.

C. Abstracting Latency and Decoupled Execution

The supported pipelined cores have different depths. Yet, our goal is that the same ISAX can be integrated without modifications to the logic, independent of its own latency. Thus, an ISAX must be allowed to return a result in earlier or later stages than the underlying core would natively expect it. SCAL abstracts away these discrepancies.

To enable further application cases, we extend the original SCAIE-V tool so that we now support five user-selectable ISAX execution modes (Table I). For example, the *in-pipeline* execution mode can be used for a combinational *popcount* [13] instruction, *stalling multicycle* for a low-area multicycle AES [14], *static decoupled* for a pipelined dot product with a pipeline deeper than the core's, *dynamic decoupled* for an ISAX whose latency depends on an external sensor, and the *always decoupled* mode for a Zero Overhead Loop [15]. Multiple ISAXes can have different execution modes.

To encapsulate this flexibility, we model the timing with:

- EarliestStage (*ES*) - the earliest stage the SCAIE-V node may be used in (introduced in SCAIE-V 1.0)
- LatestStage (*LS*) - the latest stage in which the node may be used; remains undefined for nodes with a decoupled mode (introduced in SCAIE-V 1.0)
- CommitStage (*CS*) - the stage in which the core actually implements the node (new in SCAL / SCAIE-V 2.0)
- SpawnStage (*SS*) - the stage from which SCAL integrates the node as decoupled. (new in SCAL / SCAIE-V 2.0)

TABLE I
EXECUTION MODES FOR STATE UPDATES. HH: DYNAMIC HAZARD HANDLING, ST: STATIC TIMING, ISAX RUN-TIME DEFINED AT COMPILE TIME.

Execution mode	Behavior	Tool Attributes	HH	ST	Example
in-pipeline	ISAX latency matches pipeline's depth; result is returned in-pipeline	-	✓	✓	Fig. 3a)
stalling multicycle	ISAX latency $>$ pipeline's depth; result is returned in-pipeline; core is stalled	-	✓	✓	Fig. 3c)
static decoupled	ISAX executes decoupled, in parallel to the main pipeline	"is decoupled"	✓	✓	Fig. 3e)
dynamic decoupled	ISAX executes decoupled, in parallel to the main pipeline	"is dynamic decoupled"	✓	X	Fig. 3d)
always decoupled	similar to an independent hardware block; may update the core's state at any point in time	instruction's opcode = don't cares	X	X	Fig. 3b)

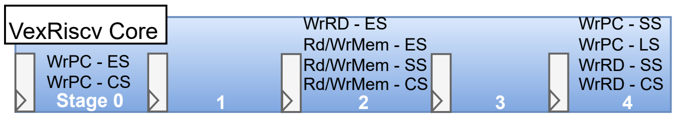

Fig. 2. Pipeline requirements in case of VexRiscv for SCAIE-V nodes.

Our tool receives as input from the user the stage number in which a node must be scheduled (UserStage) and the optional attributes. It then inserts logic into the core in the stage:

```
stage = UserStage
if(opcode == "-") // "-" = don't care
    stage = CommitStage // always block
else if(stage > SpawnStage)
  if("is decoupled" ||
     "is dynamic decoupled")
    stage = CommitStage
  else // default is with stalling
    stage = EarliestStage
```

Listing 1: Updated stage computation for the SCAIE-V node.

To better clarify these concepts, we illustrate them through an example based on the 5-stage VexRiscv core. Fig. 2 presents the core's pipeline requirements, meaning the *ES*, *LS*, *CS*, and *SS* stages, for four of the SCAIE-V nodes: **WrPC**, **WrRD**, **RdMem**, and **WrMem**. The ISAX may request these interfaces in different cycles (*US*). Based on the *US* and the specified tool attributes, SCAIE-V instantiates a specific execution mode and adds hardware logic in a particular stage. Some examples are shown in Table II, where the first row represents the user requirements, and the following rows show the pipeline stage in which SCAIE-V adds logic for the ISAX.

As can be seen in Table II, PC updates after the writeback stage are not supported. In practice, such a late branch can lead to a performance penalty due to expensive flushes. Also, it would require stalling to prevent later instructions from already committing state updates before potentially flushing (writeback or memory). Although we do not allow such *PC* updates, application scenarios such as Zero Overhead Loops [15] are still supported through the *always decoupled* mode.

In the following, we show the generated hardware for the different execution modes. In Fig. 3a), the ISAX reads an operand and returns a result. We have in-pipeline execution,

TABLE II
STAGE MAPPING OF OUR TOOL FOR THE 5-STAGE VEXRISCV USING THE TOOL ATTRIBUTES AND THE GIVEN USERSTAGE (US).

