# CREATING DOMAIN-SPECIFIC PROCESSORS USING CUSTOM RISC-V ISA INSTRUCTIONS

## 1 Introduction

When System-on-Chip (SoC) developers include processors in their designs, they face choices in solving their computational challenges. Complex SoCs will usually have a variety of processor cores responsible for varied functions such as running the main application programs, communications, signal processing, security, and managing storage. Traditionally, such cores have been in distinct categories such as MCUs, DSPs, GPUs and application processors. Additionally, some unique architectures and instruction sets were developed for very specialized applications. However, a downside of unique instruction sets is the lack of a software ecosystem.

Today, the distinctions between classic core categories are blurring. This is because if a core is designed the right way, more than one usage can be covered by that processor. Furthermore, by creating a processor that is tuned to the needs of the SoC, the silicon efficiency in terms of area and power can be improved.

A recent catalyst for creating domain-specific processors has been the RISC-V ISA (Instruction Set Architecture). Since the usage of the ISA is open and royalty-free, it is an attractive basis on which to implement a processor design. Furthermore, the existence of a base instruction set for each word length means that software using the base instruction set can be ported to all RISC-V processors with that word length.

The RISC-V ISA is designed in a modular way, meaning that the ISA has several groups of instructions (ISA extensions) that can be enabled or disabled as needed. This allows implementing precisely the instruction groups that the domain needs, without having to pay for area or power that will not be used.

One of the groups is special; it has no standard predefined instructions. Designers can add any instruction they need for the application that they want to accelerate. This is

a powerful feature, as it does not break any software compatibility and leaves space for invention and differentiation at the same time. This whitepaper describes how to add domain-specific instructions (custom ISA extensions) and how to build all the needed tools in SDK, as well as implementing the custom ISA extension in HDL (e.g. Verilog). The end result is an optimized domain-specific processor.

# 2 RISC-V Instruction Set Architecture

The RISC-V ISA is organized into groups of instructions (the base ISA & extensions). You can mix and match them as you want. For instance, you may have a RISC-V processor that implements the bare minimum, or a RISC-V processor that implements all ISA extensions, depending on the design needs.

The following table lists the main ISA extensions that have been ratified by RISC-V Foundation, and ISA extensions that are currently under development.

| ISA Extension | Ratified | Notes |
|---|---|---|
| **I/E** | *Yes* | Instructions for basic Integer operations. This is the only group that is mandatory. **I** requires 32 registers, **E** requires only 16. |
| **M** | *Yes* | Instructions for multiplication and division |
| **C** | *Yes* | Compact instructions that have only 16-bit encoding. This extension is very important for applications requiring a low memory footprint. |
| **F** | *Yes* | Single-precision floating-point instructions |
| **D** | *Yes* | Double-precision floating-point instructions |
| **A** | *Yes* | Atomic memory instructions |
| **B** | *No* | Bit manipulation instructions. The extension contains instructions used for bit manipulations, such as rotations or bit set/clear instructions. |
| **V** | *No* | Vector instructions that can be used for HPC. |
| **P** | *No* | DSP and packed SIMD instructions needed for embedded DSP processors. |

The table will be extended in the future as more ISA extensions are added.

Although the list is already extensive, a situation may arise when there is no suitable ready-made ISA extension that fits the design needs. In this case, the RISC-V specification allows adding a custom ISA extension. This can be the company's "secret sauce" and a key differentiator. Thanks to the nature of the RISC-V ecosystem, custom ISA extensions don't break compliance with the main specification—even with additional instructions, your processor is still fully RISC-V compliant and can run generic software stack taken from the ecosystem.

*Figure 1* shows how a custom ISA extension fits in a software stack. On the lowest level, there is a RISC-V-compliant processor with a custom ISA extension. It runs an OS, either real-time OS (RTOS)or a rich OS. It can be compiled with any compiler compatible with a standard RISC-V processor (no special ISA extensions). On top of the OS, there are three applications. **App1** is a generic application that does not require any acceleration. You can use a publicly available, off-the-shelf compiler (e. g. GCC) to compile it, or even a pre-compiled application can be used; either way the RISC-V processor will be able to run it. **App2** and **App3** are the important ones that need to run as fast as possible. These must be compiled with a compiler specifically aware of the custom ISA extension. The compiler can utilize the new instructions that will accelerate **App2** and **App3**.

