OpenASIP 2.0: Co-Design Toolset for RISC-V Application-Specific Instruction-Set Processors

Kari Hepola, Joonas Multanen, Pekka Jääskeläinen
Faculty of Information Technology and Communication Sciences (ITC)
Tampere University
Tampere, Finland
{kari.hepola, joonas.multanen, pekka.jaaskelainen}@tuni.fi

***Abstract*—Application-specific instruction-set processors (ASIPs) are interesting for improving performance or energy-efficiency for a set of applications of interest while supporting flexibility via compiler-supported programmability. In the past years, the open source hardware community has become extremely active, mainly fueled by the massive popularity of the open-standard RISC-V instruction set architecture. However, the community still lacks an open source ASIP co-design tool that supports rapid customization of RISC-V-based processors with an automatically retargetable programming toolchain.**

To this end, we introduce OpenASIP 2.0: A co-design toolset that is built on top of our earlier ASIP customization toolset work by extending it to support customization of RISC-V-based processors. It enables RTL generation as well as high-level language programming of RISC-V processors with custom instructions. In this paper, in addition to describing the toolset's key technical internals, we demonstrate it with customization cases for AES, CRC and SHA applications. With the example custom instructions easily integrated using the toolset, the run time was reduced by 44% on average compared to the standard RISC-V ISA. The speedups were achieved with a negligible datapath area overhead of 1.5%, and a 1.4% reduction in the maximum clock frequency.

***Index Terms*—ASIP, RISC-V, co-design tools, compilers**

I. Introduction

Application-specific instruction-set processors (ASIPs) offer a middle ground in terms of flexibility and performance between general purpose processors and fixed function accelerators, while allowing instruction set customization for an application domain. This enables to tailor the architecture to achieve better performance or energy-efficiency in targeted applications, while maintaining programmability. [1]

However, programming of ASIPs requires special attention as the compiler toolchain must be aware of the custom instructions so that they can be used when compiling programs. Moreover, the manual customization of processor implementations and their details, such as *function units* (FUs), is laborious and error-prone work. To overcome these barriers, co-design toolsets offer support for programming and hardware generation of ASIPs. Even though the RISC-V ISA has gained traction both in academia and industry, the open source community still lacks an open source implementation of a co-design toolset targeted for RISC-V-based ASIPs.

Processor co-design tools based on *processor description languages* (PDLs) simplify processor customization, while also enabling the retargetability of the compiler toolchain. However, PDLs tend not to differentiate between instructions and operations, preventing the description of operation semantics separately from the sources and targets of operands. In addition, reuse of already implemented operations is not typically supported when customizing instruction sets with PDLs. [2]

To increase processor hardware-software co-design productivity while keeping performance and area overheads minimal, we introduce the first open source, retargetable toolset for generating and programming RISC-V-based ASIPs supporting custom instructions.[1] Furthermore, we describe an automatic generation of function unit RTL from *directed acyclic graph* (DAG) based operation representations that allow reuse of operation descriptions, and integrate it with generation of RISC-V custom instructions.

We demonstrate the toolset by designing and integrating an *instruction set extension* (ISE) for AES, CRC and SHA applications, which reduced the run time by 44% on average with a small area overhead of 1.5% when synthesized with a 28 nm technology.

II. Related Work

The rising popularity of the RISC-V ISA has created motivation for RISC-V generators with retargetable compiler toolchains. Multiple proprietary ASIP co-design toolsets such as Synopsys ASIP Designer [3], Codasip Studio [4] and Andes [5] ship with support for custom instructions as well as modification for the microarchitectural implementation of RISC-V-based processors. However, the tools are not open source, lacking the freedom of the hardware community to extend and adopt them for their academic, hobbyist and commercial purposes.

The open source community maintains toolsets for the customization of RISC-V implementations such as the Chisel-based Rocket Chip Generator [6] and the TL-Verilog-based WARP-V project [7]. Even though the open source tools have extensive support for configuring the microarchitecture, such as pipelining and branch prediction, they lack support for an automatically retargetable programming toolchain and the

[1]The code will be published together with this paper.

automatic hardware generation of custom instructions, which hinders the design space exploration of RISC-V-based ASIPs.