US / Attributes	WrRD		WrMem		WrPC	
	7	3	7	3	7	3
-	2 (Fig.3c)	3 (Fig.3a)	2	2	x	3
is decoupled	4 (Fig.3e)	3 (Fig.3a)	2	2	x	3
is dynamic decoupled	4 (Fig.3d)	3 (Fig.3a)	2	2	x	3
opcode = don't care	4 (Fig.3b)	4 (Fig.3b)	2	2	0	0

where the hardware overhead is given by decoding (*Dec** box), multiplexing, and hazard handling logic.

Fig. 3b) illustrates the scenario of a functional unit that may update the core's state at any time, denoted here as *always decoupled*. An application scenario would be a malfunction in the system, signaled by an external trigger, which should overwrite and clear the current result. It is mandatory that besides the result (**WrRD**), the functional unit also provides a valid bit (**WrRD_valid**). Without it, SCAL can not know when the update is required, as *always* blocks are not bound to an opcode. The hardware overhead for this execution mode is minimal, given only by the multiplexing.

Case c) is similar to a), except the ISAX latency is *longer* than the pipeline. Without additional tool attributes, this is, by default, implemented as a *stalling multicycle* ISAX. A counter keeps track of the latency so that SCAL reads a valid result from the ISAX in the correct cycle.

SCAL generates the hardware in e) if the user wants to avoid stalling the core and requests the *static decoupled* execution mode through tool attributes. Here, the ISAX is executed in *parallel* to the core. A scoreboard implements the hazard handling mechanism (*DH* box in Fig. 3e). When the result is available, the fire logic (*Fire* box in Fig. 3e) checks that the core may be stalled based on the **ISAX_spawnAllowed** signal and stalls the ongoing instruction to write the result. Because, for this execution mode, SCAL knows the ISAX latency at compile time, it generates the "valid result" bit **WrRD_valid** internally through a shift register. This bit is optionally AND-gated with the ISAX-provided **WrRD_valid** bit. The register number for the result is given in the instruction word (bits 11:7) and is stored in a FIFO until the ISAX is finished. A

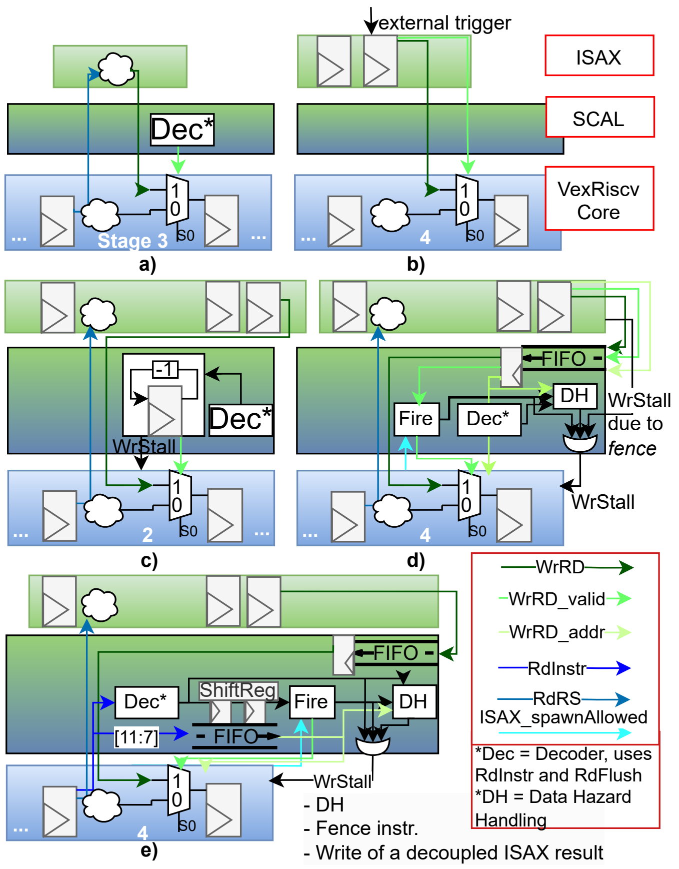

Fig. 3. Integration of an ISAX reading operands and returning a result with different execution modes: in-pipeline (a), as an always block (b), stalling multicycle (c), or decoupled with dynamic (d) and static (e) run-time.

FIFO is adequate since the static latency guarantees ordered results. In contrast, for a dynamic (variable) latency, as in Fig. 3d), the ISAX must provide the result register number and the valid bit. For the decoupled modes, the user can cancel such ongoing ISAXes through the *kill* ISAX and stall the core until they are finished through the *fence* ISAX, both added by SCAIE-V. SCAL already implements these for static-latency ISAXes. However, for the *dynamic decoupled* mode, the ISAX must implement them according to its execution status. For *kill*, it has to flush its internal registers; for *fence*, the ISAX has to stall the core (**WrStall**) while executing.