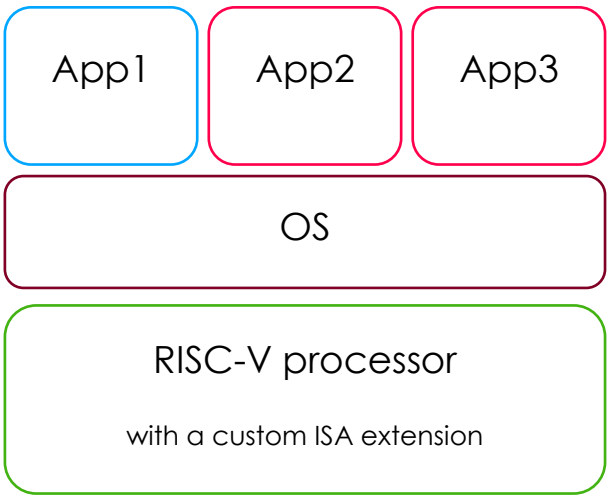



Figure 1

*Figure 2* shows another example of a RISC-V-compliant processor with a custom ISA extension. **App1** does not use the custom ISA extension. **App2** and **App3** use a generic API. The API is implemented by a library that is aware of the custom ISA extension, which again can accelerate **App2** and **App3**. Both **App2** and **App3** can be reused in an off-the-shelf RISC-V processor. All that is needed is a library that implements the required API. In this system, moving **App2** and **App3** from RISC-V *with* a custom ISA extension to RISC-V *without* the extension is easy and does not require any application porting.

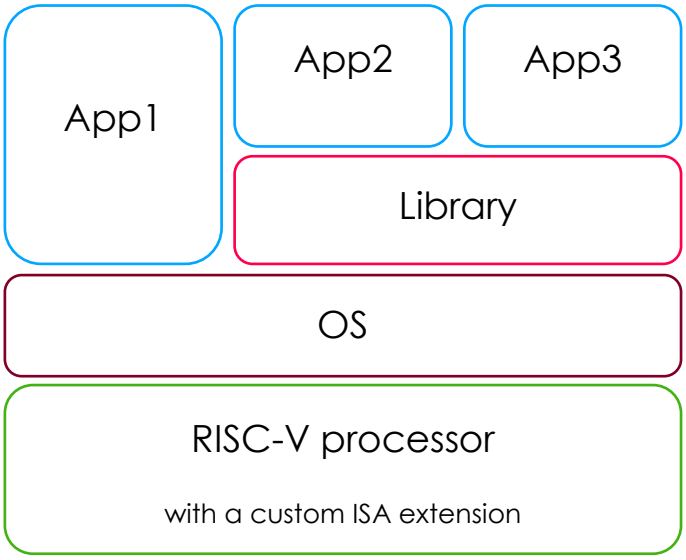



Figure 2

In the following sections, we will cover in more detail the custom ISA extensions and how Codasip helps with their design and verification.

# 3 Custom Instruction Set Extensions

Custom instruction-set architectures, or custom ISA extensions, are not new and have been around for a while. However, they usually require a lot of effort. First, you need to identify the instructions which improve your processor design. Then you need to add them into the C compiler, simulators, debuggers and other tools, and verify that the changes add the same thing to all these different tools. Adding a custom instruction usually requires some manual effort, too. Typically, you need a team who will add the new instructions into the SDK so that the programming tools can pass and compile the instructions. You also need to add new code to the instruction set simulator. Finally, the RTL must be extended, and any changes to the RTL must be verified. Depending on the amount of manual effort, ISA extensions can be rather expensive in terms of time and resources.

To reduce the cost of ISA extensions, it is needed to automate as much work as possible, from the identification of suitable instructions to RTL verification. This is exactly what Codasip does very well. Codasip provides an EDA tool called Studio that enables you to customize the off-the-shelf processors that Codasip also offers. You can start your work with a ready-made RISC-V-compliant processor by Codasip and just add custom ISA extensions according to your needs, or you can write you own RISC-V processor from scratch.

Custom instructions can be simple, such as variants of multiple-and-accumulate instructions, or they can be custom control instructions, such as zero overhead loops

(hardware loops). You can also have special load/store instructions with post- or pre-increments. This illustrates that custom instructions differ by complexity, which influences the capabilities of the C compiler and the performance of the resultant processor. Simple instructions may be used by the C compiler without having to alter the original C code. In other words, you can have one app, and you can compile it for x86 or RISC-V. If the instruction is too complex, the only way to use it is inline assembly or C intrinsic. The limit is around ~25 operations and ~3 outputs. On the other hand, more complex instructions usually improve performance, so the result is worth the effort.