With Codasip Studio and ASIP Designer, users describe the processor architecture and implementation with the codAL and nML PDLs that are used to reconfigure the compiler toolchain and generate the processor RTL. While this allows to describe the processor architecture on a high level, description of instructions is tightly coupled with the architecture definition, which prevents reuse of already described operations. [2][3][8]

Kultala et al. [9] describe a DAG-based operation set description incorporated with OpenASIP co-design toolset [10]. The DAG-based operation set description utilizes operation set libraries and reusing of already implemented operations as part of the operation description. This simplifies the description of operations and allows reuse between architectures. However, in its current state, the DAG-based descriptions are only used for automatic instruction selection, and cannot therefore be used to automatically generate hardware implementations for operations. We extended the toolset by adding automatic hardware generation from DAG-based operation descriptions to minimize the design effort when customizing ASIPs.

III. Processor Customization in OpenASIP

In this section, we overview OpenASIP, formerly known as *TTA-based Co-Design Environment* (TCE). OpenASIP is an open source co-design toolset that has compilation, simulation and hardware generation support for ASIPs based on *transport triggered architecture* (TTA). TTA is a highly modular statically scheduled exposed datapath [11] architecture. Unlike with *operation triggered* architectures [12], TTA programming interface is based on explicit moves that result in execution of operations as a side-effect. The low level programming interface and modular structure enables TTA template to be used for describing other more dynamic processor architectures.

We chose OpenASIP as a base for our customization flow due to the modular structure and the massive amount of previous functionality done over the past two decades that could be reused and extended for RISC-V ASIP customization.

A. *Customization Flow*

Fig. 1 presents the processor customization flow in OpenASIP. The toolset ships with a retargetable compiler and hardware generator that adapt to a high-level architecture description. The hardware descriptions of processors generated by the toolset can be simulated in RTL simulation and synthesized with third party vendor tools.

Architecture definition file (ADF) is an XML-based file that describes the architectural information that is needed to program the processor. In addition to architectural information, the architecture definition file describes parts of the microarchitectural details of RISC-V implementations, such as FU configurations and operation latencies that are visible in the programming interface of TTA-based processors. The function units described in the architecture definition include operations that are described in separate operation set databases. The operation set databases only describe the semantics of the

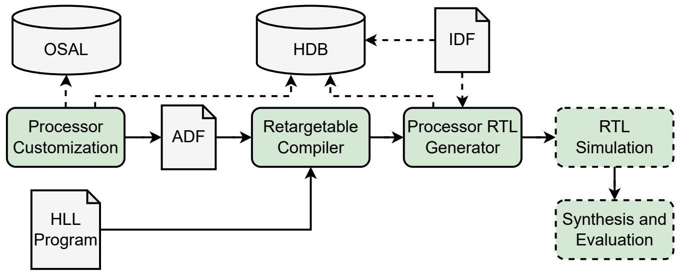


Fig. 1: Processor customization in OpenASIP with a retargetable compiler and automatic hardware generation.

operations, while the operation latency is defined in the architecture definition's FU description.

In the OpenASIP flow, the architecture definition does not describe instruction encodings. Instead, they are automatically generated to a *binary encoding map* (BEM) from the architecture definition. Binary encoding map is used in the generation of decode logic and program binaries.

When the processor RTL is generated, the user can give the processor generator tool an *implementation definition file* (IDF) that links the function unit descriptions in the architecture definition to hardware implementations of the matching FUs in the *hardware databases* (HDBs). If an implementation definition file is not passed to the processor generator tool, the function unit generator proposed in this work can automatically generate the hardware from the operation set entries if the operation behaviour is described as a sequence of already implemented operations in a separate *operation set abstraction layer* (OSAL), or if the operation implementations are described with an HDL in the hardware database.