It is important to note that the ISAXes in scenarios 3 a), c), and e) have identical interfaces despite their different execution modes. This was an essential goal to achieve, providing portability independent of the ISAX latency and the underlying core. Only for the special cases of *always-decoupled* and *dynamic-decoupled* does the ISAX interface change to allow the ISAX itself to initiate state updates.

D. Abstracting Operation Semantics

To ease ISAX generation, it is desirable to abstract the semantics of the SCAIE-V operations. Whether a value is written to the program counter or the register file should not make a difference for an ISAX.

One step in reaching this is abstracting PC updates. This is now implemented within SCAL, which sets the **WrFlush** signal on its own in case of a valid PC update. Another issue arises when a **WrPC** occurs after a state update. Although this is not possible in the base core, it may occur after ISAX integration because SCAIE-V allows later **WrPC** operations. If a memory transfer occurs in a stage before the ISAX **WrPC**, SCAL stalls the core until the **WrPC** is resolved.

Another issue is posed by data hazards, which were already considered in the original SCAIE-V tool. This work further extends SCAIE-V, and SCAL now generates two helping signals (**valid** and **valid_data**) to ease hazard handling within the core. The **valid** bit signals that this is indeed a valid ISAX instruction that wants to commit a result. The **valid_data** bit communicates that the payload is also valid and was provided by the user. In case of a data hazard, the following cases can occur in a pipeline stage:

- **valid** = 0, **valid_data** = -, no DH with ISAX
- **valid** = 1, **valid_data** = 0, pipeline must be stalled
- **valid** = 1, **valid_data** = 1, data forwarding possible

While these two signals are generated by SCAL and sent to the processor, data forwarding must be implemented in the core's datapath. These helper signals are used in the Piccolo and ORCA implementations. No hazards can occur in the multicycle PicoRV32, while VexRiscv has a user-friendly system that automatically handles hazards for new plugins.

E. Abstracting ISAX-Internal State

Some ISAXes require internal state, e.g., the accumulator in a multiply-and-accumulate ISAX. In former versions of SCAIE-V, this state was instantiated and managed (including hazard handling) by the ISAX itself. With SCAL, all state handling, whether at the architectural or ISAX-internal scopes, is consistently provided by the adaptation layer instead. **RdInternalState** and **WrInternalState** operations are supported for the new user-defined **InternalState**, and all SCAL execution modes are allowed.

F. Control Synchronisation

ISAXes and the core need to be synchronized with regard to control events such as stalls and flushes, and thus, they can communicate bidirectionally on these using SCAIE-V operations. **WrStall** and **WrFlush** can be requested by the ISAX to stall or flush the base core. SCAL distributes these signals to the appropriate stages in the core (Fig. 4). Analogously, the core can stall the ISAX by sending it a **RdStall** operation. We have modified the SCAIE-V stalling concept for SCAL to support multicycle ISAXes, which start and commit in the same pipeline stage. For example, VexRiscv reads operands and starts memory accesses in stage 2, so a *stalling multicycle* ISAX with **RdRS** and **WrMem** adds SCAIE-V logic in stage 2. Yet, the ISAX may start only after data hazards are resolved. The former SCAIE-V could not distinguish if **RdStall** is set due to hazards in the core or due to **WrStall**, which would wrongly stall the counter in Fig. 3c). Thus, we separated **RdStall** and **WrStall** within the core (red and green arrows).

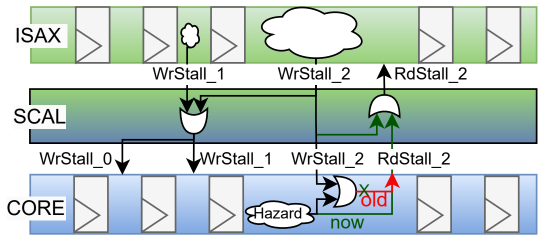

Fig. 4. Stalling mechanism. RdStall is computed similarly in all stages.

TABLE III
SYNTHESIS RESULTS FOR THE UNMODIFIED CORES.

	ORCA	PicoRV32	VexRiscv	Piccolo
FFs	1408	574	857	6184
LUTs	3108	889	1353	14924
Freq.(MHz)	77.262	194.742	95.283	36.528

IV. EVALUATION

Prior work [11] already demonstrated the benefits of SCAIE-V for a selection of different ISAXes. In this evaluation, we focus on the flexibility of SCAL to adapt different implementations of the same ISAX to multiple different base cores. We thus integrate the same ISAX but with different latencies, SCAIE-V destination nodes, and execution modes into the ORCA [16], PicoRV32 [6] (non-pipelined, FSM-based), VexRiscv [17], and Piccolo [18] cores.