There is an easy way to integrate the inline assembly or intrinsic into a library. The library has a generic implementation as well. The benefit of such a library is that you can have one implementation of the final application, and you can compile it for several targets. Each target may use a different implementation. The application does not need to be aware of the final implementation.

The following example shows a code snippet of such library. It represents a simple byte-swap function. If the specified macro is present, the C compiler has a special instruction that performs the swap. Otherwise, standard approach is used.

```
// universal byteswap() function
inline uint32_t byteswap(const uint32_t word) {
#ifdef ISA_BYTE_SWAP
    // C compiler intrinsic
     return __byteswap(word);
#else
     return ((word >> 24) & 0x000000ff) |
            ((word <<  8) & 0x00ff0000) |
            ((word >>  8) & 0x0000ff00) |
            ((word << 24) & 0xff000000);

#endif
}
```

The code generated for the first part is quite straightforward; just one instruction.

```
byteswap:
  byteswap x10, x10
  c.jr ra // return from function
```

The second part is done in twelve instructions on RV32IMC. X10 holds the value of a word and it holds the return value in the end as well.

```
byteswap:
  srl x15, x10, 8
  lui x14, %hi( 65280 )
  add x14, x14, 65280 & 0xfff
```

```
c.and x15, x14
srl x14, x10, 24
c.or x15, x14
sll x14, x10, 8
lui x13, 16711680>>12 &0xfffff
c.and x14, x13
sll x13, x10, 24
c.or x14, x13
or x10, x14, x15
c.jr ra // return from function.
```

This not only increases performance of the application; it also reduces the code size significantly. You can have a number of similar instructions, including pop-count, various bit manipulation instructions, control instructions, etc.

The next section explains how custom ISA extensions are handled by Studio.

# 4 Codasip Studio

Studio is an EDA tool for processor design used by leading-edge companies to create special processors and used by Codasip to design Bk RISC-V cores. It can generate all the necessary tools in SDK as well as processor's implementation in Verilog, SystemVerilog or VHDL, and UVM-based verification environment. All these outputs are generated from the processor description in CodAL™. CodAL is a mixed architecture-description language based on the C language. CodAL captures not only the ISA itself, but also the processor's resources and other particulars of the processor microarchitecture.

Each processor description consists of two parts: a functional model of the processor, and an implementation model. These two models share common parts, such as opcodes or instruction coding. More importantly, the models enable Studio to generate a UVM-based verification environment as well.

When it comes to ISA extensions, designers usually start with a full-blown RISC-V-compliant processor written in CodAL, delivered by Codasip. All that they need to do is to add a custom ISA extension. Note that we use Studio internally as well—to build our own RISC-V-compliant processors.

The process starts by identifying suitable custom instructions. There are many ways to do this; a convenient one is to use the profiler in Studio. A designer runs the key application on an off-the-shelf processor, and the profiler provides specific sequences of instructions that represent computational hotspots along with a list of functions that take a lot of computation time. This information helps the designer add new instructions.

The first step is to change the functional model of the processor. The designer needs to define the assembly and binary form of the instructions. Then, importantly, semantics of the instruction. The semantics are written in CodAL as well.

In the byteswap example, assuming 32bit RISC-V, the code would look like this:

```
element i_comp_2reg {
  // use of two registers
  use xregs as dst, src;
  // textual form of the instruction
  assembly { "byteswap" dst "," src };
  // encoding of the instruction
  binary { PADDING src  OPC_BYTESWAP[OPC_FRAG1] dst
OPC_BYTESWAP[OPC_FRAG0]};
  // behavior of the instruction
  semantics {
// input from register
    uint32 val;
    codasip_compiler_builtin();
    codasip_preprocessor_define("ISA_BYTE_SWAP");
// read the input from registers
    val = rf_gpr_read(src);
// do the computation and store the result to the register
    rf_gpr_write(dst, val[ 7.. 0] :: val[15.. 8] ::
                      val[23..16] :: val[31..24]);
    };
};
```

The next step is to define the implementation. Let us consider simple straightforward implementation doing the whole instruction in one clock cycle. The only task here is to update the ALU.

```
  ...
  case ALU_BYTESWAP:
    ex_result = alu_op1[ 7.. 0] :: alu_op1[15.. 8] ::
                alu_op1[23..16] :: alu_op1[31..24];
    break;
  ...
```

Once we have a CodAL description, Studio can generate all the tools in SDK, implementation, and UVM-based verification environment.