B. *Operation Set Customization*

The operation set abstraction layer consists of operation entries that describe all the operations that can be used by the toolset. The abstraction layer is divided further into subcomponents, where the XML-based operation property file describes all the information that is needed to use the operation in the compiler flow. Furthermore, the semantics of the operation can be described as a DAG if the operation is implemented as a sequence of already implemented operations. An example of such an operation is shown in Fig. 2. A sigma function used to accelerate SHA encryption is represented by using basic operations and immediate values, with one input IO(1) and one output IO(3). One unused input operand, IO(2), is used to pad the operation to the RISC-V R-format.

IV. Implementation

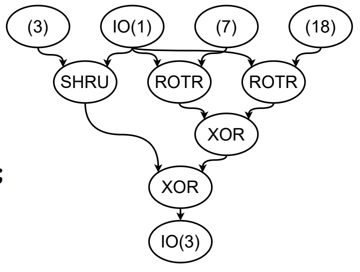
This section describes the extensions made to the OpenASIP toolset, which enables rapid hardware generation and retargetable compiler support for RISC-V processors with custom instructions.

A. *Function Unit Generator*

During processor generation, implementation definition files link hardware database HDL implementations of FUs to the

```
Var a, b, c, d;
OP(ROTR, IO(1), 7, a);
OP(ROTR, IO(1), 18, b);
OP(SHRU, IO(1), 3, c);
OP(XOR, a, b, d);
OP(XOR, d, c, IO(3));
```

(a)



(b)

Fig. 2: Code description (a) and a DAG (b) of a SHA-256 sigma function with an unused input operand.

descriptions of FUs found in the architecture definition file. While this allows users to heavily modify the hardware implementation, it is impractical for rapid design space exploration as the user must write the RTL implementations of entire FUs by hand when the architecture is modified.

We extend the hardware generation flow of OpenASIP with a function unit generator tool that automatically generates hardware descriptions of FUs from operation descriptions, in case the implementation definition file does not define all or any of the FU implementations. In addition, the function unit generator automatically generates operation pipelines based on the operation latency described in the architecture definition.

To incorporate custom operations to processors, two options were implemented into the function unit generator. First, the user can describe the operation as an HDL *snippet*. Instead of an FU implementation, operation snippets describe only the semantics of a single operation with VHDL or Verilog. Creation of the FU modules, as well as connecting the snippet operands to FU inputs and outputs is handled by the function unit generator, as long as pre-defined signal names for operands are used. The snippet is added as a hardware database entry, after which it can be used to automatically generate the hardware description when the operation is included in the architecture definition. The snippet option, of course, requires the user to be knowledgeable in writing HDL as demonstrated in Fig. 3 that represents a byte reflection snippet with an unused input operand, op2, to pad the operation to the RISC-V R-format.

```
for bits in 0 to 7 loop
  op3(bits) <= op1(7-bits);
end loop;
```

Fig. 3: A VHDL snippet for a byte reflection operation.

The second option is to describe the custom operation as a DAG consisting of already implemented operations. If the operation can be constructed using existing operations, the user does not need knowledge of HDL coding. The operations included in the toolset have snippet entries in the hardware database, which allows users to automatically generate hardware when those operations are used, as well as to use them in description of DAG-based custom operations as demonstrated in Fig. 2.

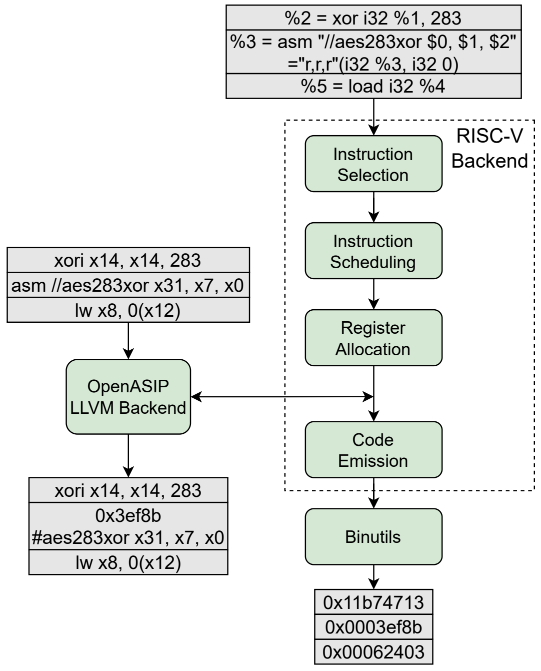


Fig. 4: The compilation flow with a hook to the OpenASIP LLVM backend for processing custom instructions.