A. Evaluation Setup

We provide FPGA and ASIC results. For FPGA, we use the **xc7z020** [19] device, and the resource usage corresponds to the minimal positive slack. The baseline synthesis results are given in Table III and were obtained for the four standalone cores. Consequently, it does not contain the wrapper used when adding the ISAXes, which would guide the tool in optimizing the logic for the core's unused interface signals. Thus, a direct comparison of the core's area is accurate only for the evaluation scenarios that synthesize the cores separately as well (Sections IV-D-IV-E). For these scenarios, the highest overheads in FFs and LUTs are below 1 % for Piccolo, below 4 % for VexRiscv, below 4 % for ORCA, and below 3 % in FFs and 20 % in LUTs for PicoRV32. For PicoRV32, SCAIE-V must keep track of certain interface signals that become invalid over time, such as the instruction word. PicoRV32's particularly small area exacerbates the relative overheads.

For each core in our 22nm ASIC evaluation, we have selected a target frequency for which the area results produced by the proprietary tool flow are stable across all scenarios (VexRiscv: 1.1 GHz, PicoRV32: 1.4 GHz, Orca: 1.0 GHz, Piccolo: 556 MHz).

For the charts with two vertical axes, the right one corresponds to the Piccolo core.

B. Abstracting Latency

Scenario: In this scenario, the ISAX multiplies two operands and writes the result in the register file. We vary the implementation between combinational (0 cycles of latency) up to pipelined with 3 cycles of latency. We used the *is decoupled* tool attribute here without a specific reason, as further attributes are analyzed in Section IV-C.

Result discussion: For a combinational implementation, the result is returned in the earliest core stage in which the operands may be read (ORCA: 2, VexRiscv: 2, PicoRV32: 1, Piccolo: 0). Naturally, the more stages the ISAX requires, the more FFs are used for pipelining (Fig. 5a). Depending on the stage in which the result is returned, the data hazard mechanism of the core must be adapted, having an impact on the LUT number (Fig. 5b). When the ISAX requires 3 cycles, it actually takes place *later* than the writeback stage of the cores. Thus, it is integrated by SCAL as decoupled (as in Fig. 3e) and requires more LUTs.

As expected, the frequency drops significantly when the multiplication ISAX is implemented combinationally (Fig. 5c). Each core computes the common ALU operations in a different stage, so the ISAX may or may not fit the core's datapath well. For example, Piccolo computes operations in the first cycle. Thus, although its frequency is lower for the combinational ISAX, it does not differ significantly when increasing the ISAX pipeline depth. On the other hand, ORCA reads operands in stage 2 and computes ALU operations in the following cycle. Thus, adding the multiplication in cycle 2 dramatically reduces the frequency. Depending on the core's datapath, a one- or two-cycle-ISAX may bring some further frequency improvement (VexRiscv) or not (ORCA).

The ASIC area (Fig. 5d) of the combinational implementation is higher for some cores compared to the pipelined versions. If we correlate this with the frequency drop we observed on FPGA (or ASIC slack values), we can assume that for the combinational implementation, the ASIC tool instantiates larger gates to try to meet the timing requirements.

This scenario highlights well how a single ISAX configuration does not fit all cores equally; being able to port an ISAX to a different core in the search for the best performance is a great advantage. A final quintessential remark to this scenario is that thanks to SCAL, the ISAX interface remains the same across different cores and latencies, even for the decoupled integration. It requires from SCAL two operands and returns one result. Naturally, the pipelined versions also use the stall and flush signals to be in sync with the base core.

Comparison to SCAIE-V: In the original SCAIE-V tool, experimenting with different ISAX multiplication latencies would have been difficult because the result had to be returned in the core's writeback stage. Apart from allowing different latencies, SCAL also takes over some logic that was previously added into the core, such as instruction decoding and discarding a result when the stage is flushed. However, the total area overhead remains the same, as logic has just shifted from the core into SCAL. Moreover, in the original tool, the three-cycle multiplication would have had a different interface than the others. Like with SCAL, it would have been integrated as decoupled and required, apart from the multiplication result, the destination address, and the valid bit. SCAL can now infer

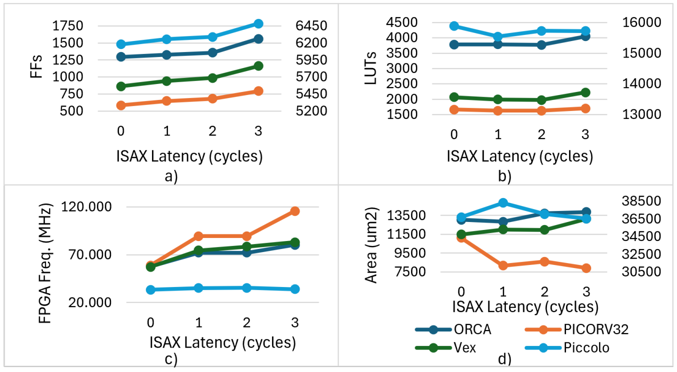

Fig. 5. Synthesis results when varying the ISAX latency.

these signals, as in Fig. 3e). Last but not least, data hazard handling was solely done within the core in the prior SCAIE-V. Now, SCAL provides the core two helper signals to ease hazard handling due to ISAXes (Section III-D).