One of the most important tools in SDK is the C compiler. Codasip uses both LLVM and GCC; typically, the new instruction is generated for the LLVM compiler. In this case, we can see that LLVM recognizes a pattern for byteswap and it uses the **bswap** function in its intermediate representation in the instruction semantics. The generated representation would be as follows:

```
def i_comp_2reg__regs__regs__:
  CodasipMicroClass_<(outs regs:$op0), (ins regs:$op1)>
```

```
{
  let AsmString = "byteswap $op0, $op1";
  let Pattern = [(set regs:$op0, (i32
      (bswap (i32 CheckFI_i32_regs:$op1))))];
  let Size = 4;
  let isReMaterializable = 1;
  let mayLoad = 0;
  let mayStore = 0;
  let AddedComplexity = 1;
}
```

All other tools are aware of the new instruction, so assembler and disassembler, debugger, and profiler can recognize the byteswap instruction as well.

Regarding implementation in HDL, Studio supports three major HDL languages: Verilog, SystemVerilog, and VHDL. The generated Verilog code for the byteswap example would be as follows:

```
always @(*) begin
    case ( alu_opcode )
...
// <file>.codal:<line>:<column>
     4'h9: mux_ex_result_D = {alu_op1_Q[ 7: 0],alu_op1_Q[15: 8],
                              alu_op1_Q[23:16],alu_op1_Q[31:24]};
    ...
    endcase
end
```

As shown, Studio produces all necessary output based on the CodAL code. Generating both SDK and RTL is fully automated.

The last step is verification. Studio includes a functional view and an implementation view. The functional view is used for a reference model, and the implementation view is used as a DUT, meaning that the generated implementation is checked against the reference model. Studio generates UVM-based environment which requires some stimuli. Studio comes with an extensive set of predefined tests and it also supports generation of random streams of instructions, including custom ones. Designers can add their own direct tests as well. Three sources (predefined tests, generated tests, and direct tests) ensure full coverage, including atypical corner cases.

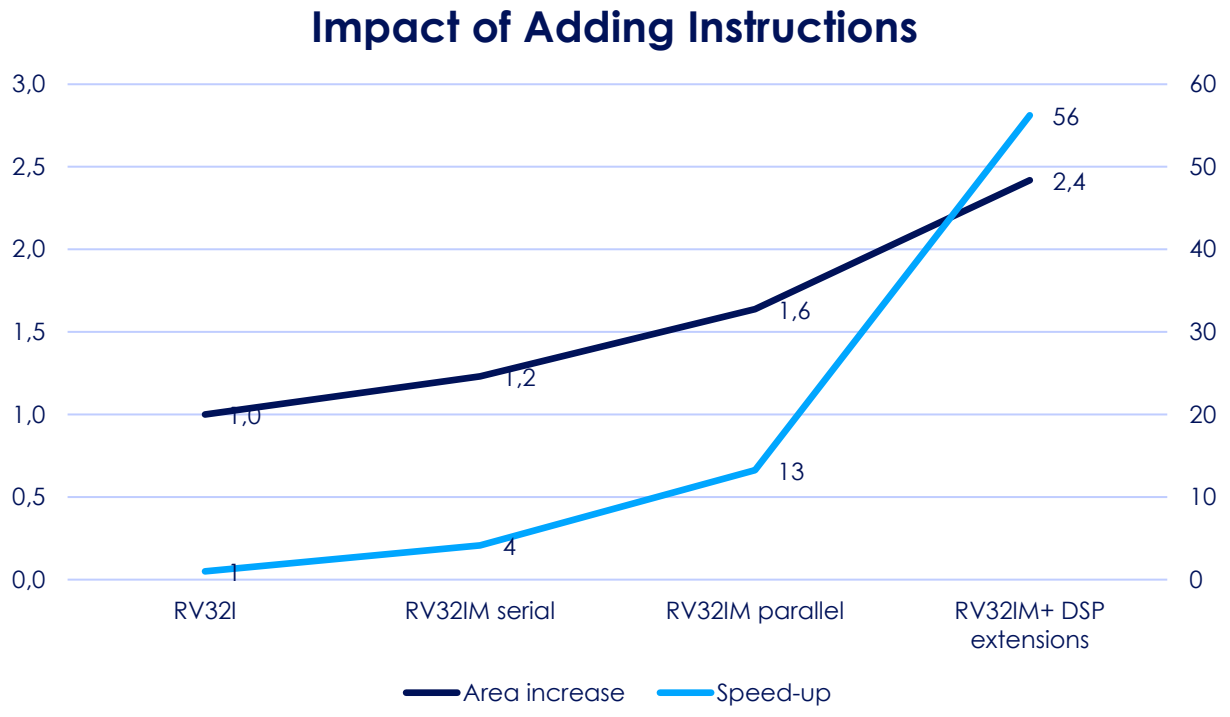
# 5 Use Case: Processor Optimized for Audio Processing