B. RISC-V Customization

Our approach for RISC-V customization uses operation set libraries that contain operation descriptions. To incorporate operations between different instruction types, architecture definition and binary encoding map templates were extended for the description of operation triggered instruction formats. Operations are added to architecture definition and binary encoding map instruction formats as entries, which allows to reuse the operation descriptions between different instruction types. This way, operations are described independently of the instruction type.

To utilize the extensive support for customization and generation of TTA-based ASIPs, the core functionality of the hardware generation flow was reused, and the decode and control logic generation extended for RISC-V implementations. As encodings for all instructions are described in a binary encoding map, decode logic for custom instructions is automatically added when the decoder RTL is generated. The current toolset implementation supports generation of the RV32E/I(M) subsets with optional custom instructions, configurable single-issue in-order pipeline with 3-4 stages, tailoring of the bypass network and operation latencies.

C. Runtime Compiler Adaptation

To utilize the upstream LLVM RISC-V backend and bintools without needing to reconfigure and compile them each time the processor architecture is modified, we extended the LLVM RISC-V backend by adding a hook to OpenASIP's LLVM backend. In the compiler frontend, OpenASIP is loaded to the compilation flow as a dynamic library together with

```
#define _OA_RV_AES283XOR(i1, i2, o1) do {
unsigned __oa_op_output_1 = (unsigned)0;
asm ("//aes283xor \%0, \%1, \%2":
  "=r"( __oa_op_output_1):
  "r"((unsigned)(i1)),
  "r"((unsigned)(i2)));
o1 = __oa_op_output_1;
} while(0)
```

Fig. 5: Example of a custom instruction intrinsic that defines the instruction name and operands.

the architecture definition file. As the RISC-V backend and bintools are not modified during run time, they cannot recognize the custom instructions. Therefore, they must be handled by OpenASIP's LLVM backend and replaced with binaries that describe the encoding and operand fields of the custom instructions.

Fig. 4 shows the compiler hook in more detail. The high-level source code is processed and optimized in the compiler front- and middle-end after which the intermediate representation is passed to the compiler backend for target-specific phases of the compilation. OpenASIP hooks into the compilation flow as a preemit pass between the register allocation and code emission phases of the RISC-V backend. At this stage, the backend has allocated registers for the custom instructions which allows to replace them with matching instruction binaries based on the generated binary encoding map. In addition to the binary, the custom instruction is inserted as a comment to increase readability of the generated assembly.

In the high-level source code, custom instructions are invoked with intrinsics. An example of such an intrinsic is presented in Fig. 5. In addition to the operands, the intrinsic describes the name of the instruction. As the RISC-V backend cannot recognize custom instructions, we included the instruction name as a comment to the inline assembly intrinsic that is then processed by the OpenASIP backend when the instruction is expanded before code emission.

An important phase in the processing of custom instructions is the generation of the binary encoding map when the OpenASIP LLVM backend is hooked into from the RISC-V LLVM backend. As discussed in Section III, the binary encoding map is automatically generated from the architecture definition file and it identifies the encoding for each instruction. RISC-V ISA reserves free opcodes that can be used to add custom instructions to the instruction set. When an instruction in the architecture definition is not recognized as an implemented standard RISC-V instruction, a binary encoding is generated for it by utilizing the free opcodes of the RISC-V ISA. When the OpenASIP LLVM backend detects a RISC-V custom instruction, it is able to construct the matching binary by combining the allocated register indexes and the operation code from the generated binary encoding map.