C. Supporting Different Execution Modes

Scenario: In this scenario, the ISAX still computes the multiplication of two operands, but it uses the different execution modes supported by our tool, as presented in Table I. For the *stalling multicycle*, *static*, and *dynamic decoupled* versions, the ISAX returns the result in 7 cycles. Its critical path is equal to the implementation with 3 cycles from Section IV-B. For the *always* block scenario, we set it to two cycles to match the cores' writeback stage and not overwrite other instructions.

Result discussion: As expected, the *always* mechanism has the lowest hardware overhead (Fig. 6a,b). This is because SCAL does not handle hazards for always-decoupled ISAXes, as the responsibility lies with software scheduling for this execution mode. The resource usage is comparable between static and dynamic decoupled execution. For the static latency ISAX, SCAL buffers the destination address and valid signals and implements the *fence* and *kill* instructions (see Section III-C). In the case of a dynamic run-time, these must be implemented by the ISAX. However, as the functionality is similar, the total area is comparable. The *stalling multicycle* version does not have to support a hazard mechanism; thus, its footprint is smaller.

The lower frequency (Fig. 6c) in the case of the *always* block is expected, as we chose a two-cycle implementation, which has a longer critical path than the seven-cycle one. For PicoRV32, the multicycle logic is integrated into the FSM state reading the operands, which is sensitive to frequency drop (see Section IV-B). The multiplication ISAX should register its inputs and outputs to avoid such lower frequencies. This is not done in SCAL, as it would lead to a cycle penalty even for simpler ISAXes. These observations demonstrate again the importance of a fast evaluation for different ISAX configurations and base cores, which is facilitated through SCAL. We see no difference in frequency between the dynamic and static decoupled execution modes. Still, supporting both of them enables more applications.

TABLE IV
FPGA AND ASIC RESULTS, FOR FIG. 5-8.

		ORCA	PicoRV32	Vex	Piccolo
0 cycles (Fig. 5)	FFs	1296	584	865	6181
	LUTs	3784	1662	2065	15884
	Freq. (MHz)	57.8	58.8	57.2	33.3
	Area (um2)	13052	11125	11491	36725
1 cycle (Fig. 5)	FFs	1328	649	942	6256
	LUTs	3790	1624	1990	15544
	Freq. (MHz)	72.2	89.5	74.6	35.2
	Area (um2)	12835	8169	12007	38336
2 cycles (Fig. 5)	FFs	1358	682	986	6290
	LUTs	3768	1626	1975	15732
	Freq. (MHz)	72.2	89.5	78.6	35.4
	Area (um2)	13725	8601	11964	37062
3 cycles (Fig. 5)	FFs	1562	794	1161	6484
	LUTs	4047	1701	2220	15724
	Freq. (MHz)	80.4	115.7	83.1	34.1
	Area (um2)	13875	7905	13152	36550
Stalling Multicycle (Fig. 6)	FFs	1485	805	1087	6406
	LUTs	3880	1733	2097	15775
	Freq. (MHz)	80.1	115.7	90.5	34.5
	Area (um2)	13533	8412	11433	37361
Decoupled (Fig. 6)	FFs	1600	977	1199	6520
	LUTs	4072	1992	2279	15957
	Freq. (MHz)	80.4	123.0	83.3	34.8
	Area (um2)	14012	9045	12886	40368
Dyn. Dec. (Fig. 6)	FFs	1619	1028	1233	6539
	LUTs	4080	2011	2268	15960
	Freq. (MHz)	80.4	123.0	85.7	34.8
	Area (um2)	13907	9132	12793	36951
Always (Fig. 6)	FFs	1353	670	953	6282
	LUTs	3717	1613	2111	15640
	Freq. (MHz)	72.1	89.5	81.0	34.8
	Area (um2)	13312	8716	11704	37505
WrMem (Fig. 7)	FFs SCAL	0	2	2	1
	LUTs SCAL	4	8	13	7
	FFs Core	1407	583	865	6182
	LUTs Core	3195	1061	1357	15074
	Freq. (MHz)	71.9	89.5	74.7	35.4
	Area (um2)	13045	8806	11975	37680
WrPC (Fig. 7)	FFs SCAL	0	2	0	0
	LUTs SCAL	4	7	4	4
	FFs Core	1409	590	882	6188
	LUTs Core	3190	1048	1336	14931
	Freq. (MHz)	72.2	89.2	83.9	35.9
	Area (um2)	12819	8877	11828	36302
WrInternal (Fig. 7)	FFs SCAL	33	35	33	33
	LUTs SCAL	13	10	13	13
	FFs Core	1409	583	882	6189
	LUTs Core	3145	938	1337	14918
	Freq. (MHz)	72.5	91.0	83.1	36.2
	Area (um2)	13032	4752	9648	36189
WrRD (Fig. 7)	FFs SCAL	2	2	0	2
	LUTs SCAL	12	6	4	12
	FFs Core	1409	583	888	6190
	LUTs Core	3232	1032	1377	14942
	Freq. (MHz)	72.2	89.5	74.6	35.2
	Area (um2)	12835	8169	12007	38336
1 ISAX (Fig. 8)	FFs SCAL	2	2	0	2
	LUTs SCAL	12	6	4	12
	FFs Core	1409	583	888	6190
	LUTs Core	3232	1032	1377	14942
	Freq. (MHz)	72.2	89.5	74.6	35.2
8 ISAXes (Fig. 8)	FFs SCAL	2	16	0	2
	LUTs SCAL	118	170	112	118
	FFs Core	1409	583	888	6190
	LUTs Core	3211	1033	1373	15047
	Freq. (MHz)	71.1	89.5	75.7	35.3