Many commonly used algorithms do not perform very well on a general-purpose core. Typical examples are ones for signal processing, cryptography, and machine learning. A real example is when Microsemi was interested in using RISC-V to replace a well-known commercial microcontroller core in their audio processing product range.

They faced multiple business challenges for the new product generation:

- Low power consumption was needed for Internet of Things (IoT) applications.
- Looking to minimize processor IP costs, especially eliminating royalties.
- Aiming to minimize mask-making and derivative design costs.
- Aiming to improve time-to-market.
- Wanting to improve both computational performance and code density.

RISC-V was an attractive option for cost reasons due to the lack of ISA royalties, and various core options were assessed for audio processing. The tests were undertaken using a minimal Codasip Bk3 RISC-V core as a starting point and adding some standard extensions as well as custom instructions using Codasip Studio.




The starting point was the minimal set of instructions RV32I. The corresponding base configuration RISC-V core had 16k gates. As the base instruction set has no multiplication instructions and the algorithms contained many multiplication instructions, it was not surprising that the computational performance was too slow.

Adding the multiplication extension RV32IM and using a sequential multiplier improved performance 4× while increasing the core area by 20 %. Using a parallel multiplier resulted in performance 13× of that of the original RV32I core with a 60 % penalty area.

Using multiplication extensions and hardware multiply was intuitively sensible. However, the real improvements came after profiling the software and devising a set of custom DSP extensions. Both tasks were done in Studio.

After profiling the resulting domain-specific processor, the final performance was 56× better than the original processor while only requiring 2.4× of the gatecount. This performance improvement without a major cost in gatecount was very cost-effective. Not only that, but the code size was reduced to 26 % of that of the base configuration.

By minimizing the combined core and instruction memory area the customer was able to manage their silicon cost and power consumption. In fact, they were able to avoid moving to a more expensive process node.

# 6 Use Case: Accelerating a Cryptography Algorithm

To further illustrate that extending a RISC-V core with just a single instruction can provide measurable improvements, let us consider the case of extending a Codasip Bk3 core for Veridify's (formerly SecureRF) WalnutDSA algorithm for authentication. Like other cryptography algorithms, the Walnut DSA performance on RISC-V was taking longer than expected. Joint investigation identified a bottleneck around multiplication in the Galois Field (32), which was taking 24 cycles where one cycle was desirable.

A single cycle instruction (quad_gmul32) was defined in CodAL and implemented in combinational logic by Studio. A wrapper was created for the new instruction and the algorithm was used as before.

The modified Bk3 RISC-V processor was ~2 % larger than the original version, however it delivered 3× speed improvement. The code size was also reduced by 30 % which would improve the area and power consumption of the instruction memory.

# 7 Conclusion

Domain-specific processors are emerging as a silicon-efficient way of delivering added processing performance. The RISC-V ISA is ideal for creating such processors thanks to its modularity and in-built support for custom instructions. The RISC-V ISA offers a wide range of ISA groups to choose from, with only the basic group (I/E) mandatory and the rest optional. The concept allows designers to select precisely what they need. If this is still insufficient for achieving the desired results, the RISC-V specification allows adding extra instructions while keeping RISC-V-compliant. This is a significant advantage, enabling you to reuse the community software stack and accelerate only the crucial parts.

Adding custom instructions can be done manually, but it is a time-consuming and error-prone task that also requires substantial amount of resources. Codasip targets this issue with its EDA tool Codasip Studio. A RISC-V processor and any added custom instructions are described in a high-level architecture description language CodAL, which is then used by Studio to generate an SDK and implementation of the processor. The high level of automation allows adding new instructions within hours, days, or weeks at most.

A domain-specific processor created by using RISC-V custom instructions can be very efficient compared with off-the-shelf processor cores. In the audio processing example, a 2.4× increase in gatecount yielded a 56× increase in performance. Adding a single instruction resulted in a 3× speedup of a digital signature algorithm. Similar gains have been made in other application areas. There are no limitations to the usage of Codasip Studio across domains in order to produce unique and measurably improved results.