As mentioned previously, in the current compiler implementation, the programmer invokes the custom instructions with the use of automatically generated intrinsics. Automatic pattern matching based custom instruction selection is left for future work. From the programmer's perspective, it is useful if the compiler can independently utilize custom instructions with the use of instruction selection patterns, but typically with more exotic operation chains, the programmer must directly invoke them anyhow due to practical issues with automated instruction selection. An example of this is inter-block instructions, such as saturation arithmetics that require instruction selection across basic blocks [13].

V. Design Case Studies

For the case study examples, we chose the advanced encryption standard (AES) benchmark implemented in CHStone [14], a cyclic redundancy check (CRC) implementation [15] and the secure hash algorithm (SHA) implemented in the Embench benchmark suite. These applications were chosen because they benefit from custom instructions that are cheap to implement in hardware, which makes them interesting candidates for instruction set customization.

We demonstrated the customization flow by generating custom instructions for the applications both with the DAG-based descriptions and HDL snippets, and evaluate the performance and area results against an identical RV32IM implementation without custom instructions which serves as a baseline for the evaluation.

To identify suitable custom instructions, we compiled the benchmarks for the RV32IM subset and ran the compiled programs with an instruction set simulator. We used the profile data generated by the simulator to identify commonly executed sequences of instructions that are interesting for designing a custom instruction set extension.

A. Operation Implementations

The AES algorithm executes a sequence of shifting and a conditional xor of a value frequently. Implementing such a sequence of operations as a custom instruction is particularly interesting as it can be used to reduce the amount of expensive control flow instructions in the program. The semantics of the instruction are simple to describe as an HDL snippet or as a DAG with the use of a select operation.

The CRC-32 algorithm uses byte and word reflection to reverse the bit order of received data and the remainder value. The reflection operation is complex to describe as a DAG as it uses *bit manipulation*. However, it is cheap to implement in hardware with the use of an HDL snippet because it only includes shuffling the bits of an operand as shown in Fig. 3.

The RISC-V cryptography ISE [16] introduces custom instructions for the acceleration of SHA implementations. The instructions combine the rotation and xoring operations required to implement the SHA functionality. In this work, we implemented the SHA-256 sigma and sum instructions as custom instructions. As they combine multiple basic operations, they are ideal to implement as a DAG as seen in Fig. 2.

$$
\begin{array}{ll}
\text{sha256sig0} & : rotr^{7}(in) \oplus rotr^{18}(in) \oplus (in \gg 3) \\
\text{sha256sig1} & : rotr^{17}(in) \oplus rotr^{19}(in) \oplus (in \gg 10) \\
\text{sha256sum0} & : rotr^{2}(in) \oplus rotr^{13}(in) \oplus rotr^{22}(in) \\
\text{sha256sum1} & : rotr^{6}(in) \oplus rotr^{11}(in) \oplus rotr^{25}(in) \\
\text{reflect8} & : \forall i \in [0,7], out[i] = in[7-i] \\
\text{reflect32} & : \forall i \in [0,31], out[i] = in[31-i] \\
\text{aes283xor} & : \begin{cases} (in \ll 1), & (in \gg 7) \neq 1 \\ (in \ll 1) \oplus 283, & (in \gg 7) = 1 \end{cases}
\end{array}
$$

Fig. 6: Included custom operations.

B. Evaluation

All of the custom instructions presented in Fig. 6 were added to an architecture definition that was configured with three pipeline stages. The generated RTL was evaluated by synthesizing the cores with Synopsys Design Compiler and a 28 nm process technology without hardware support for division. Table I presents the areas and maximum clock frequencies of the cores. The addition of custom instructions only increased the area utilization by 1.5% and decreased the maximum clock frequency by 1.4%, thanks to the simplicity of the operation semantics. The operations consist of constant shifts and xor operations that can be implemented in hardware at a low cost.

TABLE I: Post-synthesis properties of the cores.