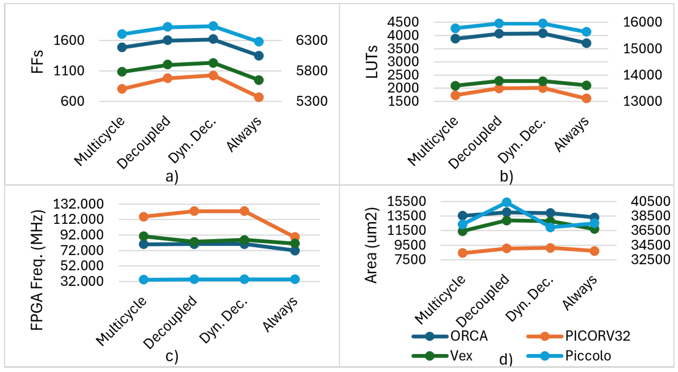
Fig. 6. Synthesis results when using different execution modes.

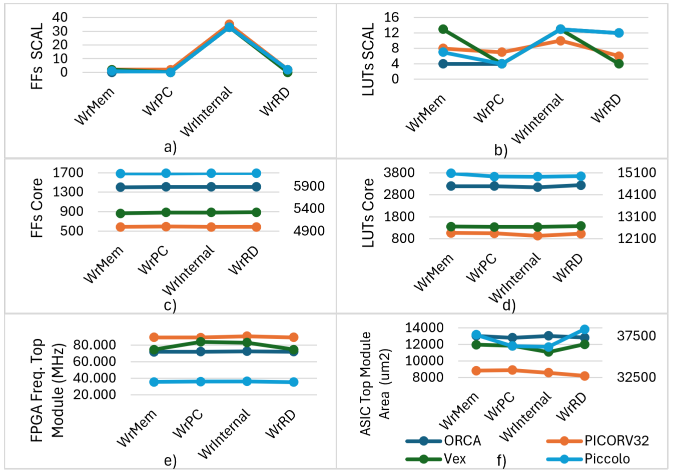
Fig. 7. Analysis when writing the ISAX result to different state elements.

The observations regarding FPGA utilization also apply to the ASIC area (Fig. 6d). For Piccolo, we observed a correlation between the area and the slack, as the ASIC tools usually instantiate larger gates to reach the timing requirements.

Comparison to SCAIE-V: The prior SCAIE-V implementation supported the following execution modes (Table I): *in-pipeline*, *stalling multicycle*, and *dynamic decoupled*. We now add the decoupled execution mode with static run-time to allow the same interface for ISAXes running decoupled or in-pipeline, as explained in the previous section. To reach this, SCAL constructs a shift register for the valid bit and a FIFO for the destination register number (Fig. 3e). We note in Fig. 6 that these do not add much area overhead nor impact the frequency. We also add the *always decoupled* execution mode, which allows for persistent interaction and core state overrides. Moreover, we updated the *stalling multicycle* mechanism, but we discuss this in more detail in an application scenario in Section IV-D. In the former SCAIE-V [11], the logic for committing results (the *Fire* box in Fig. 3d-e) and handling hazards for decoupled modes required around 100 LUTs and 30 FFs. We reuse this mechanism as part of SCAL.

D. Abstracting Operation Semantics

Scenario: In this scenario, we use the one-cycle ISAX from IV-B, but we write its result to different destinations: an internal register file instantiated within SCAL (WrInternal), program counter (WrPC), memory (WrMem), and register file (WrRD). The ISAX datapath is identical in all four cases. The operations' semantics are hidden within SCAL and the core. Thus, for the FPGA area analysis, we ran the synthesis for SCAL and the core separately (Fig. 7a-d). The synthesis for the core was done solely for the core's top module, without the top wrapper. Thus, it does not contain top-level optimizations.

Result discussion: It can be observed that when changing the state element for the result, the modifications within the core are small (Fig. 7c-d). Part of the SCAIE-V logic is now shifted into SCAL to reduce the interface complexity within the core (Fig. 7a-b). Naturally, the highest FF usage is when instantiating a new internal state element to store this result. For PicoRV32, SCAL also uses FFs to store the `RdIValid` decoding bits, as the core invalidates the instruction word register while prefetching. Two FFs are needed for ORCA and Piccolo to handle register file data hazards (see Section III-D). More variation is seen in the LUT usage (Fig. 7b). For `WrPC`, SCAL must generate the flushing mechanism, and for `WrInternal`, it implements a data hazard system. In the case of ORCA and Piccolo, it generates the two control signals for data hazards due to `WrRD`, which can be seen in their corresponding LUT usage. For VexRiscv, we observe higher resource usage when writing the result to the memory. This is because SCAL has to implement it as a *stalling multicycle* instruction since the ISAX requests a `WrMem` in stage 3, while VexRiscv starts memory transfers in stage 2 (Table II). Yet, these microarchitectural differences are successfully hidden within SCAL.

The frequency (Fig. 7e) is slightly higher when writing the result to the new internal state managed by SCAL (`WrInternal`), leaving the core's main data path untouched.

Comparison to SCAIE-V: The former SCAIE-V tool could not instantiate ISAX state elements. Moreover, its limitations would not allow the integration into the VexRiscv core of the one-cycle multiplication writing to the memory. This core reads the operands and starts a memory transfer in the same cycle in stage two (Fig. 2). Yet, the ISAX from this scenario reads the operands in one cycle and writes the result to the memory in the following one. Hence, it does not match the core's pipeline. By adding the concept of *SpawnStage* from Listing 1, we allow this ISAX to still be integrated within the core as decoupled or through stalling. To allow the latter integration method, we update the *stalling multicycle* mechanism such that the ISAX reads the operands in stage n, runs for the number of cycles according to its latency, and returns the result in the same core's stage n. Previously the result was returned in the following stage.

E. Supporting Multiple ISAXes

Scenario: In this scenario, we use the one-cycle ISAX from IV-B and instantiate it *eight times*. Although it is not a real case

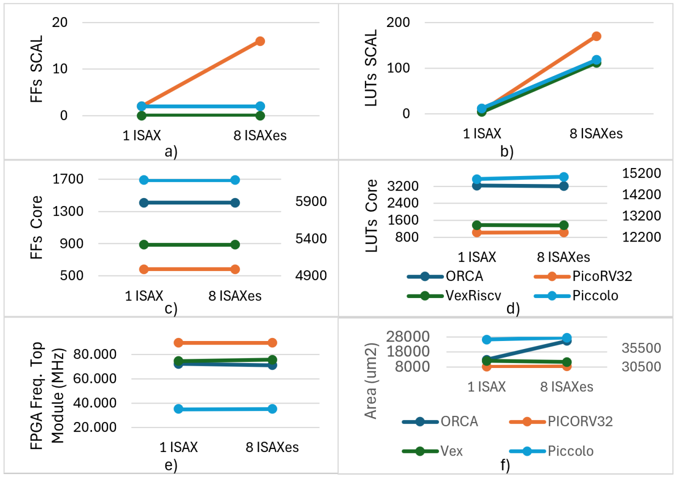

Fig. 8. FPGA results when instantiating multiple ISAXes.

application, it shows the impact of multiple ISAXes. Having the same interface, they can be generated independently.

Result discussion: No handshake or arbitration scheme is needed within ISAXes, as these are provided by SCAL. Moreover, SCAL has only a single interface to the core, independent of the number of ISAXes. This can be observed in the core's resource usage (Fig. 8), which remains unaltered. Again, the overhead and complexity have been successfully moved into SCAL. Furthermore, the PicoRV32 core overwrites some registers depending on the current state (see Section III-B); SCAL takes a safer path and preserves them locally, causing a slightly higher FF usage compared to the other cores. The 2 FFs used by ORCA and Piccolo are for data hazard handling (see Section III-D). Naturally, the LUT number increases for multiple ISAXes to multiplex the result.