	RV32IM	RV32IM + ISE
Area (µm^2)	21337	21666
Fmax (GHz)	1.43	1.41

The applications were compiled with the toolchain with and without the custom ISE, and were run in ModelSim to acquire clock cycle information. The cycle counts acquired from RTL simulation are combined with the matching maximum clock frequencies to calculate the run time of the applications in Fig. 7. As seen in the figure, the custom instructions reduce the run time significantly, approximately 44% on average. The SHA benchmark benefits the most from the use of custom instructions as the sum and sigma functions appear in the program frequently, which reduced the run time by 49%.

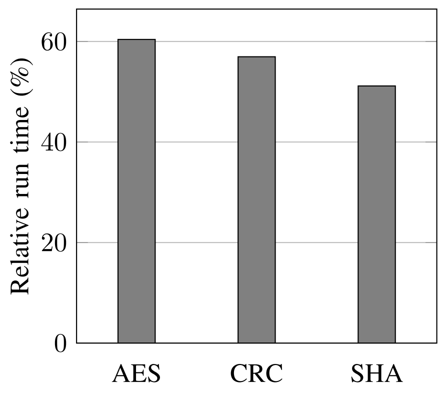


Fig. 7: Run time relative to the RV32IM baseline.

VI. Conclusions and Future Work

In this paper, we presented a hardware-software co-design toolset for programming and customizing of RISC-V-based ASIPs. It includes a retargetable compiler toolchain and support for automatic hardware generation from a high-level architecture definition. We demonstrated the toolset by tailoring an ISE for the AES, CRC and SHA applications. The generated ISE reduced run time by 44% on average with a small 1.5% overhead in the core area. The demonstration showed that the instruction set customization can be done with little design effort with the proposed toolset while vastly improving the application performance with minimal hardware overhead.

In the future we plan to investigate adding automatic custom instruction selection from the DAG-based operation descriptions for RISC-V code generation.

Acknowledgments

This work was supported by European Union's Horizon 2020 research and innovation programme under Grant Agreement No 871738 (CPSoSaware) and Academy of Finland (decision #331344). We would also like to thank Lasse Lehtonen, Kati Tervo and Topi Leppänen for participating in the development of the FuGen feature of the toolset.

References

[1] K. Keutzer, S. Malik, and A. Newton, "From ASIC to ASIP: the next design discontinuity," in *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002, pp. 84–90.
[2] P. Mishra and N. Dutt, *Processor description languages*. Elsevier, 2011.
[3] Synopsys. ASIP Designer Application-Specific Processor Design Made Easy. https://www.synopsys.com/dw/doc.php/ds/cc/asip-brochure.pdf
[4] Codasip Studio. https://codasip.com/products/codasip-studio
[5] Andes co-design tools. https://www.andestech.com/
[6] K. Asanović *et al.*, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep., 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html
[7] WARP-V. https://github.com/stevehoover/warp-v
[8] Codasip. What is CodAL? https://codasip.com/2021/02/26/what-is-codal
[9] H. Kultala, P. Jääskeläinen, and J. Takala, "Operation set customization in retargetable compilers," in *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*. IEEE, 2011, pp. 761–765.
[10] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg, *HW/SW Co-design Toolset for Customization of Exposed Datapath Processors*. Springer International Publishing, 2017, pp. 147–164.
[11] P. Jääskeläinen, H. Kultala, T. Viitanen, and J. Takala, "Code density and energy efficiency of exposed datapath architectures," *Journal of Signal Processing Systems*, vol. 80, pp. 49–64, 2015.
[12] J. Hoogerbrugge and H. Corporaal, "Transport-triggering vs. operation-triggering," in *International Conference on Compiler Construction*. Springer, 1994, pp. 435–449.
[13] G. H. Blindell, *Instruction Selection: Principles, Methods, and Applications*. Springer, 2016.
[14] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *journal of information processing*, vol. 17, pp. 242–254, 2009.
[15] Free Source Code: CRC Implementation in C. https://barrgroup.com/downloads/code-crc-c
[16] RISC-V scalar cryptographic extension v1.0.0. https://github.com/riscv/riscv-crypto/releases/tag/v1.0.0-scalar