Comparison to SCAIE-V: The former SCAIE-V tool updated the core to decode the ISAXes in order to multiplex between their **`WrRD`** and the core's datapath. When using the SCAL layer, the core must only check the incoming **`WrRD_valid`** bit. For the VexRiscv core version used in this paper, the original tool would add 188 LUTs within the core, while our proposed mechanism adds 20 LUTs within the core and 112 LUTs inside SCAL. This reflects that the ISAX decoding for **`WrRD`** arbitration has been shifted to SCAL, reducing the design effort within the core. In Figure 5c) of the SCAIE-V paper [11], the authors show a slight increase in the core's resource utilization when adding all ISAXes together versus integrating them independently. This increase within the core is reduced in our work through SCAL (Fig. 8c, d).

V. Conclusion

ISAXes speed up applications, yet their integration into a base core is typically complex, error-prone, and not portable. We extended an existing ISAX integration tool, SCAIE-V, to provide an adaptation layer (SCAL) that mediates between the ISAX and the core. We demonstrated that SCAL eases ISAX generation by maintaining a stable interface across different ISAX latencies, execution modes, and cores. SCAL also reduces the integration effort within the core, which eases the support of new base processors. We provide our proposed solution as an open-source tool to contribute to the growing RISC-V custom extensions ecosystem [20].

VI. Acknowledgements

We acknowledge the financial support by the German Federal Ministry of Education and Research in the projects "MANNHEIM-FlexKI" (grant: 01IS22086A-L) and "Scale4Edge" (grants: 16ME0122K-140, 16ME0465).

References

[1] "Codasip CodAL." https://codasip.com/products/codasip-studio/codal/.
[2] "OpenHW Group eXtension Interface." https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/index.html, 2022. Online; accessed Nov. 2024.
[3] VectorBlox Computing Inc., "FPGA-Optimized lightweight Vector Extensions for VectorBlox ORCA." https://riscv.org/wp-content/uploads/2016/07/Tue1515_VectorBlox_ORCA_with_LVE.pdf, 2022. Online; accessed Nov. 2024.
[4] Div., "Sapphire custom instruction interface." https://www.efinixinc.com/blog/blog-2023-risc-v-custom-instruction.html, 2024. Online; accessed Nov. 2024.
[5] Div., "Intel Nios V processor custom instruction." https://www.intel.com/content/www/us/en/docs/programmable/773194/current/custom-instruction-types.html, 2024. accessed Nov. 2024.
[6] "PicoRV32 RISC-V CPU." https://github.com/YosysHQ/picorv32.
[7] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip generator," Tech. Rep. UCB/EECS-2016-17.
[8] Div., "Intel Nios II processor custom instruction." https://www.intel.com/content/www/us/en/docs/programmable/683242/current/custom-instruction-types.html, 2024. accessed Nov. 2024.
[9] Div., "PicoRV32 and the PCPI interface." https://github.com/cliffordwolf/picorv32, 2022. Online; accessed Nov. 2024.
[10] B. Green, D. Todd, J. C. Calhoun, and M. C. Smith, "TIGRA: A tightly integrated generic RISC-V accelerator interface," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 779–782, 2021.
[11] M. Damian, J. Oppermann, C. Spang, and A. Koch, "SCAIE-V: An open-source scalable interface for ISA extensions for RISC-V processors," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, (New York, NY, USA), p. 169–174, Association for Computing Machinery, 2022.
[12] J. Oppermann, B. M. Damian-Kosterhon, F. Meisel, T. Mürmann, E. Jentzsch, and A. Koch, "Longnail: High-level synthesis of portable custom instruction set extensions for RISC-V processors from descriptions in the open-source CoreDSL language," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, (New York, NY, USA), p. 591–606, Association for Computing Machinery, 2024.
[13] "RTL implementation of the population count (popcount) module." https://fpgacpu.ca/fpga/Population_Count.html.
[14] "Advanced encryption standard (AES)." https://www.nist.gov/publications/advanced-encryption-standard-aes-0.
[15] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.
[16] "Mirror of the ORCA RISC-V CPU." https://github.com/cahz/orca.
[17] "VexRiscv RISC-V CPU." https://github.com/SpinalHDL/VexRiscv.
[18] "Piccolo RISC-V CPU." https://github.com/bluespec/Piccolo.
[19] Div., "AMD Zynq 7000 SoCs." https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-7000.html. Accessed Nov. 2024.
[20] "SCAIE-V 2.0 repository." https://github.com/esa-tu-darmstadt/SCAIE-V-2.0/.